موسسه بابان

انتشارات بابان و انتشارات راهیان ارشد

درس و کنکور ارشد

مهندسی نرم‌افزار

(ویژه کنکور ارشد ۱۴۰۲)

ویژه‌ی داوطلبان کنکور کارشناسی ارشد مهندسی IT

براساس کتاب مرجع

راجر اس. پرسمن

ارسطو خلیلی‌فر

تقدیم به:

تمامی آنانی که برای پیشرفت و سعادت خود و بشریت

تلاش می کنند.

ارسطو خلیلی فر

فهرست مطالب

۱ مفاهیم اولیه مهندسی نرم‌افزار

در طی ده‌ها سال از ایجاد و بکارگیری کامپیوتر تاکنون، علوم کامپیوتر در زمینه‌های مختلف، پیشرفت چشمگیری داشته است. در زمینه‌ی سخت‌افزار از کامپیوترهای لامپی به سمت ابرکامپیوترها پیشرفت کرده است و در نرم‌افزار، برنامه‌های به زبان ماشین به نرم‌افزارهای هوشمند و زبان‌های نسل چهارم توسعه یافته است. در مورد کاربرد نیز، کامپیوتر از انجام کارهای محدود و خاص بیرون آمده و اکنون در سطوح مختلف، همچون خانه‌ها، مدارس، دانشگاه‌ها، ادارات، سازمان‌ها و اماکن تجاری موارد استفاده وسیعی را به خود اختصاص داده است.

با وجودی که بیش از چند دهه از پیدایش نرم‌افزار نمی‌گذرد، این پدیده‌ی شگفت‌آور قرن بیستم، به عنوان یکی از مؤلفه‌های کلیدی فناوری اطلاعات تأثیر شگرفی بر کلیه‌ی جوانب زندگی بشر داشته است. روش‌های درمان بیماری‌ها، روش‌های یادگیری، روش‌های کسب و کار و به طور خلاصه کلیه‌ی جوانب زندگی به شدت تحت تأثیر قرار گرفته است. بدین ترتیب بشر توانسته از مرزها و قلمروهای پیشین عبور کند و قدم در دنیای پررمز و راز هستی نهد. دسترسی به فضای بیکران آسمان‌ها از یک سو و ورود به دنیای اتم‌ها در مقیاس نانو از سوی دیگر، نمونه‌های آشنایی از تأثیرات و جلوه‌های بکارگیری فناوری اطلاعات، به خصوص نرم‌افزار است.

بنابراین نرم‌افزار به عنصری کلیدی در تکامل محصولات و سیستم‌های مبتنی بر کامپیوتر تبدیل شده است. طی ۵۰ سال اخیر، نرم‌افزار از یک ابزار تحلیل اطلاعات و حل مسئله، به صنعتی مستقل تکامل یافته است.

نرم‌افزار

نرم‌افزار، ماهیتی منطقی است، که بر اساس مورد کاربرد، گاه درون یک محصول سخت‌افزاری مانند تلفن همراه و گاه درون یک محیط عملیاتی مانند دانشگاه قرار می‌گیرد و سپس تمام یا بخشی از روال کسب و کار را که به صورت دستی و سنتی انجام می‌شده است. به شیوه‌ای مدرن، مکانیزه و کامپیوتری انجام می‌دهد.

ویژگی‌های نرم‌افزار

برای درک مفهوم نرم‌افزار به بررسی آن دسته از ویژگی‌های نرم‌افزار که آن را از دیگر محصولات تولید شده توسط انسان متمایز می‌سازد، می‌پردازیم. هنگامی که سخت‌افزاری ساخته می‌شود، فرآیند تولید سخت‌افزار (ارتباط، برنامه‌ریزی، مدل‌سازی (تحلیل و طراحی)، ساخت (پیاده‌سازی و تست) و استقرار)، در نهایت به یک شی فیزیکی منتهی می‌شود. در حالی که نرم‌افزار یک عنصر سیستماتیک و منطقی است و نه فیزیکی، بنابراین نرم‌افزار دارای خصوصیاتی است که منجر به تفاوتی چشمگیر با سخت‌افزار می‌شود.

۱- نرم‌افزار توسعه می‌یابد.

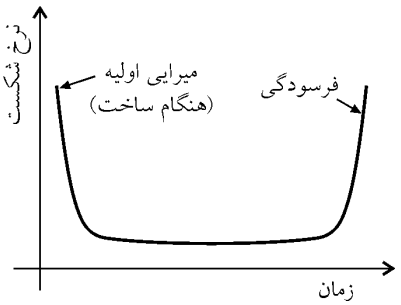
یعنی به مفهوم کلاسیک ساخته نمی‌شود، اگرچه شباهت‌هایی بین توسعه نرم‌افزار و ساخت سخت‌افزار وجود دارد، اما این دو فعالیت با یکدیگر تفاوت‌های اساسی دارند. هر چند در هر دوی آنها، کیفیت بالا، حاصل فرآیند تولید (ارتباط، برنامه‌ریزی، مدل‌سازی (تحلیل و طراحی)، ساخت (پیاده‌سازی و تست) و استقرار) خوب خواهد بود. اما مرحله‌ی ساخت در مورد سخت افزار می‌تواند یک سری مشکلات کیفی داشته باشد که در مورد نرم‌افزار وجود ندارد یا به راحتی حل‌شدنی و رفع‌شدنی خواهد بود. هر دو فعالیت وابسته به مردم هستند اما ارتباط بین افراد متخصص و کار صورت گرفته کاملاً متفاوت است، هر دو فعالیت مستلزم ساختن یک محصول هستند، اما روش‌ها کاملاً متفاوتند، توسعه‌ی نرم‌افزار بیشتر نیازمند فعالیت‌های فکری و منطقی می‌باشد در حالی که در تولید محصولات فیزیکی، فعالیت‌های یدی و فیزیکی بیشتر به کار می‌آیند.

توجه: هزینه‌های نرم‌افزار در مهندسی آن متمرکز است، به عبارت دیگر هزینه‌ی تولید و توسعه‌ی نرم‌افزار بیشتر بر روی فعالیت‌های مهندسی و مدل‌سازی (تحلیل و طراحی) متمرکز می‌باشد در حالی که هزینه‌ی تولید و توسعه‌ی محصول فیزیکی بیشتر بر روی مواد خام و اولیه و تولید (پیاده‌سازی) آنها متمرکز می‌باشد.

این بدان معناست که پروژه‌های نرم‌افزاری را نمی‌توان همانند پروژه‌های تولید معمولی مدیریت کرد، در واقع مدیریت پروژه‌های نرم‌افزاری بسیار متفاوت از مدیریت پروژه‌های فیزیکی می‌باشد.

۲- نرم‌افزار فرسوده نمی‌شود.

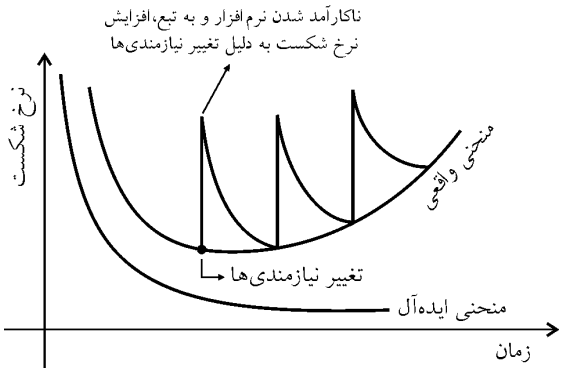
شکل مقابل میزان شکست را به عنوان تابعی از زمان در خصوص سخت‌افزار نشان می‌دهد. این رابطه که اغلب «**منحنی وان**» نامیده می‌شود، نشانگر این است که سخت‌افزار در اوایل عمرش میزان عدم موفقیت نسبتاً بالایی دارد، این شکست‌ها اغلب به اندازه‌گیری های نادرست و به تبع مدل‌سازی (تحلیل و طراحی) نادرست یا نقص تولیدی در مرحله



منحنی شکست سخت‌افزار

ساخت (پیاده‌سازی) مثل شکننده شدن قطعه‌ای به دلیل استفاده از آلیاژ نامناسب نسبت داده می شوند. نقایص اصلاح شده و میزان شکست برای مدتی به سطح ثابتی می‌رسد. اما با گذشت زمان، سخت‌افزار شروع به فرسوده شدن کرده و میزان شکست کار دوباره افزایش می‌یابد.

اما نرم‌افزار در معرض عوامل محیطی که سخت‌افزار را فرسوده می‌کند، نمی‌باشد. بنابراین از نظر تئوری منحنی عدم موفقیت در مورد نرم‌افزار شکل یک منحنی ایده‌آل را می‌گیرد که در شکل مقابل نشان داده شده است.



مشکلات شناسایی نشده در اوایل کار نرم‌افزار باعث عدم موفقیت در کار می‌شود. این نقایص برطرف می‌شود و به تبع نرخ شکست کاهش می‌یابد و اگر به صورت ایده‌آل، نیازمندی‌های نرم‌افزار تغییر نکند، نمودار شکست نرم‌افزار مطابق منحنی ایده‌آل خواهد بود. بدین معنی که نرم‌افزار در حالت ایده‌آل پس از مدتی که از تولید آن گذشت و اشکالات مربوط به آن برطرف شد دیگر برای همیشه و بدون اشکال قابل استفاده می‌باشد. اما واقعیت بدین گونه نیست در واقع تغییر نیازهای کاربر باعث بروز خلل در کارایی یک نرم‌افزار می‌شود و با گذشت زمان به علت بروز این تغییرات، نرم‌افزار دیگر قادر نخواهد بود تا نیازهای کاربر را برآورده کند. به بیان دیگر تغییرات نیازمندی‌های مشتری، سبب ناکارآمد شدن نرم‌افزار می‌گردد. بنابراین نرم‌افزار برای برآورده کردن نیازهای جدید کاربر نیازمند اعمال تغییرات می‌باشد. اگر نیازمندی‌های نرم‌افزار به صورت واقعی تغییر کند، نمودار شکست نرم‌افزار مطابق منحنی واقعی خواهد بود. در منحنی واقعی پس از هر تغییر نیاز در نرم‌افزار نرخ شکست افزایش می‌یابد و با برآورده کردن آن، کاهش می‌یابد. در یک بیان دقیق‌تر تغییر نیازمندی‌ها، سبب ناکارآمد شدن نرم‌افزار می‌گردد که پس از رفع نیازها نرم‌افزار مجددا کارآمد خواهد شد. اما در کل آینده نرم‌افزار مورد استفاده رو به زوال است، زیرا در گذر زمان این‌بار این تغییرات تکنولوژی است که سبب ناکارآمد شدن نرم‌افزار می‌شود که منجر به این می‌شود که نرم‌افزار کنار گذاشته شود.

۳- مونتاژ قطعات

صنعت در حال حرکت به سمت مونتاژ قطعات است، اما نرم‌افزارها بیشتر بر اساس نیاز مشتریان و به صورت سفارشی ساخته می‌شوند و در دنیای سخت‌افزار، استفاده مجدد از قطعات، بخشی طبیعی از فرآیند مهندسی است در واقع ماهیت سخت‌افزار این امکان را می‌دهد، تا به طور جداگانه هر یک از اجزای محصول (حتی در مکان‌های متفاوت) به صورت جداگانه ساخته شود و در نهایت با یکدیگر مونتاژ گردند.

اما در مهندسی نرم‌افزار این امر به تازگی مورد توجه قرار گرفته است و استفاده از مؤلفه‌های آماده جهت ساخت نرم‌افزار به تازگی مرسوم شده است. به عبارت دیگر هنوز مزایای استفاده از مولفه‌های نرم افزاری آماده به خوبی و به طور کامل روشن نشده است. امروزه، ایده‌ی استفاده مجدد نه تنها الگوریتم‌ها، بلکه ساختار داده‌ها را نیز در بر می‌گیرد. اجزاء مدرن قابل استفاده مجدد (کلاس)، هم دارای داده می‌باشند و هم شیوه پردازش مخصوص آن داده‌ها را شامل هستند که مهندسی نرم‌افزار را قادر می‌سازد تا برنامه‌های کاربردی جدیدی را از روی قطعات قابل استفاده مجدد بسازد. به طور مثال، امروزه رابطه‌های گرافیکی کاربر با استفاده از اجزای قابل استفاده مجدد ساخته می‌شوند که ایجاد پنجره‌های گرافیکی، منوهای بازشونده، جعبه متن‌ها و دکمه‌ها را میسر می‌سازد.

بحران نرم‌افزاری

حدود ۵۰ سال پیش، یعنی در اوایل پیدایش نرم‌افزار، مصرف‌کنندگان این محصول نوین، همان طراحان و تولیدکنندگان آن بودند. در آن زمان، نرم‌افزار عمدتاً برای محاسبات و حل مسائل ریاضی استفاده می‌شد. وجود زبان‌های سطح پایین و محدودیت‌های سخت‌افزاری (کمبود حافظه و سرعت پردازش کم) از دیگر مشخصه‌های دوران اولیه پیدایش نرم‌افزار است. در آن روزهای اولیه، نرم‌افزار، مقوله‌ای جدا از سخت‌افزار نبود و حتی برای فروش سخت‌افزار، به طور رایگان در آن تعبیه می‌شد! اما با گسترش دامنه کاربرد کامپیوتر و به دنبال آن نرم‌افزار در زمینه‌های مختلف، به تدریج شرایطی به وجود آمد که استفاده‌کنندگان و کاربران نرم‌افزار از طراحان و تولیدکنندگان آن جدا شدند و سازمان‌ها و شرکت‌هایی به وجود آمدند که کارشان صرفاً تولید نرم‌افزار بود. حالا دیگر نرم‌افزار قیمت داشت و اتفاقاً برخلاف روند کاهش قیمت در سخت‌افزارها، روز به روز قیمت نرم‌افزار افزوده می‌شد. نیازهای جدید استفاده کنندگان فراتر از محاسبات بود، آنها به مدیریت اطلاعات نیاز داشتند.

در همان نسل‌های ابتدایی تکامل نرم‌افزار تولید و فروش کامپیوترهای شخصی به دلیل تقاضای فراوان از سوی مصرف‌کنندگان و خریداران به شدت افزایش یافت. بنابراین نیاز به برنامه‌های مختلف کامپیوتری به شدت احساس گردید و این امر سبب تولید فراوان نرم‌افزار شد بدون آنکه قانونی، عمل نظارت بر تولید نرم‌افزار را داشته باشد و همین مسئله نارضایتی‌های زیادی را برای مصرف‌کنندگان این نرم‌افزارها به بار آورد، این مشکلات و چالش‌ها به قدری جدی و پرهزینه بود که از آن به «**بحران نرم‌افزار**» یاد می‌شد. این بحران در سال‌های ۱۹۶۰ تا ۱۹۷۰ به شکل پیچیده‌ای به اوج خود رسید و درست در همان سال‌ها بود که بحث «**مهندسی نرم‌افزار**» به شکل جدی‌تر مطرح شد.

اگر بخواهیم به تعدادی از دلایل بحران نرم‌افزاری و مشکلات به وجود آمده اشاره کنیم می‌توان به موارد زیر اشاره کرد:

۱- هزینه‌های بالایی که برای تولید نرم‌افزار صرف می‌شد.
۲- نرم‌افزار تولید شده تمام نیازهای مشتری را برآورده نمی‌کرد.

۳- تحویل به موقع نرم‌افزار امکان‌پذیر نبود.
۴- پیشرفت سخت‌افزار بسیار سریع بوده و امکان رقابت نرم‌افزار با آن ممکن نبود.
۵- خطاهای موجود در نرم‌افزار بسیار زیاد بوده و برای رفع آن مشکلاتی وجود داشت.
۶- امکانات توسعه نرم‌افزار، قدرت نگهداری و پشتیبانی بسیار محدود بود.

همه موارد گفته شده دست به دست هم داد و باعث گردید، نرم‌افزار با بحران مواجه شود. بنابراین همگی به فکر تولید نرم‌افزار مطابق اصول مهندسی افتادند.

خصوصیات پروژه‌های موفق

بر اساس آمارهای معتبری که توسط مؤسساتی مانند IDC[1] و Standish Group و در پی بررسی هزاران پروژه‌ی نرم‌افزاری که در ابعاد و زمینه‌های مختلف تهیه شده است، درصد زیادی از پروژه‌های نرم‌افزاری در دنیا با شکست و عدم موفقیت مواجه می‌شوند. از نگاه مشتری یک پروژه موفق نرم‌افزاری، پروژه‌ای است که بر اساس سه خصوصیت اساسی زیر تولید گردد:

۱- بازه‌ی زمانی از قبل برنامه‌ریزی شده (بازه‌ی زمانی مشخص)
۲- بودجه‌ای از قبل پیش‌بینی شده و با صرف کمترین هزینه (مقرون به صرفه)
۳- دقیقاً مطابق با نیازمندی‌های واقعی کاربران (کیفیت مطلوب)

اما همانند آنچه پیش از این درباره‌ی دوران بحران نرم‌افزاری بیان کردیم، این خصوصیات اساسی به درستی محقق نمی‌گردید. سرانجام برای اولین بار، در سال ۱۹۶۸ و در یک کنفرانس علمی که توسط ناتو در کشور آلمان برگزار شد، بر لزوم مهندسی این دستاورد جدید بشر، یعنی **نرم‌افزار**، تأکید گردید. از آن به بعد با گسترش روش‌های مهندسی، ابزارها، دانش و تجربه، صنعت نرم‌افزار به یکی از صنایع برتر جهانی تبدیل شد.

مهندسی نرم‌افزار

مهندسی نرم‌افزار شاخه‌ای از مهندسی است که با بهره‌گیری از دانش علمی، راه‌حل‌های مقرون به صرفه‌ای را در قالب دستاوردهای نرم‌افزاری به منظور حل مسائل و مشکلات علمی و خدمت به جامعه‌ی بشری ارائه می‌نماید. مهندسی نرم‌افزار یک رویکرد سیستماتیک، قاعده‌مند و قابل اندازه‌گیری برای توسعه، اجرا و نگهداری نرم‌افزار است. در مهندسی نرم‌افزار از قوانین مهندسی دقیق برای تهیه‌ی نرم‌افزار مقرون به صرفه استفاده می‌شود، به طوری که قابل اطمینان باشد و در محیط واقعی به طور کارآمد فعالیت کند. در یک بیان کامل، مهندسی نرم‌افزار نظامی است یکپارچه، شامل فرآیندها، روش‌ها و ابزارها که منجر به ایجاد نرم‌افزاری در بازه‌ی زمانی از قبل برنامه‌ریزی شده، بودجه‌ای از قبل پیش‌بینی شده و دقیقاً مطابق با نیازمندی‌های واقعی کاربران می گردد. این تعریف، مستقل از ابعاد و ماهیت پروژه، مطابق آنچه پیش از این در مورد خصوصیات

[1] International Data Corporation

پروژه‌های موفق نرم‌افزاری بیان کردیم، تولید پروژه‌های نـرم‌افـزاری را بـه سـمت موفقیـت سـوق می‌دهد. انتخاب فرآیند مناسب، روش مناسب و ابزار مناسب براساس ماهیت نرم‌افزار، کارآمدی به ارمغان خواهد آورد. مهندسی نرم‌افزار نیز بر انتخاب فرآیند مناسب، روش مناسب و ابـزار مناسـب بر اساس ماهیت نرم‌افزار تاکید کرده‌است. به عبارت دیگر اگر فرآیند مناسب، روش مناسب و ابزار مناسب بر اساس ماهیت نرم‌افزار انتخاب گردد، آنگاه می‌توان انتظار داشـت محصـول نـرم‌افـزاری ایجاد گردد که در بازه‌ی زمانی از قبل برنامه‌ریزی شده، بودجه‌ای از قبل پـیش‌بینـی شـده و دقیقـاً مطابق با نیازمندی‌های واقعی کاربران باشد. یک پدر خوب الزاما مدیر خوب هم نیست!

به قول حضرت مولوی

هرکسی را بهرکاری ساختند میل آن را در دلش انداختند

مهندس نرم‌افزار

مهندس نرم‌افزار فردی است حرفه‌ای که خود را در برابر مشتری مسئول می‌دانـد و همـواره در پی دستیابی به ارزش موردنیاز اوست. رسالت و مأموریت یک مهندس نرم‌افزار، حل خواسـته‌هـای مشتری است. بنابراین بایستی فرآیند کار را به طور کامل انجام دهـد و بـه نتیجـه دلخـواه مشـتری دست یابد. مهندسین نرم‌افزار به عنوان افراد حرفه‌ای، به جای انجام وظیفه‌های جداگانه بـه عنـوان یک شغل، مسئول نتیجه‌گیری از کل کار هستند.

بیمار برای گرفتن فشار خون و یا بازدید قلب به نـزد پزشـک نمـی‌رود. هـدف از مراجعـه بـه پزشک، بهبودی و به دست آوردن سلامتی دوباره است. هدف پزشک، به عنوان یک فرد حرفـه‌ای، دستیابی به نتیجه نهایی است، نه فعالیت‌هایی که در راه رسیدن به آن انجام می‌دهد.

در واقع مهندس نرم‌افزار فردی است که با استفاده از فرآیندها، روش‌ها و ابزارهای موجـود بـه کمک علم و دانش خود و پس از تجزیه و تحلیل مسئله آن را پیاده‌سازی و مدیریت کنـد و نهایتـاً محصول تلاش خود را که در بازه‌ی زمانی از قبل برنامه‌ریزی شده، بودجه‌ای از قبل پیش‌بینی شده و دقیقاً مطابق با نیازمندی‌های واقعی کاربران است که معرف زحمات و معلومات اوست در اختیار مشتری قرار دهد.

بنابراین یک مهندس نرم‌افزار باید با مسـائل، مفـاهیم، فرآینـدها، روش‌هـا و ابزارهـا، اصـول و قواعد مرتبط با حوزه‌ی تخصصی خود به خوبی آشنا باشد. این مهارت موجب مـی‌شـود کـه یـک مهندس نرم‌افزار بتواند در مقاطع زمانی مختلـف در فرآینـد انجـام یـک پـروژه بسـته بـه شـرایط، نیازمندی‌ها، اهداف و محدودیت‌ها، برای انجام هر فعالیتی، مناسب‌تـرین فرآینـد، روش و ابـزار را انتخاب نماید که متعاقبـاً منجـر بـه کـاهش هزینـه‌هـا، زمـان تولیـد و نیـز افـزایش کیفیـت (رفـع نیازمندی‌های مشتری) خواهد شد. بدین ترتیب بستر لازم برای حرکت به سمت انجام یک پـروژه ی مهندسی موفق مهیا می‌گردد.

بسیاری از مردم نسبت به یک مهندس نرم‌افزار دیدی نادرست دارند و تصور می‌کنند کـه یـک مهندس نرم‌افزار کسی است که عمل برنامه‌نویسی را انجام می‌دهد، ولی وظیفـه‌ی اصـلی مهنـدس نرم‌افزار چیزی غیر از این تصور است. یک مهنـدس نـرم‌افـزار مسـأله را از دیـدگاه‌هـای مختلـف

بررسی می‌نماید و با استفاده از اصول مهندسی نرم‌افزار یعنی فرآیندها، روش‌ها و ابزارها به تجزیه و تحلیل آن پرداخته و بهترین راه‌حل را برای انجام پروژه‌های نرم‌افزاری انتخاب و پیاده‌سازی می‌کند، بنابراین می‌بینید که برنامه‌نویسی فقط می‌تواند جزئی از کارهای یک مهندس نرم‌افزار باشد و وظیفه اصلی او چیز دیگری است. او برای انجام رساندن درست پروژه آن را مدیریت کرده و با نظارت کامل بر مراحل انجام پروژه به تست آن می‌پردازد. او پس از انجام تست، مسئول مراقبت و نگهداری پروژه است و در صورتی که نیاز به تکامل یک پروژه باشد، او این کار را انجام می‌دهد، البته برای انجام این کار می‌تواند از گروه همراه خود نیز استفاده کند.

توجه: در مباحث مهندسی نرم‌افزار کیفیت محصول بر اساس میزان رضایت‌مندی مشتری از محصول نرم‌افزاری که قرار است نیازمندی‌های او را برآورده سازد سنجش می‌شود.

مؤلفه‌های نرم‌افزار

نرم‌افزار محصولی است که به واسطه‌ی فرآیند تولید نرم‌افزار و تحت نظارت مهندسی نرم‌افزار، تحلیل، طراحی و پیاده‌سازی و تست می‌گردد تا در نهایت بر اساس ورودی‌های موردنظر مشتری، خروجی‌های مورد انتظار مشتری را برآورده سازد و شامل مؤلفه‌های زیر می‌باشد:

۱- **ساختار داده‌ای:** محل نگهداری داده‌های محیط عملیاتی به شکل متغیرها و جداول.

۲- **عملکرد:** دستورات یا کدهای قابل اجرا که باعث انجام وظایف مورد نظر می‌شود که به برنامه کاربردی موسوم است.

۳- **مستندات:** شامل توصیف مدل‌های تحلیل و طراحی نزد سازنده و راهنمای کاربر نزد کاربران نهایی و مشتری.

توجه: هر یک از این مؤلفه‌ها شامل یک پیکربندی است که بر اساس اصول مهندسی نرم‌افزار (فرآیندها، روش‌ها و ابزارها) ایجاد می‌گردد.

انواع نرم‌افزارها

در یک دسته‌بندی کلی برای نرم‌افزارهای موجود، می‌توان آنها را به دو گروه اصلی زیر تقسیم نمود:

نرم‌افزارهای کاربردی

برنامه‌هایی که برای رفع نیازهای کاربران کامپیوتر نوشته می‌شود، به بیان دیگر نرم‌افزارهای کاربردی به‌طور مستقیم به انسان سرویس می‌دهند. مانند نرم‌افزارهای حسابداری و نرم‌افزار فرهنگ لغات.

نرم‌افزارهای سیستمی

برنامه‌هایی که برای بهره‌برداری از سخت‌افزار یا سرویس‌دادن به سایر برنامه‌ها نوشته شده‌اند. به‌بیان دیگر نرم‌افزارهای سیستمی به طور مستقیم به نرم افزارهای دیگر و به طور غیرمستقیم به انسان سرویس می‌دهند. مانند سیستم عامل‌ها و کامپایلرها.

در یک دسته‌بندی دقیق‌تر برای نرم‌افزارهای موجود، می‌توان آنها را به گروه‌های زیر تقسیم نمود:

نرم‌افزارهای بی‌درنگ

در نرم‌افزارهای بی‌درنگ باید خروجی و پاسخ نهایی در یک زمان مشخص و از پیش تعیین شده حاصل شود. در این نرم‌افزارها، زمان نقشی کلیدی ایفا می‌کند و زمان پاسخ باید به موقع و تضمین شده باشد. نرم‌افزارهای بی‌درنگ معمولاً به عنوان یک دستگاه کنترلی در یک کاربرد خاص (مثلاً صنعتی) به کار گرفته می‌شوند. در این نرم‌افزارها دیر پاسخ دادن به همان بدی پاسخ ندادن است. در این نوع نرم‌افزارها هدف اصلی طراحان، پاسخگویی سریع (در مهلت تعیین شده) به رویدادها و درخواست‌ها می‌باشد و راحتی کاربران و بهره‌وری منابع در درجه‌های بعدی اهمیت، قرار دارند.

انسان در وادی زندگی نیازهای گوناگونی دارد، یکی از نیازهای اساسی انسان، نیاز به امنیت است. اما گاهاً، ممکن است در معرض عوامل محیطی و بیرونی و یا حتی درونی امنیت انسان در شرایط هشیاری یا ناهشیاری به مخاطره بیفتد. بنابراین نیاز است تا مکانیزمی همواره هوشیار و همیشه بیدار و با اشراف لحظه به لحظه، مخاطرات پیرامون انسان را رصد و تحت کنترل خود قرار دهد تا در موقع لزوم و به صورت آنی، بی‌درنگ، در لحظه و در زمان حقیقی و واقعی (تا دیر نشده) با تهدید مقابله کند، نرم‌افزارهای بی‌درنگ این نگهبان همیشه هوشیار و همیشه بیدار هستند. مانند نرم‌افزارهای ترمز اتومبیل، کنترل ضربان قلب اتاق بیهوشی، کنترل فشار کابین هواپیما و ...

نرم‌افزارهای مدیریت پایگاه داده

این نرم‌افزارها، برای کاربردهای پایگاه داده، مورد استفاده قرار می‌گیرند، مانند نرم‌افزار SQL Server به عنوان یک DBMS که ایجاد جداول و پرس و جوهای مربوط به یک پایگاه داده را فراهم می‌کند. در این نوع نرم‌افزارها حجم داده‌ها بالا و حجم محاسبات پایین است.

نرم‌افزارهای علمی و مهندسی

این نرم‌افزارها، برای کاربردهایی با محاسبات پیچیده و سنگین مورد استفاده قرار می‌گیرند، مانند نرم‌افزارهای محاسبات ریاضی (مثل ضرب ماتریس‌ها)، علوم زمین‌شناسی و ستاره‌شناسی، کنترل سیستم‌های صنعتی. در این نوع نرم‌افزارها حجم داده‌ها پایین و حجم محاسبات بالا است.

نرم‌افزارهای نهفته (توکار)

این نرم‌افزارها در محصولات صنعتی که به بازار عرضه می‌شوند، گنجانده می‌شوند و دارای کارکردهای محدود (مانند کنترل فعالیت‌های ماشین لباسشویی) و یا دارای کارکردهای حیاتی (مانند کنترل ترمز اتومبیل، کنترل ضربان قلب) می‌باشند. اغلب این نرم‌افزارها، از نوع نرم‌افزارهای بی‌درنگ نیز هستند، به بیان دیگر، اغلب نرم‌افزارهای بی‌درنگ، نهفته هستند.

نرم‌افزارهای خط تولیدی

این نرم‌افزارها بر طراحی و آماده‌سازی یک سرویس ویژه که مشتریان بسیاری خواهان آن هستند، تأکید می‌کند. این نرم‌افزارها بر رقابت‌های تجاری تمرکز دارند و تلاش می‌کنند، محصولاتی تولید کنند که مشتری‌های زیادی در بازار داشته باشد (مانند نرم‌افزارهای پردازش متن، گرافیک کامپیوتری، برنامه‌های تفریحی و برنامه‌های چندرسانه‌ای).

نرم‌افزارهای مبتنی بر وب

این نرم‌افزارها طیف وسیعی از برنامه‌هایی را شامل می‌شود که از طریق شبکه‌های کامپیوتری در دسترس هستند. در فرم ساده، این نرم‌افزارها می‌توانند شامل چند لینک صفحات و فایل‌های مختلف بوده و اطلاعاتی را به کاربر نمایش دهند. در فرم‌های پیچیده‌تر، این نرم‌افزارها می‌توانند محاسبات و پردازش‌هایی را در بستر شبکه انجام داده و یا حتی اطلاعات کاربران را در ساختار پایگاه داده ذخیره و بازیابی نمایند.

نرم‌افزارهای هوش مصنوعی

این نرم‌افزارها از الگوریتم‌های غیرعددی برای حل مسائل پیچیده کمک می‌گیرند. مسائلی که با محاسبات متعارف قابل حل نبوده و روش حل آسانی ندارند. از حوزه‌های مختلف این نرم‌افزارها، می‌توان به نرم‌افزارهای ربات‌ها، شناسایی الگو (تصویر و صدا)، شبکه‌های عصبی مصنوعی و یا بازی‌های کامپیوتری اشاره کرد.

نرم‌افزارهای متن باز

این نرم‌افزارها با در اختیار گذاشتن کدهای منبع، این امکان را فراهم می‌کنند که مشتریان بتوانند اصلاحاتی را با توجه به اقتضای محلی خودشان در آنها اعمال کنند.

تست‌های فصل اول

۱- یک سیستم نرم‌افزاری باید ..... . (مهندسی کامپیوتر – آزاد ۷۱)

۱) اقتصادی باشد.
۲) ضریب اطمینان آن بسیار بالا باشد.
۳) از استانداردهای خاصی تبعیت کند.
۴) همه موارد

۲- انفورماتیک: (مهندسی کامپیوتر – آزاد ۷۲)

۱) یعنی شناسایی اجزای داخلی یک کامپیوتر و نحوه کار آن اجزا
۲) به کاربردهای سیستم‌ها و تکنیک‌ها در جهت ذخیره‌سازی و انتقال بهینه اطلاعات علمی اطلاق می‌شود.
۳) مترادف با علوم نظری کامپیوتری می‌باشد.
۴) معادل با اطلاعات (Information) می‌باشد.

۳- مهندسی نرم‌افزار: (مهندسی کامپیوتر – آزاد ۷۲)

۱) یعنی به‌کارگیری عملی علوم کامپیوتر، مدیریت، اقتصاد و ... در جهت طراحی و ساخت سیستم‌های نرم‌افزاری کلان
۲) یعنی به‌کارگیری علوم کامپیوتر در طراحی و ساخت سیستم‌های عامل
۳) یعنی برنامه‌ریزی ریزبرنامه (Micro Code) برای پردازنده اصلی کامپیوتر
۴) هیچ‌کدام

۴- کدام‌یک از عبارات زیر در مورد نرم‌افزار صحیح می‌باشد؟ (مهندسی IT – آزاد ۸۹)

۱) نرم‌افزار ساخته می‌شود.
۲) نرم‌افزار با گذشت زمان فرسوده می‌گردد.
۳) هزینه نرم‌افزار در مهندسی آن متمرکز است.
۴) نرم‌افزار یک عنصر سیستمی فیزیکی است نه منطقی.

پاسخ تست‌های فصل اول

۱- گزینه (۴) صحیح است.

بر اساس آمارهای معتبری که توسط مؤسساتی مانند IDC و Standish Group و در پی بررسی هزاران پروژه‌ی نرم‌افزاری که در ابعاد و زمینه‌های مختلف تهیه شده است، درصد زیادی از پروژه‌های نرم‌افزاری در دنیا با شکست و عدم موفقیت مواجه می‌شوند. از نگاه مشتری یک پروژه موفق نرم‌افزاری، پروژه‌ای است که بر اساس سه خصوصیت اساسی زیر تولید گردد:

۱- بازه‌ی زمانی از قبل برنامه‌ریزی شده (بازه‌ی زمانی مشخص)

۲- بودجه‌ای از قبل پیش‌بینی شده و با صرف کمترین هزینه (مقرون به صرفه)

۳- دقیقاً مطابق با نیازمندی‌های واقعی کاربران (کیفیت مطلوب)

۲- گزینه (۲) صحیح است.

انفورماتیک به معنی فناوری‌هایی برای ذخیره‌سازی، پردازش، بازیابی، انتقال و مدیریت اطلاعات به کار می‌رود.

گزینه دوم به نسبت کامل‌تر از بقیه می‌باشد. زیرا فقط به ذخیره‌سازی و انتقال اطلاعات اشاره کرده است. در حالی که پردازش اطلاعات نیز مهم است. گزینه‌های اول و سوم با موضوع مرتبط نیستند. گزینه چهارم اطلاعات را معادل انفورماتیک در نظر گرفته در حالی که اطلاعات جزئی از انفورماتیک است.

۳- گزینه (۴) صحیح است.

مهندسی نرم‌افزار نظامی است یکپارچه شامل فرآیندها، روش‌ها و ابزارها که منجر به ایجاد نرم‌افزاری در بازه‌ی زمانی از قبل برنامه‌ریزی شده، بودجه‌ای از قبل پیش‌بینی شده و دقیقاً مطابق با نیازمندی‌های واقعی کاربران می‌گردد. این تعریف مستقل از ابعاد و ماهیت پروژه، تولید پروژه‌های نرم‌افزاری را به سمت موفقیت سوق می‌دهد.

۴- گزینه (۳) صحیح است.

بررسی ویژگی‌های نرم‌افزار که آن را از سایر محصولات ساخت دست بشر متمایز می‌سازد، درک بهتری از مفهوم نرم‌افزار و به تبع آن مهندسی نرم‌افزار فراهم می‌کند. تقریباً عموم ساخته‌های بشری، حالت سخت‌افزاری و نمود فیزیکی دارند در حالی که نرم‌افزار یک عنصر سیستمی منطقی است. از این رو، نرم‌افزار دارای ویژگی‌هایی است که تفاوت عمده‌ای با ویژگی‌های محصولات سخت‌افزاری دارد.

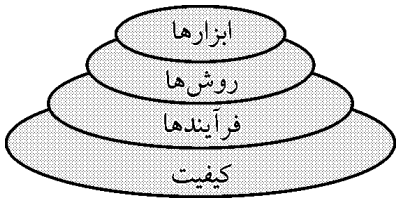
۱) نرم‌افزار به مفهوم کلاسیک ساخته نمی‌شود بلکه یک رویکرد مهندسی بر آن اعمال می‌شود. به بیان دیگر نرم‌افزار توسعه می‌یابد یا طراحی می‌شود، یعنی به مفهوم کلاسیک ساخته نمی‌شود. اگر چه میان توسعه‌ی نرم‌افزار و ساخت سخت‌افزار شباهت‌هایی وجود دارد لیکن این دو عمل، تفاوت بنیادی دارند. هزینه‌های نرم‌افزار در مهندسی آن متمرکز است. بنابراین پروژه‌های نرم‌افزاری را نمی‌توان مانند سایر پروژه‌های ساخت، مدیریت نمود.

۲) نرم‌افزار فرسوده نمی‌شود، یعنی نرم‌افزار در معرض عوامل محیطی که سخت‌افزار را فرسوده می‌کند نمی‌باشد بلکه نیازها تغییر می‌کند. برای افزایش قدرت پاسخگویی نرم‌افزار به نیازها باید آن را تصحیح و بروزرسانی کرد.

۳) بیشتر نرم‌افزارها به جای اسمبل شدن از مؤلفه‌های موجود به صورت سفارشی ساخته می‌شوند در حالی که محصولات دیگر عموماً توسط مؤلفه‌های موجود قابل تولید هستند. در دنیای سخت‌افزار، استفاده مجدد از قطعات، بخش طبیعی از فرآیند مهندسی است ولی این امر به تازگی در دنیای نرم‌افزار مورد توجه قرار گرفته است.

۲ مدل‌های فرآیند تولید نرم‌افزار

لایه‌های مهندسی نرم‌افزار



مهندسی نرم‌افزار یک فرآیند لایه‌ای است. به بیان دیگر مهندسی نرم‌افزار از چهار لایه تشکیل شده است. در شکل مقابل، چهار لایه نشان داده شده است:

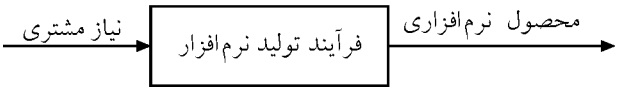
کیفیت

مهندسی نرم‌افزار یک کوشش لایه‌ای برای تولید یک محصول نرم‌افزاری با کیفیت که نیازمندی‌های مورد انتظار مشتری را برآورده می‌سازد می‌باشد. در صورتی که ابزارها، روش‌ها و فرآیندها به گونه‌ای درست و مطابق با کاربرد انتخاب و استفاده شوند می‌توان این‌گونه بیان نمود که کیفیت که همان برآورده ساختن نیازمندی‌های مورد انتظار مشتری می‌باشد برآورده شده است.

برای مثال کیفیت خانه برای برآورده ساختن آرامش و نیازهای مشتری، مهم است. اما شیوه‌های رسیدن به این مهم در شهرهای شمالی و جنوبی شباهت‌ها و تفاوت‌هایی دارد. به طور مثال در شهرهای شمالی بارندگی به وفور وجود دارد، پس روش‌ها و ابزارهای ساختمانی باید به گونه‌ای انتخاب شوند که در برابر بارندگی مقاوم باشند. مثل روش سقف شیروانی با استفاده از ابزار سفال!

فرآیندها (فرآیند تولید نرم‌افزار)

هر پروژه‌ی نرم‌افزاری، چه بزرگ و چه کوچک مراحلی را طی می‌نماید که در طی آن مجموعه‌ای از نیازمندی‌های مشتری به یک محصول نرم‌افزاری تبدیل می‌گردد. الگو و قالبی که چگونگی طی مراحل مختلف یک پروژه را تعریف می‌نماید، اصطلاحاً **فرآیند تولید نرم‌افزار** نامیده می‌شود. شکل زیر فرآیند تولید نرم‌افزار و ورودی و خروجی آن را نشان می‌دهد:



همانطور که ملاحظه می‌نمایید، ورودی این فرآیند، نیاز یا خواسته‌های مشتری و خروجی آن، یک محصول نرم‌افزاری است. یک فرآیند تولید در یک پروژه به ما می‌گوید که برای دستیابی به هدف مطلوب که همان تولید یک فرآورده‌ی نرم‌افزاری با کیفیت مطلوب است، چه کسی[1]، چه کاری را[2]، چه موقع[3] و چگونه[4] باید انجام دهد، در واقع، بدون داشتن تعریف مشترکی از فرآیند تولید نرم‌افزار، هماهنگی انجام کار تیمی در یک پروژه‌ی نرم‌افزاری، امکان‌پذیر نخواهد بود.

توجه نمایید که مدل فرآیندی که انتخاب می‌کنیم دقیقاً به نرم‌افزاری که تولید می‌کنیم بستگی دارد. ممکن است که یک مدل فرآیند برای تولید سیستم الکترونیکی هواپیما مناسب باشد، در حالی که مدل فرآیندی دیگر، برای ایجاد یک وب‌سایت استفاده شود. پس انتخاب مدل فرآیند براساس ماهیت نرم‌افزار صورت می‌گیرد. در ادامه‌ی این فصل به طور مفصل در مورد انواع مدل‌های فرآیند تولید نرم‌افزار صحبت خواهیم کرد.

روش‌ها

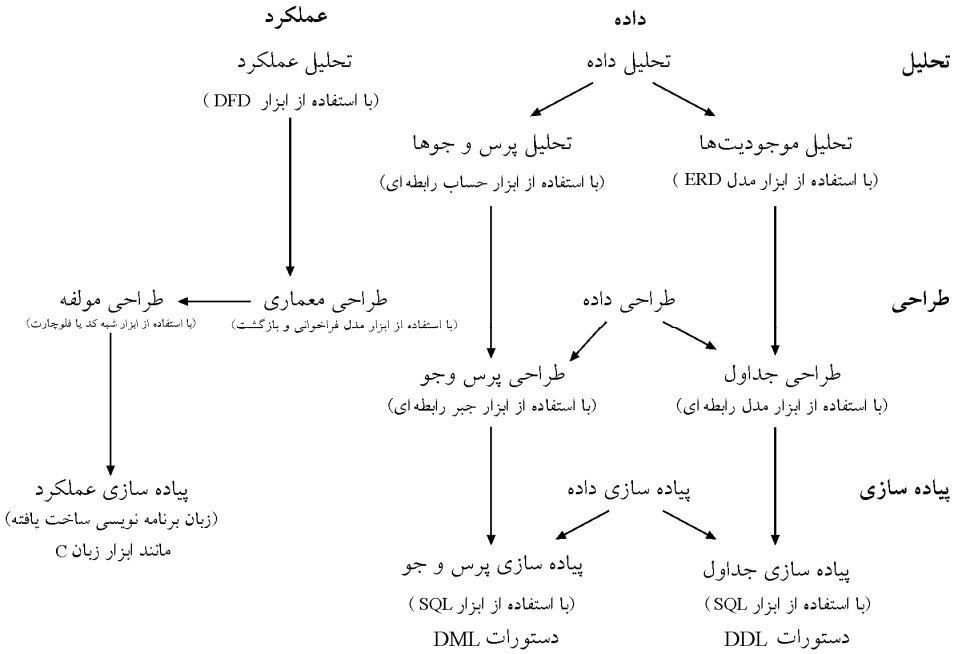
شیوه‌های فنی برای ساخت نرم‌افزار را «**روش**» می‌گوییم و به دو شکل روش ساخت‌یافته و روش شیءگرا می‌باشد. در روش ساخت‌یافته می‌گوییم چه عملکردهایی داریم و این عملکردها به چه داده‌هایی نیاز دارند و داده و عملکرد را به طور جداگانه و به روش ساخت‌یافته تحلیل، طراحی و پیاده‌سازی می‌کنیم. اما در روش شیءگرا می‌گوییم چه داده‌هایی داریم و این داده‌ها چه عملکردهایی دارند. در واقع داده و عملکرد را در قالب یک بسته (کلاس) در کنار هم قرار می‌دهیم و به روش شیءگرا (مانند مفاهیم کلاس، وراثت و چندریختی) تحلیل، طراحی و پیاده‌سازی می‌کنیم. آنچه مهندسی‌نرم‌افزار به عنوان یک هدف و اصل سودآور برای سازنده نرم‌افزار و به‌تبع مقرون به صرفه‌شدن برای مشتری دنبال می‌کند، اصل قابلیت استفاده مجدد قطعات پروژه‌های نرم‌افزاری فعلی در پروژه‌های آتی است. زمان و هزینه‌ی فرآیند تولید نرم‌افزار به دلیل استفاده از قطعات آماده (قابلیت استفاده مجدد) کاهش چشمگیری دارد. در واقع یک قطعه با قابلیت استفاده‌ی مجدد یک بار ساخته می‌شود و بارها و بارها استفاده می‌شود و این یعنی سودآوری! بنابراین واضح است که شرط لازم برای ساخت قطعات قابل استفاده مجدد، قطعه قطعه کردن نرم‌افزار یا به عبارتی پیمانه‌ای‌کردن یا بُرش نرم‌افزار است. بنابراین روش یعنی چگونگی و نحوه بُرش نرم‌افزار، در متدولوژی ساخت‌یافته این بُرش یا به عبارتی پیمانه‌ای‌کردن نرم‌افزار براساس تابع تابع کردن نرم‌افزار انجام می‌گردد و در متدولوژی شیءگرا این بُرش یا به عبارتی

[1] Who
[2] What
[3] When
[4] How

پیمانه‌ای‌کردن نرم‌افزار براساس کلاس کلاس کردن نرم‌افزار انجام می‌گردد. همانطور که گفتیم شرط لازم برای ساخت قطعات قابل استفاده مجدد، قطعه قطعه کردن نرم‌افزار یا به عبارتی پیمانه‌ای‌کردن یا بُرش نرم‌افزار است. در مورد شرط کافی برای ساخت قطعات قابل استفاده مجدد که با مفاهیمی همچون اصل پنهان‌سازی اطلاعات، استقلال عملیاتی، انسجام بالا و اتصال پایین مرتبط است در فصل مفاهیم طراحی ساخت‌یافته و مفاهیم شیءگرایی به تفصیل صحبت خواهیم کرد.

ابزارها

ابزارهای فنی برای ساخت نرم‌افزار را «**ابزار**» می‌گوییم و به دو شکل ابزار ساخت‌یافته و ابزار شیءگرا می‌باشند. شکل زیر ابزارهای ساخت‌یافته را در روش ساخت‌یافته نمایش می‌دهد:



توجه: ابزارهای شیءگرا، در فصل مربوط به مباحث شیءگرایی معرفی خواهند شد. مانند ابزار UML که سرواژه عبارت Unified Modeling Language است و به معنی زبان مدل‌سازی یکپارچه می‌باشد که جهت مدل‌سازی متدولوژی شیءگراء مورد استفاده قرار می‌گیرد.

ابزارهای کاغذی (Tool) : به معنی استفاده از ابزار به صورت دستی و نقشی بر روی کاغذ بدون استفاده از کامپیوتر مانند رسم نمودار ER بدون کامپیوتر یا رسم نمودارهای UML بدون کامپیوتر یا نوشتن کدهای برنامه‌نویسی بر روی کاغذ بدون کامپیوتر.

ابزارهای کامپیوتری (CASE TOOL) : (Computer-Aided Software Engineering Tool)

به معنی استفاده از ابزار به کمک کامپیوتر، مانند رسم نمودار ER به کمک کامپیوتر توسط

نرم‌افزار ویزیو یا رسم نمودارهای UML به کمک کامپیوتر توسط نرم‌افزار رشنال رُز یا نوشتن کدهای برنامه‌نویسی به کمک کامپیوتر توسط کامپایلرها.

توجه: ابزارهای کامپیوتری (CASE TOOL) می‌توانند، در کلیه مراحل فرآیند تولید نرم‌افزار (ارتباطات، برنامه‌ریزی، مدل‌سازی (تحلیل و طراحی)، ساخت (پیاده‌سازی و تست) و استقرار) مورد استفاده قرار گیرند.

تکنیک‌های نسل چهارم (4GT)[1]

تکنیک‌های نسل چهارم، دسته‌ای از ابزارهای CASE هستند که در فعالیت **پیاده‌سازی**، کد برنامه را به صورت خودکار تولید می‌نمایند. البته شرط لازم برای این کار، توصیف مشخصات نرم‌افزار در یک سطح انتزاعی بالا در فعالیت مدل‌سازی (نمایش نرم‌افزار توسط اشکال و علائم گرافیکی) است. مانند تولید خودکار کدهای HTML توسط نرم‌افزار Dreamweaver یا FrontPage، تولید کدهای برنامه توسط نرم‌افزار رشنال رُز که نمودارهای آن در سطح انتزاعی بالا توسط نمودارهای UML ایجاد شده است.

متدولوژی

تمامی پروژه‌های نرم‌افزاری از چهار لایه‌ی مذکور تشکیل شده‌اند. به عبارت دیگر هر پروژه‌ی نرم‌افزاری، مدل فرآیند تولیدی برای انجام پروژه‌اش دارد. همچنین روش‌های مختلفی را در قسمت‌های مختلف پروژه برای انجام فعالیت‌های تعریف شده در فرآیند تولید نرم‌افزار به کار می‌برند و ممکن است برای هر یک از این روش‌ها از ابزار خاصی استفاده کنند. افزون بر این، هر پروژه نرم‌افزاری راهکاری برای تضمین کیفیت یک نرم‌افزار دارد زیرا هدف اصلی مهندسی نرم‌افزار تولید نرم‌افزار با کیفیت بالا می‌باشد. متدولوژی در واقع نحوه‌ی ارتباط این چهار لایه را با یکدیگر مشخص می‌کند. به بیان دیگر، متدولوژی مجموعه‌ای از فرآیندها، روش‌ها و ابزارهای مرتبط با هم و همه از یک متدولوژی خاص همچون ساخت‌یافته یا شیءگرا برای ایجاد یک محصول نرم‌افزاری، مطابق با استانداردهای مهندسی نرم‌افزار می‌باشد و بر دو طبقه‌ی ساخت‌یافته و شیءگرا می‌باشد.

توجه: اساس به وجود آمدن متدولوژی شیءگرا به وجود آمدن نیازهای جدید بوده که توسط متدولوژی ساخت‌یافته قابل پوشش دادن نبوده‌اند. متدولوژی شیءگرا برخی از نیازمندی‌های جدید را پوشش داده است و این طور نبوده‌است که استفاده از متدولوژی ساخت‌یافته را منسوخ کند.

متدولوژی ساخت‌یافته (مهندسی نرم‌افزار ساخت‌یافته)

متدولوژی ساخت‌یافته یا مهندسی نرم‌افزار ساخت‌یافته نظامی است یکپارچه شامل مدل

[1] Fourth Generation Techniques

فرآیندهای ساخت‌یافته (سنتی)، روش ساخت‌یافته و ابزارهـای سـاخت‌یافتـه کـه منجـر بـه ایجـاد نرم‌افزاری در بازه‌ی زمانی از قبل برنامه‌ریزی شده، بودجه‌ای از قبل پیش‌بینی شده و دقیقاً مطابق با نیازمندی‌های واقعی مشتری می‌گردد.

در متدولوژی ساخت‌یافته، در اولین مرحله مدل‌سازی (مدل تحلیل)، سیستم به دو وجـه «داده» و «عملکرد» تفکیک می‌شود. سپس طی روندی سلسله مراتبی و مطابق با روش بالا بـه پـایین، هـر یک از این وجوه خود به مؤلفه‌های فرعی تجزیه می‌شوند. این روند تا به جایی ادامه مـی‌یابـد کـه جزئیات توابع برنامه جهت پیاده‌سازی مشخص شوند.

توجه: متدولوژی SSADM متـداول‌تـرین نمونـه از متـدولوژی سـاخت‌یافتـه براسـاس روش ساخت‌یافته، مدل فرآیند تولید آبشاری و ابزارهای ساخت‌یافتـه مـی‌باشـد. و بـر دو نـوع داده‌گـرا (جدولی) مانند نرم‌افزار حقوق و دستمزد که داده‌هـا درون جـداول ذخیـره مـی‌شـوند و تـابع‌گـرا (متغیری) مانند نرم‌افزار ماشین‌حساب که داده‌ها درون متغیرها ذخیره می‌شوند، می‌باشد.

توجه: نسبت نمونه متدولوژی ساخت‌یافته SSADM به متدولوژی سـاخت‌یافتـه، مثـل نسـبت سیستم‌عامل ویندوز سنتی به مفاهیم سنتی سیستم‌عامل است.

متدولوژی شیءگرا (مهندسی نرم‌افزار شیءگرا)

متدولوژی شیءگرا یا مهندسی نرم‌افزار شیءگرا نظـامی اسـت یکپارچـه شـامل مـدل فـرآینـد شیءگرا (مدرن)، روش شیءگرا (مبتنـی بـر مفـاهیم کـلاس، وراثـت و چنـدریختی) و ابزارهـای شیءگرا که منجر به ایجاد نرم‌افزاری در بازه‌ی زمانی از قبل برنامه‌ریـزی شـده، بودجـه‌ای از قبـل پیش‌بینی شده و دقیقاً مطابق با نیازمندی‌های واقعی مشتری می‌گردد.

در متدولوژی شیءگرا، در اولین مرحله‌ی مدل‌سازی (مدل تحلیل) سیستم در قالب کـلاس‌هـا (شامل داده (صفت) و عملکرد (متد)) نشان داده می‌شـود. سـپس طـی رونـدی سلسـله مراتبـی و مطابق با روش بالا به پایین، کلاس‌ها با جزئیات بیشتری مشخص می‌شوند. این روند تا بـه جـایی ادامه می‌یابد که جزئیات کلاس‌های برنامه جهت پیاده‌سازی مشخص شوند.

توجه: متدولوژی RUP متداول‌تـرین نمونـه از متـدولوژی شـیءگرا براسـاس روش شـیءگرا (مبتنی بر مفاهیم کلاس، وراثت و چندریختی)، مدل فرآیند مبتنی بر مؤلفه‌ی شـیءگرا بـا رویکـرد تکرار و تکامل و ابزارهای شـیءگرا (مثـل ابـزار مـدل‌سـازی UML و زبـان برنامـه‌نویسـی C++) می‌باشد.

توجه: نسبت نمونه متدولوژی شیءگرای RUP به متدولوژی شـیءگـرا، مثـل نسـبت سیسـتم عامل ویندوز مدرن به مفاهیم مدرن سیستم عامل است.

توجه: متدولوژی شیءگرا و متدولوژی RUP به تفصیل در فصـل مربوطـه شـرح داده خواهـد شد. در ادامه به تشریح مفاهیم مربوط به متدولوژی ساخت‌یافته خواهیم پرداخت:

فعالیت‌های چارچوبی فرآیند تولید نرم‌افزار

به طور کلی فعالیت‌های مرتبط با فرآیند تولید نرم‌افزار صرف‌نظر از اندازه، پیچیـدگی پـروژه و

زمینه‌ی کاربردی آن و مستقل از متدولوژی ساخت‌یافته و شیءگرا به پنج فعالیت ارتباط، برنامه‌ریزی، مدل‌سازی (تحلیل و طراحی)، ساخت (پیاده‌سازی و تست) و استقرار تقسیم می‌شود، به بیان دیگر فعالیت‌ها در هر دو دسته‌ی متدولوژی ساخت‌یافته و شیءگرا همین‌ها خواهند بود، اما کارهایی که در هر فعالیت در متدولوژی ساخت‌یافته و شیءگرا انجام می‌شود شباهت‌ها و تفاوت‌هایی خواهد داشت.

در ادامه فعالیت‌های چارچوبی فرآیند تولید نرم‌افزار براساس متدولوژی ساخت‌یافته بیان خواهد شد:

۱- ارتباطات (مهندسی نیازمندی‌های مشتری یا مهندسی خواسته‌های مشتری)

فعالیت ارتباطات یا مهندسی نیازمندی‌های مشتری نظامی است یکپارچه، شامل فرآیندها، روش‌ها و ابزارها که منجر به تهیه لیست نیازمندی‌های مشتری می‌گردد.

فرآیند تهیه لیست نیازمندی‌های مشتری

هر پروژه‌ی نرم‌افزاری، چه بزرگ و چه کوچک مراحلی را طی می‌نماید که در طی آن لیست نیازمندی‌های مشتری تهیه می‌گردد. الگو و قالبی که چگونگی طی مراحل مختلف تهیه لیست نیازمندی‌های مشتری را تعریف می‌نماید، اصطلاحاً **فرآیند تهیه لیست نیازمندی‌های مشتری** نامیده می‌شود.

روش‌های تهیه لیست نیازمندی‌های مشتری

روش‌های تشخیص برای تهیه لیست نیازمندی‌های مشتری را «روش‌های تهیه لیست نیازمندی‌های مشتری» می‌گوییم، یکی از روش‌های پرکاربرد روشی موسوم به «**روش QFD**» می‌باشد.

توجه: QFD سرواژه عبارت Quality Function Deployment و به معنی استقرار عملکرد کیفیت می‌باشد.

توجه: روش QFD جلوتر شرح داده خواهد شد.

ابزارهای تهیه لیست نیازمندی‌های مشتری

ابزارهای تشخیص برای تهیه لیست نیازمندی‌های مشتری را «ابزارهای تهیه لیست نیازمندی‌های مشتری» می‌گوییم و به پنج شکل «**گفتگو**»، «**مشاهده**»، «**پرسش‌نامه**»، «**مکانیزم نمونه‌سازی دورانداختنی**» و «**مکانیزم نمونه‌سازی تکاملی**» می‌باشد.

توجه: مکانیزم نمونه‌سازی دورانداختنی و تکاملی جلوتر شرح داده می‌شود.

توجه: فعالیت ارتباطات از طریق ارتباط با مشتری توسط ارتباط‌گر و ابزارها و روش‌های مطرح شده انجام می‌گردد.

مراحل فرآیند تهیه لیست نیازمندی‌های مشتری

به طور کلی مراحل مرتبط با فرآیند تهیه لیست نیازمندی‌های مشتری صرف‌نظر از اندازه، پیچیدگی پروژه و زمینه‌ی کاربردی آن و مستقل از متدولوژی ساخت‌یافته و شیءگرا به هفت مرحله **شناخت اولیه نیازمندی‌ها**، **شناخت بیشتر نیازمندی‌ها**، **تشریح نیازمندی‌های شناخته‌شده**، **مذاکره**، **تعیین مشخصات**، **اعتبارسنجی نیازمندی‌ها** و **مدیریت نیازمندی‌ها** تقسیم می‌شود، به بیان دیگر مراحل در هر دو دسته‌ی متدولوژی ساخت‌یافته و شیءگرا همین‌ها خواهند بود، اما کارهایی که در هر فعالیت در متدولوژی ساخت‌یافته و شیءگرا انجام می‌شود شباهت‌ها و تفاوت‌هایی خواهد داشت.

در ادامه مراحل فرآیند تهیه لیست نیازمندی‌های مشتری بیان خواهد شد:

۱- درک (شناخت اولیه نیازمندی‌ها)

در مرحله شناخت اولیه نیازمندی‌ها، شناختی پایه‌ای از نیازمندی‌های مشتری انجام می‌گردد، که این شناخت اولیه مستلزم ارتباط و گپ و گفت اولیه سازنده و مشتری است.

۲- استخراج (شناخت بیشتر نیازمندی‌ها)

در مرحله شناخت بیشتر نیازمندی‌ها، شناختی بیشتر از نیازمندی‌های مشتری انجام می‌گردد، که این شناخت بیشتر مستلزم ارتباط و گپ و گفت بیشتر سازنده و مشتری است.

توجه: توسط استفاده از روش QFD می‌توان به شناخت خواسته‌هایی از مشتری بپردازیم که برای مشتری ارزشمندتر است.QFD سه نوع خواسته را مشخص می‌کند:

خواسته‌های عادی: به خواسته‌هایی که مشتری بیان می‌کند و انتظار هم دارد که سازنده این خواسته‌ها را برآورده سازد، خواسته‌های عادی گفته می‌شود که در صورت برآورده‌سازی این خواسته‌ها توسط سازنده، مشتری راضی خواهد بود.

خواسته‌های مورد انتظار: به خواسته‌هایی که مشتری بیان نمی‌کند ولی به صورت پیش فرض انتظار هم دارد که سازنده این خواسته‌ها را برآورده سازد، خواسته‌های مورد انتظار گفته می‌شود که در صورت عدم برآورده‌سازی این خواسته‌ها توسط سازنده، مشتری ناراضی خواهد بود. مانند زمان پاسخ کوتاه در تعامل با نرم‌افزار توسط مشتری، یا سهولت در نصب نرم‌افزار.

خواسته‌های هیجان‌انگیز: به خواسته‌هایی که مشتری بیان نمی‌کند و انتظار هم ندارد که سازنده این خواسته‌ها را برآورده سازد، خواسته‌های هیجان‌انگیز گفته می‌شود که در صورت برآورده‌سازی این خواسته‌ها توسط سازنده، مشتری بسیار بسیار هیجان‌زده و راضی خواهد بود. مانند قابلیت چیدمان و آرایش صفحات برنامه به صورت دلخواه.

توجه: برآورده‌سازی «**خواسته‌های عادی**» و «**خواسته‌های مورد انتظار**» از سوی سازنده اجباری و معیار سنجش مشتری است و برآورده‌سازی «**خواسته‌های هیجان‌انگیز**» از سوی سازنده، اختیاری و معیار سنجش مشتری نیست ولی اگر از سوی سازنده این دسته از خواسته‌ها برآورده

گـردد، مشـتری هیجـان زده خواهـد شـد. راز موفقیـت «اسـتیو جـابز[1]» رهبـر فقیـد کمپـانی اپـل برآورده‌سـازی «خواسـته‌هـای هیجـان‌انگیـز» عـلاوه بـر بـرآورده‌سـازی «خواسـته‌هـای عـادی» و «خواسته‌های مورد انتظار» بود. موفقیت یعنی توجه کردن به جزئیات. «استیو جابز»

۳- تشریح نیازمندی‌های شناخته شده

در مرحله تشریح نیازمندی‌های شناخته‌شده، نیازمندی‌هـایی کـه در دو مرحلـه شـناخت اولیـه نیازمندی‌ها و شناخت بیشتر نیازمندی‌ها کشـف شـده‌انـد، بـا بیـان ذکـر جزئیـات بیشـتر، تشـریح می‌شوند.

۴- مذاکره

در مرحله مذاکره، پس از آنکه نیازمندی‌های شناخته‌شده، تشریح شدند، نوبت به مذاکره مجدد مابین سازنده و مشتری می‌رسد، تا توافقات لازم را بر سر نهایی‌شدن نیازمنـدی‌هـای شـناخته‌شـده انجام دهند.

۵- تعیین مشخصات

پس از توافقات لازم بر سر نهایی‌شدن نیازمندی‌های شناخته‌شده در مرحله مذاکره، در ادامـه و در مرحله تعیین مشخصـات، مشخصـات سیسـتمی کـه بایـد ایجـاد گـردد، تحـت عنـوان لیسـت نیازمندی‌های مشتری نوشته می‌شود. همچنین این لیست مـی‌توانـد بـه صـورت گرافیکـی توسـط نمودار مورد کاربرد یا use case diagram مدل‌سازی شود.

توجه: نمودار مورد کاربرد یا use case diagram در فصل شیءگرایی تشریح می‌گردد.

۶- اعتبارسنجی نیازمندی‌ها

در اعتبارسنجی نیازمندی‌ها، آنچه در مرحله تعیین مشخصـات حاصـل گردیـد، جهـت کنتـرل نهایی، توسط سازنده و مشتری مورد بررسی نهایی قرار مـی‌گیـرد و در صـورت وجـود اشـکالات مربوط به ناسازگاری در نظرات سازنده و مشتری، سازگاری لازم صورت می‌گیرد.

۷- مدیریت نیازمندی‌ها

نیازمندی‌های مشتری، در طول چرخه حیات نرم‌افزار مدام تغییر می‌کند، شاید تنهـا چیـزی کـه در دنیا ثابت است، تغییر باشد. بنابراین در مرحله مدیریت نیازمنـدی‌هـا، تغییراتـی کـه در چرخـه حیات نرم‌افزار حاصل می‌گردد تحت کنترل و مدیریت قرار می‌گیرد.

انواع نیازمندی‌ها

۱- وظیفه‌مندی[2] (کارکردی، عملکردی)

نیازمندی‌های وظیفه‌مندی، کمّی و قابل اندازه‌گیری هستند و در قالـب قابلیـت‌هـا، کارکردهـا،

[1] Steve Jobs
[2] Functional

ویژگی‌ها و سرویس‌های سیستم در حال توسعه یا تولید محقق می‌گردند. به عبارت دیگر، نیازمندی‌های کارکردی به بیان سرویس‌هایی که سیستم باید فراهم نماید، می‌پردازد. چگونگی واکنش سیستم در برابر ورودی‌های خاص و چگونگی رفتار سیستم در شرایط خاص نیز توسط این نیازمندی‌ها تعریف می‌شود.

مانند نیازمندی‌های مربوط به محاسبه‌ی فاکتوریل یک عدد و یا اجرای تابع فیبوناچی در یک نرم‌افزار و یا نیازمندی‌های مربوط به یک نرم‌افزار حقوق و دستمزد.

توجه: نیازمندی‌های وظیفه‌مندی، موسوم به نیازها یا «**خواسته‌های عادی**» است.

۲- غیروظیفه‌مندی[1] (غیرکارکردی، غیر عملکردی)

نیازمندی‌های غیروظیفه‌مندی، نیازمندی‌های کیفی و نه الزاماً قابل اندازه‌گیری هستند که به بیان کیفیت مورد انتظار از نیازمندی‌های وظیفه‌مندی و همچنین محدودیت‌هایی نظیر محدودیت‌های زمانی، مالی و استانداردها می‌پردازند. برخی از انواع نیازمندی‌های غیروظیفه‌مندی عبارتند از:

قابلیت استفاده، سهولت یادگیری، قابلیت اعتماد، کارایی، زمان پاسخ، قابلیت پشتیبانی، قابلیت نگهداری، قابلیت حمل، بهره‌وری، محدودیت‌های طراحی، پیاده‌سازی و فیزیکی و همچنین نیازمندی‌های واسط کاربردی.

مانند محاسبه تابع فاکتوریل توسط الگوریتم حلقه با مرتبه اجرایی O(n) و یا توسط الگوریتم بازگشتی با مرتبه اجرایی O(n) و مصرف زیاد حافظه به دلیل استفاده از استک در پی هر فراخوانی.

و یا مانند محاسبه تابع فیبوناچی توسط الگوریتم حلقه با مرتبه اجرایی O(n) و یا توسط الگوریتم بازگشتی با مرتبه اجرایی نمایی زیاد $O(۲^{\frac{n}{۲}})$ و چاق و به تبع، کند و هم مصرف زیاد حافظه به دلیل استفاده از استک، در پی هر فراخوانی.

توجه: محصول نرم‌افزاری باید برآورده کننده نیازمندی‌های هم وظیفه‌مندی و هم غیروظیفه‌مندی مشتری باشد. محصول نرم‌افزاری که نیازمندی‌های وظیفه‌مندی را برآورده می‌کند، ولی برآورنده نیازمندی‌های غیروظیفه‌مندی نباشد، معمولاً با نارضایتی مشتریان همراه می‌شود.

توجه : انتخاب نوع الگوریتم براساس شرایط، حائز اهمیت می‌باشد.

توجه: نیازمندی‌های غیروظیفه‌مندی، موسوم به نیازمندی‌ها یا «**خواسته‌های مورد انتظار**» است.

۲- برنامه‌ریزی

برنامه‌ریزی یعنی هنر حرکت از مبدأ موجود به مقصد مطلوب برای رسیدن به نتیجه‌ای مطلوب براساس خواسته‌های مورد نیاز در یک زمان مشخص.

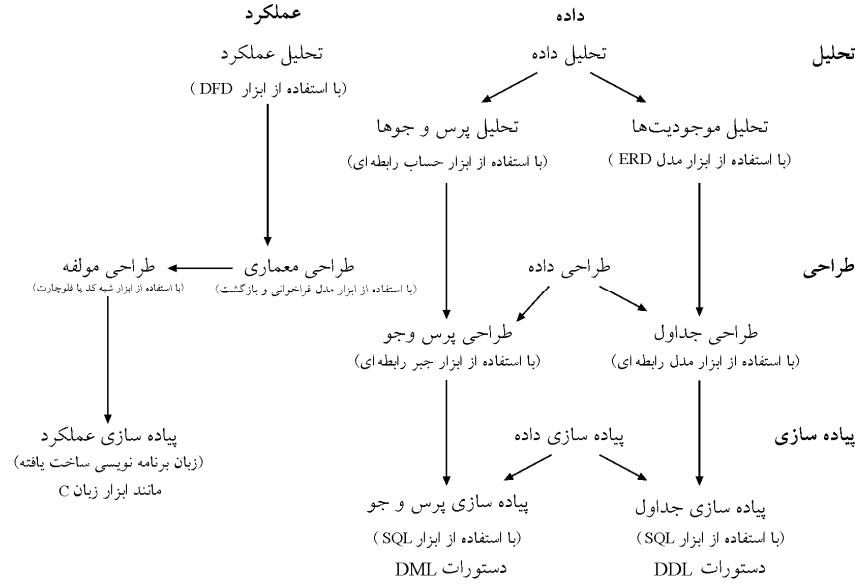
«هر تلاشی منجر به نتیجه‌ای مطلوب نمی‌گردد، بلکه این تلاشی مطلوب است که منجر به نتیجه‌ای مطلوب می‌گردد.»

[1] Non-functional

لازمه‌ی تلاش مطلوب، برنامه‌ریزی است. برنامه‌ریزی می‌تواند اجرای هر کار پیچیده‌ای را ساده‌تر سازد. هر کار مهندسی مستلزم برنامه‌ریزی می‌باشد. مهندسی نرم‌افزار نیز مانند هر فعالیت مهندسی دیگری، نیازمند برنامه‌ریزی است. فعالیت برنامه‌ریزی، برنامه‌ای را برای فعالیت‌های مختلف بخش‌های مختلف فرآیند تولید نرم‌افزار پایه‌ریزی می‌کند. این فعالیت، وظیفه‌های فنی که باید هدایت شوند، ریسک‌هایی که محتمل می‌شوند (مانند عدم شناسایی برخی نیازمندی‌ها، از دست دادن داده‌ها و مدیران)، منابع مورد نیاز، واحدهای کاری که باید ایجاد شوند و برنامه زمان‌بندی برای کارها را تشریح می‌کند. مدیریت، برنامه‌ریزی را به مرحله‌ی اجرا می‌برد.

۳- مدل‌سازی (تحلیل و طراحی)

یک مدل، ساده شده یک واقعیت است. ایجاد یک مدل برای سیستم‌های نرم‌افزاری قبل از ساخت یا بازساخت آن، به اندازه داشتن نقشه برای ساختن یک ساختمان ضروری وحیاتی است. بسیاری از شاخه‌های مهندسی، توصیف چگونگی محصولاتی که باید ساخته شوند را ترسیم می‌کنند و همچنین دقت زیادی می‌کنند که محصولاتشان طبق این مدل‌ها و توصیف‌ها ساخته شوند. مدل‌های خوب و دقیق در برقراری یک ارتباط کامل بین افراد پروژه، نقش زیادی می‌توانند داشته باشند. علت اصلی مدل کردن سیستم‌های پیچیده این است که نمی‌توان به یکباره کل سیستم را تجسم کرد و ممکن است سیستم دارای ابهامات بسیاری باشد. لذا برای رفع این ابهامات و نیز برای فهم کامل سیستم و یافتن و نمایش ارتباط بین قسمت‌های مختلف آن، از مدل‌سازی استفاده می‌شود. فعالیت مدل‌سازی خود شامل دو مرحله‌ی مدل تحلیل و مدل طراحی می‌باشد. مدل تحلیل پس از فعالیت ارتباطات (جمع‌آوری نیازمندی‌ها) و قبل از مدل طراحی انجام می‌شود، در واقع خروجی مدل تحلیل، ورودی مدل طراحی می‌باشد. شکل زیر گویای این مطلب می‌باشد:



مدل تحلیل

پس از جمع‌آوری لیست نیازمندی‌های مشتری در فعالیت ارتباطات نوبت به مدل تحلیل (مدل‌سازی لیست نیازمندی‌های مشتری) می‌رسد. مدل‌سازی که فعالیتی فنی به شمار می‌رود نیازمندی‌ها را باید به گونه‌ای مدل نماید که برای سازنده و مشتری قابل فهم باشد.

در مدل تحلیل به روش ساخت‌یافته دو وجه مدل تحلیل داده و مدل تحلیل عملکرد وجود دارد. مدل تحلیل داده شامل تحلیل موجودیت‌ها و تحلیل پرس و جوها می‌باشد. تحلیل موجودیت‌ها توسط ابزار مدل ER و تحلیل پرس و جوها توسط ابزار حساب رابطه‌ای مدل می‌شوند. مدل تحلیل عملکرد توسط ابزار DFD مدل می‌شود.

مدل طراحی

پس از مدل تحلیل، نوبت به مدل طراحی می‌رسد، مدل طراحی به روش ساخت‌یافته شامل چهار بخش **طراحی‌داده، طراحی‌معماری، طراحی‌مؤلفه** و **طراحی‌واسط** می‌باشد. طراحی داده بر دو بخش طراحی جدول و طراحی پرس و جو می‌باشد. طراحی جدول از بخش طراحی داده، تحلیل موجودیت (ERD) از مدل تحلیل را به عنوان ورودی دریافت کرده و توسط مدل رابطه‌ای، طراحی جدول را انجام می‌دهد. طراحی پرس و جو از بخش طراحی داده، تحلیل پرس و جو از مدل تحلیل را به عنوان ورودی دریافت کرده و توسط جبر رابطه‌ای، طراحی پرس و جو را انجام می‌دهد.

طراحی معماری، تحلیل عملکرد (DFD) از مدل تحلیل را به عنوان ورودی دریافت کرده و توسط یکی از سبک‌های معماری (مانند فراخوانی و بازگشت)، طراحی معماری را انجام می‌دهد.

طراحی معماری یا معماری نرم‌افزار، ساختار کلی نرم‌افزار و شیوه‌های یکپارچگی یک سیستم را بیان می‌کند. به عبارت دیگر، ساختار سلسله مراتبی **مولفه‌های برنامه (توابع یا پیمانه‌ها)**، شیوه تعامل مولفه‌ها با یکدیگر و **ساختمان داده‌های** مورد نیاز مولفه‌ها را نشان می‌دهد. معماری نرم‌افزار یک مدل قابل درک از چگونگی سازمان‌دهی سیستم است. در واقع نشان‌گر ساختمان داده‌ها و مؤلفه‌های برنامه‌ای است که برای ساختن یک سیستم کامپیوتری لازم است. به طور دقیق‌تر معماری نرم‌افزار شامل دو سطح از طراحی می‌باشد یعنی طراحی داده و طراحی معماری. در واقع این ساختار مانند یک نقشه ساختمان، مبنای ساخت نرم‌افزار قرار می‌گیرد.

توجه: در طراحی معماری، اسکلت، ساختار و چیدمان کلی مولفه‌های (توابع) برنامه به این معنی که چه مولفه‌ای (تابعی) چه مولفه‌ای (تابعی) دیگر را صدا می‌زند، بدون ذکر جزئیات داخلی مولفه‌ها (توابع) مشخص می‌گردد (ساختار درختی برنامه بدون ذکر جزئیات مولفه‌ها (توابع)). مانند اسکلت یک ساختمان که گویای جایگاه مؤلفه‌های ساختمان است اما هنوز آجرچینی نشده است. (اسکلت یک ساختمان بدون آجرچینی).

توجه: به طراحی معماری، طراحی کلی نیز گفته می‌شود.

توجه: طراحی معماری ساخت‌یافته در فصل مفاهیم طراحی ساخت‌یافته، بیشتر توضیح داده خواهد شد.

توجه: سبک معماری فراخوانی و بازگشت جلوتر شرح داده خواهد شد.

طراحی مؤلفه، طراحی معماری از همان فعالیت مدل طراحی را به عنوان ورودی دریافت کرده و طراحی مؤلفه را توسط ابزارهایی همچون **شبه کد** یا **فلوچارت** ایجاد می‌کند.

طراحی مولفه، فعالیت تبدیل طراحی معماری به نرم‌افزار است. در این مرحله، سطح انتزاع طراحی معماری به سطح انتزاع نرم‌افزار کاربردی نزدیک می‌گردد. طراحی در سطح مؤلفه‌ها، نرم‌افزار را در سطحی از انتزاع تصویر می‌کند که به کد نزدیک است. طراحی مولفه، به عنوان نقشه راهی دقیق، و نزدیک به زبان پیاده‌سازی، در فعالیت پیاده‌سازی نرم‌افزار، منجر به صرفه جویی در زمان و هزینه‌های تولید می‌گردد. در طراحی مولفه، مهندس نرم‌افزار باید ساختمان داده‌ها، واسط‌ها و الگوریتم‌ها را با جزییات کافی به نمایش در آورد تا راهنمای تولید کد منبع زبان برنامه‌نویسی باشد.

توجه: در طراحی مؤلفه، اسکلت، ساختار و چیدمان کلی مولفه‌های (توابع) برنامه به این معنی که چه مولفه‌ای (تابعی) چه مولفه‌ای (تابعی) دیگر را صدا می‌زند، با ذکر جزئیات داخلی مولفه‌ها (توابع) مشخص می‌گردد (ساختار درختی برنامه با ذکر جزئیات مولفه‌ها (توابع)). مانند اسکلت یک ساختمان که گویای جایگاه مؤلفه‌های ساختمان است و آجرچینی هم شده است. (اسکلت یک ساختمان به همراه آجرچینی).

توجه: به طراحی مولفه، طراحی جزئی، طراحی تفصیلی و طراحی رویه‌ای نیز گفته می‌شود.

طراحی واسط یا همان واسط کاربر، براساس ورودی‌ها و خروجی‌های مورد نیاز کاربران نهایی به شکل نقشی بر روی کاغذ یا طرحی بر روی کامپیوتر ایجاد می‌گردد. مانند نحوه چیدمان منوها و فرم‌ها.

۴- ساخت (پیاده‌سازی و تست)

پس از مدل طراحی نوبت به پیاده‌سازی و تست می‌رسد. پیاده‌سازی جداول از بخش پیاده‌سازی داده، طراحی جدول از مدل طراحی را به عنوان ورودی دریافت کرده و توسط دستورات DDL در SQL، پیاده‌سازی جداول را انجام می‌دهد.

پیاده‌سازی پرس و جو از بخش پیاده‌سازی داده، طراحی پرس و جو از مدل طراحی را به عنوان ورودی دریافت کرده و توسط دستورات DML در SQL پیاده‌سازی پرس و جو را انجام می‌دهد. پیاده‌سازی عملکرد، طراحی مؤلفه از مدل طراحی را به عنوان ورودی دریافت کرده و توسط یک زبان برنامه‌نویسی (ساخت‌یافته یا شیءگرا) پیاده‌سازی عملکرد را انجام می‌دهد.

توجه: دقت‌کنید که می‌توان مدل تحلیل و طراحی را به روش و ابزارهای ساخت‌یافته انجام داد ولی فعالیت پیاده‌سازی را توسط یک زبان برنامه‌نویسی شیءگرا انجام داد و از امکانات شیءگرایی زبان استفاده نکرد، اما عکس این مطلب امکان‌پذیر نیست، یعنی نمی‌توان مدل تحلیل و طراحی را به روش شیءگرا انجام داد ولی فعالیت پیاده‌سازی را توسط یک زبان ساخت‌یافته انجام داد زیرا زبان ساخت‌یافته امکانات شیءگرایی (همچون کلاس، وراثت و چندریختی) را پشتیبانی نمی‌کند.

پس از پیاده‌سازی نوبت به تست می‌رسد، در این مرحله کلیه‌ی موارد پیاده‌سازی شده از نظر خطاهای نحوی و خطاهای معنایی براساس لیست نیازمندی‌های مشتری (چک لیست) که در فعالیت ارتباطات تهیه شده بود مورد وارسی قرار می‌گیرد تا مشخص شود نرم‌افزار براساس ورودی‌های مورد نظر مشتری، خروجی‌های مورد انتظار مشتری را برآورده می‌سازد یا خیر.

۵- استقرار

پس از فعالیت تست نوبت به فعالیت استقرار می‌رسد. در این مرحله، نرم‌افزار به مشتری تحویل داده می‌شود و مشتری با بررسی محصول دریافتی، بازخوردهای به دست آمده براساس همین ارزیابی‌ها را به تیم نرم‌افزاری ارائه می‌دهد. این بازخوردها می‌توانند مبنایی برای ارتقاء و یا تصحیح نسخه‌ی بعدی نرم‌افزار باشد.

این پنج فعالیت چارچوبی را می‌توان طی تولید برنامه‌های کوچک و ساده، در ایجاد برنامه‌های تحت وب و برای مهندسی سیستم‌های کامپیوتری پیچیده و عظیم به کار برد. جزئیات مدل‌های فرآیند تولید نرم‌افزار در هر مورد کاملاً متفاوت خواهد بود، ولی فعالیت‌های چارچوبی همین‌ها خواهد بود.

برای بسیاری از پروژه‌های نرم‌افزاری، فعالیت‌های چارچوبی به موازات پیشرفت پروژه به صورت تکراری به کار برده می‌شوند. یعنی فعالیت‌های ارتباطات، برنامه‌ریزی، مدل‌سازی، ساخت و استقرار به طور مکرر در چند دور تکرار پروژه به کار برده می‌شوند. در هر دور تکرار پروژه، یک نسخه (افزایش)[1] از نرم‌افزار ایجاد می‌شود که زیرمجموعه‌ای از قابلیت‌های عملیاتی و ویژگی‌های نرم‌افزار کامل را در اختیار مشتری قرار می‌دهد. با تولید هر افزایش، نرم‌افزار کامل و کامل‌تر می‌شود.

توجه: فعالیت‌های چارچوبی فرآیند تولید نرم‌افزار براساس متدولوژی شیءگرا در فصل شیءگرایی به تفصیل شرح داده خواهد شد.

فعالیت‌های چتری فرآیند تولید نرم‌افزار

انجام درست فعالیت‌های چارچوبی تا حدود بسیار زیادی تحت کنترل انسان می‌باشد ولی برای تولید موفق یک پروژه‌ی نرم‌افزاری به تنهایی کافی نیستند. زیرا در وادی زندگی علاوه بر قوانین انسانی، قوانین طبیعی نیز وجود دارند، اگر فعالیت‌های چارچوبی درست بودند، اما اطلاعات موجود در هارد دیسک به دلیل عوامل طبیعی از بین رفتند، چه کار کنیم، اگر فعالیت‌های چارچوبی درست بودند، اما به دلیل ماهیت انسانی بودن یک انسان و عوامل طبیعی همچون مرگ و میر و زلزله، مدیران و همکاران خود را از دست دادیم، چه کار کنیم و واقعاً اگر همه‌ی موارد تحت کنترل انسان برای رسیدن به موفقیت درست بودند، عوامل طبیعی که خارج از کنترل انسان هستند را چه کار کنیم ...

[1] Increment

به قول حضرت حافظ

در بیابان‌گر به شوق کعبه خواهی زد قدم سرزنش‌هاگر کند خار مغیلان غم مخور

در وادی مهندسی نرم‌افزار، توسط فعالیت‌های چتری، عوامل طبیعی خارج از کنترل انسان و هر آنچه مربوط به موفقیت پروژه باشد، نظارت و تحت کنترل دقیق قرار می‌گیرد، در واقع فعالیت‌های چارچوبی فرآیند تولید نرم‌افزار توسط تعدادی از فعالیت‌های چتری تکمیل می‌شود. به طور کلی، فعالیت‌های چتری در سرتاسر یک پروژه‌ی نرم‌افزاری به کار برده می‌شوند و به تیم نرم‌افزاری کمک می‌کنند تا پیشرفت، کیفیت، تغییر و ریسک را کنترل کنند. به بیان دیگر فعالیت‌های چتری بر محقق شدن خصوصیات پروژه‌های موفق نرم‌افزاری (بازه‌ی زمانی از قبل برنامه‌ریزی شده، بودجه‌ای از قبل پیش‌بینی شده و با صرف کمترین هزینه و دقیقاً مطابق با نیازمندی‌های واقعی کاربران) در حین انجام فعالیت‌های چارچوبی، تأکید می‌کند.

انواع فعالیت‌های چتری

۱- کنترل و پیگیری پروژه‌ی نرم‌افزاری

برای تحویل به موقع محصول، نیاز به یک زمان‌بندی اولیه است، اما از آن مهم‌تر کنترل و پیگیری این زمان‌بندی در طول پروژه است. بنابراین کنترل و پیگیری پروژه‌ی نرم‌افزاری به تیم پروژه امکان می‌دهد تا از پیشرفت واقعی پروژه با توجه به برنامه زمان‌بندی شده آگاه شده و در صورت لزوم اقدامات لازم را برای حفظ برنامه‌ی زمان‌بندی انجام دهند. در واقع در این فعالیت مقایسه می‌شود تا در صورت لزوم و در مواردی که از زمان زمان‌بندی عقب هستیم، کارهایی برای جبران این عقب ماندگی انجام پذیرد.

۲- مدیریت ریسک

گام اول مدیریت ریسک، شناسایی ریسک (تحلیل ریسک) است و گام دوم مقابله با ریسک، برای مقابله با ریسک باید هزینه کرد. مثلاً ریسک از دست دادن اطلاعات را می‌توان با هزینه و خریداری یک سیستم پشتیبان‌گیری از اطلاعات مرتفع نمود یا ریسک از دست دادن مدیران را می‌توان با هزینه و استخدام یک نیروی انسانی پشتیبان در کنار مدیران دارای درجه اهمیت بالا مرتفع نمود. در واقع در این فعالیت، ریسک‌های محتملی که ممکن است بر روی خروجی‌های پروژه و یا کیفیت محصول نهایی پروژه یا به عبارت دقیق‌تر بر روی خصوصیات پروژه‌های موفق نرم‌افزاری (نیاز، زمان و هزینه) تأثیر ناگواری بگذارند شناسایی، مدیریت و در صورت امکان تقلیل می‌یابند. دقت‌کنید که احتمال وقوع ریسک، تابعی از زمان است، بنابراین مدیریت ریسک می بایست در سراسر چرخه‌ی حیات نرم‌افزار، حضوری فعال و پررنگ داشته باشد.

۳- تضمین کیفیت نرم‌افزار (SQA : Software Quality Assurance)

فعالیت‌های لازم، کافی و دقیق در فعالیت‌های مختلف چارچوب فرآیند تولید نرم‌افزار (ارتباطات، برنامه‌ریزی، مدل‌سازی، ساخت و استقرار) برای حصول اطمینان از کیفیت نرم‌افزار (نرم‌افزاری مطابق با خواسته‌های مشتری) را معین می‌کند. اگر فعالیت‌های مربوط به فازهای

مختلف چارچوپ فرآیند تولید نرم‌افزار درست باشند، نتایج خودشان درست خواهند بود.

۴- بازبینی‌های فنی (FTR : Formal Technical Review)

تمامی فرآورده‌های تولید شده در فعالیت‌های مختلف چارچوب فرآیند تولید نرم‌افزار برای آشکار کردن خطاها قبل از انتشار آنها در فعالیت بعدی و برطرف کردن آنها مورد ارزیابی و بازبینی قرار می‌گیرند. به بیان دیگر بعد از انجام هر مرحله از توسعه‌ی نرم‌افزار، نتایج حاصله مورد ارزیابی قرار می‌گیرند تا خطاهای مستقر در این مرحله به مراحل بعد انتشار نیابند. این ارزیابی توسط یک گروه از افراد پروژه انجام شده و باعث افزایش کیفیت نرم‌افزار می‌شود.

۵- اندازه‌گیری

در این فعالیت، اندازه‌ها، معیارها و شاخص‌های محصول، پروژه و فرآیند، تعریف و جمع‌آوری می‌شوند که با استفاده از این اطلاعات، مدیر و تیم پروژه قادر به تحویل نرم‌افزار با کیفیت بالا و مطابق با استانداردهای از پیش تعیین شده می‌باشند. در واقع این اطلاعات سبب مدیریت بهتر پروژه‌های نرم‌افزاری و تطابق دادن آن با استانداردها و به دست آمدن محصولی با کیفیت بالاتر می‌شود. به بیان دیگر با استفاده از فعالیت اندازه‌گیری، **معیارهایی کمّی**، برای ارزیابی و پیشرفت **فرآیند، محصول (پروژه)** ارائه می‌شود.

۶- مدیریت پیکربندی نرم‌افزار (Software Confiquration Management)

اثرات هرگونه تغییرات را در سرتاسر فرآیند تولید نرم‌افزار مدیریت می‌کند. مدیریت پیکربندی نرم‌افزار را می‌توان معادل مدیریت و کنترل تغییرات در نظر گرفت. بنابراین مدیریت پیکربندی نرم‌افزار همانند مدیریت تغییرات، هم روی تغییراتی که پس از تحویل محصول به مشتری رخ می‌دهند، اعمال می‌شود و هم تغییراتی که قبل از تحویل به مشتری رخ داده‌اند را کنترل می‌کند.

۷- مدیریت قابلیت استفاده مجدد

ضوابطی برای محصول کاری که قابلیت استفاده مجدد دارد (که شامل تمام مؤلفه‌ها می‌شود) تعریف شده و نیز مکانیزمی برای تولید مؤلفه‌هایی با قابلیت استفاده‌ی مجدد پایه‌ریزی می‌شود. در این دیدگاه، مؤلفه‌ی در حال تولید، باید هم نیازهای پروژه‌ی فعلی را برطرف سازد و هم نیازهای احتمالی پروژه‌های مشابه که در آینده تولید خواهند شد را پوشش دهد.

۸- تولید و پیش تولید محصولات کاری

شامل فعالیت‌های لازم برای تولید محصولات کاری مانند مدل‌ها، مستندات، فرم‌ها و لیست‌ها می‌شود.

مدل‌های فرآیند تولید نرم‌افزار

فرآیند تولید نرم‌افزار، مجموعه‌ای از فعالیت‌های چارچوبی (ارتباطات، برنامه‌ریزی، مدل‌سازی، ساخت و استقرار) است که هدفشان تولید نرم‌افزاری با کیفیت و مطابق با خواسته‌های مورد انتظار مشتری می‌باشد. مدل فرآیند تولید نرم‌افزار براساس ماهیت نرم‌افزاری که قرار است تولید شود

انتخاب می‌گردد. همه‌ی مدل‌های فرآیند تولید نرم‌افزار (ساخت‌یافته و شیءگرا) از فعالیت‌های چارچوبی پیروی می‌کنند اما در جریان کار شباهت‌ها و تفاوت‌هایی دارند. مدل‌های فرآیند تولید نرم‌افزار بر دو طبقه‌ی سنتی و مدرن هستند.

مدل‌های فرآیند تولید نرم‌افزار سنتی (ساخت‌یافته)

مدل‌های فرآیند تولید نرم‌افزار سنتی بر دو دسته‌ی کلی غیرتکاملی سنتی و تکاملی سنتی هستند:

مدل‌های غیرتکاملی سنتی

مدل‌های فرآیند غیرتکاملی سنتی یا تجویزی[1] در ابتدا برای نظم بخشیدن به فرآیند تولید نرم‌افزار پیشنهاد شدند. این مدل‌های سنتی به میزان نسبتاً قابل قبولی به کار مهندسی نرم‌افزار ساختار بخشیده‌اند و راهنمای اثربخشی برای تیم‌های نرم‌افزاری بوده‌اند. این مدل‌ها را تجویزی می‌نامند، زیرا مجموعه‌ای از فعالیت‌های چارچوبی و چتری را برای هر پروژه تجویز می‌کنند. در این مدل‌ها، تولید نرم‌افزار مطابق فعالیت‌های چارچوبی، مراحل مختلفی دارد که هر مرحله دارای ورودی، فعالیت و خروجی خاص خود می‌باشد. خروجی هر مرحله در این مدل‌ها، ورودی مرحله بعدی است و از فعالیت ارتباطات تا استقرار ادامه می‌یابند.

مدل آبشاری[2]

مدل آبشاری، که گاه از آن به عنوان چرخه‌ی حیات کلاسیک یاد می‌شود، روشی ترتیبی برای تولید نرم‌افزار پیشنهاد می‌کند. این مدل با فعالیت ارتباطات شروع می‌شود و با فعالیت‌های برنامه‌ریزی، مدل‌سازی (تحلیل و طراحی) و ساخت (پیاده‌سازی و تست) پیش می‌رود و با فعالیت استقرار پایان می‌یابد، تحویل پروژه انجام می‌گیرد و کار تمام می‌شود. و دیگر فرصت و تکرار دیگری در کار نخواهد بود تا خواسته‌های فراموش شده یا جدید به پروژه اضافه گردد.

خصوصیت اصلی این مدل این است که هیچ‌گونه بازخوردی بین مراحل این مدل وجود ندارد. مانند آب که نمی‌تواند در آبشار به عقب برگردد، در این مدل نیز بعد از ورود به یک فعالیت به فعالیت‌های قبلی نمی‌توان بازگشت. این مدل زمانی کاربرد دارد که کلیه‌ی نیازمندی‌های مشتری در همان ابتدای پروژه مشخص، ثابت و بدون تغییر باشد. بنابراین این مدل در تولید پروژه‌های کوچک ساده (مانند سیستم حقوق و دستمزد واحد حسابداری که نیازمندی‌های مشخص و ثابتی دارد.) و یا تولید مجدد پروژه‌های بزرگ و حتی پیچیده‌ی از قبل از ساخته شده (به دلیل مشخص بودن لیست نیازمندی‌ها در تولید مجدد) کارایی بالایی دارد و کارآمد خواهد بود.

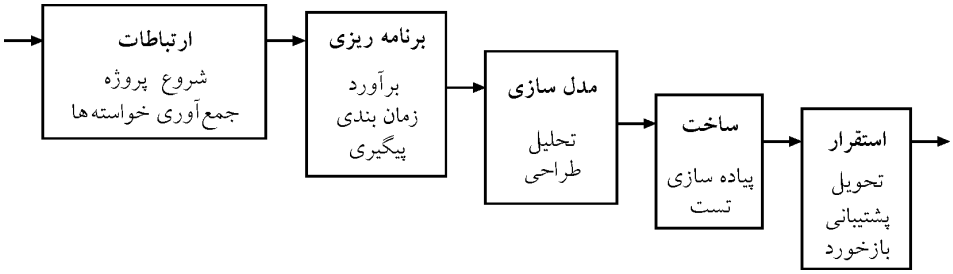
اما در پروژه‌های بزرگ و پیچیده به دلیل عدم مشخص بودن تمام نیازمندی‌ها در ابتدای پروژه و حتی تغییرات نیازمندی‌ها در حین انجام پروژه کارایی پایینی خواهد داشت. مدل آبشاری در

[1] Prescriptive
[2] Waterfall model

شرایطی که خواسته‌ها به طور کامل و جامع مشخص، ثابت و پایدار است و قرار است کـه کـار تـا پایان به شیوه‌ای خطی پیش برود، می‌تواند به عنوان مدلی مفید، مورد استفاده قرار گیرد.

برای مثال، در بسیاری از پروژه‌های سـاختمان‌سـازی، نیازمنـدی‌هـا از قبـل مشـخص هسـتند، بنابراین فرآیند تولید پروژه را می‌توان براساس مدل آبشاری بنا نهاد. اما در دنیای نـرم‌افـزار چنـین پروژه‌هایی به ندرت وجود دارد. به گونه‌ای که بسیاری از متخصصان نرم‌افزار بر این عقیده‌انـد کـه تمام پروژه‌هایی که نیازمندی‌هایش به طور جامع و کامل مشخص باشند، قبلاً توسط دیگران انجـام شده‌اند!

توجه: به مدل آبشاری، مدل خطی (Linear Model) و مدل ترتیبی (Sequential Model) نیـز گفته می‌شود.



مدل آبشاری

معایب مدل آبشاری

۱- پروژه‌های واقعی به ندرت جریان ترتیبی پیشنهاد شده توسط این مـدل را دنبـال مـی‌کننـد. ترتیبی بودن روال فعالیت‌های ارتباطات تا اسـتقرار در ایـن مـدل نیازمنـد مشـخص بـودن تمـامی نیازمندی‌های پروژه در ابتدای کار می‌باشد اما ماهیت اغلب پروژه‌های نرم‌افزاری بدین گونه نیست که تمامی نیازمندی‌ها در ابتدای پروژه مشخص باشند. بنابراین ایـن مـدل در مـواردی کـه همـه‌ی نیازمندی‌ها در ابتدای پروژه مشخص نباشد به دلیل ماهیت ترتیبی بودن مدل و عـدم بازگشـت بـه عقب کارایی لازم را نخواهد داشت. زیرا لازم است همه چیز از ابتدا مشـخص باشـند، کـه اغلـب محال است! در یک بیان ساده اینطور می‌توان بیان کرد که ماهیت اغلب پروژه‌های نرم‌افزاری بـدین شکل است که به ندرت پیش می‌آید که یک مرحله را به طور کامل تمام کنند و وارد مرحله بعدی شوند. از آنجا که در اغلب پروژه‌های نرم‌افزاری نیازمندی‌ها ذره ذره شناسایی می‌شوند و منجر بـه تکامل نرم‌افزار در طی فعالیت‌های چارچوبی بعدی می‌شـوند، مـدل آبشـاری در ایـن مـوارد کـارا نخواهد بود.

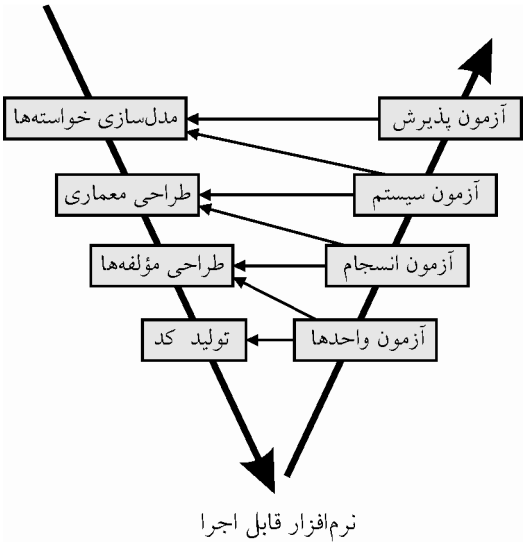
۲- اغلب برای مشتری دشوار است که همه‌ی نیازهای خود را در همان ابتدای کار بیان کند، او دوست دارد برخی از نیازمندی‌ها را در ابتدای کار بیان کند و برخی دیگر را در حـین انجـام کـار. این علاقه و دوست داشتن مشتری با مدل آبشاری برآورده نمی‌شود. اغلب، شناسایی نیازمندی‌های واقعی سیستم، یک فرآیند مستمر است که این مورد با ماهیت مدل آبشاری سازگاری ندارد. به جز

در پروژه‌های تولید سیستم‌های خاص که بارها و بارها نمونه‌ی مشابه‌شان تولید شده است، نیازمندی‌های یک سیستم را اغلب نمی‌توان قبل از شروع فرآیند تولید و پیش از اقدام به پیاده‌سازی آن، به طور کامل و جامع شناسایی نمود.

۳- مشتری باید حوصله داشته باشد. زیرا نسخه‌ی عملیاتی را دیر می‌بیند. به عبارت دیگر تا پایان تولید نهایی نرم‌افزار امکان دیدن آن توسط مشتری وجود ندارد. بنابراین، مشتری باید تا پایان کار پروژه صبور باشد، در این میان، ایجاد یک نقص در برنامه‌ی تولید شده و یا تولید چیزی غیر از خواسته‌های مشتری ممکن است فاجعه بار باشد، زیرا مشتری پس از فعالیت ارتباطات و بیان خواسته‌ها، دیگر در حین فعالیت‌های بعدی پروژه حضور نداشته است. بنابراین مهمترین مشکل رویکرد آبشاری، ضعف ذاتی آن در غلبه بر ریسک است. در اینجا، منظور از ریسک، کلیه‌ی شرایط، عوامل و نگرانی‌هایی است که می‌توانند مانع از دستیابی به موفقیت (شناسایی نیازهای دقیق مشتری، مقرون به صرفه بودن و در زمان مورد انتظار بودن) شوند. بسیاری از ریسک‌ها تنها در زمان پیاده‌سازی و تست آشکار می‌شوند. از آنجایی که در مدل آبشاری، پیاده‌سازی و تست سیستم به انتهای پروژه موکول می‌شود، در صورت آشکار شدن یک ریسک، فرصت کمی برای مدیریت آن وجود خواهد داشت و اغلب هزینه‌های زیادی برای مقابله با آن باید صرف شود. برای مثال، وجود نقص در مدل طراحی ممکن است ناشی از وجود نقص در مدل تحلیل و شناسایی نیازمندی‌ها باشد. این مشکل تنها در زمان پیاده‌سازی و تست، آشکار می‌گردد، یعنی زمانی که تصحیح آن باعث افزایش هزینه‌ها و طولانی‌تر شدن پروژه و یا حتی بسته شدن و شکست آن می‌شود. ضعف مدل آبشاری در مواجه با ریسک‌های پیچیده در پروژه‌های امروزی، مهمترین عامل انزوای این مدل در دنیای مهندسی نرم‌افزار مدرن است. فرض کنید مشتری سفارش ساخت یک میز را به نجار می‌دهد و نجار به سرعت آن را می‌سازد اما بعداً، از در اتاق مورد نظر خانه‌ی مشتری عبور نمی‌کند. زیرا مشتری و سازنده یادشان رفت بر سر اندازه‌های میز توافق کنند. این یعنی فاجعه، البته در این مورد خاص کمی فاجعه! این است که می‌گوییم اغلب مشتری در همان ابتدای کار نمی‌داند چه می‌خواهد یا یادش می‌رود دقیقاً چه می‌خواهد، انسان است و جایزالخطا بودنش، بنابراین باید بپذیریم که انسان فراموش می‌کند دقیقاً چه می‌خواهد، اما دوست دارد به او فرصت دهیم تا در حین کار و با دیدن نمونه‌هایی از پروژه بقیه‌ی خواسته‌هایش را بگوید. او اینطور خوشحال می‌شود. مدل‌های بعدی سعی در خوشحال نمودن مشتری دارند.

مدل V

شکل دیگری در نمایش مدل آبشاری به عنوان مدل V شناخته می‌شود. مدل V که در شکل زیر نمایش داده شده است.

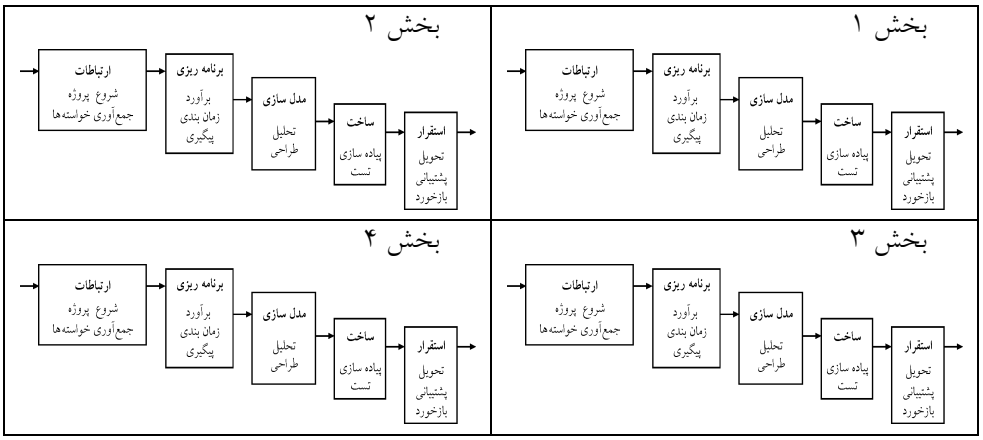


رابطه‌ی تضمین کیفیت با کنش‌های مرتبط با ارتباط، مدل‌سازی و فعالیت‌های ساخت اولیه را تصویر می‌کند. با حرکت تیم نرم‌افزاری به طرف پایین و سمت چپ V، خواسته‌های اساسی مساله رفته رفته پالایش می‌شوند و جزئیات بیشتری از آنها تعیین می‌شود و مساله و راهکار آن بهتر نمایش داده می‌شود. هنگامی‌که کدها نوشته شد، تیم در طرف راست V به طرف بالا حرکت می‌کند و اساسا یک سری آزمون اجرا می‌کند (کنش‌های تضمین کیفیت) تا هر کدام از مدل‌های ایجاد شده در مدت حرکت تیم به طرف پایین را وارسی کند. در جهان واقعیت، هیچ اختلاف بنیادی میان چرخه‌ی حیات کلاسیک و مدل V وجود ندارد. مدل V راهی برای تجسم بخشیدن به چکونگی وارسی و اعتبارسنجی در ابتدای کار نرم‌افزار فراهم می‌آورد.

مدل توسعه سریع RAD[1]

مدل RAD، شکل پُرسرعت مدل آبشاری می‌باشد، با این تفاوت که پروژه به بخش‌های مختلف تقسیم شده و هر بخش، توسط یک تیم، مطابق مدل آبشاری ایجاد می‌گردد و در پایان نتیجه‌ی تیم‌ها، برای خلق محصول نهایی ترکیب می‌گردد. مدل RAD، سرعت خود را مدیون بهره‌گیری از تکنیک بخش‌بندی و موازی‌سازی بخش‌های مختلف پروژه است. چنانچه نیازمندی‌ها به خوبی شناسایی شده و دامنه پروژه کوچک باشد این مدل قادر است یک سیستم کاملاً عملیاتی را در مدت زمان بسیار کوتاه (مثلاً بین ۶۰ تا ۹۰ روز) تولید نماید. در این مدل نرم‌افزار به قسمت‌های مختلف تقسیم شده و همواره سعی می‌شود که نرم‌افزار موردنظر سریع‌تر تولید شود. نکته قابل توجه این است که نرم‌افزار موردنظر باید خاصیت تفکیک‌پذیری داشته باشد تا بتوان این مدل را پیاده‌سازی کرد.

[1] Rapid Application Development



مدل RAD

ویژگی‌های مدل RAD

۱- شرط لازم برای انجام پروژه‌های نرم‌افزاری توسط مدل RAD، قابلیت بخش‌بندی پروژه است.

۲- از آنجا که در مدل RAD، هر بخش از مدل آبشاری استفاده می‌کند، پس بنا بر ویژگی‌های مدل آبشاری، باید تمامی نیازمندی‌های پروژه (لیست نیازمندی‌ها) در ابتدای پروژه مشخص باشد. در این صورت این مدل می‌تواند ظرف مدت بسیار کوتاهی (۶۰ تا ۹۰ روز) محصول نهایی را ایجاد نماید.

۳- در پروژه‌های بزرگ، تعداد بخش‌های مختلف پروژه زیاد می‌شود، به همین دلیل به تیم‌های نرم‌افزاری بیشتری نیاز خواهد بود. بنابراین با افزایش حجم پروژه باید نیروی انسانی کافی برای پیمانه‌ها وجود داشته باشد.

۴- این مدل در پروژه‌هایی با ریسک‌های فنی بالا، به دلیل عدم امکان شناسایی نیازمندی‌های مشتری در ابتدای پروژه کارآمد نخواهد بود.

۵- موفقیت مدل RAD، وابسته به تعامل مناسب سازنده و مشتری است و هر دو باید برای انجام سریع فعالیت‌ها با یکدیگر هماهنگ باشند تا بتوانند در موعد مقرر تولید نهایی را تحویل مشتری دهند. چون اگر یک قسمت از این پروژه انجام نشده باشد. تحویل پروژه میسر نیست، بنابراین مدیریت این مدل اهمیت فراوانی دارد.

مکانیزم نمونه‌سازی دورانداختنی

همانطور که پیش از این نیز گفتیم، علاوه بر ابزارهای مشاهده، مصاحبه و گفتگو، مکانیزم نمونه‌سازی دورانداختنی نیز برای شناسایی نیازمندی‌های مشتری در فعالیت ارتباط مورد استفاده قرار می‌گیرد. در مکانیزم نمونه‌سازی دورانداختنی، یک پیاده‌سازی عملیاتی از سیستم با ابزارهای ارزان‌قیمت، فقط و فقط به منظور شناسایی نیازمندی‌های مشتری، ایجاد و سپس دور انداخته

می‌شود، سپس سیستم نهایی براساس فرآیند تولید مجزایی تولید می‌گردد، توجه کنید که سیستم نهایی براساس فرآیند تولید مجزای دیگری تولید می‌گردد، مثلاً پس از شناسایی تمامی نیازمندی‌های مشتری توسط مکانیزم نمونه‌سازی دورانداختنی، لیست تمامی نیازمندی‌های مشتری آماده است، بنابراین در ادامه برای تولید نرم‌افزار از مدل آبشاری می‌توان استفاده نمود.

هدف از مکانیزم دورانداختنی، استخراج نیازمندی‌های مشتری است. شروع مکانیزم نمونه‌سازی دورانداختنی براساس نیازمندی‌هایی است که کمتر درک شده‌اند. یکی از مزایای این روش، کاهش ریسک نیازمندی‌ها است.

توجه: به مکانیزم‌نمونه‌سازی دورانداختنی، **الگوسازی بسته** نیز گفته می‌شود.

توجه: در ادامه و در سراسر این کتاب مکانیزم نمونه‌سازی دورانداختنی را، به اختصار «نمونه‌سازی» در نظر خواهیم گرفت.

فرض کنید مشتری سفارش ساخت یک میز با چوب گران قیمت را به یک نجار می‌دهد و نجار به سرعت آن را می‌سازد. اما بعداً از در اتاق مورد نظر مشتری عبور نمی‌کند و با وجود صرف وقت و هزینه، نیاز مشتری برای داشتن یک میز در اتاقش برآورده نمی‌شود. زیرا مشتری و سازنده یادشان رفت بر سر اندازه‌های میز توافق کنند، این یعنی فاجعه. البته در این مورد خاص کمی فاجعه. جملاتی که بیان کردیم در ذهن خود تجسم کنید. خب یادشان رفت، چه کار می‌توان کرد، انسان است و جایزالخطا بودنش.

اما آیا نمی‌توان به آنها اشکال گرفت که چرا وقتی مطمئن نبودید که دقیقاً چه می‌خواهید و چه می‌خواهید بسازید با هم بر سر ساخت توافق کردید؟!

این یک مثال خیلی ساده بود، اغلب، انجام پروژه‌های نرم‌افزاری بزرگ صرف وقت و هزینه‌های سرسام‌آوری دارد و اگر نرم‌افزاری ساخته شود که در زمان مورد انتظار به کار مشتری نیاید می‌توان قاطع گفت که اینجا واقعاً فاجعه رخ داده است. فرض کنید نرم‌افزاری می‌بایست برای انتخابات ریاست جمهوری که در یک تاریخ از قبل مشخص شده برگزار می‌گردد، می‌رسید، اما نمی‌رسد!

زیرا براساس خواسته‌های دقیق وزارت کشور به عنوان مجری انتخابات ساخته نشده است و حتی نیازهایی فراموش شده است. روز انتخابات فرا می‌رسد، اما نرم‌افزار آماده نیست، فرصت برای جبران هم نیست، این یعنی فاجعه، به معنای واقعی کلمه.

اما چه می‌توان کرد؟ راهکار چیست؟ در یک جمله، راهکار نمونه‌سازی است. نمونه‌سازی راهکاری برای تشخیص دقیق خواسته‌های مشتری است.

در مثال مشتری و نجار آیا بهتر نبود، ابتدا میز با چوب ارزان قیمت ساخته می‌شد و پس از مشاهده مشتری و میزان رضایتمندی او و حتی ایجاد فرصت برای بیان خواسته‌های جدیدش، و بعد از مشخص شدن دقیق و کامل لیست نیازمندی‌های مشتری، ساخت میز با چوب گران قیمت آغاز می‌گردید؟

در مثال نرم‌افزار انتخابات ریاست جمهوری آیا بهتر نبود ابتدا توسط ابزارهای ارزان و برنامه‌نویسان ارزان و به تبع هزینه پایین، نمونه‌هایی سریع از بخش‌های مختلف نرم‌افزار آماده

می‌شد و پس از مشاهده وزارت کشور و میزان رضایت‌مندی و حتی بیان خواسته‌های جدید و مشخص شدن دقیق لیست نیازمندی‌های وزارت کشور، ساخت نرم‌افزار با ابزارهای گران قیمت و برنامه‌نویسان گران قیمت براساس لیست دقیق نیازمندی‌های وظیفه‌مندی و غیروظیفه‌مندی وزارت کشور آغاز می‌گردید؟ پاسخ مثبت است، بله، واضح است که بهتر بود.

هرگاه برای شناسایی خواسته‌های مشتری و تهیه‌ی لیست نیازی‌ها ابهام داشتید، نمونه‌سازی کنید، نمونه‌ای ارزان قیمت از بخش‌های مختلف پروژه توسط ابزارهای ارزان و برنامه‌نویسان ارزان بسازید نشان مشتری دهید. کم‌کم، کم‌کم بقیه خواسته‌هایش را هم می‌گوید.

یادتان باشد در حال ساخت نمونه برای مشاهده مشتری و شناسایی مابقی نیازمندی‌های او هستید، لیست نیازمندی‌های مشتری که دقیقاً مشخص شد و بعد به توافق نهایی هم رسیدید، نمونه‌های ساخت شده را دور بریزید، زیرا به هدفی که می‌خواستید رسیدید، نیازها مشخص شدند و دیگر به نمونه‌ها نیازی ندارید!

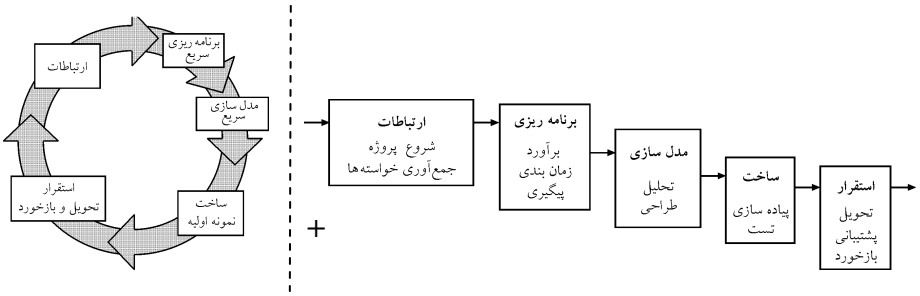
نگران نباشید، دور بریزید، شما با این راهکار موفق شدید به طور دقیق بدانید مشتری واقعاً چه می‌خواهد. **تا همین جا موفقیت بزرگی نصیب شما شده است!**

حال ساخت محصول نهایی را به برنامه‌نویسان گران قیمت خود بسپارید، با خیالی آسوده، زیرا این بار براساس لیست دقیق نیازمندی‌های وظیفه‌مندی و غیروظیفه‌مندی، دقیقاً همان چیزی در حال تولید است که مشتری می‌خواهد.

مدل نمونه‌سازی

شرح مطالب مربوط به مکانیزم نمونه‌سازی دورانداختنی را به یاد آورید، در آنجا گفتیم که در مکانیزم نمونه‌سازی دورانداختنی، یک پیاده‌سازی عملیاتی از سیستم با ابزارهای ارزان قیمت فقط و فقط به منظور شناسایی نیازمندی‌های مشتری، ایجاد و سپس دور انداخته می‌شود، سپس سیستم نهایی براساس فرآیند تولید مجزایی تولید می‌گردد، همچنین تأکید کردیم که سیستم نهایی براساس فرآیند تولید مجزای دیگری تولید می‌گردد. اگر آن فرآیند مجزای دیگر، مدل آبشاری باشد، به حاصل جمع مکانیزم نمونه‌سازی دورانداختنی و مدل آبشاری «مدل نمونه‌سازی» گفته می‌شود.

در واقع مدل نمونه‌سازی از ترکیب مکانیزم نمونه‌سازی دورانداختنی و مدل آبشاری ایجاد شده است. در ابتدای کار توسط مکانیزم نمونه‌سازی دورانداختنی تمامی لیست نیازمندی‌های مشتری تکمیل می‌گردد، سپس توسط مدل آبشاری به شکل خطی نرم‌افزار ایجاد می‌گردد و کار تمام می‌شود. در واقع نیازمندی‌های مشتری طی تکرارهای مکانیزم نمونه‌سازی دورانداختنی شناخته می‌شوند، سپس طی یک روال خطی، توسط مدل آبشاری، نرم‌افزار تولید می‌گردد و کار تمام می‌شود و دیگر تکامل نمی‌یابد زیرا مدل نمونه‌سازی یک مدل غیر تکاملی است و **پس از ساخت نهایی** امکان ادامه‌ی پروژه و تغییر وجود ندارد.



مدل نمونه سازی

اغلب، مشتری مجموعه‌ای از اهداف کلی را برای نرم‌افزار تعریف می‌کند، اما جزئیات ورودی، پردازش یا شرایط مورد نیاز خروجی را تعریف نمی‌کند. در حالت‌های دیگر، سازنده ممکن است از کارایی الگوریتمی خاص، یا توانایی یک سیستم عامل، یا نحوه‌ی تعامل کاربران نهایی و نرم‌افزار اطمینان نداشته باشد، اگر نرم‌افزار بر مبنای همین نیازمندی‌های مبهم ایجاد گردد ممکن است نتواند همه‌ی نیازهای مشتری را برآورده سازد و بدین ترتیب پروژه با شکست مواجه می‌شود. در این موارد و بسیاری از موار دیگر، مکانیزم نمونه‌سازی دورانداختنی که در ابتدای مدل نمونه‌سازی قرار دارد روش مناسبی برای به دست آوردن نیازمندی‌های دقیق مشتری است.

توجه کنید که مکانیزم نمونه‌سازی دورانداختنی راهکاری برای تشخیص لیست نیازمندی‌های مشتری است که در ابتدای مدل نمونه‌سازی قرار دارد. بنابر خاصیت مکانیزم نمونه‌سازی دورانداختنی سازنده قادر است نمونه‌ای از نرم‌افزاری را که می‌خواهد، تولید کند، هر چند به طور مختصر و مفید به طوری که مشتری ارتباط میان خود و نرم‌افزار را احساس کرده و متوجه عملکرد نسبی نرم‌افزار شود.

مدل نمونه‌سازی با جمع آوری خواسته‌های مشتری به کمک مکانیزم نمونه‌سازی دورانداختنی آغاز می‌گردد، سازنده در ابتدا با مشتری به گفتگو می‌پردازد و نظرات مشتری را در رابطه با نیازهایی که توقع دارد تا نرم‌افزار مورد نظرش در آینده برآورده سازد جویا می‌شود و آنها را جمع‌آوری می‌کند. سپس یک مدل‌سازی سریع (تحلیل و طراحی) اتفاق می‌افتد تا آن دسته از ویژگی‌هایی که به چشم مشتری می‌آیند (مثل شکل و شمایل صفحات ورودی) به طور سریع مدل‌سازی گردد. به دنبال این مدل‌سازی، یک نمونه‌ی اولیه از نرم‌افزار ساخته می‌شود و توسط مشتری مورد ارزیابی قرار می‌گیرد. نتایج ارزیابی برای پالایش و تکمیل نیازمندی‌های نرم‌افزاری که قرار است ساخته شود، استفاده می‌گردد. مجدداً براساس نیازمندی‌های پالایش شده، نمونه‌ی دیگری که نسبت به نمونه‌ی قبل‌تر کامل‌تر است ساخته شده و در اختیار مشتری قرار داده می‌شود تا ارزیابی گردد. این چرخه، تا زمانی که لیست نیازمندی‌های مشتری تکمیل گردد، تکرار می‌شود. در کل ایده‌ی مدل نمونه‌سازی، استفاده از مکانیزم نمونه‌سازی دورانداختنی به عنوان راهکاری برای تشخیص لیست نیازمندی‌های مشتری است. توجه کنید که نمونه‌های ساخته شده، دورانداخته می‌شوند.

معایب مدل نمونه‌سازی

۱- چون نمونه‌ای از نرم‌افزار بدون رعایت مسائل کیفی در اختیار مشتری قرار می‌گیرد و مشتری در ابتدا نمی‌تواند نرم‌افزار کامل را مشاهده کند، ممکن است تصور غلطی از نرم‌افزار نهایی پیدا کند، زیرا مشتری ظاهراً یک نسخه‌ی کاری از نرم‌افزار را می‌بیند. ولی نمی‌داند که این نمونه‌ی اولیه فقط یک «**ماکت**» است که با «**موم**» سرهم‌بندی شده است. خبر ندارد که کیفیت قربانی سرعت ساخت نمونه‌ی اولیه شده است. زمانی که مشتری اطلاع می‌یابد در آینده این محصول باید به طور مجدد به گونه‌ای ایجاد شود که کیفیت مطلوب به دست آید، ممکن است ناراحت شده و تقاضا کند همان محصول نمونه‌ی اولیه با کمی تغییر به محصول نهایی تبدیل گردد و یا کلاً پروژه را لغو کند!

۲- سازنده اغلب برای دستیابی سریع‌تر به مدل نمونه‌ی اولیه به مسائل کیفی توجه نمی‌کند. برای‌مثال زبان برنامه‌نویسی نامناسب یا برنامه نویس ارزان قیمت ناآشنا به ایجاد الگوریتم‌های بهینه برای نوشتن مدل نمونه انتخاب می‌نماید آن هم فقط به دلیل سهولت کار یا این که صرفاً چون این برنامه فقط نمونه‌ای بیش نیست، اقدام به این انتخاب‌ها کند و پس از نوشتن برنامه‌ی نمونه، سازنده این برنامه را کنار نگذاشته و به تکمیل همان نمونه‌ی اولیه بدون کیفیت بپردازد و همان را به مشتری تحویل دهد که در آینده مشکلاتی را برای سازنده ایجاد خواهد کرد.

نتیجه اینکه، این مدل زمانی سرانجام خوشی برای سازنده و مشتری خواهد داشت که در ابتدای کار بر سر مسائل مختلف پروژه توافق کنند. مثلاً مشتری باید بداند که ساخت نمونه‌ی اولیه فقط به عنوان راهکاری جهت شناسایی نیازهای نرم‌افزار بوده و نرم‌افزار نهایی حتماً دارای معیارهای کیفیتی خواهد بود.

مدل‌های تکاملی سنتی

در طول سال‌های گذشته، بسیاری از افراد در دانشگاه‌ها و نیز شرکت‌های پیشروی صنعت نرم‌افزار، تلاش‌های زیادی برای ارائه و معرفی مدل‌ها و رویکردهای دیگری که جایگزین رویکرد مدل آبشاری شود، انجام داده‌اند. حاصل این تلاش‌ها، ارائه‌ی ده‌ها مدل فرآیند دیگر بوده است. یکی از راهکارها و تجارب موفق، رویکردی است مبتنی بر تکرار و تکامل که در مقابل رویکرد مدل آبشاری قرار دارد. در رویکرد مبتنی بر تکرار و تکامل، فرصت یادگیری و بهبود تدریجی در سرتاسر چرخه‌ی تولید فراهم است. بدین ترتیب، در طول پروژه، امکان تصحیح به موقع اشتباهات وجود خواهد داشت. در صورت بروز اشتباه در یک تکرار، امکان جبران آن در تکرار بعدی وجود دارد. در حالی که همانطور که پیش از این نیز بیان شد، در مدل آبشاری بسیاری از اشتباهات در انتهای پروژه آشکار می‌شود و در نتیجه فرصت کمی برای تصحیح آنها وجود خواهد داشت. رویکرد مبتنی بر تکرار و تکامل به برنامه‌ریزی مستمر و پویا در طول پروژه نیازمند است، در حالی که در مدل آبشاری، برنامه‌ریزی، یک بار و آن هم در ابتدای پروژه صورت می‌پذیرد. مدل‌های افزایشی و حلزونی براساس رویکرد مبتنی بر تکرار و تکامل ایجاد شده‌اند، اما مدل حلزونی به دلیل تأکید بر مدیریت ریسک در هر تکرار توسط تحلیل ریسک، توانایی قابل توجهی در مدیریت

ریسک دارد. در اینجا منظور از ریسک، کلیه‌ی شرایط، عوامل و نگرانی‌هایی است که می‌تواند مانع از دستیابی به موفقیت (شناسایی نیازهای دقیق مشتری، مقرون به صرفه بودن و در زمان مورد انتظار بودن) شوند. در واقع، یک تکرار، عبارت است از انجام متوالی فعالیت‌های لازم در یک بازه‌ی زمانی کوتاه و انجام چندین باره‌ی آن در طول بازه‌ی زمانی یک پروژه. بنابراین، به جای انجام فعالیت‌ها، به صورت یکباره و متوالی، چندین بار و در بازه‌های کوچک‌تری این مجموعه فعالیت‌ها را تکرار می‌نماییم. این مدل‌ها با رویکردی مبتنی بر تکرار و تکامل در هر تکرار نسخه‌هایی از نرم‌افزار را ارائه می‌دهند که هریک از قبلی کامل‌تر است. بنابراین برخلاف سایر مدل‌های فرآیند غیرتکاملی که با تحویل نرم‌افزار پایان می‌یابند، مدل‌های تکاملی را می‌توان طوری تطبیق داد که در سرتاسر عمر نرم‌افزار کامپیوتری قابل به کارگیری باشد.

توجه: در مدل‌های تکاملی در هر دور از تکرار نسخه‌ی کامل‌تری از نرم‌افزار تولید می‌شود.

مکانیزم نمونه‌سازی تکاملی

در نمونه‌سازی تکاملی، یک نمونه‌ی اولیه تولید می‌شود. در ادامه با اعمال اصلاحات بر روی نمونه‌ی اولیه، طی چند مرحله سیستم نهایی تولید می‌گردد. هدف از نمونه‌سازی تکاملی، تحویل یک سیستم عملیاتی به کاربران نهایی و شروع فرآیند تولید براساس نیازمندی‌هایی است که بهتر و بیشتر درک شده‌اند. کاربرد آن در مواردی است که نیازمندی‌های مشتری به طور کامل در ابتدای کار مشخص نباشد و یا نیاز به توسعه‌ی مبتنی بر تکرار و تکامل داشته باشیم. از مزایای نمونه‌سازی تکاملی می‌توان به درگیر کردن مشتری با فرآیند تولید سیستم که منجر به شناسایی بهتر نیازمندی‌ها می‌گردد، اشاره نمود.

توجه: به مکانیزم نمونه‌سازی تکاملی، **الگوسازی باز** نیز گفته می‌شود.

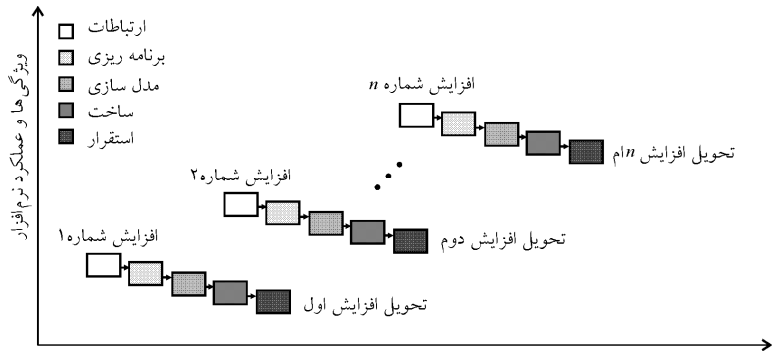
مدل افزایشی[1]

مدل افزایشی، مراحل مدل آبشاری را با رویکرد تکرار و تکامل مکانیزم نمونه‌سازی تکاملی ترکیب نموده است. بنابراین مدل افزایشی مبتنی بر مدل آبشاری و مکانیزم نمونه‌سازی تکاملی است.

نمونه‌سازی تکاملی به فرآیند تولید نرم‌افزار روح، حرکت و تکرار می‌دهد و مدل آبشاری فعالیت‌های چارچوبی هر تکرار (افزایش) را مشخص می‌کند. همچنین همانطور که پیش از این نیز گفتیم، هرگاه نیاز به شناسایی خواسته‌های مبهم مشتری وجود داشت می‌توان از مکانیزم نمونه‌سازی دورانداختنی استفاده نمود. بنابراین قبل از هر تکرار (افزایش) جهت شناسایی نیازمندی‌ها می‌توان از مکانیزم نمونه‌سازی دورانداختنی نیز استفاده نمود و نمونه را دور انداخت. اما حرکت، تکرار و تکامل همچنان می‌تواند توسط نمونه‌سازی تکاملی ادامه یابد تا محصول نهایی آماده گردد.

[1] Incremental Model

این مدل از یک سری فعالیت‌های چارچوبی تکراری تشکیل شده است که هر تکرار (افزایش) شبیه به مدل آبشاری است. با این تفاوت که روی قسمتی از نرم‌افزار انجام می‌شود. هر کدام از این قسمت‌ها یک «قطعه» قابل تحویل را ایجاد می‌کند. شکل زیر مدل افزایشی و مراحل هر افزایش را نشان می‌دهد:



مدل افزایشی

در این مدل، با توجه به خاصیت نمونه‌سازی تکاملی، پروژه به تدریج کامل می‌شود یعنی هر مرحله‌ای که می‌گذرد، پروژه کامل‌تر شده و در افزایش‌های بعدی این تکامل ادامه می‌یابد تا به محصول نهایی برسد.

مثال: نرم‌افزار واژه‌پرداز که با استفاده از مدل افزایشی توسعه یافته است، به صورت زیر تحویل داده می‌شود:

افزایش اول: عملیاتی از قبیل مدیریت فایل، تولید و ویرایش مستندات

افزایش دوم: قابلیت‌های پیچیده‌تر در ویرایش و تولید مستندات

افزایش سوم: چک کردن املاء و دستور زبان

افزایش چهارم: قابلیت‌های پیشرفته‌ی صفحه‌بندی

توجه: مشاهده می‌شود که محصول هر افزایش در مثال فوق، قطعه‌ای قابل تحویل است.

هنگامی که از یک مدل افزایشی استفاده می‌شود، افزایش نخست غالباً هسته‌ی اصلی محصول است، یعنی نیازهای اولیه رفع می‌شوند، اما بسیاری از ویژگی‌های مکمل (که برخی معلوم و برخی نامعلوم هستند) تحویل داده نمی‌شوند. هسته‌ی اصلی محصول، توسط مشتری مورد استفاده و ارزیابی قرار می‌گیرد. سپس در افزایش بعدی، قابلیت‌ها و ویژگی‌های دیگر مورد انتظار مشتری به هسته‌ی اصلی محصول اضافه می‌گردد. این فرآیند به دنبال تحویل هر قطعه تکرار می‌شود تا اینکه محصول کامل تولید شود.

توجه: مدل افزایشی، مانند مدل نمونه‌سازی و مدل‌های تکاملی دیگر، ماهیتی تکراری دارد. اما برخلاف مدل نمونه‌سازی، مدل افزایشی بر تحویل قطعه‌ای عملیاتی در هر افزایش تأکید دارد. قطعات اولیه، بخش‌هایی اولیه از محصول نهایی هستند، اما قابلیت ارائه خدمات به کاربر را دارند.

توجه: در مدل نمونه‌سازی، به واسطه‌ی مکانیزم نمونه‌سازی دورانداختنی فرآیند تکرار تنها در مرحله‌ی جمع‌آوری نیازمندی‌ها انجام می‌شود. اما در مدل افزایشی فرآیند تکرار در تمام طول فرآیند تولید نرم‌افزار انجام می‌شود.

توجه: مدل افزایشی به واسطه‌ی استفاده از مکانیزم نمونه‌سازی تکاملی بر تولید تکاملی نرم‌افزار تأکید دارد. اما مدل نمونه‌سازی به واسطه‌ی استفاده از مکانیزم نمونه‌سازی دورانداختنی بر شناسایی تکاملی نیازمندی‌ها تأکید دارد و نه تولید تکاملی نرم‌افزار. در واقع در مدل نمونه‌سازی پس از شناسایی نیازمندی‌ها توسط مکانیزم نمونه‌سازی دورانداختنی در ادامه نرم‌افزار به شیوه‌ی مدل آبشاری در قالب یک قطعه و یکجا ایجاد می‌گردد. تولید تکاملی بدین معنی است که نرم‌افزار قطعه، قطعه، و ذره ذره، و کم کم، کم کم تکامل می‌یابد و نه در قالب یک قطعه و یکجا. بنابراین در مواقعی که تغییرات زیاد است و نیاز است نسخه‌هایی از نرم‌افزار ارائه گردد و در نسخه‌های بعدی نرم‌افزار تکامل یابد مدل نمونه‌سازی کارایی نخواهد داشت و مدل‌های تکاملی (افزایشی و پیچشی) مناسب خواهند بود.

مدل افزایشی، مواقعی مفید است که منابع مالی و انسانی لازم برای تکمیل پیاده‌سازی پروژه در مهلت کاری مقرر، در دسترس نباشد. بنابراین افزایش‌های اولیه را می‌توان با افراد کمتری پیاده‌سازی کرد. اگر هسته‌ی اصلی محصول به خوبی به دست آید، افراد دیگری را (در صورت لزوم) می‌توان اضافه کرد و افزایش بعدی را پیاده‌سازی کرد.

به علاوه، افزایش‌ها می‌توانند به گونه‌ای برنامه‌ریزی شوند که **ریسک‌های تکنیکی** را مدیریت کنند. برای مثال، یک سیستم جامع و کلی ممکن است نیاز به سخت‌افزار جدیدی داشته باشد که فعلاً در حال تولید است و تاریخ تحویل آن قطعی نیست. در نتیجه می‌توان افزایش‌های اولیه را به نحوی برنامه‌ریزی کرد که به این سخت‌افزار نیازی نداشته باشد. بدین ترتیب این امکان به وجود می‌آید که بخشی از عملکردها بدون تأخیر غیر عادی به کاربر نهایی تحویل داده شود.

توجه: مدل افزایشی در مواقعی که نیازمندی‌های اولیه‌ی مشتری به خوبی تعریف شده‌اند ولی نیازمندی‌های دیگری باقی مانده‌اند و نیاز به تکرار و تکامل می‌باشد مناسب است.

توجه: مدل افزایشی علاوه بر مدیریت ریسک‌های تکنیکی، از مکانیزم نمونه‌سازی دورانداختنی به عنوان راهکاری برای کاهش ریسک مربوط به شناسایی نیازمندی‌ها در هر افزایش استفاده می‌کند و به ریسک‌های دیگر پروژه مانند از دست دادن مدیران و اطلاعات به دلیل عدم مدیریت ریسک (تحلیل ریسک) توجه ندارد.

مدل پیچشی[1] (مارپیچی یا حلزونی)

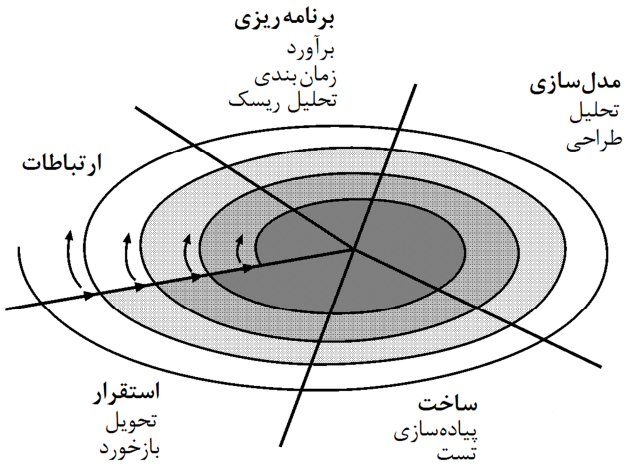
رویکرد مدل پیچشی را همانند مدل افزایشی در نظر بگیرید با این تفاوت اساسی که در مدل پیچشی مدیریت ریسک (تحلیل ریسک) در فعالیت برنامه‌ریزی به طور جدی و در تمام جوانب پروژه انجام می‌گیرد. اما در مدل افزایشی فقط **ریسک مربوط به شناسایی درست نیازمندی‌ها** در

[1] Spiral Model

هر تکرار توسط مکانیزم نمونه‌سازی دورانداختنی و **ریسک تکنیکی** مدیریت می‌گردد. مدل پیچشی را به مانند یک کلاف به دور خود پیچیده شده در نظر بگیرید. سپس این کلاف را باز کنید و به صورت یک خط راست در ذهن خود تجسم کنید، حال آن را به قطعاتی شامل فعالیت‌های چارچوبی (ارتباطات، برنامه‌ریزی، مدل‌سازی، ساخت و استقرار) تقسیم کنید، آن چه مشاهده خواهید کرد قطعاتی است مانند، قطعات (افزایش‌های) موجود در هر تکرار مدل افزایشی. حال قطعات تقسیم شده را مجدداً به هم بچسبانید. سپس به آن به شکل یک کلاف، پیچش دهید تا مجدداً شکل مدل پیچشی را به خود بگیرد!

مدل پیچشی نیز همانند مدل افزایشی، مراحل مدل آبشاری را با رویکرد تکرار و تکامل مکانیزم نمونه‌سازی تکاملی ترکیب نموده است. بنابراین مدل پیچشی مبتنی بر مدل آبشاری و مکانیزم نمونه‌سازی تکاملی است. نمونه‌سازی تکاملی به فرآیند تولید نرم‌افزار روح، حرکت و تکرار می‌دهد و مدل آبشاری فعالیت‌های چارچوبی هر تکرار (پیچش) را مشخص می‌کند. همچنین همانطور که پیش از این نیز گفتیم، هرگاه نیاز به شناسایی خواسته‌های مبهم مشتری وجود داشت می‌توان از مکانیزم نمونه‌سازی دورانداختنی استفاده نمود. بنابراین قبل از هر تکرار (پیچش) جهت شناسایی نیازمندی‌ها می‌توان از مکانیزم نمونه‌سازی دورانداختنی استفاده نمود و نمونه را دور انداخت. اما حرکت، تکرار و تکامل همچنان می‌تواند توسط نمونه‌سازی تکاملی ادامه یابد تا محصولی نهایی آماده گردد.

این مدل از یک سری فعالیت‌های چارچوبی تکراری تشکیل شده است که هر تکرار (پیچش) شبیه به مدل آبشاری است. با این تفاوت که روی قسمتی از نرم‌افزار انجام می‌شود. هر کدام از این قسمت‌ها یک «قطعه» قابل تحویل را ایجاد می‌کند. شکل زیر مدل پیچشی و مراحل هر پیچش را نشان می‌دهد:



مدل پیچشی

در این مدل با توجه به خاصیت نمونه‌سازی تکاملی، پروژه به تدریج کامل می‌شود، یعنی هر مرحله‌ای که می‌گذرد، پروژه کامل‌تر شده و در پیچش‌های بعدی این تکامل ادامه می‌یابد تا به محصول نهایی برسد. هنگامی که از یک مدل پیچشی استفاده می‌شود، پیچش‌های نخست غالباً هسته‌ی اصلی محصول است، یعنی نیازهای اولیه رفع می‌شوند، اما بسیاری از ویژگی‌های مکمل (که برخی معلوم و برخی نامعلوم هستند) تحویل داده نمی‌شوند.

هسته‌ی اصلی محصول، توسط مشتری مورد استفاده و ارزیابی قرار می‌گیرد. سپس در پیچش بعدی، قابلیت‌ها و ویژگی‌های دیگر مورد انتظار مشتری به هسته‌ی اصلی محصول اضافه می‌گردد. این فرآیند به دنبال تحویل هر قطعه تکرار می‌شود تا اینکه محصول کامل تولید شود.

توجه: مدل پیچشی، مانند مدل نمونه‌سازی و مدل‌های تکاملی دیگر، ماهیتی تکراری دارد، اما برخلاف مدل نمونه‌سازی، مدل پیچشی بر تحویل قطعه‌ای عملیاتی، در هر افزایش تأکید دارد. قطعات اولیه، بخش‌هایی اولیه از محصول نهایی هستند، اما قابلیت ارائه‌ی خدمات به کاربر را دارند.

توجه: در موارد خاص در صورتی که حتی نیازمندی‌های اولیه نیز مشخص نباشند، پیچش‌های نخست می‌تواند مدلی کاغذی باشد و در تکرارهای بعدی هسته‌ی اصلی محصول ایجاد گردد.

توجه: در مدل نمونه‌سازی، به واسطه‌ی مکانیزم نمونه‌سازی دورانداختنی فرآیند تکرار تنها در مرحله‌ی جمع‌آوری نیازمندی‌ها انجام می‌شود اما در مدل پیچشی فرآیند تکرار در تمام طول فرآیند تولید نرم‌افزار انجام می‌شود.

توجه: مدل پیچشی به واسطه‌ی استفاده از مکانیزم نمونه‌سازی تکاملی بر تولید تکاملی نرم‌افزار تأکید دارد. اما مدل نمونه‌سازی به واسطه‌ی استفاده از مکانیزم نمونه‌سازی دورانداختنی بر شناسایی تکاملی نیازمندی‌ها تأکید دارد و نه تولید تکاملی نرم‌افزار. در واقع در مدل نمونه‌سازی پس از شناسایی نیازمندی‌ها توسط مکانیزم نمونه‌سازی دورانداختنی در ادامه نرم‌افزار به شیوه‌ی مدل آبشاری در قالب یک قطعه و یکجا ایجاد می‌گردد. تولید تکاملی یعنی نرم‌افزار قطعه قطعه، ذره ذره و کم کم، کم کم تکامل می‌یابد و نه در قالب یک قطعه و یکجا. بنابراین در مواقعی که تغییرات زیاد است و نیاز است نسخه‌هایی از نرم‌افزار ارائه گردد و در نسخه‌های بعدی نرم‌افزار تکامل یابد مدل نمونه‌سازی کارایی نخواهد داشت. و مدل‌های تکاملی (افزایشی و پیچشی) مناسب خواهند بود.

مدل پیچشی، مدلی امن، مطمئن و قابل اعتماد است. زیرا در هر تکرار یا پیچش در فعالیت مربوط به برنامه‌ریزی توسط عمل تحلیل ریسک، تمامی ریسک‌های مربوط به پروژه را به دقت در نظر می‌گیرد، شناسایی و در ادامه برطرف می‌نماید. از آنجا که تحلیل ریسک در ابتدای هر چرخش انجام می‌شود، بنابراین امکان وقوع ریسک بسیار پایین خواهد آمد. در نتیجه کیفیت نرم‌افزار تولیدی بالا خواهد رفت. همچنین مدیر پروژه بر امور جاری کنترل دقیق دارد و همه‌ی مسایل و هزینه‌ها با توافق طرفین (مشتری و سازنده) صورت می‌گیرد. به این کنترل دقیق بر امور جاری

پروژه در هر تکرار اصطلاحاً «**نقاط عطف لنگرگاهی**[1]» نیز گفته می‌شود.

توجه: مدل پیچشی در مواقعی که نیازمندی‌های اولیه به خوبی تعریف شده‌اند ولی نیازمندی‌های دیگری باقی مانده‌اند و نیاز به تکرار و تکامل می‌باشد مناسب است و یا حتی در مواقعی که نیازمندی‌های اولیه بخوبی تعریف نشده‌اند و مسائل ناشناخته‌ی بسیار زیادی وجود دارد مناسب است. اگر قرار باشد پروژه در امنیت بالایی از هر لحاظ ادامه یابد نیاز به مدیریت ریسک به طور مستمر می‌باشد که این خاصیت در مدل پیچشی به واسطه‌ی تحلیل ریسک وجود دارد.

توجه: مدل پیچشی از مکانیزم نمونه‌سازی دورانداختنی به عنوان راهکاری برای کاهش ریسک مربوط به شناسایی نیازمندی‌ها در هر پیچش، استفاده می‌کند و همچنین به دلیل تحلیل ریسک به واسطه‌ی مدیریت ریسک تمامی ریسک‌های مربوط به کلیت پروژه (شناسایی نیازمندی‌های دقیق مشتری، مقرون به صرفه بودن و در زمان مورد انتظار بودن) را کنترل می‌کند. در بیان ساده، مدل پیچشی به واسطه‌ی مدیریت ریسک (تحلیل ریسک) بستری امن، برای تولید نرم‌افزار ایده‌آل مشتری فراهم می‌سازد.

توجه: در صورتی که علاوه بر تحلیل ریسک، تحلیل برنده برنده، به معنی انجام توافقات برد برد بین مشتری و سازنده در تمامی مراحل پروژه به فعالیت برنامه‌ریزی اضافه گردد، مدل پیچشی، **مدل پیچشی برنده برنده** خواهد بود.

معایب مدل پیچشی

۱- قانع کردن مشتری به ویژه در این مورد که تکامل پروژه قابل کنترل می‌باشد، کار دشواری است. در واقع ما برای انجام مراحل مختلف پروژه نیازمند تأمین بودجه از سوی مشتری هستیم و ممکن است خروجی‌های بخش‌های مختلف تکامل پروژه از نظر مشتری چندان مطلوب نباشد و در تأمین بودجه‌ی مورد نیاز کوتاهی کند.

۲- این مدل براساس ارزیابی، در نظر گرفتن و تعیین کردن ریسک در هر چرخش بنا شده است که خود این ارزیابی و مدیریت ریسک نیازمند تخصص مورد نیاز خودش می‌باشد، بنابراین موفقیت این مدل فرآیند، کاملاً وابسته به تصمیمات و سیاست‌هایی است که فرد و یا گروه مدیریت ریسک اتخاذ می‌کنند.

۳- اگر یک ریسک بزرگ شناسایی و مدیریت نشود، ممکن است مشکلات غیرقابل پیش بینی را در نرم‌افزار به وجود آورد.

معایب مدل‌های تکاملی سنتی

مشخصه‌ی اصلی نرم‌افزارهای مدرن امروزی، تغییر پیوسته، در فواصل زمانی بسیار به هم فشرده و با تأکید بسیار بر رضایت مشتری است. **شاید تنها چیزی که در دنیا ثابت است، تغییر باشد.** در بسیاری موارد، زمان رساندن محصول به بازار، مهمترین خواسته مدیریتی است. اگر زمان

[1] Anchor Point Milestones

مقرر برای ارائه به بازار از دست برود، خود پروژه‌ی نرم‌افزاری ممکن است دیگر بی‌معنا شود.

تصور می‌شد مدل‌های فرایند تکاملی سنتی این مشکلات را برطرف سازند و با این وجود، آنها نیز به عنوان طبقه‌ای عمومی از مدل‌های فرآیند تولید نرم‌افزار، نقطه ضعف‌هایی دارند.

به رغم مزایای غیرقابل تردید مدل‌های تکاملی سنتی، دغدغه‌های نیز وجود دارد:

۱- مدل‌های تکاملی سنتی به دلیل قطعی نبودن تعداد چرخه‌های (تکرارهای) لازم برای ساخته شدن محصول، برای برنامه‌ریزی پروژه ایجاد مشکل می‌کند. اکثر تکنیک‌های برآورد و مدیریت پروژه بر پایه چیدمان خطی فعالیت‌ها (یعنی مشخص بودن تعداد گام‌های لازم برای ساخته شدن محصول) استوارند، لذا مدل‌های تکاملی سنتی به خوبی در مدل‌های مدیریتی نمی‌گنجند.

توجه: متدولوژی RUP به دلیل قطعی بودن تعداد چرخه‌های (تکرارهای) لازم برای ساخته شدن محصول (تعداد تکرار برابر چهار گام است) برای برنامه‌ریزی بسیار مناسب است. بنابراین متدولوژی RUP به خوبی در مدل‌های مدیریتی می‌گنجد.

۲- مدل‌های تکاملی سنتی چندان سریع نیستند. زیرا اگر تکامل آنها به سرعت انجام شود، فرآیند دچار بی‌نظمی می‌شود (مثلاً نیازمندی‌ها به خوبی شناسایی نمی‌شوند) و اگر خیلی کند باشد، ممکن است روی بهره‌وری تأثیر منفی بگذارد (مثلاً پروژه در زمان مورد انتظار نرسد).

۳- در مدل‌های تکاملی سنتی، تمرکز بیشتر بر روی انعطاف‌پذیری و قابلیت گسترش‌پذیری می‌باشد تا بر روی کیفیت. زیرا اگر بخواهیم پروژه‌ای با کیفیت بالا را گسترش دهیم لازم است زمان زیادی را صرف کنیم (به خصوص در چرخه‌های اولیه) که این خود ممکن است فرصتی را که در بازار برای این نرم‌افزار وجود دارد از بین ببرد.

توجه: امروزه، چالش پیش روی مهندسان نرم‌افزار، استفاده از مدل فرآیندی است که بتواند بین دو فاکتور مهم سرعت (تولید به موقع) و کیفیت (رضایت مشتری) موازنه برقرار کند.

مدل توسعه‌ی هم‌روند[1]

اغلب سازمان‌های نرم‌افزاری، در یک بازه‌ی زمانی، احتمالاً چندین پروژه را در دست تولید دارند. بسیاری از مهندسان نرم‌افزار معتقدند مدل‌های قبلی نمی‌توانند تصویر دقیقی از وضعیت یک پروژه در اختیار مدیران قرار دهند. به عنوان مثال ممکن است، پنج پروژه از پروژه‌های سازمان در مرحله‌ی ایجاد باشند، اما احتمالاً وضعیت آن‌ها با یکدیگر متفاوت است (مثلاً یکی در مرحله‌ی تولید کد است در حالی که دیگری در حال تست قرار دارد). مدل توسعه‌ی هم‌روند که با نام مهندسی هم‌روند نیز شناخته می‌شود، جهت کنترل اجرای چند پروژه‌ی همزمان مورد استفاده قرار می‌گیرد که هر یک از فعالیت‌های چارچوبی فرآیند تولید نرم‌افزار مربوط به هر پروژه می‌تواند در وضعیت‌های مختلفی قرار داشته باشد، وضعیت‌های مختلف مربوط به هر یک از فعالیت‌های چارچوبی توسط یک گراف نشان داده می‌شود.

[1] Concurrent Development Model

وضعیت‌های مختلف فعالیت‌های چارچوبی

۱- انجام نشده: مانند زمانی که فعالیت ارتباط با مشتری آغاز شده است، اما فعالیت طراحی هنوز شروع نشده است.

۲- در حال توسعه: فعالیت چارچوبی که در حال انجام است، در این وضعیت قرار دارد.

۳- در حال مرور و معیارها: کارهای انجام شده براساس لیست نیازمندی‌ها و استانداردها، شاخص‌ها و معیارهای مشتری مرور و اعتبارسنجی می‌شوند.

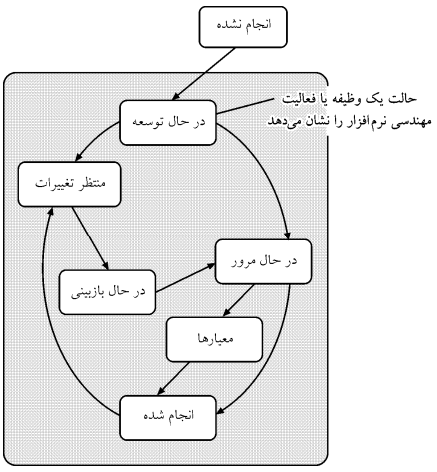
۴- منتظر تغییرات: اگر فعالیتی در وضیعت در حال توسعه یا انجام شده باشد و تغییراتی در روند انجام پروژه ایجاد گردد، فعالیت موجود به وضعیت منتظر تغییرات جهت انجام تغییرات می‌رود.

۵- در حال بازبینی: اگر فعالیتی در وضعیت منتظر تغییرات قرار گرفت، پس از اعمال تغییرات، تغییرات انجام شده در وضعیت بازبینی باید مورد اعتبار سنجی قرار گیرد و در ادامه براساس معیارها مرور گردد.

۶- انجام شده: هر مرحله از کار بعد از اینکه با توجه به معیارها و شاخص‌ها بازبینی شد و مورد تأیید قرار گرفت، به این مرحله وارد می‌شود و تا بروز تغییرات جدید در این مرحله می‌ماند.

همان طور که گفتیم، هر فعالیت چارچوبی از یک مجموعه وضعیت‌ها تشکیل شده است و بسته به اینکه فعالیت در چه حالتی قرار دارد، وضعیت‌های متفاوتی خواهد داشت. در این مدل، برای هر یک از فعالیت‌های چارچوبی (ارتباط، برنامه‌ریزی، مدل‌سازی، ساخت و استقرار) که جهت تکمیل هر پروژه و رسیدن به نتیجه‌ی نهایی انجام می‌گیرد. یک گراف به صورت شبکه‌ای از وضعیت‌های مختلف یک فعالیت چارچوبی، رسم می‌شود، بدین معنی که هر یک از فعالیت‌های چارچوبی موجود در یک پروژه، گراف مخصوص به خود را دارند، به عنوان مثال فعالیت مدل‌سازی، گراف مختص به خود را دارد که وضعیت‌های مختلف آن را نمایش می‌دهد. فعالیت‌های دیگر نیز به همین منوال هستند.

هر کدام از این فعالیت‌ها در هر لحظه می‌تواند در یکی از وضعیت‌های نشان داده شده در شکل مقابل قرار گیرد:



مدل توسعه‌ی هم‌روند

به عنوان مثال وقتی فعالیت ارتباط با مشتری مربوط به یک پروژه (تعیین خواسته‌ها و نیازمندی‌ها)، یک تکرار را به پایان می‌برد و در وضعیت «انجام شده و سپس منتظر تغییرات» قرار می‌گیرد، در ادامه فعالیت تحلیل که هنگام کامل شدن ارتباط اولیه با مشتری در حالت «انجام نشده» قرار داشت به حالت «در حال توسعه» وارد می‌شود. حال اگر مشتری تغییراتی را به اطلاع برساند، وضعیت فعالیت تحلیل به «منتظر تغییرات» تبدیل می‌شود، در واقع از وضعیت «در حال توسعه» به وضعیت «منتظر تغییرات» انتقال می‌یابد. نتیجه اینکه در هر پروژه چندین فعالیت چارچوبی انجام می‌شود که هر کدام در وضعیت‌های متفاوتی قرار دارند. در این مدل، یکسری رویداد تعریف می‌شود که وقوع آنها باعث انتقال از وضعیتی به وضعیت دیگر برای هر یک از فعالیت‌های چارچوبی می‌گردد. برای مثال، وقوع یک ناسازگاری در فعالیت طراحی در طول مراحل اولیه آن که ناشی از وجود اشکال در فعالیت تحلیل بوده است، باعث انتقال فعالیت تحلیل از وضعیت «انجام شده» به حالت «منتظر تغییرات» و سپس در «حال بازبینی» می‌شود.

خصوصیات مدل توسعه‌ی هم‌روند

۱- در دنیای واقعی، مدل توسعه‌ی هم‌روند را در تولید تمامی انواع نرم‌افزارها می‌توان به کار برد و این مدل، تصویری درست از وضعیت جاری یک پروژه را نمایش می‌دهد.

۲- در مدل توسعه‌ی هم‌روند، به جای تعریف ترتیبی برای فعالیت‌های چارچوبی، شبکه‌ای از فعالیت‌های چارچوبی برای هر پروژه خواهیم داشت. به نحوی که رخدادهای یک فعالیت روی وضعیت آن فعالیت و سایر فعالیت‌های دیگر تأثیر می‌گذارد.

مدل‌های فرآیند خاص

مدل‌های فرآیند خاص، بسیاری از خصوصیات مدل‌های متداول ارائه شده در بخش‌های قبل را دارند. اما فرق آنها در این است که در شرایط خاص محیطی قابل استفاده هستند. در واقع شرایط خاصی وجود دارند که استفاده از این مدل‌های فرآیندی خاص می‌تواند بهترین نتیجه را در آنها به همراه داشته باشد.

مدل توسعه‌ی مبتنی بر مؤلفه ساخت‌یافته

مؤلفه یک قطعه‌ی آماده، کاربردی و پیاده‌سازی شده است که دارای واسط لازم جهت اتصال با سایر قطعات و استقرار در بخشی از یک سیستم عملیاتی می‌باشد.

در دیدگاه ساخت‌یافته به مؤلفه، پیمانه نیز گفته می‌شود. در حیطه‌ی مهندسی نرم‌افزار ساخت‌یافته، **واحد مولفه**، یک قطعه عملیاتی مبتنی بر تابع است که بر دو نوع می‌باشد:

۱) تابع کاربردی: مانند تابع جمع که برنامه‌نویس آن را می‌نویسد. اگر تابع کاربردی، شرایط قابل حمل بودن (مانند تعریف متغیر درون تابع به صورت‌محلی و صریح و عدم استفاده از متغیرهای سراسری) را داشته باشد می‌تواند به عنوان یک قطعه‌ی آماده‌ی قابل استفاده‌ی مجدد در پروژه‌های بعدی مورد استفاده مجدد قرار گیرد.

۲) تابع سیستمی: مانند تابع سینوس که کامپایلر تعاریف آن را فراهم می‌کند و می‌تواند به

عنوان یک قطعه‌ی آماده و قابل استفاده‌ی مجدد، در پروژه‌ها مورد استفاده مجدد قرار گیرد.

مدل توسعه‌ی مبتنی بر مؤلفه‌ی ساخت‌یافته از بسیاری از خصوصات مدل پیچشی استفاده می‌کند. این مدل از نظر ماهیت، مدلی تکاملی است و ساختاری تکرارشونده را برای توسعه‌ی نرم‌افزار در پیش می‌گیرد. اما در تولید برنامه‌های کاربردی از مؤلفه‌های آماده استفاده می‌کند، در واقع این مدل برنامه را با استفاده از ترکیب و سرهم کردن قطعات نرم‌افزاری از پیش ساخته شده می‌سازد.

روال استقرار تابع در معماری نرم‌افزار (ساختار برنامه یا اسکلت برنامه) براساس مدل توسعه‌ی مبتنی بر مؤلفه‌ی ساخت‌یافته به صورت زیر است:

۱- شناسایی توابع مورد نیاز برنامه

۲- جستجوی توابع در کتابخانه‌ی توابع سیستمی یا کاتالوگ توابع کاربردی

توجه: توابع کاربردی ایجاد شده در پروژه‌های قبلی در کاتالوگ توابع کاربردی، نگهداری می‌شوند.

۳- در صورت وجود توابع مورد نیاز در کتابخانه‌ی توابع سیستمی یا کاتالوگ توابع کاربردی، تابع استخراج و دوباره استفاده می‌شود. در غیراینصورت تابع مورد نیاز ایجاد می‌گردد.

۴- تابع در معماری نرم‌افزار (اسکلت برنامه) استقرار می‌یابد.

۵- انجام تست برای اطمینان از صحت کار انجام می‌شود.

توجه: دقت کنید که مراحل فوق مبتنی بر تکرار و تکامل صورت می‌گیرد، یعنی پروژه به تدریج و براساس تکرار، تکامل می‌یابد.

توجه: زمان و هزینه‌ی فرآیند تولید نرم‌افزار براساس مدل توسعه‌ی مبتنی بر مؤلفه ساخت‌یافته به دلیل استفاده از قطعات آماده (قابلیت استفاده مجدد) کاهش چشمگیری دارد. در واقع یک قطعه با قابلیت استفاده‌ی مجدد یک بار ساخته می‌شود و بارها و بارها استفاده می‌شود و این یعنی سودآوری!

مدل روش‌های رسمی[1]

مدل روش‌های رسمی (فرمال، قراردادی و صوری) شامل مجموعه‌ای از فعالیت‌ها است که سعی دارد پروژه‌ی نرم‌افزاری را در قالب روابطی رسمی و ریاضی، سیستم‌های کامپیوتری را تعریف، توسعه، پیاده‌سازی و ارزیابی نماید. در این مدل، با استفاده از تحلیل‌های ریاضی، بسیاری از ابهامات، نواقص و عدم سازگاری نرم‌افزار را تا حد زیادی می‌توان به سادگی کشف و تصحیح نمود. گونه‌ای از روش‌های رسمی وجود دارد که به مهندسی نرم‌افزار **اتاق تمیز**[2] معروف است.

توجه: این مدل، امکان کشف و تصحیح خطاهای زیادی را که تا زمان اجرا غیرقابل تشخیص هستند را در طول مراحل اولیه‌ی تولید نرم‌افزار، فراهم می‌کند.

---

[1] Formal Method
[2] Clean Room

توجه: یکی از مهمترین کاربردهای مدل روش‌های رسمی، ساخت سیستم‌های حساس و امن مانند ساخت نرم‌افزارهای کنترل هواپیما و موشک است.

معایب مدل‌های روش‌های رسمی

۱- استفاده از مدل روش‌های رسمی بسیار وقتگیر و گران است.

۲- آموزش گسترده‌ی سازندگان نرم‌افزار به علت اینکه تعداد اندکی از تولیدکنندکان نرم‌افزار پیش‌زمینه‌ی لازم برای به کار بردن روش‌های فرمال (رسمی) را دارند.

۳- از این مدل نمی‌توان برای برقراری ارتباط با مشتریان معمولی که دید فنی ندارند، استفاده نمود.

مدل فرآیند تولید نرم‌افزار مدرن (شیءگرا)

مدل فرآیند تولید نرم‌افزار مدرن، به صورت تکاملی مدرن می‌باشد.

مدل تکاملی مدرن

مدل تکاملی مدرن، مبتنی بر مؤلفه شیءگرا نام دارد و براساس رویکرد تکرار و تکامل می‌باشد.

مدل توسعه‌ی مبتنی بر مؤلفه شیءگرا

مؤلفه یک قطعه‌ی آماده، کاربردی و پیاده‌سازی شده است که دارای واسط لازم جهت اتصال با سایر قطعات و استقرار در بخشی از یک سیستم عملیاتی می‌باشد.

در دیدگاه شیءگرا، به مؤلفه پیمانه نیز گفته می‌شود. در حیطه مهندسی نرم‌افزار شیءگرا، **واحد مؤلفه**، یک قطعه‌ی عملیاتی مبتنی بر کلاس است که بر دو نوع می‌باشد:

۱) کلاس کاربردی: مانند کلاس دانشجو که برنامه‌نویس آن را می‌نویسد و به دلیل داشتن شرایط قابل حمل به شکل ذاتی، می‌تواند به عنوان یک قطعه‌ی آماده و قابل استفاده‌ی مجدد در پروژه‌های بعدی مورد استفاده مجدد قرار گیرد.

۲) کلاس سیستمی: مانند کلاس جعبه‌ی متن که کامپایلر تعاریف آن را فراهم می‌کند و می‌تواند به عنوان یک قطعه‌ی آماده و قابل استفاده‌ی مجدد، در پروژه‌ها مورد استفاده مجدد قرار گیرد.

این مدل از نظر ماهیت، مدلی تکاملی است و ساختاری تکرارشونده را برای توسعه‌ی نرم‌افزار در پیش می‌گیرد. اما در تولید برنامه‌های کاربردی از مؤلفه‌های آماده استفاده می‌کند، در واقع این مدل، برنامه را با استفاده از ترکیب و سرهم‌کردن قطعات نرم‌افزاری از پیش ساخته شده می‌سازد.

روال استقرار کلاس در معماری نرم‌افزار (ساختار برنامه یا اسکلت برنامه) براساس مدل توسعه‌ی مبتنی بر مؤلفه شیءگرا به صورت زیر است:

۱- شناسایی کلاس مورد نیاز برنامه

۲- جستجوی کلاس در کتابخانه‌ی کلاس‌های سیستمی یا کاتالوگ کلاس‌های کاربردی

توجه: کلاس‌های کاربردی ایجاد شده در پروژه‌های قبلی در کاتالوگ کلاس‌های کاربردی، نگهداری می‌شوند.

۳- در صورت وجود کلاس‌های مورد نیاز در کتابخانه‌ی کلاس‌های سیستمی یا کاتالوگ کلاس‌های کاربردی، کلاس استخراج و دوباره استفاده می‌شود. در غیراینصورت کلاس مورد نیاز ایجاد می‌گردد.

۴- کلاس در معماری نرم‌افزار (اسکلت برنامه) استقرار می‌یابد.

۵- انجام تست برای اطمینان از صحت کار انجام می‌شود.

توجه: دقت کنید که مراحل فوق مبتنی بر تکرار و تکامل صورت می‌گیرد، یعنی پروژه به تدریج و براساس تکرار، تکامل می‌یابد.

توجه: زمان و هزینه‌ی فرآیند تولید نرم‌افزار براساس مدل توسعه‌ی مبتنی بر مؤلفه شیءگرا به دلیل استفاده از قطعات آماده (قابلیت استفاده مجدد) کاهش چشمگیری دارد. در واقع یک قطعه با قابلیت استفاده‌ی مجدد یک بار ساخته می‌شود و بارها و بارها استفاده می‌شود و این یعنی سودآوری!

تست‌های فصل دوم

۱- در شرکتی کار می‌کنید که در بازار رقابتی و در حال رشد شبکه‌های نوری فعالیت می‌کند. باید در مدت کوتاه و معینی، نرم‌افزاری برای پیکربندی و تست سخت‌افزار بنویسید (به عنوان مثال، نرم‌افزاری برای راه‌اندازی و پیکربندی مسیریاب‌های نوری در شبکه). نرم‌افزار باید به همراه سخت‌افزار به فروش برسد، در غیر این صورت سخت‌افزار استفاده‌ای نخواهد داشت. متأسفانه سخت‌افزار پیوسته تغییر می‌کند و تا چند هفته پیش از عرضه‌ی محصول به بازار، نهایی نخواهد شد. مدل فرایند مناسب برای توسعه‌ی این نرم‌افزار چیست؟ (مهندسی IT – دولتی ۸۳)

۱) نمونه‌سازی (Prototyping)
۲) افزایشی (Incremental)
۳) آبشاری (Waterfall)
۴) Rapid Application Development

۲- کدام عبارت صحیح است؟ (مهندسی IT – دولتی ۸۴)

۱) بسته به شرایط محیطی که یک نرم‌افزار برای آن توسعه می‌یابد، می‌توان از ترکیبی از متدولوژی‌های توسعه نرم‌افزار استفاده نمود. از آنجایی که متدولوژی به مفهوم ترکیبی از فرآیند و ابزار می‌باشد، می‌توان فرایند یک متدولوژی را با ابزارهای استفاده شده در متدولوژی دیگر تلفیق نموده و مورد استفاده قرار داد.

۲) بسته به شرایط محیطی که یک نرم‌افزار برای آن توسعه می‌یابد، می‌توان از ترکیبی از متدولوژی‌های توسعه نرم‌افزار استفاده نمود. از آنجایی که متدولوژی به مفهوم ترکیبی از فرآیند و ابزار می‌باشد، نمی‌توان فرایند یک متدولوژی را با ابزارهای استفاده شده در متدولوژی دیگر تلفیق نموده و مورد استفاده قرار داد.

۳) بسته به شرایط محیطی که یک نرم‌افزار برای آن توسعه می‌یابد، نمی‌توان از ترکیبی از متدولوژی‌های توسعه نرم‌افزار استفاده نمود. از آنجایی که متدولوژی به مفهوم ترکیبی از فرآیند و ابزار می‌باشد، می‌توان فرایند یک متدولوژی را با ابزارهای استفاده شده در متدولوژی دیگر تلفیق نموده و مورد استفاده قرار داد.

۴) بسته به شرایط محیطی که یک نرم‌افزار برای آن توسعه می‌یابد، نمی‌توان از ترکیبی از متدولوژی‌های توسعه نرم‌افزار استفاده نمود. از آنجایی که متدولوژی به مفهوم ترکیبی از فرآیند و ابزار می‌باشد، نمی‌توان فرآیند یک متدولوژی را با ابزارهای استفاده شده در متدولوژی دیگر تلفیق نموده و مورد استفاده قرار داد.

۳- شما مدیر بخش نرم‌افزار در یک شرکت نرم‌افزاری بزرگ هستید. از شما خواسته می‌شود که برای پروژه‌ی خودکارسازی سیستم کنترل ترافیک یک فرودگاه بزرگ پیشنهاد فنی آماده کنید. سیستم بزرگ و دستی فعلی به شکل معقولی کار می‌کند، اما مشتری اعتقاد دارد که خودکارسازی سیستم باعث صرفه‌جویی در هزینه می‌گردد. مشتری در رابطه با مسائل قراردادی بسیار انعطاف‌پذیر است. سیستمی که ایجاد می‌شود باید بسیار ایمن و قابل اطمینان باشد و شرکت شما باید مسئولیت حقوقی تمامی صدمات ناشی از نامناسب عمل کردن سیستم را به عهده بگیرد. علاوه بر این، در رابطه با امکان‌پذیری

سیستم، مسائل ناشناخته‌ای وجود دارد که در طی مراحل توسعه‌ی سیستم مشخص خواهند گردید. برای توسعه‌ی این سیستم کامپیوتری چه مدل فرآیندی را انتخاب می‌کنید؟ (مهندسی IT – دولتی ۸۴)

۱) مدل پیچشی (Spiral) ۲) مدل آبشاری (Waterfall)

۳) روش‌های صوری (Formal Methods) ۴) مدل Rapid Application Development

---

۴- کدام عبارت صحیح است؟ (مهندسی IT – دولتی ۸۴)

۱) در متدولوژی شیءگرا، همواره از روش برنامه‌نویسی شیءگرا (Object Oriented Programming) استفاده می‌شود. معمولاً در متدولوژی ساخت‌یافته (SSADM) دیدگاه تابع‌گرا در تجزیه و تحلیل سیستم و ایجاد نرم‌افزار حاکم است و می‌توان بسته به شرایط در مرحله برنامه‌نویسی از روش شیءگرا نیز استفاده نمود.

۲) در متدولوژی شیءگرا، می‌توان از روش برنامه‌نویسی شیءگرا (Object Oriented Programming) یا از روش برنامه‌نویسی تابع‌گرا استفاده نمود. معمولاً در متدولوژی ساخت‌یافته (SSADM) دیدگاه تابع‌گرا در تجزیه و تحلیل سیستم و ایجاد نرم‌افزار حاکم است و می‌توان بسته به شرایط در مرحله برنامه‌نویسی از روش شیءگرا نیز استفاده نمود.

۳) در متدولوژی شیءگرا، می‌توان از روش برنامه‌نویسی شیءگرا (Object Oriented Programming) یا از روش برنامه‌نویسی تابع‌گرا استفاده نمود. معمولاً در متدولوژی ساخت‌یافته (SSADM) دیدگاه تابع‌گرا در تجزیه و تحلیل سیستم و ایجاد نرم‌افزار حاکم است و در مرحله برنامه‌نویسی همواره از روش تابع‌گرا استفاده می‌شود.

۴) در متدولوژی شیءگرا، همواره از روش برنامه‌نویسی شیءگرا (Object Oriented Programming) استفاده می‌شود. معمولاً در متدولوژی ساخت‌یافته (SSADM) دیدگاه تابع‌گرا در تجزیه و تحلیل سیستم و ایجاد نرم‌افزار حاکم است و در مرحله برنامه‌نویسی همواره از روش تابع‌گرا استفاده می‌شود.

۵- کدام‌یک از موارد زیر صحیح است؟ (مهندسی IT – دولتی ۸۴)

۱) متدولوژی‌های تابع‌گرا (Function Oriented) همواره بهتر از بقیه متدولوژی‌های توسعه نرم‌افزار بوده است.

۲) متدولوژی‌های شیءگرا (Object Oriented) همواره از بقیه متدولوژی‌های توسعه نرم‌افزار بهتر می‌باشد.

۳) متدولوژی‌های شیءگرا (Object Oriented) و متدولوژی‌های تابع‌گرا (Function Oriented) هر دو زیرمجموعه‌ای از متدولوژی‌های نسل دوم یا ساخت‌یافته (Structured) می‌باشند.

۴) هیچ‌کدام

۶- مدیریت نیازها (requirements management) عبارت است از: (مهندسی IT – دولتی ۸۳)

۱) مجموعه‌ای از فعالیت‌ها برای مدیریت مرحله تجزیه و تحلیل نیازها

۲) مجموعه‌ای از فعالیت‌ها برای به دست آوردن نیازها در مرحله تجزیه و تحلیل نیازها
۳) مجموعه‌ای از فعالیت‌ها برای دنبال نمودن (track) نیازها و تغییرات آنها در طی انجام پروژه
۴) مجموعه‌ای از فعالیت‌ها برای به دست آوردن نیازها و دنبال نمودن (track) نیازها و تغییرات آنها در طی انجام پروژه

۷- کدام‌یک از گزینه‌های زیر مربوط به مشخصه‌ی نیازمندی‌های غیرعملیاتی نیست؟ (مهندسی IT – آزاد ۸۴)

۱) سرویس کاربر ۲) نمایش داده‌ها ۳) زمان پاسخگویی ۴) حافظه موردنیاز

۸- به چه منظور از مدل آبشاری جهت توسعه‌ی نرم‌افزار استفاده می‌شود؟ (مهندسی IT – آزاد ۸۴)

۱) جهت پایین آمدن هزینه‌ی نگهداری سیستم
۲) به منظورساده کردن مدیریت فرآیند نرم‌افزار
۳) جهت جلوگیری از تکرار فرآیندها
۴) چون می‌تواند مدل‌های دیگر را نیز دربرگیرد.

۹- کدام‌یک از مراحل زیر در مراحل تولید و به‌کارگیری نرم‌افزار به ترتیب دارای کمترین و بیشترین هزینه می‌باشند؟ (مهندسی IT – آزاد ۸۴)

۱) تجزیه و تحلیل، پیاده‌سازی ۲) امکان‌پذیری، نگهداری
۳) طراحی، تجزیه و تحلیل ۴) تعریف، نگهداری

۱۰- کدام‌یک از موارد زیر صحیح است؟ (مهندسی IT – دولتی ۸۵)

۱) نتایج متدولوژی SSADM با نتایج حاصل از متدولوژی شیءگرا کاملاً با یکدیگر مشابه و متناظر است.
۲) متدولوژی ساخت‌یافته (SSADM) و شیءگرا (O.O) کاملاً از یکدیگر مستقل بوده و نتایج غیرمشترکی را دارند.
۳) از روی نتایج حاصل از متدولوژی SSADM می‌توان برخی از خروجی‌های متدولوژی شیءگرا (O.O) را به دست آورد و بالعکس.
۴) متدولوژی شیءگرا (O.O) فقط برای طراحی و متدولوژی SSADM فقط برای تحلیل نرم‌افزار استفاده می‌شود.

۱۱- تفاوت متدولوژی آبشاری با متدولوژی ساخت‌یافته SSADM چیست؟ (مهندسی IT – دولتی ۸۵)

۱) انجام هر مرحله از متدولوژی SSADM در مدت زمان محدودی بوده و نتایج هر مرحله غیرقابل تغییر و ورودی مرحله بعد می‌باشد.
۲) در متدولوژی SSADM مدت زمان انجام مراحل محدود و معین بوده ولی نتایج هر مرحله قابل تغییر می‌باشد.
۳) انجام متدولوژی آبشاری نیازمند دانستن وضع موجود در یک سازمان یا سیستم می‌باشد.
۴) اساساً متدولوژی آبشاری و SSADM با هم یکسان بوده و فرقی ندارند.

۱۲- منظور از نرم‌افزارهای CASE چیست؟ (مهندسی IT – دولتی ۸۵)

۱) نرم‌افزارهایی است که انجام برخی یا تمامی مراحل موردنظر در یک متدولوژی توسعه نرم‌افزار را پشتیبانی و تسهیل نماید.

۲) نرم‌افزارهایی است که باید تمامی مراحل یک متدولوژی نرم‌افزاری یا مجموعه‌ای از متدولوژی را انجام دهد.

۳) نرم‌افزارهایی را شامل می‌شود که می‌توان به کمک آنها نمودارهای موردنظر در یک یا چند متدولوژی توسعه نرم‌افزار را ترسیم نمود.

۴) نرم‌افزارهایی است که فقط عمل تبدیل طرح یک سیستم نرم‌افزاری را به شکل کد و نرم‌افزار نهایی انجام می‌دهد.

۱۳- کدام‌یک از عبارات زیر، بیان بهتری برای مفهوم توسعه نرم‌افزار (RAD) می‌باشد؟ (مهندسی IT – دولتی ۸۵)

۱) اجرای سریع متدولوژی SSADM را RAD گویند.

۲) روشی است که در دوره‌های کوتاه زمانی، نرم‌افزار قسمت‌های مختلف یک سازمان (Enterprise) را ایجاد کرده و سپس آنها را با هم تلفیق می‌کند.

۳) مجموعه فرآیندهای سازمانی اولویت‌بندی شده و متدولوژی توسعه نرم‌افزار مشخصی در دوره‌های کوتاه برای تهیه نرم‌افزار هر فرآیند استفاده می‌شود.

۴) هر دو گزینه ۲ و ۳ با هم بیان مناسبی از RAD است.

۱۴- در کدام‌یک از مدل‌های فرآیند نرم‌افزار بر کوتاه نمودن چرخه توسعه نرم‌افزار (تولید سریع) تأکید می‌گردد؟ (مهندسی IT – آزاد ۸۵)

۱) مدل Incremental
۲) مدل مارپیچی
۳) مدل مارپیچی Win Win
۴) مدل RAD

۱۵- کدام‌یک از گزینه‌های زیر جزء لایه‌های مهندسی نرم‌افزار نمی‌باشد؟ (مهندسی IT – آزاد ۸۶)

۱) فرآیند (Process)
۲) محصول (Product)
۳) روش‌ها (Methods)
۴) ابزار (Tools)

۱۶- کدام عبارت در مورد مدل‌ها (روش‌ها)ی توسعه نرم‌افزار صحیح است؟ (مهندسی IT – دولتی ۸۷)

۱) مدل‌های الگوسازی و آبشاری را می‌توان در مدل حلزونی جمع کرد.

۲) مدل‌های آبشاری و حلزونی را می‌توان در مدل الگوسازی جمع کرد.

۳) مدل‌های الگوسازی و حلزونی را می‌توان در مدل آبشاری جمع کرد.

۴) هر سه مورد صحیح است.

۱۷- فرض نمایید به شما پیشنهاد ایجاد یک نرم‌افزار ارایه می‌گردد. نرم‌افزار مذکور به عناصری وابسته می‌باشد که پیوسته در حال تغییر بوده و در فاصله کوتاهی قبل از عرضه محصول، نهایی می‌گردند. مدل فرآیند مناسب برای توسعه‌ی این نرم‌افزار کدام است؟ (مهندسی IT – آزاد ۸۷)

۱) مدل افزایشی (Incremental)
۲) مدل ترتیبی خطی (Liner)
۳) مدل نمونه‌سازی (Prototyping)
۴) مدل Rapid Application Development

۱۸- کدام مورد به عنوان لایه‌های مهندسی نرم‌افزار صحیح‌تر می‌باشد؟ (مهندسی IT – دولتی ۸۸)

۱) مدل فرآیند – ابزار – متدولوژی – کیفیت
۲) مدل فرآیند – متدولوژی – ابزار – کیفیت
۳) کیفیت – مدل فرآیند – متدولوژی – ابزار
۴) کیفیت – مدل فرآیند – روش‌ها – ابزار

۱۹- مهندسی نرم‌افزار اتاق تمیز (Clean Room)، به کدام دسته از مدل‌های فرآیند نرم‌افزار تعلق دارد؟ (مهندسی IT – آزاد ۸۸)

۱) مدل بسط هم زمان
۲) مدل مارپیچی Win-Win
۳) مدل توسعه بر مبنای مؤلفه
۴) مدل روش‌های رسمی

۲۰- کدام‌یک از مدل‌های فرآیند توسعه‌ی نرم‌افزار زیر تأکید بر قابلیت استفاده مجدد محصولات تولید شده‌ی نرم‌افزاری ندارد؟ (مهندسی IT – دولتی ۸۹)

۱) افزایشی (Incremental)
۲) Rapid Application Development
۳) Component Base Development
۴) نمونه‌سازی (Prototype)

۲۱- کدام‌یک از مدل‌های فرآیند زیر بر اصول ریاضی در توسعه نرم‌افزار تأکید می‌نماید؟ (مهندسی IT – آزاد ۸۹)

۱) مدل گام به گام (Incremental Model)
۲) مدل روش‌های رسمی
۳) مدل مارپیچی
۴) مدل بسط همزمان

۲۲- نیازهای غیروظیفه‌مندی (Non functional) در کدام‌یک از مراحل زیر مورد رسیدگی قرار می گیرد؟ (مهندسی IT – آزاد ۸۹)

۱) تحلیل
۲) پیاده‌سازی
۳) طراحی
۴) آزمون

۲۳- تیمی از مهندسین نرم‌افزار با تجربه سیستم جدیدی را در دست تهیه دارند. اگرچه سیستم جدید نسبتاً بزرگ است، اما انتظار نمی‌رود که با سیستم‌هایی که قبلاً توسط این تیم تهیه شده است تفاوت‌های فاحشی داشته باشد. کدام‌یک از مدل‌های چرخه تولید نرم‌افزار (SDLC) زیر برای این پروژه مناسب‌تر است؟ (مهندسی IT – دولتی ۹۰)

۱) آبشاری
۲) حلزونی (مارپیچی)
۳) نمونه‌سازی
۴) تکاملی

۲۴- دو پروژه زیر را در نظر بگیرید:
الف) مشتری تأکید بر انجام سریع پروژه حداکثر در مدت سه ماه می‌نماید. تغییرات در درخواست مشتری اندک بوده و امکان تقسیم پروژه به قسمت‌های مستقل وجود دارد.

ب) پروژه نیازمند عواملی می‌باشد که در طول انجام پروژه امکان دارد تغییر نماید و تا چند روز به عرضه نهایی محصول قطعی نمی‌گردد.
مدل‌های فرآیند مناسب جهت پروژه‌های الف و ب به ترتیب از راست به چپ عبارتند از:
(مهندسی IT – آزاد ۹۰)

۱) مدل RAD و مدل Spiral
۲) مدل RAD و مدل Linear
۳) مدل Spiral و مدل RAD
۴) مدل Spiral و مدل Spiral

۲۵- کدام‌یک از ویژگی‌های زیر جزء خصایص مدل روش‌های رسمی نمی‌باشد؟ **(مهندسی IT – آزاد ۹۰)**

۱) از آنجا که به صورت کلیشه‌ای می‌باشد، هزینه‌ی زمانی اندکی دارد.
۲) با اعمال یک نظم ریاضی شدید سیستم کامپیوتری را توسعه می‌دهد.
۳) در نرم‌افزارهای بحرانی – امنیتی (مثلاً نرم‌افزارهای مرتبط با دستگاه‌های پزشکی و هوافضا) کاربردهای فراوانی دارد.
۴) استفاده از این مدل به عنوان راهکار ارتباطی با مشتریانی که دید فنی ندارند، دشوار است.

۲۶- کدام‌یک از روش‌های زیر، روشی برای استخراج خواسته‌ها نیست؟ **(مهندسی IT – دولتی ۹۱)**

۱) تحلیل ریسک ۲) مصاحبه ۳) مشاهده ۴) ساختن نمونه اولیه

۲۷- کدام‌یک از روش‌های زیر برای پروژه‌ای که از تکنولوژی mobile برای اطلاع‌رسانی در مورد خاصی استفاده می‌کند مناسب است، در صورتی که ما با این تکنولوژی آشنایی نداشته باشیم؟
(مهندسی IT – دولتی ۹۳)

۱) RAD Model
۲) Sequential Model
۳) Rapid Prototyping
۴) Incremental Model

۲۸- کدام‌یک از روش‌های زیر قادر هستند به راحتی ابهامات، ناسازگاری‌ها و نواقص یک سیستم را حین تولید مشخص نمایند؟ **(مهندسی IT – دولتی ۹۳)**

۱) Formal Method
۲) Object Oriented Analysis, RUP
۳) Component Based Development
۴) Concurrent Development Model

۲۹- در کدام یک از شیوه‌های زیر، مهندسی نیازمندی‌ها، مکانیسم و ابزار مناسب را فراهم می‌سازد؟
(مهندسی IT – دولتی ۹۴)

۱) ابهام در مشخصات راه‌حل
۲) اعتبارسنجی مشخصات
۳) تحلیل نیازها
۴) هر سه موارد بالا

۳۰- کدام یک از عناصر زیر متعلق به جمع‌آوری نیازها (requirements elicitation) است؟
(مهندسی IT – دولتی ۹۴)

۱) تحلیل نیازها
۲) ارزیابی خطر
۳) مشاهده و بازدید
۴) پیاده‌سازی سیستم

۳۱- کدام یک از موارد زیر اگر وجود داشته باشد از روش تولید سریع برنامه کاربردی (Rapid Application Development) نباید استفاده کرد؟ **(مهندسی IT – دولتی ۹۵)**

۱) ریسک فنی بالا ۲) واحدمند (Modular) بودن سیستم
۳) منابع انسانی کافی برای پروژه‌های بزرگ ۴) تعامل کامل کاربر و تولیدکننده سیستم

۳۲- کدام مدل فرآیندی، تکاملی و ریسک-رانه (Risk-Driven) محسوب می‌شود؟
(مهندسی IT – دولتی ۹۷)

۱) مدل V ۲) مدل مارپیچی (Spiral)
۳) مدل آبشاری (Waterfall) ۴) مدل افزایشی (Incremental)

۳۳- در دو نمونه زیر، معادله درجه دوم و مثال بیمارستان، به ترتیب نوع نیازمندی کدام است؟
الف- $x = c - b + sqrt(b^{۲} - ۴ \times a \times c) / ۲ \times a$
ب- در یک بیمارستان، کاربر باید قادر باشد تا در بانک، اطلاعات کاملی از بیماران را جستجو کند یا حتی زیر مجموعه‌ای از آن را انتخاب کند. **(مهندسی IT – دولتی ۹۸)**

۱) دامنه‌ای – عملکردی ۲) عملکردی - عملکردی
۳) دامنه‌ای – غیرعملکردی ۴) غیرعملکردی - عملکردی

پاسخ تست‌های فصل دوم

۱- گزینه (۲) صحیح است.

مدل‌های RAD و Sequential (ترتیبی) یا Linear (خطی) یا Waterfall (آبشاری) جزو مدل‌های غیرتکاملی و مدل افزایشی (Incremental Model) جزو مدل‌های تکاملی سنتی فرآیند تولید نرم‌افزار هستند.

Prototyping (نمونه‌سازی دورانداختنی) یا Rapid Prototyping (نمونه‌سازی سریع) یک مکانیزم، صرفاً جهت شناسایی نیازمندی‌های مشتری است، در یک بیان کلی‌تر نمونه‌سازی دورانداختنی، گفتگو، مشاهده و مصاحبه روش‌هایی جهت شناسایی نیازمندی‌های مشتری و تهیه لیست نیازمندی‌ها می‌باشند. توجه کنید که نمونه‌سازی دورانداختنی یک مدل فرآیند تولید نرم‌افزار به شمار نمی‌آید، بلکه یک مکانیزم صرفاً جهت شناسایی نیازمندی‌های مشتری می‌باشد. به حاصل جمع مکانیزم نمونه‌سازی دورانداختنی و مدل آبشاری «مدل نمونه‌سازی» گفته می‌شود که این مدل نیز یک مدل غیرتکاملی به حساب می‌آید. در واقع مدل نمونه‌سازی از ترکیب مکانیزم نمونه‌سازی دورانداختنی و مدل آبشاری ایجاد شده است. به این نحو که در ابتدای کار توسط مکانیزم نمونه سازی دورانداختنی، تمامی لیست نیازمندی‌های مشتری تکمیل می‌گردد، سپس توسط مدل آبشاری به شکل خطی نرم‌افزار ایجاد می‌گردد و کار تمام می‌شود. در واقع نیازمندهای مشتری طی تکرارهای مکانیزم نمونه‌سازی دورانداختنی شناخته می‌شوند. سپس طی یک روال خطی، توسط مدل آبشاری، نرم‌افزار تولید می‌گردد و کار تمام می‌شود و دیگر تکامل نمی‌یابد. زیرا مدل نمونه سازی یک مدل غیرتکاملی است و پس از ساخت نهایی امکان ادامه‌ی پروژه و تغییر وجود ندارد.

پروژه‌ای که نرم‌افزار آن باید در کنار سخت افزار مرتبط با آن به فروش برسد و سخت افزار آن پیوسته تغییر می‌کند و دستخوش تغییر است، باعث می‌شود تا نرم‌افزار آن نیز پیوسته تغییر کند و دستخوش تغییر باشد.

در چنین شرایطی که نیازها، نرم‌افزار و سخت افزار به طور پیوسته در حال تغییر و تحول هستند، مناسب‌ترین مدل فرآیند تولید نرم‌افزار، مدل‌های تکاملی سنتی (افزایشی و پیچشی)، مدل تکاملی سنتی خاص یعنی مبتنی بر مؤلفه ساخت‌یافته (مبتنی بر تابع) و مدل تکاملی مدرن یعنی مبتنی بر مؤلفه‌ی شیءگرا (مبتنی بر کلاس) هستند. زیرا در اثر مرور زمان و با تغییرات سخت‌افزاری می‌توان نرم‌افزار را نیز در هر بار تکرار (Iteration) تغییر داد.

از آنجا که در صورت سوال، از وجود قطعات و مولفه‌های آماده و قابل استفاده مجدد صحبت نشده است، مدل‌های مبتنی بر مولفه ساخت‌یافته و مبتنی بر مولفه شیءگرا را کنار می‌گذاریم، همچنین از آنجا که شرایط امن بودن پروژه در صورت سوال مطرح نشده است، مدل پیچشی را نیز کنار می‌گذاریم، همچنین مدل‌های غیرتکاملی RAD و آبشاری (Waterfall) را نیز با توجه به شرایط پروژه کنار می‌گذاریم.

بنابراین با توجه به گزینه‌ها پرواضح است که مدل افزایشی پاسخ درست خواهد بود.

مدل افزایشی، مراحل مدل آبشاری را با رویکرد تکرار و تکامل مکانیزم نمونه‌سازی تکاملی

ترکیب نموده است. در این مدل، با توجه به خاصیت نمونه‌سازی تکاملی، پروژه به تدریج کامل می‌شود یعنی هر مرحله‌ای که می‌گذرد، پروژه کامل‌تر شده و در افزایش‌های بعدی این تکامل ادامه می‌یابد تا به محصول نهایی برسد و این تکامل نیز ادامه خواهد داشت.

دقت کنید که می‌توان از مکانیزم نمونه‌سازی دورانداختنی در ابتدای هر تکرار (افزایش) در مدل افزایشی به جهت کاهش ریسک مربوط به شناسایی دقیق لیست نیازمندی‌های مشتری استفاده نمود.

به طور کلی نمونه‌سازی دورانداختنی، به عنوان راه‌کاری برای شناسایی لیست نیازمندی‌های مشتری می‌باشد و در اغلب مدل‌ها می‌توان آن را گنجاند به غیر از مدل آبشاری و مدل RAD .

از آنجا که مدل RAD یک مدل غیرتکاملی است، پرواضح است که مدل مناسبی در این پروژه نخواهد بود، اما به غیر از این مطلب شرط لازم برای استفاده از مدل RAD تقسیم‌پذیر بودن پروژه است که از این مسأله هم صحبتی نشده است.

۲- گزینه (۱) صحیح است.

می‌توان مدل تحلیل و طراحی را به روش و ابزارهای ساخت‌یافته انجام داد ولی فعالیت پیاده‌سازی را توسط یک زبان برنامه‌نویسی شیءگرا انجام داد و از امکانات شیءگرایی زبان استفاده نکرد اما عکس این مطلب امکان‌پذیر نیست، یعنی نمی‌توان مدل تحلیل و طراحی را به روش شیءگرا انجام داد ولی فعالیت پیاده‌سازی را توسط یک زبان ساخت‌یافته انجام داد، زیرا زبان ساخت‌یافته امکانات شیءگرایی (همچون کلاس، وراثت و چندریختی) را پشتیبانی نمی‌کند.

۳- گزینه (۱) صحیح است.

مدل‌های RAD و Sequential (ترتیبی) یا Linear (خطی) یا Waterfall (آبشاری) جزو مدل‌های غیرتکاملی، روش‌های صوری (Formal Methods) جزو مدل‌های غیرتکاملی مطمئن و مبتنی بر روابط ریاضی و مدل پیچشی (Spiral Model) جزو مدل‌های تکاملی سنتی فرآیند تولید نرم‌افزار هستند.

پروژه‌ای که در ابتدای کار، لیست نیازمندی‌های مشخصی ندارد (مسائل ناشناخته وجود دارد)، در روند ساخت، مدام دچار تغییر و تحول می‌شود. از آنجا که پروژه مربوط به کنترل ترافیک یک فرودگاه بزرگ است، بنابراین پس از ساخت پروژه هم به دلیل تغییر قوانین و نیازهای پروژه امکان تغییر و تحول وجود دارد.

در چنین شرایطی که نیازها در ابتدای کار مشخص نیستند و پس از تحویل محصول باز هم امکان تغییر و تحول در محصول وجود دارد، مناسب‌ترین مدل فرآیند تولید نرم‌افزار، مدل‌های تکاملی سنتی (افزایشی و پیچشی)، مدل تکاملی سنتی خاص یعنی مبتنی بر مؤلفه ساخت‌یافته (مبتنی بر تابع) و مدل تکاملی مدرن یعنی مبتنی بر مؤلفه‌ی شیءگرا (مبتنی بر کلاس) هستند.

از آنجا که در صورت سوال، از وجود قطعات و مولفه‌های آماده و قابل استفاده مجدد صحبت نشده است، مدل‌های مبتنی بر مولفه ساخت‌یافته و مبتنی بر مولفه شی گرا را کنار می‌گذاریم، همچنین مدل روش‌های صوری را نیز به دلیل وجود مسائل ناشناخته در پروژه و غیرتکاملی بودن

ماهیت این مدل، کنار می‌گذاریم، همچنین مدل افزایشی را به دلیل اهمیت ایمنی مطرح شده در پروژه و عدم پشتیبانی ایمنی توسط مدل افزایشی، کنار می‌گذاریم، همچنین مدل‌های غیرتکاملی RAD و آبشاری (Waterfall) را نیز با توجه به شرایط پروژه کنار می‌گذاریم.

بنابراین با توجه به گزینه‌ها و اهمیت ایمنی و مدیریت کلیه ریسک‌های پروژه بر اساس شرایط پروژه، پرواضح است که مدل پیچشی (Spiral Model) پاسخ درست خواهد بود.

مدل پیچشی، مراحل مدل آبشاری را با رویکرد تکرار و تکامل مکانیزم نمونه‌سازی تکاملی ترکیب نموده است. در این مدل، با توجه به خاصیت نمونه‌سازی تکاملی، پروژه به تدریج کامل می‌شود یعنی هر مرحله‌ای که می‌گذرد، پروژه کامل‌تر شده و در پیچش‌های بعدی این تکامل ادامه می‌یابد تا به محصول نهایی برسد و این تکامل نیز ادامه خواهد داشت.

توجه کنید که می‌توان از مکانیزم نمونه‌سازی دورانداختنی در ابتدای هر تکرار (پیچش) در مدل پیچشی به جهت کاهش ریسک مربوط به شناسایی دقیق لیست نیازمندی‌های مشتری استفاده نمود.

به طور کلی نمونه‌سازی دورانداختنی، به عنوان راه‌کاری برای شناسایی لیست نیازمندی‌های مشتری می‌باشد و در اغلب مدل‌ها می‌توان آن را گنجاند به غیر از مدل آبشاری و مدل RAD .

مدل پیچشی از مکانیزم نمونه‌سازی دورانداختنی به عنوان راهکاری برای کاهش ریسک مربوط به شناسایی نیازمندی‌ها استفاده می‌کند و همچنین به دلیل تحلیل ریسک به واسطه‌ی مدیریت ریسک تمامی ریسک‌های مربوط به کلیت پروژه (شناسایی نیازمندی‌های دقیق مشتری، مقرون به صرفه بودن و در زمان مورد انتظار بودن) را کنترل می‌کند. در بیان ساده، مدل پیچشی به واسطه‌ی مدیریت ریسک (تحلیل ریسک) بستری امن، برای تولید نرم‌افزار ایده آل مشتری فراهم می‌سازد.

از آنجا که مدل RAD یک مدل غیرتکاملی است، پرواضح است که مدل مناسبی در این پروژه نخواهد بود، اما به غیر از این مطلب شرط لازم برای استفاده از مدل RAD تقسیم‌پذیر بودن پروژه است که از این مسأله هم صحبتی نشده است.

۴- گزینه (۱) صحیح است.

می‌توان مدل تحلیل و طراحی را به روش و ابزارهای ساخت‌یافته انجام داد ولی فعالیت پیاده‌سازی را توسط یک زبان برنامه‌نویسی شیءگرا انجام داد و از امکانات شیءگرایی زبان استفاده نکرد، اما عکس این مطلب امکان‌پذیر نیست، یعنی نمی‌توان مدل تحلیل و طراحی را به روش شیءگرا انجام داد ولی فعالیت پیاده‌سازی را توسط یک زبان ساخت‌یافته انجام داد زیرا زبان ساخت‌یافته امکانات شیءگرایی (همچون کلاس، وراثت و چندریختی) را پشتیبانی نمی‌کند.

متدولوژی SSADM متداول‌ترین نمونه از متدولوژی ساخت‌یافته براساس روش ساخت‌یافته، مدل فرآیند تولید آبشاری و ابزارهای ساخت‌یافته می‌باشد. و بر دو نوع داده‌گرا (جدولی) مانند نرم‌افزار حقوق و دستمزد که داده‌ها درون جداول ذخیره می‌شوند و تابع‌گرا (متغیری) مانند نرم‌افزار ماشین‌حساب که داده‌ها درون متغیرها ذخیره می‌شوند، می‌باشد.

بنابراین گزینه اول درست و گزینه‌های دوم، سوم و چهارم نادرست هستند.

۵- گزینه (۳) صحیح است.

اساس به وجود آمدن متدولوژی شیءگرا به وجود آمدن نیازهای جدید بوده که توسط متدولوژی ساخت‌یافته (تابع‌گرا یا داده‌گرا) قابل پوشش دادن نبوده‌اند. متدولوژی شیءگرا برخی از نیازمندی‌های جدید را پوشش داده است و این طور نبوده است که استفاده از متدولوژی ساخت‌یافته را منسوخ کند. متدولوژی شیءگرا و متدلوژی ساخت‌یافته هر دو زیرمجموعه‌ای از متدولوژی‌های نسل دوم یا ساختارمند می‌باشند. منظور از کلمه «ساخت‌یافته» در گزینه چهارم، ساختارمند بودن متدولوژی است و نه مفهوم متدولوژی ساخت‌یافته. بنابراین گزینه سوم درست است.

۶- گزینه (۴) صحیح است.

مهم‌ترین کار در فعالیت ارتباطات، مدیریت شناسایی نیازمندی‌ها و پیگیری تغییرات نیازمندی‌های مشتری است.

۷- گزینه (۱) صحیح است.

سرویس کاربر مربوط به نیازمندی‌های عملیاتی و نمایش داده‌ها، زمان پاسخگویی و حافظه مورد نیاز مربوط به نیازمندی‌های غیرعملیاتی است.

۸- گزینه (۲) صحیح است.

در صورتی که کلیه نیازمندی‌های مشتری در ابتدای پروژه مشخص باشد، مدل آبشاری، به عنوان مدلی کارآمد و ساده جهت مدیریت ساده فرآیند تولید نرم‌افزار مورد استفاده قرار می‌گیرد. مدل آبشاری تکرار و بازگشت به عقب ندارد و مدل‌های دیگر را در بر نمی‌گیرد، یعنی از اجتماع مدل‌های دیگر ایجاد نشده است.

۹- گزینه (۲) صحیح است.

نگهداری نرم‌افزار بیشترین هزینه و امکان‌سنجی کمترین هزینه را دارا است.

۱۰- گزینه (۳) صحیح است.

فعالیت ارتباطات و بخش مدل‌سازی لیست نیازمندی‌ها در مدل تحلیل، در متدولوژی ساخت‌یافته و متدولوژی شیءگرا یکسان است. بنابراین در این دو مرحله از هریک از متدولوژی‌های ساخت‌یافته و شیءگرا می‌توان به دیگری تغییر مسیر داد. اما پس از عبور از این دو مرحله دیگر تحت هیچ شرایطی نمی‌توان از متدولوژی شیءگرا به متدولوژی ساخت‌یافته تغییر مسیر داد. اما در متدولوژی ساخت‌یافته می‌توان مدل تحلیل و طراحی را به روش و ابزارهای ساخت‌یافته انجام داد ولی فعالیت پیاده‌سازی را توسط یک زبان برنامه‌نویسی شیءگرا انجام داد و از امکانات شیءگرایی زبان استفاده نکرد، اما عکس این مطلب امکان‌پذیر نیست، یعنی نمی‌توان مدل تحلیل و طراحی را به روش شیءگرا انجام داد ولی فعالیت پیاده‌سازی را توسط یک زبان ساخت‌یافته انجام داد زیرا زبان ساخت‌یافته امکانات شیءگرایی (همچون کلاس، وراثت و

چندریختی) را پشتیبانی نمی‌کند. بنابراین گزینه سوم درست است و گزینه‌های اول و دوم نادرست هستند.

متدولوژی ساخت‌یافته و شیءگرا هر یک به تنهایی کلیه فعالیت‌های چارچوبی مربوط به فرآیند تولید نرم‌افزار (ارتباطات، برنامه‌ریزی، مدل‌سازی (تحلیل و طراحی)، ساخت (پیاده‌سازی و تست و استقرار) را پشتیبانی می‌کنند. بنابراین گزینه چهارم نیز نادرست است.

۱۱- گزینه (۱) صحیح است.

اشکالی که به این سوال وارد است این است که مدل فرآیند تولید نرم‌افزار آبشاری یک متدولوژی در نظر گرفته شده است. بنابراین صورت سوال باید به صورت زیر اصلاح گردد:

«تفاوت مدل آبشاری با متدولوژی ساخت‌یافته SSADM چیست؟»

متدولوژی مجموعه‌ای از فرآیندها، روش‌ها و ابزارهای مرتبط با هم و همه از یک متدولوژی خاص همچون ساخت‌یافته یا شیءگرا برای ایجاد یک محصول نرم‌افزاری، مطابق با استانداردهای مهندسی نرم‌افزار می‌باشد و بر دو طبقه‌ی ساخت‌یافته و شیءگرا است.

متدولوژی ساخت‌یافته یا مهندسی نرم‌افزار ساخت‌یافته نظامی است یکپارچه شامل مدل فرآیندهای ساخت‌یافته (سنتی)، روش ساخت‌یافته و ابزارهای ساخت‌یافته که منجر به ایجاد نرم‌افزاری در بازه‌ی زمانی از قبل برنامه‌ریزی شده، بودجه‌ای از قبل پیش‌بینی شده و دقیقاً مطابق با نیازمندی‌های واقعی مشتری می‌گردد.

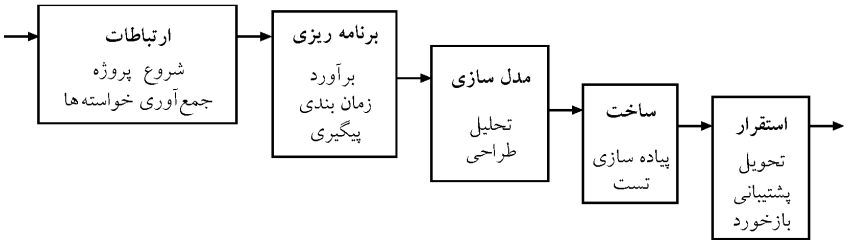
متدولوژی SSADM متداول‌ترین نمونه از متدولوژی ساخت‌یافته براساس روش ساخت‌یافته، مدل فرآیند تولید آبشاری و ابزارهای ساخت‌یافته می‌باشد.

نسبت نمونه متدولوژی ساخت‌یافته SSADM به متدولوژی ساخت‌یافته، مثل نسبت سیستم عامل ویندوز سنتی به مفاهیم سنتی سیستم عامل است.

مدل آبشاری، که گاه از آن به عنوان چرخه‌ی حیات کلاسیک یاد می‌شود، روشی ترتیبی برای تولید نرم‌افزار پیشنهاد می‌کند. این مدل با فعالیت ارتباطات شروع می‌شود و با فعالیت‌های برنامه‌ریزی، مدل‌سازی و ساخت پیش می‌رود و با فعالیت استقرار پایان می‌یابد، تحویل پروژه انجام می‌گیرد و کار تمام می‌شود. و دیگر فرصت و تکرار دیگری در کار نخواهد بود تا خواسته‌های فراموش شده یا جدید به پروژه اضافه گردد.

خصوصیت اصلی این مدل این است که هیچ‌گونه بازخوردی بین مراحل این مدل وجود ندارد. مانند آب که نمی‌تواند در آبشار به عقب برگردد، در این مدل نیز بعد از ورود به یک فعالیت به فعالیت‌های قبلی نمی‌توان بازگشت. این مدل زمانی کاربرد دارد که کلیه‌ی نیازمندی‌های مشتری در همان ابتدای پروژه مشخص، ثابت و بدون تغییر باشد.

مدل آبشاری در شرایطی که خواسته‌ها به طور کامل و جامع مشخص، ثابت و پایدار است و قرار است که کار تا پایان به شیوه‌ای خطی پیش برود، می‌تواند به عنوان مدلی مفید، مورد استفاده قرار گیرد.



مدل آبشاری

متدولوژی SSADM، یک نمونه متدولوژی ساخت‌یافته یا یک نمونه مهندسی‌نرم‌افزار ساخت‌یافته است. بنابراین از آنجا که تعریف متدولوژی ساخت‌یافته یا مهندسی‌نرم‌افزار ساخت‌یافته، براساس روش ساخت‌یافته، مدل فرآیند تولید آبشاری و ابزارهای ساخت‌یافته برای آن برقرار است، لذا SSADM منجر به ایجاد نرم‌افزاری در بازه‌ی زمانی از قبل برنامه‌ریزی شده، بودجه‌ای از قبل پیش‌بینی شده و دقیقاً مطابق با نیازمندی‌های واقعی مشتری می‌گردد.

از آنجاکه SSADM بسته‌ای کامل از فرآیند، روش و ابزار است، بنابراین می‌تواند در فضایی مناسب خصوصیات SSADM، برآورده‌سازی نیازمندی‌های مشتری، محدودیت زمانی (مدت زمان محدود) و مقرون به صرفه بودن را گارانتی کند. همچنین از آنجاکه SSADM جهت هدایت فرآیند تولید نرم‌افزار، مدل فرآیند آبشاری را مورد استفاده قرار می‌دهد، لذا مطابق خصوصیات مدل آبشاری و به تبع در SSADM نتایج هر مرحله غیر قابل تغییر و ورودی مرحله بعد می‌باشد.

همچنین از آنجاکه مدل آبشاری به تنهایی بسته‌ای کامل از فرآیند، روش و ابزار نیست و صرفا به تنهایی فقط یک مدل فرایند تولید نرم‌افزار است، بنابراین نمی‌تواند مطابق تعریف متدولوژی و مهندسی نرم‌افزار، برآورده‌سازی نیازمندی‌های مشتری، محدودیت زمانی(مدت زمان محدود) و مقرون به صرفه بودن را گارانتی کند. که همین مورد اخیر تفاوت متدولوژی SSADM و مدل آبشاری است. بنابراین گزینه اول درست و گزینه دوم نادرست است.

همچنین از آنجاکه SSADM جهت هدایت فرآیند تولید نرم‌افزار، مدل فرآیند آبشاری را مورد استفاده قرار می‌دهد، بنابراین، هم SSADM و هم مدل آبشاری، مطابق خصوصیات مدل آبشاری نیازمند دانستن وضع موجود در یک سازمان یا سیستم می‌باشند، زیرا مدل آبشاری زمانی کاربرد دارد که کلیه‌ی نیازمندی‌های مشتری در همان ابتدای پروژه مشخص، ثابت و بدون تغییر باشد. که این شباهت است و نه تفاوت SSADM و مدل آبشاری. بنابراین گزینه سوم نیز نادرست است.

SSADM یک نمونه متدولوژی ساخت‌یافته و مدل آبشاری یک مدل فرآیند تولید نرم‌افزار است. بنابراین شاید در برخی موارد مشابه باشند، اما در کل متفاوت هستند، پس گزینه چهارم نیز نادرست است.

۱۲- گزینه (۱) صحیح است.

ابزارهای CASE فقط نمودار ترسیم نمی‌نمایند بلکه می‌توانند کلیه مراحل فرآیند تولید نرم‌افزار (ارتباطات، برنامه‌ریزی، مدل‌سازی (تحلیل و طراحی)، ساخت (پیاده‌سازی و تست) و استقرار) را

انجام دهند که حتمی و اجباری نمی‌باشد.

۱۳- گزینه (۴) صحیح است.

مدل RAD، شکل پُرسرعت مدل آبشاری می‌باشد، با این تفاوت که پروژه به بخش‌های مختلف تقسیم شده و هر بخش، توسط یک تیم، مطابق مدل آبشاری ایجاد می‌گردد و در پایان نتیجه‌ی تیم‌ها، برای خلق محصول نهایی ترکیب می‌گردد. مدل RAD، سرعت خود را مدیون بهره‌گیری از تکنیک بخش‌بندی و موازی‌سازی بخش‌های مختلف پروژه است. چنانچه نیازمندی‌ها به خوبی شناسایی شده و دامنه پروژه کوچک باشد این مدل قادر است یک سیستم کاملاً عملیاتی را در مدت زمان بسیار کوتاه (مثلاً بین ۶۰ تا ۹۰ روز) تولید نماید. در این مدل نرم‌افزار به قسمت‌های مختلف تقسیم شده و همواره سعی می‌شود که نرم‌افزار موردنظر سریع‌تر تولید شود. نکته قابل توجه این است که نرم‌افزار موردنظر باید خاصیت تفکیک‌پذیری داشته باشد تا بتوان این مدل را پیاده‌سازی کرد.

SSADM یک نمونه متدولوژی ساخت‌یافته است، RAD یک مدل فرآیند تولید نرم‌افزار است.

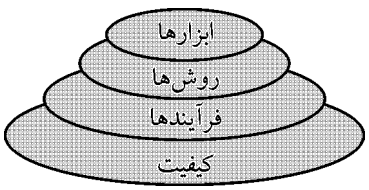
منظور از عبارت «مجموعه فرآیندهای سازمانی اولویت بندی شده» در گزینه سوم، امکان سنجی بخش‌های (فرآیندهای) مختلف محیط عملیاتی کسب و کار سنتی برای مکانیزه‌کردن بخش مورد نظر است.

۱۴- گزینه (۴) صحیح است.

مدل RAD، شکل پُرسرعت مدل آبشاری می‌باشد، با این تفاوت که پروژه به بخش‌های مختلف تقسیم شده و هر بخش، توسط یک تیم، مطابق مدل آبشاری ایجاد می‌گردد و در پایان نتیجه‌ی تیم‌ها، برای خلق محصول نهایی ترکیب می‌گردد. مدل RAD، سرعت خود را مدیون بهره‌گیری از تکنیک بخش‌بندی و موازی‌سازی بخش‌های مختلف پروژه است. چنانچه نیازمندی‌ها به خوبی شناسایی شده و دامنه پروژه کوچک باشد این مدل قادر است یک سیستم کاملاً عملیاتی را در مدت زمان بسیار کوتاه (مثلاً بین ۶۰ تا ۹۰ روز) تولید نماید. در این مدل نرم‌افزار به قسمت‌های مختلف تقسیم شده و همواره سعی می‌شود که نرم‌افزار موردنظر سریع‌تر تولید شود. نکته قابل توجه این است که نرم‌افزار موردنظر باید خاصیت تفکیک‌پذیری داشته باشد تا بتوان این مدل را پیاده‌سازی کرد.

۱۵- گزینه (۲) صحیح است.

مهندسی نرم‌افزار یک فرآیند لایه‌ای است. به بیان دیگر مهندسی نرم‌افزار از چهار لایه تشکیل شده است. در شکل مقابل، چهار لایه نشان داده شده است:



۱۶- گزینه (۱) صحیح است.

مدل پیچشی نیز همانند مدل افزایشی، مراحل مدل آبشاری را با رویکرد تکرار و تکامل

مکانیزم نمونه‌سازی تکاملی ترکیب نموده است. بنابراین مدل پیچشی مبتنی بر مدل آبشاری و مکانیزم نمونه‌سازی تکاملی است. نمونه‌سازی تکاملی به فرآیند تولید نرم‌افزار روح، حرکت و تکرار می‌دهد و مدل آبشاری فعالیت‌های چارچوبی هر تکرار (پیچش) را مشخص می‌کند.

به مکانیزم‌نمونه‌سازی تکاملی، **الگوسازی باز** نیز گفته می‌شود.

همچنین، هرگاه نیاز به شناسایی خواسته‌های مبهم مشتری وجود داشت می‌توان از مکانیزم نمونه‌سازی دورانداختنی استفاده نمود. بنابراین قبل از هر تکرار (پیچش) جهت شناسایی نیازمندی‌ها می‌توان از مکانیزم نمونه‌سازی دورانداختنی استفاده نمود و نمونه را دور انداخت. اما حرکت، تکرار و تکامل همچنان می‌تواند توسط نمونه‌سازی تکاملی ادامه یابد تا محصولی نهایی آماده گردد.

این مدل از یک سری فعالیت‌های چارچوبی تکراری تشکیل شده است که هر تکرار (پیچش) شبیه به مدل آبشاری است. با این تفاوت که روی قسمتی از نرم‌افزار انجام می‌شود. هر کدام از این قسمت‌ها یک «قطعه» قابل تحویل را ایجاد می‌کند.

به مکانیزم نمونه‌سازی دورانداختنی، **الگوسازی بسته** نیز گفته می‌شود.

۱۷- گزینه (۱) صحیح است.

مدل‌های RAD و Sequential (ترتیبی) یا Linear (خطی) یا Waterfall (آبشاری) جزو مدل‌های غیرتکاملی و مدل افزایشی (Incremental Model) جزو مدل‌های تکاملی سنتی فرآیند تولید نرم‌افزار هستند.

Prototyping (نمونه‌سازی دورانداختنی) یا Rapid Prototyping (نمونه‌سازی سریع) یک مکانیزم، صرفاً جهت شناسایی نیازمندی‌های مشتری است، در یک بیان کلی‌تر نمونه‌سازی دورانداختنی، گفتگو، مشاهده و مصاحبه روش‌هایی جهت شناسایی نیازمندی‌های مشتری و تهیه لیست نیازمندی‌ها می‌باشند. توجه کنید که نمونه‌سازی دورانداختنی یک مدل فرآیند تولید نرم‌افزار به شمار نمی‌آید، بلکه یک مکانیزم صرفاً جهت شناسایی نیازمندی‌های مشتری می‌باشد. به حاصل جمع مکانیزم، نمونه‌سازی دورانداختنی و مدل آبشاری «مدل نمونه‌سازی» گفته می‌شود که این مدل نیز یک مدل غیرتکاملی به حساب می‌آید. در واقع مدل نمونه‌سازی از ترکیب مکانیزم نمونه‌سازی دورانداختنی و مدل آبشاری ایجاد شده است. به این نحو که در ابتدای کار توسط مکانیزم نمونه‌سازی دورانداختنی، تمامی لیست نیازمندی‌های مشتری تکمیل می‌گردد، سپس توسط مدل آبشاری به شکل خطی نرم‌افزار ایجاد می‌گردد و کار تمام می‌شود. در واقع نیازمندهای مشتری طی تکرارهای مکانیزم نمونه‌سازی دورانداختنی شناخته می‌شوند. سپس طی یک روال خطی، توسط مدل آبشاری، نرم‌افزار تولید می‌گردد و کار تمام می‌شود و دیگر تکامل نمی‌یابد. زیرا مدل نمونه‌سازی یک مدل غیرتکاملی است و پس از ساخت نهایی امکان ادامه‌ی پروژه و تغییر وجود ندارد.

پروژه‌ای که نرم‌افزار آن به عناصری وابسته است که پیوسته در حال تغییر بوده و در فاصله کوتاهی قبل از عرضه محصول، باید نهایی گردد، در روند ساخت، مدام دچار تغییر و تحول

می‌شود.

در چنین شرایطی که نیازها، نرم‌افزار و عناصر مرتبط با نرم‌افزار به طور پیوسته در حال تغییر و تحول هستند، مناسب‌ترین مدل فرآیند تولید نرم‌افزار، مدل‌های تکاملی سنتی (افزایشی و پیچشی)، مدل تکاملی سنتی خاص یعنی مبتنی بر مؤلفه ساخت‌یافته (مبتنی بر تابع) و مدل تکاملی مدرن یعنی مبتنی بر مؤلفه‌ی شیءگرا (مبتنی بر کلاس) هستند. زیرا در اثر مرور زمان و با تغییرات سخت‌افزاری می‌توان نرم‌افزار را نیز در هر بار تکرار (Iteration) تغییر داد.

از آنجا که در صورت سوال، از وجود قطعات و مولفه‌های آماده و قابل استفاده مجدد صحبت نشده است، مدل‌های مبتنی بر مولفه ساخت‌یافته و مبتنی بر مولفه شی گرا را کنار می‌گذاریم، همچنین از آنجا که شرایط امن بودن پروژه در صورت سوال مطرح نشده است، مدل پیچشی را نیز کنار می‌گذاریم، همچنین مدل‌های غیرتکاملی RAD و آبشاری (Waterfall) را نیز با توجه به شرایط پروژه کنار می‌گذاریم.

بنابراین با توجه به گزینه‌ها پرواضح است که مدل افزایشی پاسخ درست خواهد بود.

مدل افزایشی، مراحل مدل آبشاری را با رویکرد تکرار و تکامل مکانیزم نمونه‌سازی تکاملی ترکیب نموده است. در این مدل، با توجه به خاصیت نمونه‌سازی تکاملی، پروژه به تدریج کامل می‌شود یعنی هر مرحله‌ای که می‌گذرد، پروژه کامل‌تر شده و در افزایش‌های بعدی این تکامل ادامه می‌یابد تا به محصول نهایی برسد و این تکامل نیز ادامه خواهد داشت.

توجه کنید که می‌توان از مکانیزم نمونه‌سازی دورانداختنی در ابتدای هر تکرار (افزایش) در مدل افزایشی به جهت کاهش ریسک مربوط به شناسایی دقیق لیست نیازمندی‌های مشتری استفاده نمود.

به طور کلی نمونه‌سازی دورانداختنی، به عنوان راه‌کاری برای شناسایی لیست نیازمندی‌های مشتری می‌باشد و در اغلب مدل‌ها می‌توان آن را گنجاند به غیر از مدل آبشاری و مدل RAD .

از آنجا که مدل RAD یک مدل غیرتکاملی است، پرواضح است که مدل مناسبی در این پروژه نخواهد بود، اما به غیر از این مطلب شرط لازم برای استفاده از مدل RAD تقسیم‌پذیر بودن پروژه است که از این مسأله هم صحبتی نشده است.

۱۸- گزینه (۴) صحیح است.

مهندسی نرم‌افزار یک فرآیند لایه‌ای است. به بیان دیگر مهندسی نرم‌افزار از چهار لایه تشکیل شده است. در شکل مقابل، چهار لایه نشان داده شده است:

۱۹- گزینه (۴) صحیح است.

مدل روش‌های رسمی (فرمال، قراردادی و صوری) شامل مجموعه‌ای از فعالیت‌ها است که سعی دارد پروژه‌ی نرم‌افزاری را در قالب روابطی رسمی و ریاضی، سیستم‌های کامپیوتری را

تعریف، توسعه، پیاده‌سازی و ارزیابی نماید. در این مدل، با استفاده از تحلیل‌های ریاضی، بسیاری از ابهامات، نواقص و عدم سازگاری نرم‌افزار را تا حد زیادی می‌توان به سادگی کشف و تصحیح نمود. گونه‌ای از روش‌های رسمی وجود دارد که به مهندسی نرم‌افزار **اتاق تمیز** معروف است.

۲۰- گزینه (۲) صحیح است.

مدل‌های تکاملی، محصولات تولید شده در تکرارهای قبلی را ارتقاء داده و از قابلیت‌های آن‌ها در تکرارهای بعدی استفاده می‌کنند. اما در مدل‌های غیرتکاملی محصول نهایی یکجا و در انتهای کار ایجاد می‌گردد.

مدل RAD جزو مدل‌های غیرتکاملی، مدل افزایشی جزو مدل‌های تکاملی سنتی، مدل مبتنی بر مولفه ساخت‌یافته (مبتنی بر تابع) جزو مدل تکاملی سنتی خاص و مدل مبتنی بر مولفه شیءگرا (مبتنی بر کلاس) جزو مدل تکاملی مدرن فرآیند تولید نرم‌افزار است.

مدل افزایشی، مراحل مدل آبشاری را با رویکرد تکرار و تکامل مکانیزم نمونه‌سازی تکاملی ترکیب نموده است. بنابراین مدل افزایشی مبتنی بر مدل آبشاری و مکانیزم نمونه‌سازی تکاملی است.

نمونه‌سازی تکاملی به فرآیند تولید نرم‌افزار روح، حرکت و تکرار می‌دهد و مدل آبشاری فعالیت‌های چارچوبی هر تکرار (افزایش) را مشخص می‌کند. همچنین، هرگاه نیاز به شناسایی خواسته‌های مبهم مشتری وجود داشت می‌توان از مکانیزم نمونه‌سازی دورانداختنی استفاده نمود. بنابراین قبل از هر تکرار (افزایش) جهت شناسایی نیازمندی‌ها می‌توان از مکانیزم نمونه‌سازی دورانداختنی نیز استفاده نمود و نمونه را دور انداخت. اما حرکت، تکرار و تکامل همچنان می‌تواند توسط نمونه‌سازی تکاملی ادامه یابد تا محصول نهایی آماده گردد.

این مدل از یک سری فعالیت‌های چارچوبی تکراری تشکیل شده است که هر تکرار (افزایش) شبیه به مدل آبشاری است. با این تفاوت که روی قسمتی از نرم‌افزار انجام می‌شود. هر کدام از این قسمت‌ها یک «قطعه» قابل تحویل را ایجاد می‌کند.

در این مدل، با توجه به خاصیت نمونه‌سازی تکاملی، پروژه به تدریج کامل می‌شود یعنی هر مرحله‌ای که می‌گذرد، پروژه کامل‌تر شده و در افزایش‌های بعدی این تکامل ادامه می‌یابد تا به محصول نهایی برسد.

هنگامی که از یک مدل افزایشی استفاده می‌شود، افزایش نخست غالباً هسته‌ی اصلی محصول است، یعنی نیازهای اولیه رفع می‌شوند، اما بسیاری از ویژگی‌های مکمل (که برخی معلوم و برخی نامعلوم هستند) تحویل داده نمی‌شوند. هسته‌ی اصلی محصول، توسط مشتری مورد استفاده و ارزیابی قرار می‌گیرد. سپس در افزایش بعدی، قابلیت‌ها و ویژگی‌های دیگر مورد انتظار مشتری به هسته‌ی اصلی محصول اضافه می‌گردد. این فرآیند به دنبال تحویل هر قطعه تکرار می‌شود تا اینکه محصول کامل تولید شود. بنابراین مدل افزایشی دیدی پیوسته به توسعه و تکامل نرم‌افزار دارد و در نسخه افزایشی خود نرم‌افزار کامل‌تری را ارائه می‌کند. در واقع ورودی هر مرحله از آن نسخه‌ای می‌باشد که در افزایش قبلی تولید شده است، پس این مدل تاکید بر قابلیت استفاده مجدد

از محصولات تولید شده قبلی را دارد.

مدل RAD، شکل پُرسرعت مدل آبشاری می‌باشد، با این تفاوت که پروژه به بخش‌های مختلف تقسیم شده و هر بخش، توسط یک تیم، مطابق مدل آبشاری ایجاد می‌گردد و در پایان نتیجه‌ی تیم‌ها، برای خلق محصول نهایی ترکیب می‌گردد. مدل RAD، سرعت خود را مدیون بهره‌گیری از تکنیک بخش‌بندی و موازی‌سازی بخش‌های مختلف پروژه است. چنانچه نیازمندی‌ها به خوبی شناسایی شده و دامنه پروژه کوچک باشد این مدل قادر است یک سیستم کاملاً عملیاتی را در مدت زمان بسیار کوتاه (مثلاً بین ۶۰ تا ۹۰ روز) تولید نماید. در این مدل نرم‌افزار به قسمت‌های مختلف تقسیم شده و همواره سعی می‌شود که نرم‌افزار موردنظر سریع‌تر تولید شود. نکته قابل توجه این است که نرم‌افزار موردنظر باید خاصیت تفکیک‌پذیری داشته باشد تا بتوان این مدل را پیاده‌سازی کرد. مدل RAD یک مدل غیر تکاملی است، پس این مدل تاکید بر قابلیت استفاده مجدد از محصولات تولید شده قبلی را ندارد. زیرا مدل‌های غیرتکاملی، محصول نهایی را یکجا و در انتهای کار ایجاد می‌کنند. تولید تکاملی یعنی نرم‌افزار قطعه قطعه، ذره ذره و کم‌کم، کم‌کم تکامل می‌یابد و نه در قالب یک قطعه و یکجا.

مؤلفه یک قطعه‌ی آماده، کاربردی و پیاده‌سازی شده است که دارای واسط لازم جهت اتصال با سایر قطعات و استقرار در بخشی از یک سیستم عملیاتی می‌باشد.

در دیدگاه ساخت‌یافته به مؤلفه پیمانه نیز گفته می‌شود. در حیطه‌ی مهندسی نرم‌افزار ساخت‌یافته، مولفه یک قطعه عملیاتی مبتنی بر تابع است که بر دو نوع می‌باشد:

۱) تابع کاربردی: مانند تابع جمع که برنامه‌نویس آن را می‌نویسد. اگر تابع کاربردی، شرایط قابل حمل بودن (مانند تعریف متغیر درون تابع به صورت‌محلی و صریح و عدم استفاده از متغیرهای سراسری) را داشته باشد می‌تواند به عنوان یک قطعه‌ی آماده‌ی قابل استفاده‌ی مجدد در پروژه‌های بعدی مورد استفاده مجدد قرار گیرد.

۲) تابع سیستمی: مانند تابع سینوس که کامپایلر تعاریف آن را فراهم می‌کند و می‌تواند به عنوان یک قطعه‌ی آماده و قابل استفاده‌ی مجدد، در پروژه‌ها مورد استفاده مجدد قرار گیرد.

مدل توسعه‌ی مبتنی بر مؤلفه‌ی ساخت‌یافته از بسیاری از خصوصات مدل‌های تکاملی سنتی استفاده می‌کند. این مدل از نظر ماهیت، مدلی تکاملی است و ساختاری تکرارشونده را برای توسعه‌ی نرم‌افزار در پیش می‌گیرد. اما در تولید برنامه‌های کاربردی از مؤلفه‌های آماده استفاده می‌کند، در واقع این مدل برنامه را با استفاده از ترکیب و سرهم کردن قطعات نرم‌افزاری از پیش ساخته شده می‌سازد.

زمان و هزینه‌ی فرآیند تولید نرم‌افزار براساس مدل توسعه‌ی مبتنی بر مؤلفه ساخت‌یافته به دلیل استفاده از قطعات آماده (قابلیت استفاده مجدد) کاهش چشمگیری دارد. در واقع یک قطعه با قابلیت استفاده‌ی مجدد یک بار ساخته می‌شود و بارها و بارها استفاده می‌شود و این یعنی سودآوری!

در دیدگاه شیءگرا به مؤلفه پیمانه نیز گفته می‌شود. در حیطه مهندسی نرم‌افزار شیءگرا، **واحد مؤلفه** یک قطعه‌ی عملیاتی مبتنی بر کلاس است که بر دو نوع می‌باشد:

۱) کلاس کاربردی: مانند کلاس دانشجو که برنامه‌نویس آن را می‌نویسد و به دلیل داشتن شرایط قابل حمل به شکل ذاتی، می‌تواند به عنوان یک قطعه‌ی آماده و قابل استفاده‌ی مجدد در پروژه‌های بعدی مورد استفاده مجدد قرار گیرد.

۲) کلاس سیستمی: مانند کلاس جعبه‌ی متن که کامپایلر تعاریف آن را فراهم می‌کند و می‌تواند به عنوان یک قطعه‌ی آماده و قابل استفاده‌ی مجدد، در پروژه‌ها مورد استفاده مجدد قرار گیرد.

این مدل از نظر ماهیت، مدلی تکاملی است و ساختاری تکرارشونده را برای توسعه‌ی نرم‌افزار در پیش می‌گیرد. اما در تولید برنامه‌های کاربردی از مؤلفه‌های آماده استفاده می‌کند، در واقع این مدل برنامه را با استفاده از ترکیب و سرهم کردن قطعات نرم‌افزاری از پیش ساخته شده می‌سازد.

زمان و هزینه‌ی فرآیند تولید نرم‌افزار براساس مدل توسعه‌ی مبتنی بر مؤلفه شیءگرا به دلیل استفاده از قطعات آماده (قابلیت استفاده مجدد) کاهش چشمگیری دارد. در واقع یک قطعه با قابلیت استفاده‌ی مجدد یک بار ساخته می‌شود و بارها و بارها استفاده می‌شود و این یعنی سودآوری!

بنابراین مستقل از ساخت‌یافته یا شیءگرا بودن مدل مبتنی بر مولفه، مولفه‌ها، قطعات و محصولات آماده شده در پروژه‌های قبلی در پروژه‌های آتی مورد استفاده قرار می‌گیرد، پس این مدل تاکید بر قابلیت استفاده مجدد از محصولات تولید شده قبلی را دارد.

نمونه‌سازی بر دو شکل نمونه‌سازی دور انداختنی و نمونه‌سازی تکاملی وجود دارد.

علاوه بر روش‌های مشاهده، مصاحبه و گفتگو، مکانیزم نمونه‌سازی دورانداختنی نیز برای شناسایی نیازمندی‌های مشتری در فعالیت ارتباطات مورد استفاده قرار می‌گیرد. در مکانیزم نمونه‌سازی دورانداختنی، یک پیاده‌سازی عملیاتی از سیستم با ابزارهای ارزان‌قیمت، فقط و فقط به منظور شناسایی نیازمندی‌های مشتری، ایجاد و سپس دور انداخته می‌شود، سپس سیستم نهایی براساس فرآیند تولید مجزایی تولید می‌گردد، توجه کنید که سیستم نهایی براساس فرآیند تولید مجزای دیگری تولید می‌گردد، مثلاً پس از شناسایی تمامی نیازمندی‌های مشتری توسط مکانیزم نمونه‌سازی دورانداختنی، لیست تمامی نیازمندی‌های مشتری آماده است، بنابراین در ادامه برای تولید نرم‌افزار از مدل آبشاری می‌توان استفاده نمود.

هدف از مکانیزم دورانداختنی، استخراج نیازمندی‌های مشتری است. شروع مکانیزم نمونه‌سازی دورانداختنی براساس نیازمندی‌هایی است که کمتر درک شده‌اند. یکی از مزایای این روش، کاهش ریسک نیازمندی‌ها است. مکانیزم نمونه‌سازی دورانداختنی صرفا یک ابزار جهت تهیه لیست نیازمندی‌های مشتری است و نمونه‌های ساخته شده دورانداخته می‌شوند، پس این مکانیزم تاکید بر قابلیت استفاده مجدد از محصولات تولید شده قبلی را ندارد.

به مکانیزم‌نمونه‌سازی دورانداختنی، **الگوسازی بسته** نیز گفته می‌شود.

اما در نمونه‌سازی تکاملی، یک نمونه‌ی اولیه تولید می‌شود. در ادامه با اعمال اصلاحات بر روی نمونه‌ی اولیه، طی چند مرحله سیستم نهایی تولید می‌گردد. هدف از نمونه‌سازی تکاملی، تحویل یک سیستم عملیاتی به کاربران نهایی و شروع فرآیند تولید براساس نیازمندی‌هایی است که

بهتر و بیشتر درک شده‌اند. کاربرد آن در مواردی است که نیازمندی‌های مشتری به طور کامل در ابتدای کار مشخص نباشد و یا نیاز به توسعه‌ی مبتنی بر تکرار و تکامل داشته باشیم. از مزایای نمونه‌سازی تکاملی می‌توان به درگیرکردن مشتری با فرآیند تولید سیستم که منجر به شناسایی بهتر نیازمندی‌ها می‌گردد، اشاره نمود. مکانیزم نمونه‌سازی دورانداختنی در تولید تکاملی نرم‌افزار مورد استفاده قرار می‌گیرد و نمونه‌های ساخته شده دورانداخته نمی‌شوند، در واقع محصولات اولیه، محصولاتی اولیه از محصولات نهایی هستند، اما قابلیت ارائه خدمات به کاربر را دارند. پس این مکانیزم تاکید بر قابلیت استفاده مجدد از محصولات تولید شده قبلی را دارد. بنابراین گزینه چهارم نیز می‌تواند به واسطه وجود مکانیزم نمونه‌سازی تکاملی، بر قابلیت استفاده مجدد محصولات تولید شده قبلی تاکید کند.

به مکانیزم نمونه‌سازی تکاملی، **الگوسازی باز** نیز گفته می‌شود.

۲۱- گزینه (۲) صحیح است.

مدل روش‌های رسمی (فرمال، قراردادی و صوری) شامل مجموعه‌ای از فعالیت‌ها است که سعی دارد پروژه‌ی نرم‌افزاری را در قالب روابطی رسمی و ریاضی، سیستم‌های کامپیوتری را تعریف، توسعه، پیاده‌سازی و ارزیابی نماید. در این مدل، با استفاده از تحلیل‌های ریاضی، بسیاری از ابهامات، نواقص و عدم سازگاری نرم‌افزار را تا حد زیادی می‌توان به سادگی کشف و تصحیح نمود. گونه‌ای از روش‌های رسمی وجود دارد که به مهندسی نرم‌افزار **اتاق تمیز** (Clean Room) معروف است.

۲۲- گزینه (۱) صحیح است.

به طور کلی نیازمندی‌های وظیفه‌مندی و غیروظیفه‌مندی نرم‌افزار در فعالیت ارتباطات به شکل نوشتاری و لیست نیازمندی‌ها و در فعالیت مدل تحلیل به شکل نموداری و مدل‌سازی شده مورد بررسی و رسیدگی اولیه قرار می‌گیرند.

محصول نرم‌افزاری که نیازمندی‌های وظیفه‌مندی را برآورده می‌کند، ولی برآورنده نیازهای غیروظیفه‌مندی و کیفی نباشد، معمولاً با نارضایتی مشتریان همراه می‌شود.

در فعالیت تست، نیازمندی‌های وظیفه‌مندی و غیروظیفه‌مندی جهت اطمینان از صحت عملکرد آنها مورد ارزیابی مجدد قرار می‌گیرند.

۲۳- گزینه (۱) صحیح است.

مدل‌های Sequential (ترتیبی) یا Linear (خطی) یا Waterfall (آبشاری) جزو مدل‌های غیرتکاملی و مدل Spiral (پیچشی یا حلزونی) جزو مدل‌های تکاملی سنتی فرآیند تولید نرم‌افزار هستند.

Prototyping (نمونه‌سازی دورانداختنی) یا Rapid Prototyping (نمونه‌سازی سریع) یک مکانیزم، صرفاً جهت شناسایی نیازمندی‌های مشتری است، در یک بیان کلی‌تر نمونه‌سازی دورانداختنی، گفتگو، مشاهده و مصاحبه روش‌هایی جهت شناسایی نیازمندی‌های مشتری و تهیه لیست نیازمندی‌ها می‌باشند. توجه کنید که نمونه‌سازی دورانداختنی یک مدل فرآیند تولید نرم‌افزار

به شمار نمی‌آید، بلکه یک مکانیزم صرفاً جهت شناسایی نیازمندی‌های مشتری می‌باشد. به حاصل جمع مکانیزم، نمونه‌سازی دورانداختنی و مدل آبشاری «مدل نمونه‌سازی» گفته می‌شود که این مدل نیز یک مدل غیرتکاملی به حساب می‌آید. در واقع مدل نمونه‌سازی از ترکیب مکانیزم نمونه سازی دورانداختنی و مدل آبشاری ایجاد شده است. به این نحو که در ابتدای کار توسط مکانیزم نمونه‌سازی دورانداختنی، تمامی لیست نیازمندی‌های مشتری تکمیل می‌گردد، سپس توسط مدل آبشاری به شکل خطی نرم‌افزار ایجاد می‌گردد و کار تمام می‌شود. در واقع نیازمندهای مشتری طی تکرارهای مکانیزم نمونه‌سازی دورانداختنی شناخته می‌شوند. سپس طی یک روال خطی، توسط مدل آبشاری، نرم‌افزار تولید می‌گردد و کار تمام می‌شود و دیگر تکامل نمی‌یابد. زیرا مدل نمونه‌سازی یک مدل غیرتکاملی است و پس از ساخت نهایی امکان ادامه‌ی پروژه و تغییر وجود ندارد.

از آنجا که سیستم جدید، در ابعاد کوچکتر قبلاً ایجاد شده است و تفاوت‌های زیادی با سیستم سابق ندارد، لذا مسائل ناشناخته زیادی وجود ندارد. بنابراین مدل آبشاری برای توسعه سیستم جدید مناسب است. گزینه‌های دیگر برحسب کاربردشان زمانی مورد استفاده قرار می‌گیرند که نیازمندی‌های ناشناخته‌ای موجود باشد. بنابراین گزینه اول درست و گزینه‌های دوم، سوم و چهارم نادرست هستند.

۲۴- گزینه (۱) صحیح است.

از آنجا که در مورد «الف» مشتری تأکید بر انجام سریع پروژه حداکثر در مدت سه ماه می‌نماید و همچنین تغییرات در درخواست مشتری اندک بوده و امکان تقسیم پروژه به قسمت‌های مستقل وجود دارد. بنابراین در چنین شرایطی مدل RAD مناسب است.

از آنجا که در مورد «ب» پروژه دستخوش تغییر در نیازمندی‌ها می‌باشد بنابراین یکی از مدل‌های تکاملی فرآیند نرم‌افزار همچون مدل افزایشی (Incremental) یا حلزونی (Spiral) می‌تواند برای این پروژه مناسب باشد. در مدل‌های تکاملی برای ایجاد سیستم‌های بسیار ایمن و قابل اطمینان، مدل حلزونی (پیچشی) می‌تواند بهترین انتخاب باشد زیرا این مدل در هر تکرار از توسعه نرم‌افزار، فعالیت کامل تحلیل ریسک را انجام می‌دهد. ضمن این که برای مورد «ب» بین مدل‌های افزایشی و حلزونی مدل افزایشی مناسب‌تر بود زیرا بحثی از امنیت مطرح نشده است.

۲۵- گزینه (۱) صحیح است.

مدل روش‌های رسمی (فرمال، قراردادی و صوری) شامل مجموعه‌ای از فعالیت‌ها است که سعی دارد پروژه‌ی نرم‌افزاری را در قالب روابطی رسمی و ریاضی، سیستم‌های کامپیوتری را تعریف، توسعه، پیاده‌سازی و ارزیابی نماید. در این مدل، با استفاده از تحلیل‌های ریاضی، بسیاری از ابهامات، نواقص و عدم سازگاری نرم‌افزار را تا حد زیادی می‌توان به سادگی کشف و تصحیح نمود. گونه‌ای از روش‌های رسمی وجود دارد که به مهندسی نرم‌افزار **اتاق تمیز** معروف است.

این مدل، امکان کشف و تصحیح خطاهای زیادی را که تا زمان اجرا غیرقابل تشخیص هستند را در طول مراحل اولیه‌ی تولید نرم‌افزار، فراهم می‌کند.

یکی از مهمترین کاربردهای مدل روش‌های رسمی، ساخت سیستم‌های حساس و امن مانند ساخت نرم‌افزارهای کنترل هواپیما و موشک است.

استفاده از مدل روش‌های رسمی بسیار وقتگیر و گران است.

آموزش گسترده‌ی سازندگان نرم‌افزار به علت اینکه تعداد اندکی از تولیدکنندگان نرم‌افزار پیش زمینه‌ی لازم برای به کاربردن روش‌های فرمال (رسمی) را دارند.

از این مدل نمی‌توان برای برقراری ارتباط با مشتریان معمولی که دید فنی ندارند، استفاده نمود.

۲۶- گزینه (۱) صحیح است.

در فعالیت ارتباط لیست نیازمندی‌های مشتری از طریق ارتباط با مشتری و ابزارهایی همچون گفتگو، مشاهده، پرسش‌نامه، نمونه‌سازی دورانداختنی و نمونه‌سازی تکاملی، جمع‌آوری و آماده می‌گردد.

تحلیل ریسک در مدل پیچشی (حلزونی) به فرآیند تولید نرم‌افزار اضافه شده‌است. گام اول مدیریت ریسک، شناسایی ریسک (تحلیل ریسک) است و گام دوم مقابله با ریسک، برای مقابله با ریسک باید هزینه کرد. برای‌مثال ریسک از دست دادن اطلاعات را می‌توان با هزینه و خریداری یک سیستم پشتیبان‌گیری از اطلاعات مرتفع نمود یا ریسک از دست دادن مدیران را می‌توان با هزینه و استخدام یک نیروی انسانی پشتیبان در کنار مدیران دارای درجه اهمیت بالا مرتفع نمود. در واقع در این فعالیت، ریسک‌های محتملی که ممکن است بر روی خروجی‌های پروژه و یا کیفیت محصول نهایی پروژه یا به عبارت دقیق‌تر بر روی خصوصیات پرژوه‌های موفق نرم‌افزاری (نیاز، زمان و هزینه) تأثیر ناگواری بگذارند شناسایی، مدیریت و در صورت امکان تقلیل می‌یابند. دقت‌کنید که احتمال وقوع ریسک، تابعی از زمان است، بنابراین مدیریت ریسک می‌بایست در سراسر چرخه‌ی حیات نرم‌افزار، حضوری فعال و پررنگ داشته باشد. بنابراین گزینه اول درست و گزینه‌های دوم، سوم و چهارم نادرست هستند.

۲۷- گزینه (۴) صحیح است.

مدل‌های RAD و Sequential (ترتیبی) یا Linear (خطی) یا Waterfall (آبشاری) جزو مدل‌های غیرتکاملی و مدل افزایشی (Incremental Model) جزو مدل‌های تکاملی سنتی فرآیند تولید نرم‌افزار هستند.

Prototyping (نمونه‌سازی دورانداختنی) یا Rapid Prototyping (نمونه‌سازی سریع) یک مکانیزم، صرفاً جهت شناسایی نیازمندی‌های مشتری است، در یک بیان کلی‌تر نمونه‌سازی دورانداختنی، گفتگو، مشاهده و مصاحبه روش‌هایی جهت شناسایی نیازمندی‌های مشتری و تهیه لیست نیازمندی‌ها می‌باشند. توجه کنید که نمونه‌سازی دورانداختنی یک مدل فرآیند تولید نرم‌افزار به شمار نمی‌آید، بلکه یک مکانیزم صرفاً جهت شناسایی نیازمندی‌های مشتری می‌باشد. به حاصل جمع مکانیزم، نمونه‌سازی دورانداختنی و مدل آبشاری «مدل نمونه‌سازی» گفته می‌شود که این مدل نیز یک مدل غیرتکاملی به حساب می‌آید. در واقع مدل نمونه‌سازی از ترکیب مکانیزم نمونه‌سازی دورانداختنی و مدل آبشاری ایجاد شده است. به این نحو که در ابتدای کار توسط

مکانیزم نمونه‌سازی دورانداختنی، تمامی لیست نیازمندی‌های مشتری تکمیل می‌گردد، سپس توسط مدل آبشاری به شکل خطی نرم‌افزار ایجاد می‌گردد و کار تمام می‌شود. در واقع نیازمندهای مشتری طی تکرارهای مکانیزم نمونه‌سازی دورانداختنی شناخته می‌شوند. سپس طی یک روال خطی، توسط مدل آبشاری، نرم‌افزار تولید می‌گردد و کار تمام می‌شود و دیگر تکامل نمی‌یابد. زیرا مدل نمونه‌سازی یک مدل غیرتکاملی است و پس از ساخت نهایی امکان ادامه‌ی پروژه و تغییر وجود ندارد.

پروژه‌ای که از تکنولوژی mobile برای اطلاع‌رسانی در مورد خاصی استفاده می‌کند، در ابتدای کار لیست نیازمندی‌های مشخصی ندارد، همچنین تکنولوژی mobile مدام در حال تغییر و تحول می‌باشد، بنابراین پس از ساخت پروژه هم امکان تغییر و تحول وجود دارد.

در چنین شرایطی که نیازها در ابتدای کار مشخص نیستند، و پس از تحویل محصول باز هم امکان تغییر و تحول در محصول وجود دارد، مناسب‌ترین مدل فرآیند تولید نرم‌افزار، مدل‌های تکاملی سنتی (افزایشی و پیچشی)، مدل تکاملی سنتی خاص یعنی مبتنی بر مؤلفه ساخت‌یافته (مبتنی بر تابع) و مدل تکاملی مدرن یعنی مبتنی بر مؤلفه‌ی شیءگرا (مبتنی بر کلاس) هستند.

از آنجا که در صورت سوال، از وجود قطعات و مولفه‌های آماده و قابل استفاده مجدد صحبت نشده است، مدل‌های مبتنی بر مولفه ساخت‌یافته و مبتنی بر مولفه شی گرا را کنار می‌گذاریم،

همچنین مدل‌های غیرتکاملی RAD و Sequential را نیز با توجه به شرایط پروژه کنار می‌گذاریم.

بنابراین با توجه به گزینه‌ها پرواضح است که مدل افزایشی پاسخ درست خواهد بود.

مدل افزایشی، مراحل مدل آبشاری را با رویکرد تکرار و تکامل مکانیزم نمونه‌سازی تکاملی ترکیب نموده است. در این مدل، با توجه به خاصیت نمونه‌سازی تکاملی، پروژه به تدریج کامل می‌شود یعنی هر مرحله‌ای که می‌گذرد، پروژه کامل‌تر شده و در افزایش‌های بعدی این تکامل ادامه می‌یابد تا به محصول نهایی برسد و این تکامل نیز ادامه خواهد داشت.

توجه کنید که می‌توان از مکانیزم نمونه‌سازی دورانداختنی در ابتدای هر تکرار (افزایش) در مدل افزایشی به جهت کاهش ریسک مربوط به شناسایی دقیق لیست نیازمندی‌های مشتری استفاده نمود. این قضیه برای مدل‌های پیچشی و مبتنی بر مؤلفه ساخت‌یافته و شیءگرا نیز صادق است.

به طور کلی نمونه‌سازی سریع، به عنوان راه‌کاری برای شناسایی لیست نیازمندی‌های مشتری می‌باشد و در اغلب مدل‌ها می‌توان آن را گنجاند به غیر از مدل آبشاری و مدل RAD .

از آنجا که مدل RAD یک مدل غیرتکاملی است، پرواضح است که مدل مناسبی در این پروژه نخواهد بود، اما به غیر از این مطلب شرط لازم برای استفاده از مدل RAD تقسیم‌پذیر بودن پروژه است که از این مسأله هم صحبتی نشده است.

۲۸- گزینه (۱) صحیح است.

مدل روش‌های رسمی (فرمال، قراردادی و صوری) شامل مجموعه‌ای از فعالیت‌ها است که سعی دارد پروژه‌ی نرم‌افزاری را در قالب روابط رسمی و ریاضی، سیستم‌های کامپیوتری را

تعریف، تولید، پیاده‌سازی و ارزیابی نماید. در این مدل، با استفاده از تحلیل‌های ریاضی، بسیاری از ابهامات، نواقص و عدم سازگاری نرم‌افزار را تا حد زیادی می‌توان بـه سـادگی کشـف و تصـحیح نمود. بنابراین گزینه اول درست است.

گزینه دوم نادرست است. زیرا، RUP یک نمونه متدولوژی شیءگرا براساس مدل فرآیند مبتنی بر مؤلفه شیءگرا، روش شیءگرا (با استفاده از مفاهیم کلاس، وراثت و چندریختی) و ابزار UML است. همچنین OOA (Object Oriented Analysis) به معنی تحلیل شیءگرا، یک فعالیـت تحلیـل شیءگرا از فعالیت‌های فرآیند تولید نرم‌افزار (ارتباط، برنامه‌ریـزی، تحلیـل، طراحـی، پیـاده‌سـازی، تست و استقرار) است.

گزینه سوم نیز نادرست است. زیرا، مدل توسعه مبتنی بر مؤلفه نیازمند وجود قطعات آمـاده در حد قابل قبولی می‌باشد و این پیش شرط، با شرایط مسأله سازگاری ندارد.

گزینه چهارم نادرست است. زیرا، مدل توسعه همروند، برای مـدیریت وضـعیت فعالیـت‌هـای مختلف چندین پروژه‌ی مختلف در حال انجام در یک سازمان نرم‌افزاری است. اغلب سازمان‌هـای نرم‌افزاری، در یک بازه‌ی زمانی، احتمالاً چندین پروژه را در دست تولیـد دارنـد. بـه عنـوان مثـال ممکن است، پنج پروژه از پروژه‌های سازمان در مرحله‌ی ایجاد باشند، اما احتمالاً وضعیت آن‌ها بـا یکدیگر متفاوت است (مثلاً یکی در مرحله‌ی تولید کد است در حالی که دیگـری در حـال تسـت قرار دارد). مدل توسعه‌ی همروند که با نام مهندسی هم‌روند نیز شـناخته مـی‌شـود، جهـت کنتـرل اجرای چندین پروژه‌ی همزمان مورد استفاده قرار می‌گیرد که هر یک از فعالیـت‌هـای چـارچوبی فرآیند تولید نرم‌افزار مربوط به هر پروژه می‌تواند در وضـعیت‌هـای مختلفـی قـرار داشـته باشـند. وضعیت‌های مختلف مربوط به هر یک از فعالیت‌هـای چـارچوبی توسـط یـک گـراف نشـان داده می‌شود.

۲۹- گزینه (۴) صحیح است.

در فعالیت ارتباط (مهندسی نیازمندی‌ها) لیست نیازمندی‌های مشتری از طریق ارتباط با مشتری و استفاده از ابزارها و مکانیسم‌هایی همچون گفتگو، مشاهده، مصاحبه، پرسش‌نامه، بازدیـد، نمونـه سازی دورانداختنی و نمونه‌سازی تکاملی، جمع‌آوری و آماده می‌گردد. این ابزارهـا و مکانیسـم‌هـا کمک می‌کنند، تا مشتری خواسته‌ها و نیازهای خود را دقیق‌تـر بیـان کنـد و سـازنده نیـز دقیـق‌تـر نیازهای مشتری را بشناسد. به این ارتباطات میان مشتری و سازنده بـرای شـناخت نیازهـا، تحلیـل نیازها نیز گفته می‌شود که منجر به شناخت نیازهایی بدون ابهـام در مشخصـات راه‌حـل، سـنجش اعتبار، اعتبارسنجی و معتبر بودن مشخصات نیازها می‌گردد.

۳۰- گزینه (۳) صحیح است.

در فعالیت ارتباط (مهندسی نیازمندی‌ها) لیست نیازمندی‌های مشتری از طریق ارتباط با مشتری و استفاده از ابزارها و مکانیسم‌هایی همچون گفتگو، مشاهده، مصاحبه، پرسش‌نامه، بازدیـد، نمونـه سازی دورانداختنی و نمونه‌سازی تکاملی، جمع‌آوری و آماده می‌گردد. این ابزارهـا و مکانیسـم‌هـا کمک می‌کنند، تا مشتری خواسته‌ها و نیازهای خود را دقیق‌تـر بیـان کنـد و سـازنده نیـز دقیـق‌تـر

نیازهای مشتری را بشناسد. به این ارتباطات میان مشتری و سازنده برای شناخت نیازها، تحلیل نیازها نیز گفته می‌شود که منجر به شناخت نیازهایی بدون ابهام در مشخصات راه‌حل، سنجش اعتبار، اعتبارسنجی و معتبر بودن مشخصات نیازها می‌گردد.

تحلیل نیازها، عدم ابهام در مشخصات راه‌حل و اعتبارسنجی مشخصات نیازمندی‌ها، نتیجه اقداماتی (ابزارهایی) همچون گفتگو، مشاهده، مصاحبه، پرسش‌نامه، بازدید، نمونه‌سازی دورانداختنی و نمونه‌سازی تکاملی است.

ارزیابی خطر (مدیریت ریسک) در فعالیت برنامه‌ریزی، از فعالیت‌هایی چارچوبی انجام می‌گردد. فعالیت‌های چتری نیز بر فعالیت برنامه‌ریزی و به تبع مدیریت ریسک نظارت دارد.

۳۱- گزینه (۱) صحیح است.

صورت سوال به این شکل است:

کدام یک از موارد زیر اگر وجود داشته باشد از روش تولید سریع برنامه کاربردی (Rapid Application Development) نباید استفاده کرد؟

۱) ریسک فنی بالا

گزینه اول پاسخ سوال است، زیرا این مدل در پروژه‌هایی با ریسک‌های فنی بالا، به دلیل عدم امکان شناسایی نیازمندی‌های مشتری در ابتدای پروژه کارآمد نخواهد بود. مدل RAD در محیط عملیاتی که نیازها به طور کامل واضح، مشخص و ثابت است، کارآمد است، در غیر اینصورت این مدل ناکارآمد خواهد بود.

۲) واحدمند (Modular) بودن سیستم

گزینه دوم پاسخ سوال نیست، زیرا شرط لازم برای انجام پروژه‌های نرم‌افزاری توسط مدل RAD، قابلیت بخش‌بندی یا پیمانه‌ای یا واحدمند (Modular) بودن سیستم یا پروژه است.

۳) منابع انسانی کافی برای پروژه‌های بزرگ

گزینه سوم پاسخ سوال نیست، زیرا در پروژه‌های بزرگ، تعداد بخش‌های مختلف پروژه زیاد می‌شود، به همین دلیل به تیم‌های نرم‌افزاری بیشتری نیاز خواهد بود. بنابراین با افزایش حجم پروژه باید نیروی انسانی کافی برای پیمانه‌ها وجود داشته باشد.

۴) تعامل کامل کاربر و تولیدکننده سیستم

گزینه چهارم پاسخ سوال نیست، زیرا موفقیت مدل RAD، وابسته به تعامل مناسب و کامل سازنده (تولید کننده سیستم) و مشتری (کاربر) است و هر دو باید برای انجام سریع فعالیت‌ها با یکدیگر هماهنگ باشند تا بتوانند در موعد مقرر تولید نهایی را تحویل مشتری دهند. چون اگر یک قسمت از این پروژه انجام نشده باشد. تحویل پروژه میسر نیست، بنابراین مدیریت این مدل اهمیت فراوانی دارد.

۳۲- گزینه (۲) صحیح است.

صورت سوال به این شکل است:

کدام مدل فرآیندی، تکاملی و ریسک-رانه (Risk-Driven) محسوب می‌شود؟

مدل V و مدل Sequential (ترتیبی) یا Linear (خطی) یا Waterfall (آبشاری) جزو مدل‌های غیرتکاملی و مدل افزایشی (Incremental Model) و مدل مارپیچی (Spiral) جزو مدل‌های تکاملی سنتی فرآیند تولید نرم‌افزار هستند.

۱) مدل V

گزینه اول پاسخ سوال نیست، زیرا زیرا مدل V یک مدل غیرتکاملی است.

۲) مدل مارپیچی (Spiral)

گزینه دوم پاسخ سوال است، زیرا رویکرد مدل پیچشی همانند مدل افزایشی است با این تفاوت اساسی که در مدل پیچشی **مدیریت ریسک** (تحلیل ریسک) در فعالیت برنامه‌ریزی به طور جدی و در تمام جوانب پروژه انجام می‌گیرد. اما در مدل افزایشی فقط **ریسک مربوط به شناسایی درست نیازمندی‌ها** در هر تکرار توسط مکانیزم نمونه‌سازی دورانداختنی و **ریسک تکنیکی** مدیریت می‌گردد.

۳) مدل آبشاری (Waterfall)

گزینه سوم پاسخ سوال نیست. زیرا مدل آبشاری یک مدل غیرتکاملی است.

۴) مدل افزایشی (Incremental)

گزینه چهارم پاسخ سوال نیست. زیرا در مدل افزایشی، فقط **ریسک مربوط به شناسایی درست نیازمندی‌ها** در هر تکرار توسط مکانیزم نمونه‌سازی دورانداختنی و **ریسک تکنیکی** مدیریت می‌گردد.

۳۳- گزینه (۲) صحیح است.

به طور کلی انواع نیازمندی‌های نرم‌افزار را می‌توانیم به دو دسته زیر تقسیم کنیم:

۱- وظیفه‌مندی (کارکردی، عملکردی)

۲- غیروظیفه‌مندی (غیرکارکردی، غیر عملکردی)

اگر یک معادله درجه دو به صورت زیر باشد:

$$ax^2+bx+c=0$$

راه حل عمومی محاسبه ریشه‌های آن به صورت زیر است:

$$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

گزاره الف یعنی $x=\left(c-b+sqrt(b^2-4\times a\times c)\right)/2\times a$ البته به کمی اصلاح پرانتزگذاری، ریشه‌های معادله درجه دوم $a(x-c)^2+b(x-c)+c=0$ را محاسبه می‌کند. که اشاره به «**کارکرد**» و

به تبع لیست نیازمندی‌های وظیفه‌مندی (کارکردی، عملکردی) یک نرم افزار دارد.

$a(x-c)^{۲}+b(x-c)+c=۰$

گزاره ب یعنی در یک بیمارستان، کاربر باید قادر باشد تا در بانک، اطلاعات کاملی از بیماران را جستجو کند یا حتی زیر مجموعه‌ای از آن را انتخاب کند. که اشاره به «**کارکرد**» و به تبع لیست نیازمندی‌های وظیفه‌مندی (کارکردی، عملکردی) یک نرم افزار دارد.

هرآنچه مربوط به طعم، مزه و خوشگل‌کاری یک نرم افزار باشد، مربوط به لیست نیازمندی‌های غیروظیفه‌مندی است. همه خودروها انسان را می‌برند (کارکردی) اما مرسدس بنز مزه (غیرکارکردی) دیگری دارد. همه خانم‌ها غذا درست می‌کنند (کارکردی) اما برخی دست‌پخت‌ها مزه (غیرکارکردی) دیگری دارد. به همین سادگی!

توجه: کلمه «**دامنه‌ای**» در گزینه سوم ارتباطی به انواع نیازمندی‌های نرم‌افزار ندارد، دامنه به معنی بخش‌ها و زیرسیستم‌های مختلف یک محیط عملیاتی مثل دانشگاه است که سیستم‌سازی آن هنوز امکان سنجی نشده است.

۳ مدل تحلیل ساخت‌یافته

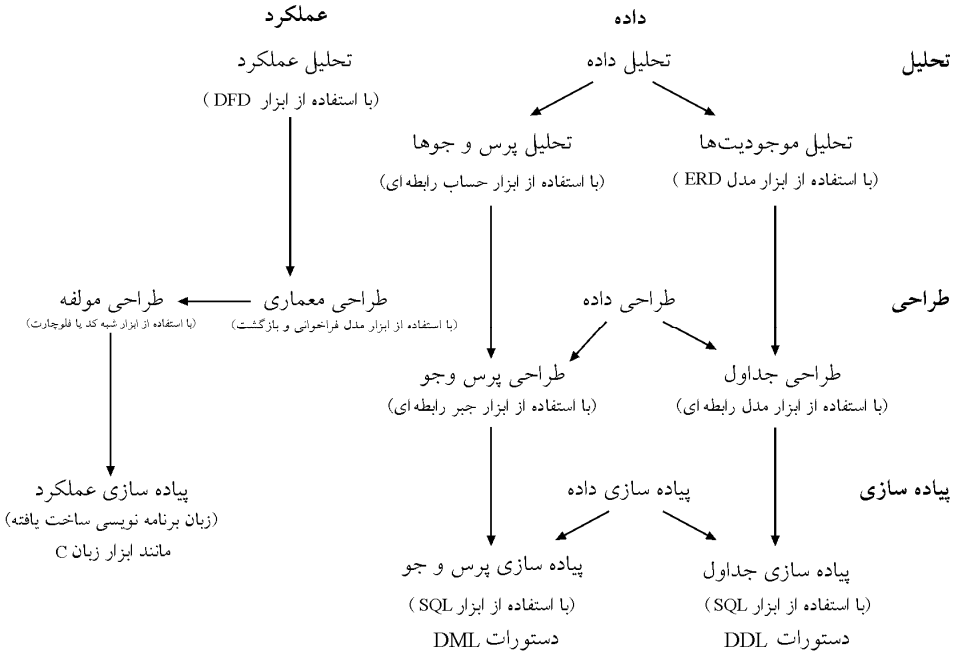
فعالیت‌های چارچوبی فرآیند تولید نرم‌افزار

به طور کلی فعالیت‌های مرتبط با فرآیند تولید نرم‌افزار صرف‌نظر از اندازه، پیچیدگی پروژه و زمینه‌ی کاربردی آن و مستقل از متدولوژی ساخت‌یافته و شیءگرا به پنج فعالیت ارتباط، برنامه‌ریزی، مدل‌سازی، ساخت و استقرار تقسیم می‌شود، به بیان دیگر فعالیت‌ها در هر دو دسته‌ی متدولوژی ساخت‌یافته و شیءگرا همین‌ها خواهند بود، اما کارهایی که در هر فعالیت در متدولوژی ساخت‌یافته و شیءگرا انجام می‌شود شباهت‌ها و تفاوت‌هایی خواهد داشت.

در ادامه فعالیت چارچوبی مدل‌سازی (مدل تحلیل) از فرآیند تولید نرم‌افزار براساس متدولوژی ساخت‌یافته بیان خواهد شد:

مدل‌سازی (تحلیل و طراحی)

یک مدل، ساده شده یک واقعیت است. ایجاد یک مدل برای سیستم‌های نرم‌افزاری قبل از ساخت یا بازساخت آن، به اندازه داشتن نقشه برای ساختن یک ساختمان ضروری وحیاتی است. بسیاری از شاخه‌های مهندسی، توصیف چگونگی محصولاتی که باید ساخته شوند را ترسیم می‌کنند و همچنین دقت زیادی می‌کنند که محصولاتشان طبق این مدل‌ها و توصیف‌ها ساخته شوند. مدل‌های خوب و دقیق در برقراری یک ارتباط کامل بین افراد پروژه، نقش زیادی می‌توانند داشته باشند. علت اصلی مدل کردن سیستم‌های پیچیده این است که نمی‌توان به یکباره کل سیستم را تجسم کرد و ممکن است سیستم دارای ابهامات بسیاری باشد. لذا برای رفع این ابهامات و نیز برای فهم کامل سیستم و یافتن و نمایش ارتباط بین قسمت‌های مختلف آن، از مدل‌سازی استفاده می‌شود. فعالیت مدل‌سازی خود شامل دو مرحله‌ی مدل تحلیل و مدل طراحی می‌باشد. مدل تحلیل پس از فعالیت ارتباطات (جمع‌آوری نیازمندی‌ها) و قبل از مدل طراحی انجام می‌شود، در واقع خروجی مدل تحلیل، ورودی مدل طراحی می‌باشد. شکل زیر گویای این مطلب می‌باشد:



مدل تحلیل

پس از جمع‌آوری نیازمندی‌ها در فعالیت ارتباطات نوبت به مدل تحلیل (مدل‌سازی لیست نیازمندی‌ها) می‌رسد. مدل‌سازی که فعالیتی فنی به شمار می‌رود نیازمندی‌ها را باید به گونه‌ای مدل نماید که برای سازنده و مشتری قابل فهم باشد.

مدل‌هایی که در فعالیت مدل‌سازی (بخش مدل تحلیل)، تهیه می‌شوند، سه مزیت مهم را در تولید برنامه‌های کامپیوتری ایجاد می‌کنند:

۱- توصیف نمادین نیازهای مشتری، توسط مدل‌سازی لیست نیازمندی‌ها، بر اساس نیازمندی‌های وظیفه‌مندی و هم نیازمندی‌های غیروظیفه‌مندی.

۲- مدل تحلیل، بستری مناسب را برای فعالیت طراحی نرم‌افزار ایجاد می‌کند.

۳- در فعالیت ساخت (بخش تست)، هنگامیکه اعتبارسنجی (validation) نرم‌افزار انجام می‌شود، مدل تحلیل، می‌تواند به عنوان یک تصویر از نرم‌افزار مورد انتظار، جهت فعالیت تست مورد استفاده قرار گیرد.

توجه: مدل تحلیل به روش ساخت‌یافته از سه بخش مدل‌سازی داده‌ای، مدل‌سازی عملکردی و مدل‌سازی رفتاری تشکیل شده است، مدل‌سازی داده‌ای شامل تحلیل موجودیت‌ها و تحلیل پرس و جوها می‌باشد. تحلیل موجودیت‌ها توسط ابزار مدل ER و تحلیل پرس و جوها توسط ابزار حساب رابطه‌ای مدل می‌شوند، مدل‌سازی عملکردی توسط ابزار DFD و مدل‌سازی رفتاری توسط ابزار STD مدل می‌شود.

در ادامه به تشریح هر یک از مدل‌های مذکور می‌پردازیم.

مدل‌سازی داده‌ای

نرم‌افزار، داده‌ها را به اطلاعات مفید برای کابر تبدیل می‌کند و این کار را از طریق مجموعه‌ای از پردازش‌ها انجام می‌دهد، بنابراین در مدل‌سازی داده‌ای، تلاش می‌شود تا مدل دقیقی از داده‌ها، موجودیت‌ها و روابط مابین موجودیت‌های موجود در محیط عملیاتی نرم‌افزار ایجاد شود.

توجه: مدل‌سازی داده‌ای، موجودیت‌ها و روابط مابین موجودیت‌های موجود در محیط عملیاتی سیستم را، مستقل از روند پردازش آنها در داخل نرم‌افزار مدل می‌کند.

توجه: ERD یا Entity Relationship Diagram یا نمودار نهاد و رابطه جهت مدل‌سازی داده‌ای مورد استفاده قرار می‌گیرد.

مدل ER

برای شناسایی هر مدل یا ساختاری ابتدا باید به بررسی بخش‌های مختلف آن مدل یا ساختار بپردازیم.

در مدل ER، سه مفهوم معنایی وجود دارد و معنای داده‌های هر محیط عملیاتی (محیطی که می‌خواهیم در مورد آن اطلاع کسب کنیم) به کمک همین سه مفهوم نمایش داده می‌شود:

۱- موجودیت یا نهاد (Entity)

۲- صفت (Attribute)

۳- ارتباط یا رابطه (Relationship)

توجه: در مدل تحلیل منظور از رابطه، ارتباط بین موجودیت‌هاست، اما در مدل طراحی منظور از رابطه، جداول مدل رابطه‌ای می‌باشد.

۱- نهاد (موجودیت)

موجودیت عبارتند از مفهوم کلی «شیء»، «چیز»، «پدیده» و به طور کلی هر آنچه که می‌خواهیم در موردش «اطلاع» داشته باشیم.

مثال: در محیط عملیاتی دانشگاه، انواع موجودیت‌های دانشجو، درس، استاد، گروه آموزشی و غیره وجود دارد.

توجه: در کتب مختلف گاهاً به نهاد، مجموعه نهادی (Entity Set) نیز گفته می‌شود.

توجه: خروجی مدلسازی یک سیستم به کمک مدل ER، نموداری است که آن را ERD می‌نامند.

توجه: برای نمایش یک موجودیت (مجموعه نهادی) در نمودار ER از یک مستطیل نام‌دار استفاده می‌گردد.

مثال: موجودیت‌های درس و دانشجو در محیط عملیاتی دانشگاه.

درس	دانشجو

توجه: توصیه می‌شود در تشخیص مجموعه‌های نهادی و تحلیل سیستم به دنبال اسامی عام بگردیم. معمولاً اسامی عام یک موجودیت یا مجموعه نهادی را نشان می‌دهند (مانند دانشجو) و اسامی خاص اعضای یک مجموعه نهادی را نشان می‌دهند (مانند دانشجو احمدی)

توجه: از آنجا که نمادهای گرافیکی نمی‌توانند تمامی جزئیات مربوط به موجودیت‌ها یا اشیاء داده‌ای را بیان کنند، جهت تشریح شرح حال موجودیت‌ها یا اشیاء داده‌ای از مستنداتی به نام شرح حال موجودیت‌ها یا اشیاء داده‌ای یا فرهنگ داده‌ها (دیکشنری داده) استفاده می‌شود. فرهنگ داده‌ها، به بیان جزئیات مربوط به موجودیت‌ها یا اشیاء داده‌ای مورد نظر می‌پردازد.

برای‌مثال بر روی نماد مستطیل شکل موجودیت‌ها اسم برده می‌شود، اما این‌که دقیقاً این موجودیت‌ها یا اشیاء داده‌ای چه هستند مشخص نمی‌شود. برای این منظور از فرهنگ داده‌ها استفاده می‌شود که موجودیت‌ها یا اشیاء داده‌ای را به صورت دقیق تشریح می‌کند.

۲- صفت

صفت در واقع خصیصه یا ویژگی یک نوع موجودیت است و هر نوع موجودیت مجموعه‌ای از صفات(موسوم به مجموعه صفات موجودیت) را دارد. هر صفت از نظر کاربران یک نام، یک نوع و یک معنای مشخص دارد. به عنوان مثال، موجودیت درس را در نظر بگیرید. صفات درس عبارتند از: شماره درس، عنوان درس، تعداد واحد درس، نوع درس (پایه، تخصصی، اختیاری و غیره)، ماهیت درس (نظری، عملی و غیره)، سطح درس (کاردانی، کارشناسی و غیره).

توجه: هر صفتی از یک مجموعه مقادیر معتبر و مجاز مقدار می‌گیرد که به این مجموعه مقادیر، اصطلاحاً دامنه یا میدان (Domain) مقادیر آن صفت می‌گویند.

انواع صفات در مدل ER

هر موجودیت دارای مجموعه‌ای از صفات است که در نمودار ER با بیضی نشان داده می‌شود.

صفت کلید

کلید عبارتند از یک یا چند صفت خاصه که در یک موجودیت منحصربه فرد باشد. به عبارت دقیق‌تر هر ترکیبی از صفت یا صفات که اگر بر اساس آن جستجو انجام گردد فقط و فقط یک مقدار را بازگرداند. بر اساس کدملی جستجو انجام دهید، واضح است که فقط و فقط یک مقدار را باز می‌گرداند. به عنوان مثالی دیگر در موجودیت دانشجو، شماره دانشجویی کلید است، چون هر دانشجو شماره‌ای یکتا دارد. ولی نام نمی‌تواند کلید باشد.

توجه: گاهی یک صفت به تنهایی نمی‌تواند کلید باشد بلکه مجموعه‌ای از دو یا چند صفت، تشکیل کلید را می‌دهند.

توجه: صفت کلید نمی‌تواد مقدار NULL بگیرد.

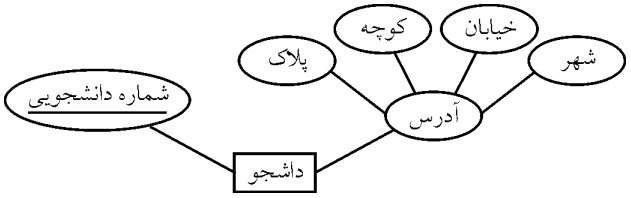
توجه: در نمودار ER زیر صفت یا صفات کلیدی یک خط می‌کشند.

صفت ساده و مرکب

صفت ساده صفتی است که مقدار آن از لحاظ معنایی ساده یا اتومیک یا تجزیه نشدنی باشد، به این معنا که اگر مقدار آن را به اجزایی تجزیه کنیم، مقادیر جزیی حاصله فاقد معنا باشد. برای‌مثال

صفت درس و شماره دانشجویی یک صفت ساده است. صفت مرکب صفتی است که از چند صفت ساده تشکیل شده باشد به گونه‌ای که تجزیه شدنی باشند و اجزاء حاصله خود صفات ساده باشند. مانند صفت آدرس که از اجزاء نام استان، نام شهر، نام خیابان، نام کوچه، شماره پلاک و کدپستی تشکیل شده است.

توجه: برای نشان دادن صفت مرکب در نمودار ER اجزای صفت مرکب خود به عنوان صفت برای صفت مذکور نشان داده می‌شود.



توجه: در بانک اطلاعاتی مبتنی بر مدل رابطه‌ای (جدولی) صفت مرکب نداریم.

صفت تک مقداری و چند مقداری

بعضی از صفات چه ساده و چه مرکب فقط می‌توانند یک مقدار را بگیرند که به این صفات، صفت تک‌مقداری می‌گویند. مانند شماره دانشجویی که نمی‌تواند بیش از یک مقدار داشته باشد. این صفات در نمودار ER بصورت معمول نمایش داده می‌شوند.

صفاتی وجود دارند که می‌توانند چندین مقدار را بگیرند مانند صفت مدرک در موجودیت استاد که می‌تواند مقادیر لیسانس، فوق لیسانس و یا دکتری را در خود بگیرد.

مثال:

توجه: به مثال‌های زیر توجه کنید.

صفت ساده تک‌مقداری: مانند کدملی

صفت ساده چندمقداری: مانند مدرک تحصیلی

صفت مرکب تک‌مقداری: مانند تاریخ تولد

صفت مرکب چندمقداری: مانند آدرس

توجه: در بانک اطلاعاتی مبتنی بر مدل رابطه‌ای (جدولی) صفت چندمقداری نداریم.

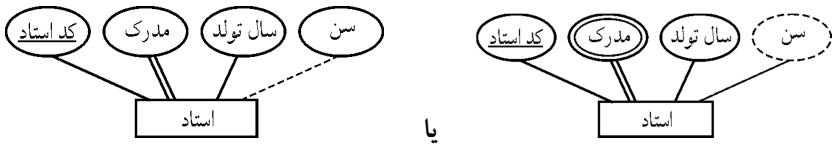
صفت مشتق (پویا)

صفتی است که در موجودیت وجود خارجی ندارد ولی در صورت لزوم می‌توان آنرا بدست

آورد. صفتی که مقادیر آن مدام در حال تغییر و تحول باشد، صفت پویا یـا مشـتق محسـوب مـی-گردد. بنابراین به دلیل تغییرات مداوم، توصیه می‌گردد صفت پویا در جداول بانک اطلاعات مـورد استفاده قرار نگیرد و مقدار آن از طریق صفت مرتبط با آن محاسبه گردد. برای‌مثـال بـرای محاسـبه صفت سن، می‌توان صفت تاریخ تولد را در نظر گرفت و از روی این صفت، سن را محاسبه نمود.

توجه: صفت مشتق را در نمودار ER با نقطه چین به موجودیت مورد نظر متصل می‌کنند.

مثال:



۳- رابطه (ارتباط)

ارتباط، وابستگی بین دو مجموعه نهادی (موجودیت) را نشان می‌دهد.

توجه: در هنگام استخراج رابطه‌ها در مدل تحلیل، به دنبال افعال می‌گردیم.

توجه: برای نمایش یک رابطه در ER از یک لوزی استفاده می‌کنیم.

مثال:

در نمودار ER فوق، رابطه تهیه، مابین موجودیت تولیدکننده و قطعه برقرار است.

خواص رابطه

بطور کلی خواص رابطه به سه شکل زیر بررسی می‌گردد:

الف) درجه ارتباط

ب) کاردینالیتی ارتباط یا کمیت ارتباط

ج) اجباری و اختیاری بودن ارتباط یا مودالیتی ارتباط یا کیفیت ارتباط

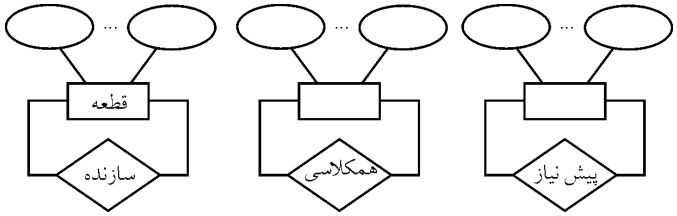
توجه: همه روابط سه خصیصه فوق را به طور همزمان دارند، اما مقادیر این خصیصه‌ها در روابـط مختلف، متفاوت است.

الف) درجه ارتباط

به تعداد موجودیت‌هایی که در یک رابطه مشارکت دارند، درجه ارتباط گفته مـی‌شـود. درجـه در مدل ER عددی صحیح و کوچکتر از ۵ است. ارتباط‌های درجه ۱، ۲ و ۳ معمول، ارتباط درجه ۴ کمیاب و غیر معمول است و ارتباط بالاتر از درجه ۴ قابل رسم کردن نیست.

ارتباط درجه ۱

مثال:

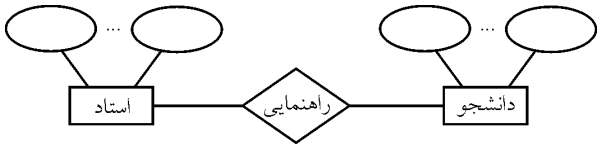


فقط یک موجودیت در هر یک از شکل‌های فوق وجود دارد. دقت‌کنید که موجودیت یک نوع است و عضوهای آن، یک مجموعه را تشکیل می‌دهند. بنابراین در ارتباط یکتایی، عضوهایی از یک مجموعه در ارتباط با عضوهایی دیگر از همان مجموعه قرار می‌گیرند. برای مثال درسی پیش نیاز درس دیگر است، یا دانشجویی همکلاسی دانشجوی دیگری است، در حالیکه همه دروس به یک موجودیت و همه دانشجویان نیز به یک موجودیت تعلق دارند.

توجه: وقتی یک ارتباط بین یک نوع موجودیت و خودش برقرار باشد، آنرا ارتباط با خود (Self-Relationship) یا بازگشتی (Recursive) می‌گویند.

ارتباط درجه ۲

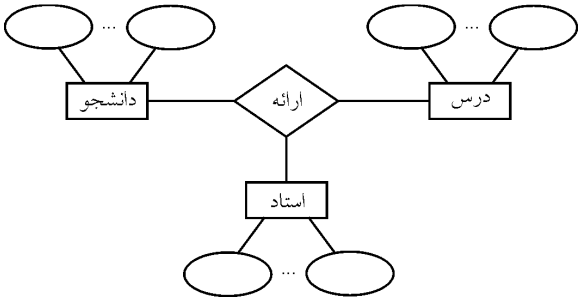
مثال:



در این ارتباط دو موجودیت استاد و دانشجو مشارکت دارند.

ارتباط درجه ۳

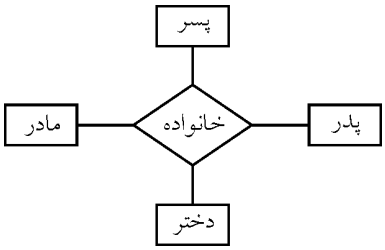
مثال:



در این ارتباط سه موجودیت استاد، درس و دانشجو مشارکت دارند.

ارتباط درجه ۴

مثال:

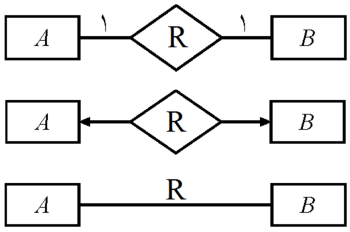


در این ارتباط چهار موجودیت پدر، مادر، پسر و دختر مشارکت دارند.

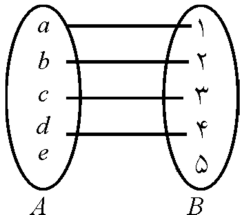
ب) کاردینالیتی ارتباط یا کمیت ارتباط

کاردینالیتی ارتباط بر سه نوع است: یک به یک، یک به چند، چند به چند.

۱- یک به یک (۱-۱)



یعنی هر نمونه موجودیت از A با حداکثر با یک نمونه موجودیت از B ارتباط دارد و هر نمونه موجودیت از B با حداکثر یک نمونه موجودیت از A ارتباط دارد.



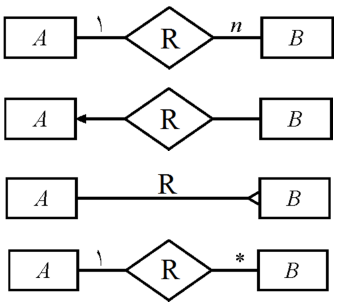
ارتباط یک به یک

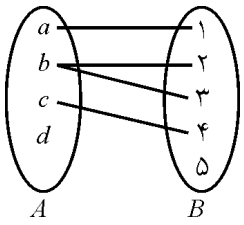
مثال:

بدین معنی که هر استاد حداکثر یک درس ارائه می‌کند و هر درس حداکثر یک استاد برای ارائه دارد.

توجه: یعنی ممکن است استادی اصلاً درس نداشته باشد و یا درسی توسط هیچ استادی در این ترم ارائه نگردد.

۲- یک به چند (۱:n)





ارتباط یک به چند

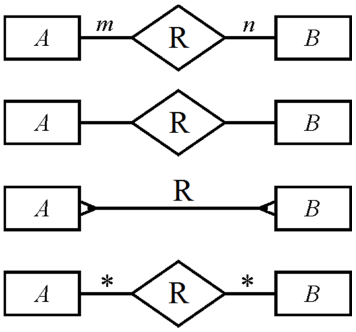
یعنی هر نمونه موجودیت از A با صفر یا بیشتر نمونه موجودیت از B ارتباط دارد و هر نمونه موجودیت از B با حداکثر یک نمونه موجودیت از A ارتباط دارد.

مثال:

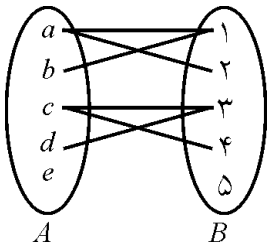
بدین معنی که هر استاد صفر یا بیشتر درس ارائه می‌کند و هر درس حداکثر یک استاد برای ارائه دارد.

توجه: یعنی ممکن است استادی اصلاً درس نداشته باشد و یا درسی توسط هیچ استادی در این ترم ارائه نگردد.

۳- چند به چند (m:n)



یعنی هر نمونه موجودیت از A با صفر یا بیشتر نمونه موجودیت از B ارتباط دارد و هر نمونه موجودیت از B با صفر یا بیشتر نمونه موجودیت از A ارتباط دارد.



مثال:

بدین معنی که هر استاد صفر یا بیشتر درس ارائه می‌کند و هر درس صفر یا بیشتر استاد برای ارائه دارد.

توجه: یعنی ممکن است استادی اصلاً درس نداشته باشد و یا درسی توسط هیچ استادی در این ترم ارائه نگردد.

توجه: در ارتباط، شرکت موجودیت‌ها در ارتباط به طور پیش‌فرض اختیاری است. برای‌مثال ممکن است استادی درسی ارائه نکند و یا درسی این ترم ارائه نشود. در ادامه ارتباط اجباری را مورد بررسی قرار خواهیم داد.

حد

مشخصه دیگر ارتباط، حد آن است که بیانگر حداقل و حداکثر تعداد نمونه موجودیت‌های شرکت‌کننده در ارتباط است. این مقادیر در پایین خط ارتباط، توسط پرانتز نشان داده می‌شود.

مثال:

در شکل بالا حد (۱۰و۰) نشان می‌دهد که یک استاد ممکن است راهنمای هیچ دانشجویی نباشد و یا حداکثر ۱۰ دانشجو را راهنمایی کند. همچنین حد (۱و۰) نشان می‌دهد که یک دانشجو ممکن است استاد راهنما نداشته باشد و یا حداکثر توسط یک استاد راهنمایی شود.

ج) اجباری و اختیاری بودن ارتباط یا مودالیتی ارتباط یا کیفیت ارتباط

این نوع رابطه به دو دسته کلی زیر تقسیم می‌شود:

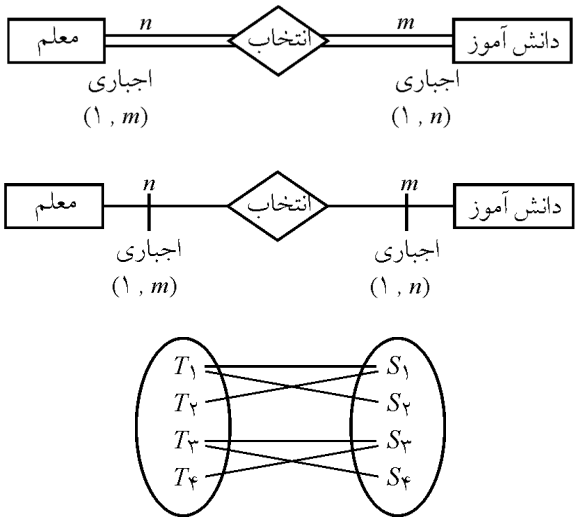
۱- اجباری یا کلی (Total)

یک رابطه اجباری است، اگر و تنها اگر تمام نمونه‌های موجودیت در رابطه شرکت کرده

باشند. اجباری بودن در نمودار ER با نماد خط مضاعف افقی یا نماد | به معنی یک و الزام شرکت در رابطه نشان داده می‌شود.

توجه: نماد خط مضاعف افقی نشانه اجباری بودن موجودیت چسبیده به آن است، اما نماد | به معنی یک و الزام شرکت در رابطه نشانه اجباری بودن موجودیت طرف مقابل است.

مثال: در نظام آموزشی وزارت آموزش و پرورش حضور معلم و دانش آموز در محیط عملیاتی مدرسه اجباری است. زیرا مفاهیمی همچون مرخصی برای دانش‌آموز و یا پژوهش به‌جای تدریس برای معلم معنا ندارد.



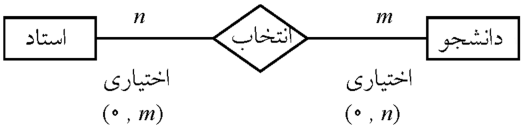
قید (1,m) نشان می‌دهد که یک معلم حداقل یک و حداکثر m دانش‌آموز دارد و قید (1,n) نشان می‌دهد که یک دانش‌آموز حداقل یک و حداکثر n معلم دارد.

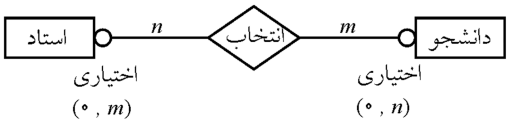
۲- اختیاری یا جزئی (Partial)

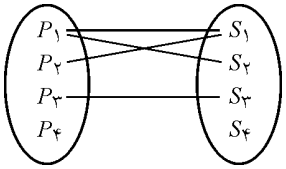
یک رابطه اختیاری است، اگر و تنها اگر حداقل یکی از نمونه‌های موجودیت در رابطه شرکت نکرده باشد. اختیاری بودن در نمودار ER با نماد خط افقی یا نماد دایره کوچک توخالی به معنی صفر و عدم الزام شرکت در رابطه نشان داده می‌شود.

توجه: نماد خط افقی نشانه اختیاری بودن موجودیت چسبیده به آن است، اما نماد دایره کوچک توخالی به معنی صفر و عدم الزام شرکت در رابطه نشانه اختیاری بودن موجودیت طرف مقابل است.

مثال: در نظام آموزشی وزارت علوم، تحقیقات و فناوری حضور استاد و دانشجو اختیاری است. زیرا مفاهیمی همچون مرخصی برای دانشجو و پژوهش به‌جای تدریس برای استاد معنا دارد.



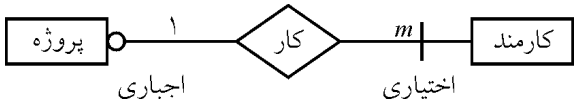




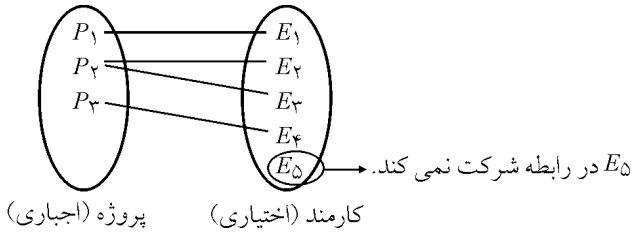
قید (0,m) نشان می‌دهد که یک استاد حداقل هیچ و حداکثر m دانشجو دارد و قید (0,n) نشان می‌دهد که یک دانشجو حداقل هیچ و حداکثر n استاد دارد.

توجه: از آنجا که حضور استاد اختیاری است، پس می‌تواند هیچ دانشجویی نداشته باشد. همین معنا برای دانشجو نیز صادق است.

مثال: نمودار ER زیر چه روابطی را نشان می‌دهد؟



شکل زیر گویای نمودار ER است.



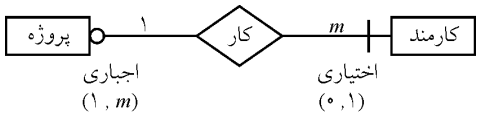
توجه: نماد | به معنی یک و الزام شرکت در رابطه نشانه اجباری بودن موجودیت طرف مقابل است.

توجه: نماد دایره کوچک توخالی به معنی صفر و عدم الزام شرکت در رابطه نشانه اختیاری بودن موجودیت طرف مقابل است.

با توجه به ارتباط یک به چند پروژه با کارمند و اجباری بودن حضور نمونه موجودیت‌های پروژه، قید (1,m) برای پروژه در نظر گرفته می‌شود، بدین معنی‌که، یک پروژه حداقل یک و حداکثر m کارمند دارد.

همچنین با توجه به ارتباط چند به یک کارمند با پروژه و اختیاری بودن حضور نمونه موجودیت‌های کارمند، قید (۱و۰) برای پروژه در نظر گرفته می‌شود، بدین معنی‌که، یک کارمند حداقل هیچ و حداکثر یک پروژه دارد.

شکل زیر گویای مطلب است:



صفت در ارتباط

ارتباط‌ها نیز می‌توانند صفت داشته باشند (اغلب در ارتباطات n:m)

برای مثال صفت تعداد قطعات (QTY) در نمودار بانک اطلاعات تولیدکنندگان قطعات می‌تواند صفت در ارتباط، رابطه تهیه باشد.

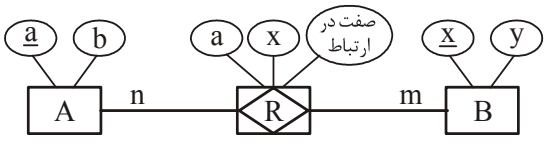
شاید تصور شود که تعداد قطعات مربوط به موجودیت قطعه است. ولی این تصور غلط است زیرا یک تولیدکننده چند قطعه و یک قطعه نیز توسط چند تولیدکننده، تهیه می‌گردد. بنابراین صفت تعداد قطعات (QTY) را باید به ارتباط **تولید** که دو موجودیت قطعه و تولیدکننده را به هم مرتبط می‌کند نسبت داد.

توجه: چنین ارتباطاتی با یک لوزی درون یک مستطیل نشان داده می‌شود و کلید آنها کلیدهای همه موجودیت‌های مربوطه را شامل می‌شود.

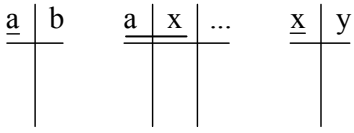
نگاشت رابطه چند به چند بین دو موجودیت (مدل تحلیل) به مدل رابطه‌ای (مدل طراحی)

مستقل از اختیاری یا اجباری بودن موجودیت‌ها، هر موجودیت به یک جدول تبدیل می‌گردد و یک جدول پُل (Bridge) نیز به عنوان ارتباط‌دهنده دو جدول مورد استفاده قرار می‌گیرد. همچنین کلید کاندید جدول پُل از ترکیب کلید کاندید دو جدول دیگر ایجاد می‌گردد. روال کلی به‌صورت زیر است:

مدل تحلیل:



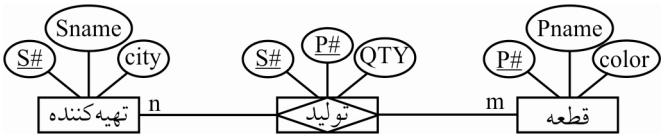
مدل طراحی:



جدول A جدول AB جدول B

مثال: ارتباط تولیدکنندگان و قطعات.

مدل تحلیل:



توجه: صفت QTY، به‌عنوان یک صفت در ارتباط، تعداد قطعات را مشخص می‌کند.

فعالیت طراحی:

S#	Sname	city
S_1	Sn1	C_1
S_2	Sn2	C_1
S_3	Sn3	C_3

جدول تولیدکننده

S#	P#	QTY
S_1	P_1	10
S_1	P_2	20
S_2	P_1	30

جدول تولید

P#	Pname	color
P_1	Pn1	Red
P_2	Pn2	Blue

جدول قطعه

مدل‌سازی عملکردی یا مدل‌سازی جریان‌گرا (Flow – Oriented Models)

مدل‌سازی عملکردی یا مدل‌سازی جریان‌گرا، مجموعه‌ای از مدل‌های به نسبت قدیمی هستند که یکی از مهم‌ترین آن‌ها DFD است.

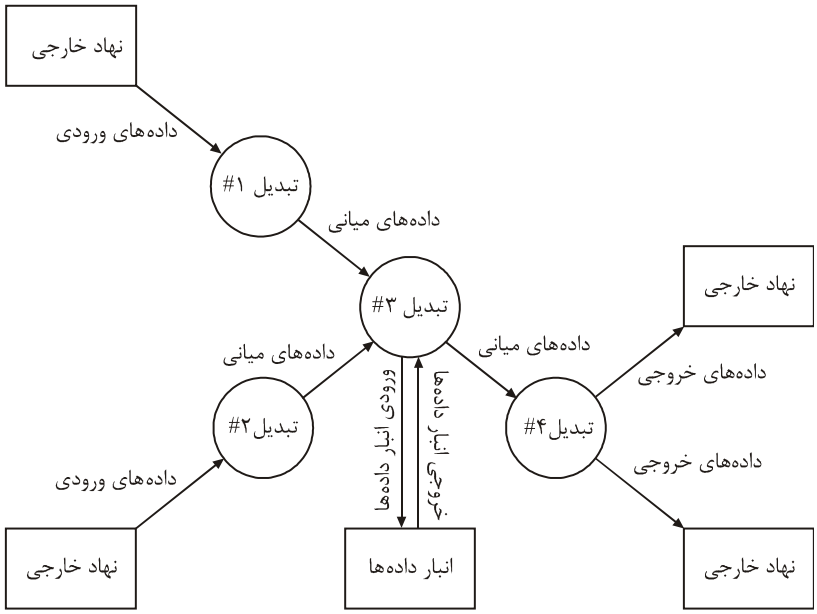
برای اینکه داده‌های خام ورودی به اطلاعات مفید خروجی تبدیل شوند، این داده‌ها در درون نرم‌افزار جریان پیدا می‌کنند و مجموعه‌ای از پردازش‌ها بر روی آن‌ها انجام می‌شود. هر پردازش یا باعث تغییر شکل اطلاعات شده یا اطلاعات را از بخشی به بخش دیگر منتقل می‌کند. بنابراین کارکردها و توابع نرم‌افزار، پردازش‌هایی هستند که انتقال یا تغییر شکل داده‌ها را پشتیبانی می‌کنند. مدل عملکردی یا کارکردی نرم‌افزار، ضمن نمایش پردازش‌های نرم‌افزار، جریان داده‌های نرم‌افزار را نیز نمایش می‌دهد.

به بیان دقیق‌تر، تمامی نرم‌افزارها مجموعه‌ای از داده‌ها را به عنوان ورودی پذیرفته، سپس مجموعه‌ای از پردازش‌ها را بر روی آن‌ها انجام داده و سپس آن‌ها را به عنوان اطلاعات خروجی

در اختیار موجودیت‌های خارج از سیستم قرار می‌دهند. در واقع در این مدل، سیستم کامپیوتری به عنوان یک **مبدل اطلاعات** نمایش داده می‌شود.

توجه: DFD یا Data Flow Diagram یا نمودار جریان داده جهت مدل‌سازی عملکردی مورد استفاده قرار می‌گیرد.

مدل شکل زیر، نمودار جریان داده نامیده می‌شود.



نمودار جریان داده

نهاد خارجی (موجودیت خارجی)

در این مدل برای نمایش نهادهای خارجی از یک مستطیل استفاده می‌شود (منظور از نهادهای خارجی، ورودی‌ها یا خروجی‌های داده‌ای هستند که می‌تواند شامل سخت‌افزار، افراد، برنامه‌های دیگر و یا سیستم‌های تبدیل اطلاعات دیگر باشد.) نهاد خارجی، جزیی از سیستم یا سیستمی دیگر است که اطلاعات مورد نیاز نرم‌افزار را تولید نموده یا اطلاعات تولید شده توسط نرم‌افزار را دریافت می‌کند.

پردازش یا حباب

برای نمایش حباب‌ها از یک دایره نام‌دار و معنادار استفاده می‌شود. دایره (فعل یا حباب)، پردازش و یا تبدیلی را که روی داده اعمال می‌شود، نشان می‌دهد. در واقع وظیفه تبدیل داده‌های ورودی به اطلاعات خروجی، توسط مجموعه‌ای از پردازش‌ها انجام می‌شود.

توجه: از آنجا که نمادهای گرافیکی نمی‌توانند تمامی جزئیات هر حباب را بیان کنند، جهت تشریح شرح حال هر حباب از مستنداتی به نام شرح حال پردازش یا مشخصات پردازش یا

PSPEC یا Process Specification استفاده می‌شود. مشخصات پردازش، به بیان روال انجام کار در حباب مورد نظر می‌پردازد، مواردی همچون، ورودی‌ها، خروجی‌ها و الگوریتم‌ها. مانند بیان روال عمل جمع دو عدد در حباب جمع.

توجه: عمل نوشتن PSPEC یا مشخصات پردازش هر حباب، توسط یک زبان مادری همچون فارسی یا انگلیسی انجام می‌گردد.

مثال: PSPEC یا مشخصات پردازش جمع دو عدد در مدل‌سازی عملکردی مدل تحلیل برای حباب جمع دو عدد، در DFD مربوطه، به صورت زیر است:

۱- شروع

۲- سه متغیر a، b و c را تعریف کن.

۳- دو عدد از ورودی خوانده و مقادیر آنها را در متغیرهای a و b قرار بده.

۴- مقادیر دو متغیر a و b را جمع کن و مقدار حاصل را در متغیر c قرار بده.

۵- مقدار متغیر c را در خروجی نمایش بده.

۶- پایان

توجه: در نمودار جریان‌داده ترتیب حباب‌ها مشخص نیست و فاقد هر گونه نمایش صریح برای توالی پردازش است. توالی را می‌توان به صورت ضمنی در این نمودار مشخص نمود.

توجه: شرح جزییات صریح منطقی (توالی، ساختارهای تکرار و ساختارهای شرط) هر حباب به معنی بیان روال انجام کار هر حباب توسط **شبه کد** یا **فلوچارت** تا زمان طراحی مولفه به عنوان بخشی از مدل طراحی نرم‌افزار به تعویق می‌افتد.

توجه: مدل طراحی و طراحی مولفه در فصل بعد تشریح خواهد شد.

جریان داده

برای نمایش جریان داده از خطوطی که دارای برچسبی نام‌دار و معنادار بر روی خود هستند استفاده می‌شود. خطوط برچسب‌دار با نام داده‌های مورد نظر نام‌گذاری می‌شوند. داده‌هایی که بر روی خطوط برچسب‌دار از حبابی به حباب دیگر به شکلی سیال در حال حرکت و جریان هستند همان داده‌هایی هستند که در بخش مدل‌سازی داده‌ای شناسایی شده‌اند.

توجه: از آنجا که نمادهای گرافیکی نمی‌توانند تمامی جزئیات هر جریان داده یا خطوط برچسب‌دار را بیان کنند، جهت تشریح شرح حال هر جریان داده یا خطوط برچسب‌دار از مستنداتی به نام شرح حال جریان داده یا شرح حال خطوط برچسب‌دار یا فرهنگ داده‌ها استفاده می‌شود. فرهنگ داده‌ها، به بیان جزئیات هر جریان داده یا خطوط برچسب‌دار مورد نظر می‌پردازد.

برای‌مثال بر روی خطوط برچسب‌دار از داده‌ها اسم برده می‌شود، اما این که دقیقاً این داده‌ها چه هستند مشخص نمی‌شود. برای این منظور از فرهنگ داده‌ها استفاده می‌شود که داده‌ها را به صورت دقیق تشریح می‌کند.

مخزن داده

برای نمایش ورودی و خروجی مابین مخزن داده یا منبع ذخیره‌سازی یا انبار داده یا پایگاه داده

و حباب (پردازش یا پردازه یا فرآیند) از خطوط مضاعف دو طرفه و بدون نام استفاده می‌شود.

توجه: بدون نام بودن خطوط مضاعف دو طرفه به این دلیل است که این خطوط مضاعف دو طرفه مانند یک کانال انتقال، اطلاعات مختلفی را ما بین حباب و منبع ذخیره‌سازی مبادله می‌کنند، بنابراین این تبادل اطلاعات، مختص به یک داده نام‌دار نمی‌باشد.

توجه: سادگی نشانه‌گذاری DFD یکی از دلایل گستردگی استفاده از این نمودار در مدل تحلیل ساخت‌یافته می‌باشد.

توجه: مخزن داده در نمودار سطح صفر DFD نمایش داده نمی‌شود. نمودار سطح صفر جلوتر شرح داده می‌شود.

سطوح نمودار DFD

با استفاده از DFD که گراف جریان داده یا چارت حبابی نیز نامیده می‌شود، می‌توان یک سیستم نرم‌افزاری را در سطوح مختلفی از انتزاع نمایش داد. در حقیقت این نمودار قابلیت افراز شدن به چند سطح را دارا است که با پیشروی در این سطوح، جریان اطلاعات و حباب‌ها را با جزییات بیشتر و انتزاع کمتری نشان می‌دهد.

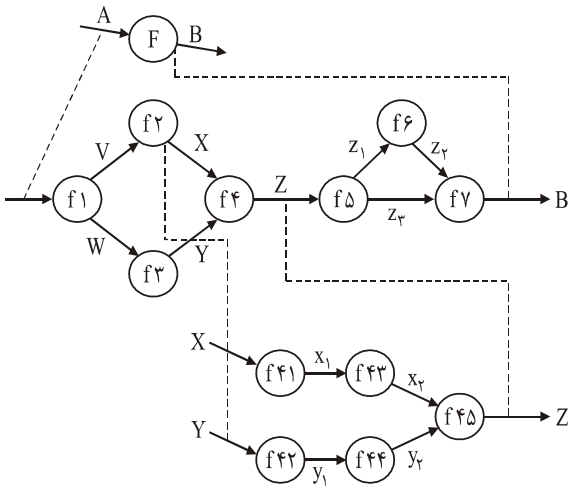
اولین نمودار جریان داده، DFD سطح صفر نام دارد که مدل بنیادی سیستم یا مدل بستر (Context Model) نیز نامیده می‌شود. در این سطح، کل سیستم به صورت فقط یک حباب همراه با ورودی و خروجی‌های آن و هم نام با نام برنامه کامپیوتری نشان داده می‌شود.

در سطح بعدی نمودار، هر کدام از حباب‌ها ممکن است به چندین حباب شکسته شود تا جزییات بیشتری را نشان دهد. هر کدام از حباب‌های نشان داده شده در سطح یک، یک زیر عملیات از سیستم کلی نشان داده شده در سطح صفر است.

برای تهیه DFD باید از پارسینگ گرامر استفاده شود. برای این منظور، تمامی فعل‌ها در داخل متن توضیحی مربوط به لیست نیازمندی‌ها به صورت حباب‌های برچسب‌دار در نظر گرفته می شوند. تمامی اسم‌ها نیز در داخل متن توضیحی مربوط به لیست نیازمندی‌ها به صورت نهادهای خارجی (مستطیل) یا جریان داده‌ای (خطوط برچسب‌دار) در نظر گرفته می‌شوند. علاوه بر این، اسامی و افعال ممکن است به هم وابسته باشند، بنابراین می‌توان از این اطلاعات برای پالایش بهتر یک حباب در DFD سطح بعد استفاده نمود. در تهیه DFD سطوح پایین‌تر باید پیوستگی اطلاعات (Information Continuity) حفظ گردد. بدین معنا که ورودی‌ها و خروجی‌های هر حباب در سطح فعلی و سطح بعدی یکسان باشد. به این خصوصیت بالانس کردن نیز گفته می‌شود و برای ساخت مدل‌های سازگار ضروری است. نتیجه این‌که هرچه در سطوح DFD پایین‌تر می‌رویم، جزییات حباب‌ها (پردازش‌ها) بیشتر مشخص می‌شوند. اما ورودی‌ها و خروجی‌های کل این مجموعه ثابت باقی می‌مانند.

توجه: در هر سطح فقط یک حباب باید تجزیه شود.

مثال:



این شکل یک مدل بنیادی برای سیستم F را نشان می‌دهد که دارای ورودی A و خروجی B به عنوان DFD سطح صفر می‌باشد.

این شکل در تبدیلات بعدی به F۱ تا F۷ افراز شده است در حالی که همچنان پیوستگی اطلاعات (Information Continuity) حفظ گشته است. یعنی ورودی و خروجی هر افراز بدون تغییر باقی مانده‌است. در این تبدیلات که نمودارهای DFD سطح یک هستند جزییات بیشتری از حباب‌ها نمایان شده است. در مرحله بعدی حباب F۴ در DFD سطح یک خود به چند حباب دیگر به نام‌های F۴۱ تا F۴۵ افراز گشته است. این نمودار DFD سطح دو نامیده می‌شود و روند تبدیلات داده‌ای را به صورت جزیی‌تری بررسی می‌کند.

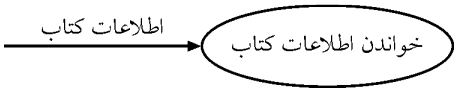
توجه: توصیه می‌شود، فرآیند تجزیه هر حباب به حباب‌های کوچکتر، تا زمانی که هر حباب یک عمل ساده را انجام می‌دهد، ادامه یابد.

توجه: همواره برای انتقال اطلاعات مابین دو نهاد خارجی، دو مخزن داده و یا یک نهاد خارجی و یک مخزن داده، نیاز به یک حباب (فرآیند یا پردازش) واسط است، تا عمل انتقال را انجام دهد.

مدل منطقی و مدل فیزیکی DFD

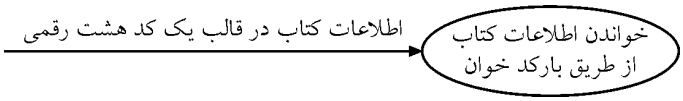
نمودارهای مدل منطقیDFD، کلی‌تر و انتزاعی‌تر هستند، و برای مدل تحلیل اولیه حباب‌ها مورد استفاده قرار می‌گیرند، بدون آنکه به جزئیات و روال انجام کار حباب‌ها بپردازند. اما نمودارهای مدل فیزیکی، جزئی‌تر و غیرانتزاعی‌تر هستند، و برای بیان جزئیات و نحوه انجام کار حباب‌ها مورد استفاده قرار می‌گیرند.

برای مثال در بخشی از یک نمودار منطقی DFD که برای یک کتاب‌فروشی مدل‌سازی شده است، می‌توان مدل زیر را مشاهده نمود:



توجه: در شکل فوق نحوه خواندن اطلاعات مشخص نشده است و می‌تواند به هر شیوه‌ای انجام شود.

در حالی که در بخشی از یک نمودار فیزیکی DFD که برای یک کتاب فروشی مدل‌سازی شده است، می‌توان مدل زیر را مشاهده نمود:



توجه: در شکل فوق نحوه خواندن اطلاعات دقیقا مشخص شده است، بدین نحو که اطلاعات کتاب باید از طریق بارکدخوان و در قالب یک کدهشت رقمی، خوانده شود.

کاربرد مدل منطقی و مدل فیزیکی DFD در توسعه سیستم‌های ساخته شده

هنگامی‌که قصد توسعه یک نرم‌افزار موجود و از پیش ساخته شده را داریم، آنچه در ابتدا با آن مواجه هستیم، نمودار فیزیکی DFD در وضعیت موجود نرم‌افزار است که برای شناخت وضع موجود سیستم فعلی مورد استفاده قرار می‌گیرد. حال اگر جزئیات مربوط به نحوه انجام کار حباب‌ها را حذف کنیم، آنگاه به نمودار منطقی DFD در وضعیت موجود نرم‌افزار می‌رسیم که همان مدل تحلیل موجود است. از آنجا که توسعه نرم‌افزار از پیش ساخته شده موجود را به عنوان یک هدف دنبال می‌کنیم، اگر به نمودار منطقی DFD در وضعیت موجود، نیازها و حباب‌های جدید را اضافه کنیم، آنگاه به نمودار منطقی DFD در وضعیت مطلوب می‌رسیم. که همان مدل تحلیل مطلوب است. در گام آخر، با بیان جزئیات و نحوه انجام کار هر یک از حباب‌ها در نمودار منطقی DFD در وضعیت مطلوب به نمودار فیزیکی DFD در وضعیت مطلوب می‌رسیم، که این مرحله، آماده ورود به مدل طراحی می‌باشد.

توجه: نمادهای استفاده شده در مدل منطقی و فیزیکی DFD یکسان است.

مدل‌سازی رفتاری

امروزه نرم‌افزارهای بی درنگ طیف وسیعی از برنامه‌های کامپیوتری را پوشش می‌دهند. در نرم‌افزارهای بی‌درنگ باید خروجی و پاسخ نهایی در یک زمان مشخص و از پیش تعیین شده حاصل شود. در این نرم‌افزارها، زمان نقشی کلیدی ایفا می‌کند و زمان پاسخ باید به موقع و تضمین شده باشد. نرم‌افزارهای بی‌درنگ معمولاً به عنوان یک دستگاه کنترلی در یک کاربرد خاص (مثلاً صنعتی) به کار گرفته می‌شوند. در این نرم‌افزارها دیر پاسخ دادن به همان بدی پاسخ ندادن است. در این نوع نرم‌افزارها هدف اصلی طراحان، پاسخگویی سریع(در مهلت تعیین شده) به رویدادها و درخواست‌ها می‌باشد و راحتی کاربران و بهره‌وری منابع در درجه‌های بعدی اهمیت، قرار دارند.

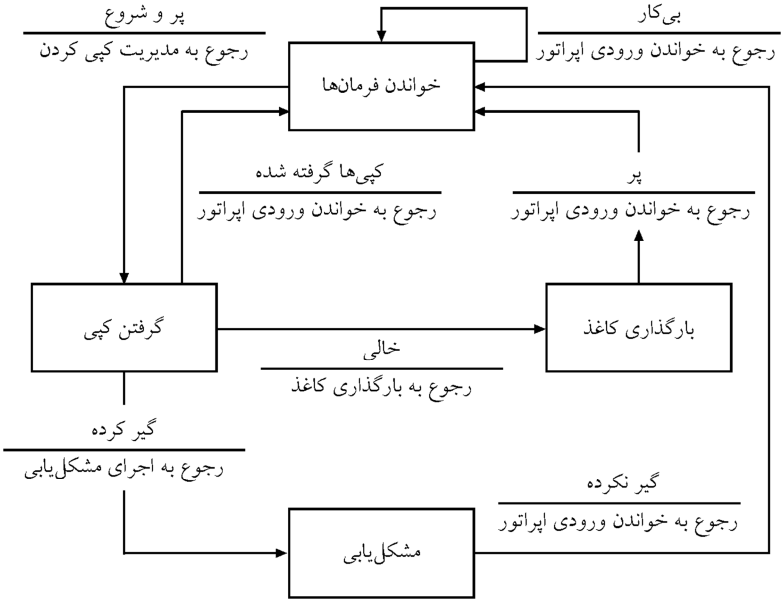
انسان در وادی زندگی نیازهای گوناگونی دارد، یکی از نیازهای اساسی انسان، نیاز به امنیت

است. اما گاهاً، ممکن است در معرض عوامل محیطی و بیرونی و یا حتی درونی، امنیت انسان در شرایط هشیاری یا ناهشیاری به مخاطره بیفتد. بنابراین نیاز است تا مکانیزمی همواره هوشیار و همیشه بیدار و با اشراف لحظه به لحظه مخاطرات پیرامون انسان را رصد و تحت کنترل خود قرار دهد تا در موقع لزوم و به صورت آنی، بی‌درنگ، در لحظه و در زمان حقیقی و واقعی (تا دیر نشده) با تهدید مقابله کند، نرم‌افزارهای بی‌درنگ این نگهبان همیشه هوشیار و همیشه بیدار هستند. مانند نرم‌افزارهای ترمز اتومبیل، کنترل ضربان قلب اتاق بیهوشی، کنترل فشار کابین هواپیما، دستگاه فتوکپی و ... .

توجه: STD یا State Transition Diagram یا نمودار انتقال حالت، جهت مدل‌سازی رفتاری در سیستم‌های بی‌درنگ مورد استفاده قرار می‌گیرد.

توجه: هر نرم‌افزاری که توسط سنسور و حسگر، عالم خارج را شنود و مورد ارزیابی قرار می‌دهد، تا در موقع لزوم و در زمان واقعی، حقیقی و تا دیر نشده عکس‌العمل نشان دهد، یک نرم‌افزار بی‌درنگ است.

نمودار انتقال حالت، نشان می‌دهد که چگونه سیستم برای پاسخگویی به رویدادها، در بین حالات خود گذر می‌کند. با وقوع هر رویداد، عموما اقدام خاصی نیز صورت گرفته و سیستم به یک حالت دیگر گذر می‌کند. منظور از حالت، هر رفتار قابل مشاهده از یک سیستم بی‌درنگ است، به بیان دیگر STD رفتار یک سیستم بی‌درنگ را با تصویر کردن حالات آن و روال‌ها و رویدادهایی که باعث تغییر حالت سیستم می‌شود، به نمایش می‌گذارد. شکل زیر نمودار انتقال حالت را برای یک دستگاه فتوکپی نشان می‌دهد. در این نمودار حالت‌های مختلفی که یک دستگاه فتوکپی می‌تواند داشته باشد، آورده شده‌اند.



در شکل فوق که یک نمودار STD ساده برای نرم‌افزار دستگاه فتوکپی می‌باشد، مستطیل‌ها بیانگر حالات مختلف سیستم و فلش‌ها نمایانگر تغییر حالت‌ها هستند. انتقال از یک حالت به حالت دیگر با برقراری یک شرط انجام می‌گردد. برای مثال اگر دستگاه در حال «**گرفتن کپی**» باشد و کاغذ تمام شود، به حالت «**بارگذاری کاغذ**» منتقل می‌شود. بنابراین روی هر خط انتقال، یک شرط و عملی که باید انجام گردد، نوشته می‌شود.

انتزاع یا تجرید (abstraction)

مدل تحلیل، مدل‌سازی عالم خارج به زبانی شبیه انسان است، اما مدل طراحی مدل‌سازی عالم داخل ماشین به زبانی شبیه زبان ماشین است. بنابراین برای اینکه ساخت برنامه کامپیوتری که به زبان ماشین است، امکان‌پذیر باشد، باید مدل تحلیل به مدل طراحی نگاشت شود تا مدل طراحی بتواند به عنوان نقشه راهی شبیه به زبان ماشین، راهگشا باشد.

مدل‌های تحلیل، کلی‌تر و انتزاعی‌تر هستند، و برای مدل‌سازی عالم خارج مورد استفاده قرار می‌گیرند، بدون آنکه به جزئیات نحوه پیاده‌سازی بپردازند، در واقع، مدل تحلیل به کلی‌گویی به زبان انسان و حذف جزئیات نحوه پیاده‌سازی می‌پردازد. اما مدل‌های طراحی، جزئی‌تر و غیرانتزاعی‌تر هستند، و برای مدل‌سازی عالم داخل ماشین مورد استفاده قرار می‌گیرند، و به بیان جزئیات نحوه پیاده‌سازی می‌پردازند، در واقع مدل طراحی به جزئی‌گویی به زبان شبیه ماشین و درج جزئیات نحوه پیاده‌سازی می‌پردازد.

با نگاهی سطح به سطح، به حل مساله، سطوح مختلفی از انتزاع را خواهیم داشت. در بالاترین سطح انتزاع، راه حل به صورت کلی به زبان محیط مساله بیان می‌گردد. در سطوح پایین‌تر، راه حل به جزئیات پیاده‌سازی نزدیک‌تر می‌شود و در پایین‌ترین سطح انتزاع راه حل به صورتی بیان می‌شود تا مستقیما قابل پیاده‌سازی باشد.

تجرید یا انتزاع، روش و نگرشی در ارائه و توضیح است که در آن فقط اطلاعات مهمی که برای حل یک مساله لازم است، گردآوری و نگهداری می‌شود و از بقیه اطلاعات صرف‌نظر می‌گردد، بنابراین تجرید یا انتزاع (abstraction)، صرفا یک روش و نگرش در ارائه و توضیح برای حل مساله است که ساخت برنامه‌های کامپیوتری را به سمت پیمانه‌ای‌کردن سوق می‌دهد.

به عنوان مثال، برای تماس الکترونیکی با یک دوست، کافی است آدرس پست الکترونیک و نام او را بدانید. برای برقراری این تماس، دانستن محل کار، قد، سن و علایق او لازم نیست. این اطلاعات اضافی، هرچند که درست و معتبر باشند، اما در برقراری تماس، کمکی نمی‌کنند. بنابراین باید حذف شوند. رعایت تجرید در فرآیند تولید یک نرم‌افزار، باعث کاهش شلوغی و پیچیدگی پروژه می‌گردد، زیرا از ذکر اطلاعات اضافی خودداری شده و افراد تیم توسعه مانند تحلیل‌گر تنها با اطلاعات ضروری سروکار خواهد داشت. برای مثال، در فرآیند تحلیل یک سیستم بانکداری، وقتی که در سطح اول انتزاع هستیم، اموری نظیر برداشت وجه، واریز وجه و غیره به صورت کلی مورد بررسی قرار می‌گیرند و باید از پرداختن به جزئیات، صرف نظر نمود. در سطوح پایین‌تر، جزئیات هر کدام توضیح داده می‌شود.

مثال: مکالمه محاوره‌ای زیر را در نظر بگیرید:
سوال: کجا هستی؟
پاسخ: بیرون
سوال: کی می‌یای؟
پاسخ: میام
همانطورکه مشاهده می‌کنید مکالمه فوق کاملا انتزاعی و با حذف جزئیات از طرف پاسخ‌دهنده انجام شده است. و برای سوال‌کننده همچنان سوالاتی نظیر کجای بیرون دقیقاً و چه زمانی دقیقاً، باقی‌مانده است، چون پاسخ‌ها کاملاً انتزاعی از طرف پاسخ‌دهنده بیان شده است.
انتزاع در مدل تحلیل در سه بخش قرار دارد:

انتزاع داده‌ای
انتزاعی‌که به کلی‌گویی در بخش مدل‌سازی داده‌ای می‌پردازد. برای‌مثال موجودیت «دانشجو»، در یک دانشگاه، بدون ذکر جزئیات. در حالی که در سطوح پایین‌تر انتزاع، می‌توان دانشجویان را به گروه‌هایی مثل، روزانه و نوبت دوم نیز تقسیم نمود.

انتزاع عملکردی یا رویه‌ای
انتزاعی که به کلی‌گویی در بخش مدل‌سازی عملکردی می‌پردازد. برای‌مثال حباب «ثبت نام دانشجو» در یک دانشگاه، بدون ذکر جزئیات.

انتزاع رفتاری یا کنترلی
انتزاعی که به کلی‌گویی در بخش مدل‌سازی رفتاری می‌پردازد. برای‌مثال حالت «کاهش فشار کابین هواپیما» در یک هواپیما، بدون ذکر جزئیات.

پالایش (refinement)
همانگونه که بیان شد، درک یکجای مساله به یکباره، اغلب غیرممکن است. بنابراین با استفاده از مفهوم انتزاع، ابتدا بر کلیت مساله تمرکز می‌شود، اما برای حل کامل مساله جزئیات آن نیز باید در نهایت بررسی شوند! فعالیت پالایش با تعیین ریز به ریز عملیات، جزئیات مساله را آشکار می‌کند. این فعالیت روندی بالا به پایین و سلسله مراتبی دارد، در واقع در سطوح پایینی پالایش، جزئیات واضح‌تر می‌شوند. هرچه به سطوح پایین‌تر سلسله مراتب نزدیک‌تر می‌شوید، جزئیات بیشتری از مساله آشکار می‌شود، تا جایی که در پایین‌ترین سطح پالایش، دستورات زبان برنامه‌نویسی ارائه می‌گردند.
انتزاع و پالایش دو مفهوم مکمل هستند. انتزاع، مهندسان نرم‌افزار را قادر می‌سازد تا داده‌ها، عملکردها و رفتارها را با حذف جزئیات تعریف کنند. در حالی که پالایش بر محتوای داخلی داده‌ها، عملکردها و رفتارها تمرکز داشته و جزئیات آنها را تشریح می‌کند. این تشریح به صورت لایه به لایه و در یک ساختار سلسله مراتبی و بالا به پایین است که با پیشروی در عمق لایه‌ها، جزئیات بیشتری آشکار می‌شود.
توجه: به پالایش، افراز یا بخش‌بندی(partitioning) نیز گفته می‌شود.

تست‌های فصل سوم

۱- کدام‌یک از عبارات زیر در رابطه با رسم نمودار جریان داده‌ها (data flow diagram) صحیح است؟ **(مهندسی IT – دولتی ۸۳)**

۱) برچسب هر جریان داده باید با یک فعل بیان گردد تا چگونگی تغییر شکل داده‌ها مشخص شود.

۲) برای نشان دادن حرکت داده‌ها بین دو موجودیت خارجی (external entity) می‌توان آن‌ها را به طور مستقیم توسط یک یا چندین جریان داده به یکدیگر متصل نمود.

۳) برای نشان دادن حرکت داده‌ها بین دو محل نگهداری داده‌ها (data store) می‌توان آن‌ها را به طور مستقیم توسط یک یا چند جریان داده به یکدیگر متصل نمود.

۴) برای این که موجودیت خارجی (external entity) بتواند داده‌ای را در محل نگهداری داده‌ها ذخیره کند، نیاز به فرآیندی می‌باشد که داده‌ها را از موجودیت خارجی دریافت کند و در محل نگهداری داده‌ها قرار دهد.

۲- کدام عبارت در مورد مدل منطقی و فیزیکی جریان داده برای یک سیستم نرم‌افزاری صحیح می‌باشد؟ **(مهندسی IT – دولتی ۸۴)**

۱) مدل منطقی جریان داده برای شناخت وضع موجود محیط نرم‌افزار در حال ایجاد به کار می‌رود.

۲) در مدل جریان داده منطقی از نمادهای متفاوتی نسبت به مدل جریان داده فیزیکی استفاده می‌شود.

۳) مدل جریان داده فیزیکی برای تحلیل نیازها و مدل جریان داده منطقی برای شناخت سیستم به کار می‌رود.

۴) مدل جریان داده فیزیکی برای شناخت وضع موجود و مدل جریان داده منطقی برای تحلیل نیازها استفاده می‌شود.

۳- کدام‌یک از موارد زیر به تحلیل‌گر این امکان را می‌دهد که وظیفه تحلیل نیازمندی‌ها را آسان‌تر و سیستماتیک‌تر انجام دهد؟ **(مهندسی IT – آزاد ۸۴)**

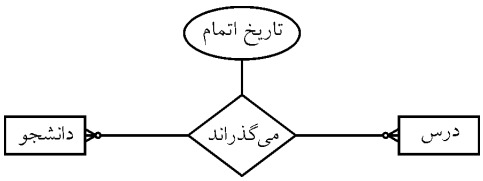
۱) مطالعه امکان‌پذیری ۲) مدل‌سازی

۳) طراحی ۴) ایجاد الگو

۴- کدام‌یک از ویژگی‌های زیر به شخص اجازه می‌دهد تا با مفاهیم و اصطلاحات آشنا با محیط مسئله کار کند، بدون اینکه مجبور به تبدیل آنها به یک ساختار ناآشنا گردد؟ **(مهندسی IT – آزاد ۸۴)**

۱) تجرید ۲) تعریف ۳) تحلیل ۴) طراحی

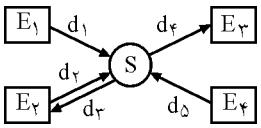
۵- کدام عبارت در مورد نمودار موجودیت – رابطه زیر درست است؟ (مهندسی IT – دولتی ۸۶)



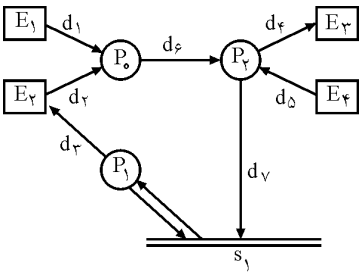
۱) رابطه «می‌گذارند» را می‌توان به یک موجودیت تبدیل کرد.
۲) با توجه به نمودار فوق، هر دانشجو بایستی حداقل یک درس را بگیرد.
۳) با توجه به نمودار فوق هر درس بایستی حداقل توسط یک دانشجو گرفته شود.
۴) هیچ‌کدام

۶- کدام عبارت در مورد نمودار سطح صفر و سطح یک سیستم S درست است؟
(مهندسی IT – دولتی ۸۶)

سطح صفر سیستم S



سطح یک سیستم S



۱) پردازه P1 یک مخزن داده است.
۲) جریان داده‌های d2 و d3 بایستی هر دو به یک موجودیت خارجی متصل باشند.
۳) پردازه P0 هیچ داده‌ای را به موجودیت خارجی نمی‌فرستد و این یک اشتباه است.
۴) ظهور انباره داده‌ای S1 در سطح یک نمودار، بدون آن که در سطح صفر نشان داده شود، خطا نیست.

۷- در یک نمودار جریان داده‌ای یک جریان داده‌ای دو طرفه و بدون نام میان یک پردازه و یک انباره‌ی داده‌ها ..... (مهندسی IT – دولتی ۸۷)

۱) یک اشتباه است.
۲) به معنای درج یک رکورد کامل در انباره‌ی داده‌هاست.
۳) به معنای حذف یک رکورد کامل در انباره‌ی داده‌هاست.
۴) به معنای خواندن و به روز درآوردن یک رکورد کامل در انباره داده‌هاست.

۸- کدام گزینه محتویات دیکشنری داده هر نرم‌افزار را بیان می‌نماید؟ (مهندسی IT – آزاد ۸۷)

۱) دیاگرام‌ها ۲) اشیاء داده‌ای ۳) مدل‌ها ۴) عناصر پیکربندی

۹- در تجزیه و تحلیل و طراحی سیستم با استفاده از ابزار DFD ترتیب اجرای کدام‌یک صحیح می‌باشد؟ (مهندسی IT – دولتی ۸۸)

۱) DFD منطقی سیستم جاری – DFD فیزیکی سیستم جاری – DFD منطقی سیستم آتی – DFD فیزیکی سیستم آتی

۲) DFD فیزیکی سیستم جاری – DFD منطقی سیستم جاری – DFD فیزیکی سیستم آتی – DFD منطقی سیستم آتی

۳) DFD فیزیکی سیستم جاری – DFD منطقی سیستم جاری – DFD منطقی سیستم آتی – DFD فیزیکی سیستم آتی

۴) DFD منطقی سیستم آتی – DFD منطقی سیستم جاری – DFD فیزیکی سیستم جاری – DFD فیزیکی سیستم آتی

۱۰- اهداف مدل تحلیل عبارتند از: (مهندسی IT – آزاد ۸۸)

۱) ۱- توصیف نیازهای مشتری ۲- برقراری مبنایی جهت ایجاد طراحی نرم‌افزار ۳- رسیدگی به نیازهای غیروظیفه‌مندی (Non functionality)

۲) ۱- رسیدگی به نیازهای وظیفه‌مندی (functionality) با توجه به چگونگی پیاده‌سازی آنها ۲- رسیدگی به نیازهای غیروظیفه‌مندی

۳) ۱- توصیف نیازهای مشتری ۲- برقراری مبنایی جهت ایجاد طراحی نرم‌افزار ۳- تعریف مجموعه‌ای از نیازها که پس از ساخته شدن نرم‌افزار قابل اعتبارسنجی باشند.

۴) ۱- رسیدگی به نیازهای وظیفه‌مندی (functionality) با توجه به چگونگی پیاده‌سازی آن‌ها ۲- برقراری مبنایی جهت ایجاد طراحی نرم‌افزار ۳- تعریف مجموعه‌ای از نیازها که پس از ساخته‌شدن نرم‌افزار قابل اعتبارسنجی باشند.

۱۱- در تحلیل ساخت‌یافته نمودار گذار حالت (STD) به مدل‌سازی کدام جنبه از سیستم می‌پردازد؟ (مهندسی IT – آزاد ۹۰)

۱) مدل‌سازی داده‌ای
۲) مدل‌سازی رفتاری
۳) مدل‌سازی عملکردی
۴) مدل‌سازی مؤلفه‌های سخت‌افزاری

۱۲- در نمودارهای جریان داده (DFD)، کدام‌یک از موارد زیر صادق است؟ (مهندسی IT – دولتی ۹۲)

۱) جریان داده باعث انتقال کنترل هم می‌شود.

۲) نمایش ذخیره و بازیابی داده‌ها مطرح نیست، بلکه فقط داده‌های در جریان، نشان داده می‌شوند.

۳) ترتیب اجرای فرآیندها (حباب‌ها) صریحاً مشخص نمی‌شود.

۴) برخی ورودی‌ها یا خروجی‌های یک فرآیند سطح بالا (غیربرگ) را می‌توان در نمودار متناظر با آن فرآیند (در سطح پایین‌تر) حذف نمود.

پاسخ تست‌های فصل سوم

۱- گزینه (۴) صحیح است.

در مدل DFD برای نمایش موجودیت‌های خارجی یا نهادهای خارجی از یک مستطیل استفاده می‌شود (منظور از نهادهای خارجی، ورودی‌ها یا خروجی‌های داده‌ای هستند که می‌تواند شامل سخت‌افزار، افراد، برنامه‌های دیگر و یا سیستم‌های تبدیل اطلاعات دیگر باشد.) نهاد خارجی، جزیی از سیستم یا سیستمی دیگر است که اطلاعات مورد نیاز نرم‌افزار را تولید نموده یا اطلاعات تولید شده توسط نرم‌افزار را دریافت می‌کند.

برای نمایش حباب‌ها از یک دایره نام‌دار و معنادار استفاده می‌شود. دایره (فعل یا حباب)، پردازش و یا تبدیلی را که روی داده اعمال می‌شود، نشان می‌دهد. در واقع وظیفه تبدیل داده‌های ورودی به اطلاعات خروجی، توسط مجموعه‌ای از پردازش‌ها انجام می‌شود.

برای نمایش جریان داده از خطوطی که دارای برچسبی نام‌دار و معنادار بر روی خود هستند استفاده می‌شود. خطوط برچسب‌دار با نام داده‌های مورد نظر نام گذاری می‌شوند. داده‌هایی که بر روی خطوط برچسب‌دار از حبابی به حباب دیگر به شکلی سیال در حال حرکت و جریان هستند همان داده‌هایی هستند که در بخش مدل‌سازی داده‌ای شناسایی شده‌اند.

برای نمایش ورودی و خروجی به مخزن داده یا منبع ذخیره‌سازی یا انبار داده یا پایگاه داده از خطوط مضاعف استفاده می‌شود.

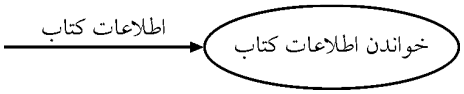
برای تهیه DFD باید از پارسینگ گرامر استفاده شود. برای این منظور، تمامی فعل‌ها در داخل متن توضیحی مربوط به لیست نیازمندی‌ها به صورت پردازش (حباب‌های برچسب‌دار) در نظر گرفته می‌شوند. تمامی اسم‌ها نیز در داخل متن توضیحی مربوط به لیست نیازمندی‌ها به صورت نهادهای خارجی (مستطیل) یا جریان داده‌ای (خطوط برچسب‌دار) در نظر گرفته می‌شوند. بنابراین گزینه اول نادرست است، زیرا برچسب هر جریان داده باید با یک اسم بیان گردد تا چگونگی تغییر شکل داده‌ها مشخص شود.

همواره برای انتقال اطلاعات مابین دو نهاد خارجی یا موجودیت خارجی، دو مخزن داده و یا یک نهاد خارجی و یک مخزن داده، نیاز به یک حباب (فرآیند یا پردازش) واسط است، تا عمل انتقال را انجام دهد. بنابراین گزینه دوم و سوم نیز نادرست و گزینه چهارم درست است.

۲- گزینه (۴) صحیح است.

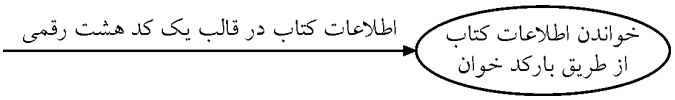
نمودارهای مدل منطقیDFD، کلی‌تر و انتزاعی‌تر هستند، و برای مدل تحلیل اولیه حباب‌ها مورد استفاده قرار می‌گیرند، بدون آنکه به جزئیات و روال انجام کار حباب‌ها بپردازند. اما نمودارهای مدل فیزیکیDFD، جزئی‌تر و غیرانتزاعی‌تر هستند، و برای بیان جزئیات و نحوه انجام کار حباب‌ها مورد استفاده قرار می‌گیرند.

برای مثال در بخشی از یک نمودار منطقی DFD که برای یک کتاب فروشی مدل‌سازی شده است، می‌توان مدل زیر را مشاهده نمود:



در شکل فوق نحوه خواندن اطلاعات مشخص نشده است و می‌تواند به هر شیوه‌ای انجام شود.

در حالی که در بخشی از یک نمودار فیزیکی DFD که برای یک کتاب فروشی مدل‌سازی شده است، می‌توان مدل زیر را مشاهده نمود:



در شکل فوق نحوه خواندن اطلاعات دقیقا مشخص شده است، بدین نحو که اطلاعات کتاب باید از طریق بارکدخوان و در قالب یک کدهشت رقمی، خوانده شود.

هنگامی که قصد توسعه یک نرم‌افزار موجود و از پیش ساخته شده را داریم، آنچه در ابتدا با آن مواجه هستیم، نمودار فیزیکی DFD در وضعیت موجود نرم‌افزار است که برای شناخت وضع موجود سیستم فعلی مورد استفاده قرار می‌گیرد. حال اگر جزئیات مربوط به نحوه انجام کار حباب‌ها را حذف کنیم، آنگاه به نمودار منطقی DFD در وضعیت موجود نرم‌افزار می‌رسیم که همان مدل تحلیل موجود است. از آنجا که توسعه نرم‌افزار از پیش ساخته شده موجود را به عنوان یک هدف دنبال می‌کنیم، اگر به نمودار منطقی DFD در وضعیت موجود، نیازها و حباب‌های جدید را اضافه کنیم، آنگاه به نمودار منطقی DFD در وضعیت مطلوب می‌رسیم. که همان مدل تحلیل مطلوب است. در گام آخر، با بیان جزئیات و نحوه انجام کار هر یک از حباب‌ها در نمودار منطقی DFD در وضعیت مطلوب به نمودار فیزیکی DFD در وضعیت مطلوب می‌رسیم، که این مرحله، آماده ورود به مدل طراحی می‌باشد.

نمادهای استفاده شده در مدل منطقی و فیزیکی DFD یکسان است.

۳- گزینه (۲) صحیح است.

مطالعه امکان‌پذیری به امکان‌سنجی انجام پروژه پیش از شروع فعالیت‌های چارچوبی مرتبط است، بنابراین گزینه اول نادرست است. مدل‌سازی در مدل تحلیل برای مدل‌سازی لیست نیازمندی‌های مشتری مورد استفاده قرار می‌گیرد. بنابراین گزینه دوم درست است. مدل طراحی پس از مدل تحلیل قرار دارد و طراحی معماری، طراحی داده، طراحی مولفه و طراحی واسط در این مدل انجام می‌گردد. ضمن اینکه مدل تحلیل به مدل طراحی کمک می‌کند، و نه مدل طراحی به مدل تحلیل. بنابراین گزینه سوم نیز نادرست است. ایجاد الگو یا الگوسازی یا نمونه‌سازی در فعالیت ارتباطات جهت شناسایی نیازمندی‌های مشتری مورد استفاده قرار می‌گیرد. بنابراین گزینه چهارم نیز نادرست است.

۴- گزینه (۱) صحیح است.

مدل تحلیل، مدل‌سازی عالم خارج به زبانی شبیه انسان است، اما مدل طراحی مدل‌سازی عالم

داخل ماشین به زبانی شبیه زبان ماشین است. بنابراین برای اینکه ساخت برنامه کامپیوتری که به زبان ماشین است، امکان‌پذیر باشد، باید مدل تحلیل به مدل طراحی نگاشت شود تا مدل طراحی بتواند به عنوان نقشه راهی شبیه به زبان ماشین، راهگشا باشد.

مدل‌های تحلیل، کلی‌تر و انتزاعی‌تر هستند، و برای مدل‌سازی عالم خارج مورد استفاده قرار می‌گیرند، بدون آنکه به جزئیات نحوه پیاده‌سازی بپردازند، در واقع مدل تحلیل به کلی‌گویی به زبان انسان و حذف جزئیات نحوه پیاده‌سازی می‌پردازد. اما مدل‌های طراحی، جزئی‌تر و غیرانتزاعی‌تر هستند، و برای مدل‌سازی عالم داخل ماشین مورد استفاده قرار می‌گیرند، و به بیان جزئیات نحوه پیاده‌سازی می‌پردازند، در واقع مدل طراحی به جزئی‌گویی به زبان شبیه ماشین و درج جزئیات نحوه پیاده‌سازی می‌پردازد.

با نگاهی سطح به سطح، به حل مساله، سطوح مختلفی از انتزاع را خواهیم داشت. در بالاترین سطح انتزاع، راه حل به صورت کلی به زبان محیط مساله بیان می‌گردد. در سطوح پایین‌تر، راه حل به جزئیات پیاده‌سازی نزدیک‌تر می‌شود و در پایین‌ترین سطح انتزاع راه حل به صورتی بیان می‌شود تا مستقیما قابل پیاده‌سازی باشد.

تجرید یا انتزاع، روش و نگرشی در ارائه و توضیح است که در آن فقط اطلاعات مهمی که برای حل یک مساله لازم است، گردآوری و نگهداری می‌شود و از بقیه اطلاعات صرف نظر می‌گردد، بنابراین تجرید یا انتزاع (abstraction)، صرفا یک روش و نگرش در ارائه و توضیح برای حل مساله است که ساخت برنامه‌های کامپیوتری را به سمت پیمانه‌ای کردن سوق می‌دهد.

به عنوان مثال، برای تماس الکترونیکی با یک دوست، کافی است آدرس پست الکترونیک و نام او را بدانید. برای برقراری این تماس، دانستن محل کار، قد، سن و علایق او لازم نیست. این اطلاعات اضافی، هرچند که درست و معتبر باشند، اما در برقراری تماس، کمکی نمی‌کنند. بنابراین باید حذف شوند. رعایت تجرید در فرآیند تولید یک نرم‌افزار، باعث کاهش شلوغی و پیچیدگی پروژه می‌گردد، زیرا از ذکر اطلاعات اضافی خودداری شده و افراد تیم توسعه مانند تحلیل‌گر تنها با اطلاعات ضروری سروکار خواهند داشت. برای مثال، در فرآیند تحلیل یک سیستم بانکداری، وقتی که در سطح اول انتزاع هستیم، اموری نظیر برداشت وجه، واریز وجه و غیره به صورت کلی مورد بررسی قرار می‌گیرند و باید از پرداختن به جزئیات، صرف نظر نمود. در سطوح پایین‌تر، جزئیات هر کدام توضیح داده می‌شود.

مثال: مکالمه محاوره‌ای زیر را در نظر بگیرید:

سوال: کجا هستی؟

پاسخ: بیرون

سوال: کی می‌یای؟

پاسخ: میام

همانطور که مشاهده می‌کنید مکالمه فوق کاملا انتزاعی و با حذف جزئیات از طرف پاسخ‌دهنده انجام شده است. و برای سوال‌کننده همچنان سوالاتی نظیر کجای بیرون دقیقا و چه

زمانی دقیقا، باقی‌مانده است، چون پاسخ‌ها کاملا انتزاعی از طرف پاسخ‌دهنده بیان شده است.

۵- گزینه (۱) صحیح است.

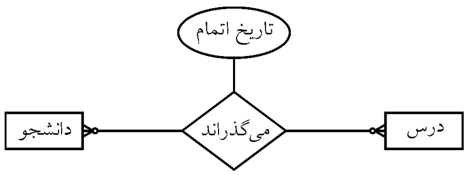
یک رابطه اجباری است، اگر و تنها اگر تمام نمونه‌های موجودیت در رابطه شرکت کرده باشند. اجباری بودن در نمودار ER با نماد خط مضاعف افقی یا نماد | به معنی یک و الزام شرکت در رابطه نشان داده می‌شود.

نماد خط مضاعف افقی نشانه اجباری بودن موجودیت چسبیده به آن است، اما نماد | به معنی یک و الزام شرکت در رابطه نشانه اجباری بودن موجودیت طرف مقابل است.

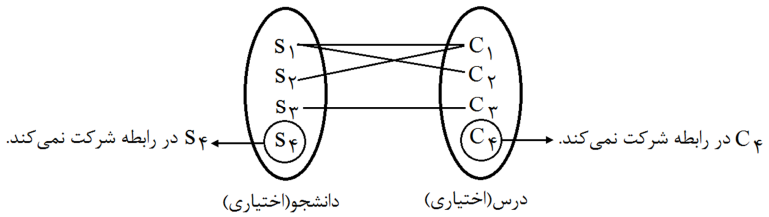
یک رابطه اختیاری است، اگر و تنها اگر حداقل یکی از نمونه‌های موجودیت در رابطه شرکت نکرده باشد. اختیاری بودن در نمودار ER با نماد خط افقی یا نماد دایره کوچک توخالی به معنی صفر و عدم الزام شرکت در رابطه نشان داده می‌شود.

نماد خط افقی نشانه اختیاری بودن موجودیت چسبیده به آن است، اما نماد دایره کوچک توخالی به معنی صفر و عدم الزام شرکت در رابطه نشانه اختیاری بودن موجودیت طرف مقابل است.

با توجه به نمودار، حضور موجودیت درس و دانشجو در رابطه اختیاری است.



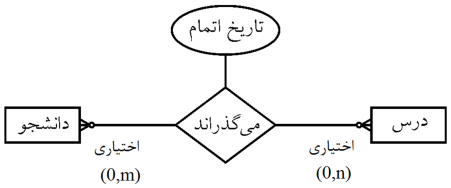
شکل زیر گویای نمودار ER است: (به‌طور مثال)



با توجه به ارتباط چند به چند دانشجو با درس و اختیاری بودن حضور نمونه موجودیت‌های دانشجو، قید (0,m) برای دانشجو در نظر گرفته می‌شود، بدین معنی‌که، یک دانشجو حداقل هیچ و حداکثر m درس دارد. بنابراین گزینه دوم نادرست است.

همچنین با توجه به ارتباط چند به چند درس با دانشجو و اختیاری بودن حضور نمونه موجودیت‌های درس، قید (0,n) برای درس در نظر گرفته می‌شود، بدین معنی‌که، یک درس حداقل هیچ و حداکثر n دانشجو دارد. بنابراین گزینه سوم نیز نادرست است.

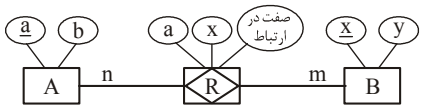
شکل زیر گویای مطلب است:



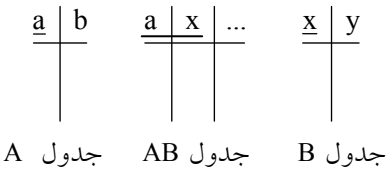
نگاشت رابطه چند به چند بین دو موجودیت (مدل تحلیل) به مدل رابطه‌ای (مدل طراحی)

مستقل از اختیاری یا اجباری بودن موجودیت‌ها، هر موجودیت به یک جدول تبدیل می‌گردد و یک جدول پُل(Bridge) نیز به عنوان ارتباط‌دهنده دو جدول مورد استفاده قرار می‌گیرد. همچنین کلید کاندید جدول پُل از ترکیب کلید کاندید دو جدول دیگر ایجاد می‌گردد. روال کلی به‌صورت زیر است:

مدل تحلیل:

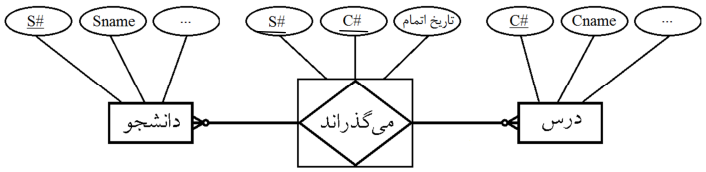


مدل طراحی:



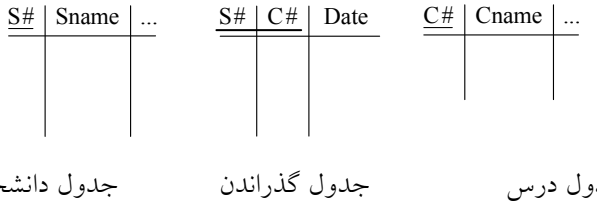
مثال: ارتباط درس و دانشجو.

مدل تحلیل:



توجه: صفت تاریخ اتمام، به‌عنوان یک صفت در ارتباط، تاریخ اتمام درس را مشخص می‌کند.

فعالیت طراحی:



۶- گزینه (۴) صحیح است.

با استفاده از DFD که گراف جریان داده یا چارت حبابی نیز نامیده می‌شود، می‌توان یک سیستم نرم‌افزاری را در سطوح مختلفی از انتزاع نمایش داد. در حقیقت این نمودار قابلیت افراز شدن به چند سطح را دارا است که با پیشروی در این سطوح، جریان اطلاعات و حباب‌ها (فرآیندها) را با جزییات بیشتر و انتزاع کمتری نشان می‌دهد.

اولین نمودار جریان داده، DFD سطح صفر نام دارد که مدل بنیادی سیستم یا مدل بستر (Context Model) نیز نامیده می‌شود. در این سطح، کل سیستم به صورت فقط یک حباب همراه با ورودی و خروجی‌های آن و هم‌نام با نام برنامه کامپیوتری نشان داده می‌شود.

در سطح بعدی نمودار، هر کدام از حباب‌ها ممکن است به چندین حباب شکسته شود تا جزییات بیشتری را نشان دهد. هر کدام از حباب‌های نشان داده شده در سطح یک، یک زیر عملیات از سیستم کلی نشان داده شده در سطح صفر است. مانند حباب‌های P0، P1 و P2 در DFD سطح یک. بنابراین گزینه اول نادرست است، زیرا P1 حباب (پردازه) است و نه مخزن داده.

در تهیه DFD سطوح پایین‌تر باید پیوستگی اطلاعات (Information Continuity) حفظ گردد. بدین معنا که ورودی‌ها و خروجی‌های هر حباب در سطح فعلی و سطح بعدی یکسان باشد. به این خصوصیت بالانس کردن نیز گفته می‌شود و برای ساخت مدل‌های سازگار ضروری است. نتیجه این که هر چه در سطوح DFD پایین‌تر می‌رویم، جزییات پردازش (تبدیل) بیشتر مشخص می‌شوند. اما ورودی‌ها و خروجی‌های کل این مجموعه ثابت باقی می‌مانند. بنابراین جریان داده‌های d2 و d3 بایستی هر دو به یک حباب متصل باشند و الزامی وجود ندارد که جریان داده‌های d2 و d3 بایستی هر دو به یک موجودیت خارجی متصل باشند، زیرا جریان داده‌های d2 و d3 می‌توانند از طریق یک حباب واسط به شکل غیرمستقیم با موجودیت‌های خارجی در ارتباط باشند. بنابراین گزینه دوم نادرست است.

الزامی وجود ندارد که یک حباب (پردازه) به طور مستقیم داده به سوی موجودیت خارجی بفرستد، بلکه یک حباب می‌تواند به طور غیرمستقیم و از طریق حباب‌های همکار خود داده به سوی موجودیت خارجی بفرستد. بنابراین گزینه سوم نادرست است.

مخزن داده در نمودار سطح صفر DFD نمایش داده نمی‌شود، مخزن داده در نمودار سطح یک و به بعد DFD نمایش داده می‌شود. بنابراین گزینه چهارم درست است.

۷- گزینه (۴) صحیح است.

برای نمایش ورودی و خروجی مابین مخزن داده یا منبع ذخیره‌سازی یا انبارداده یا پایگاه‌داده و حباب یا پردازش یا پردازه یا فرآیند، از خطوط مضاعف دوطرفه و بدون نام استفاده می‌شود.

بدون نام بودن خطوط مضاعف دو طرفه به این دلیل است که این خطوط مضاعف دوطرفه مانند یک کانال انتقال، اطلاعات مختلفی را مابین حباب و منبع ذخیره‌سازی مبادله می‌کنند، بنابراین این تبادل اطلاعات، مختص به یک داده نام‌دار نمی‌باشد.

۸- گزینه (۲) صحیح است.

از آنجا که نمادهای گرافیکی نمی‌توانند تمامی جزئیات مربوط به موجودیت‌ها یا اشیاء داده‌ای

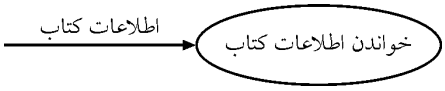
را بیان کنند، جهت تشریح شرح حال موجودیت‌ها یا اشیاء داده‌ای از مستنداتی به نام شرح حال موجودیت‌ها یا اشیاء داده‌ای یا فرهنگ داده‌ها (دیکشنری داده) استفاده می‌شود. فرهنگ داده‌ها، به بیان جزئیات مربوط به موجودیت‌ها یا اشیاء داده‌ای مورد نظر می‌پردازد.

برای‌مثال بر روی نماد مستطیل شکل موجودیت‌ها اسم برده می‌شود، اما این که دقیقاً این موجودیت‌ها یا اشیاء داده‌ای چه هستند مشخص نمی‌شود. برای این منظور از فرهنگ داده‌ها استفاده می‌شود که موجودیت‌ها یا اشیاء داده‌ای را به صورت دقیق تشریح می‌کند.

۹- گزینه (۳) صحیح است.

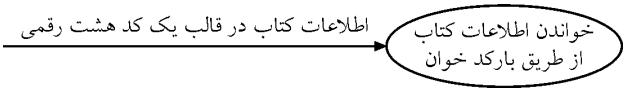
نمودارهای مدل منطقیDFD، کلی‌تر و انتزاعی‌تر هستند، و برای مدل تحلیل اولیه حباب‌ها مورد استفاده قرار می‌گیرند، بدون آنکه به جزئیات و روال انجام کار حباب‌ها بپردازند. اما نمودارهای مدل فیزیکیDFD، جزئی‌تر و غیرانتزاعی‌تر هستند، و برای بیان جزئیات و نحوه انجام کار حباب‌ها مورد استفاده قرار می‌گیرند.

برای مثال در بخشی از یک نمودار منطقی DFD که برای یک کتاب فروشی مدل‌سازی شده است، می‌توان مدل زیر را مشاهده نمود:



در شکل فوق نحوه خواندن اطلاعات مشخص نشده است و می‌تواند به هر شیوه‌ای انجام شود.

در حالی که در بخشی از یک نمودار فیزیکی DFD که برای یک کتاب فروشی مدل‌سازی شده است، می‌توان مدل زیر را مشاهده نمود:



در شکل فوق نحوه خواندن اطلاعات دقیقا مشخص شده است، بدین نحو که اطلاعات کتاب باید از طریق بارکدخوان و در قالب یک کدهشت رقمی، خوانده شود.

هنگامی که قصد توسعه یک نرم‌افزار موجود و از پیش ساخته شده را داریم، آنچه در ابتدا با آن مواجه هستیم، نمودار فیزیکی DFD در وضعیت موجود (جاری) نرم‌افزار است که برای شناخت وضع موجود سیستم فعلی مورد استفاده قرار می‌گیرد. حال اگر جزئیات مربوط به نحوه انجام کار حباب‌ها را حذف کنیم، آنگاه به نمودار منطقی DFD در وضعیت موجود(جاری) نرم‌افزار می‌رسیم که همان مدل تحلیل موجود است. از آنجا که توسعه نرم‌افزار از پیش ساخت شده موجود را به عنوان یک هدف دنبال می‌کنیم، اگر به نمودار منطقی DFD در وضعیت موجود، نیازها و حباب‌های جدید را اضافه کنیم، آنگاه به نمودار منطقی DFD در وضعیت مطلوب (آتی) می‌رسیم. که همان مدل تحلیل مطلوب است. در گام آخر، با بیان جزئیات و نحوه انجام کار هر یک از حباب‌ها در نمودار منطقی DFD در وضعیت مطلوب به نمودار فیزیکی DFD در وضعیت مطلوب (آتی) می‌رسیم، که این مرحله، آماده ورود به مدل طراحی می‌باشد.

نمادهای استفاده شده در مدل منطقی و فیزیکی DFD یکسان است.

۱۰- گزینه (۳) صحیح است.

یک مدل، ساده شده یک واقعیت است. ایجاد یک مدل برای سیستم‌های نرم‌افزاری قبل از ساخت یا بازساخت آن، به اندازه داشتن نقشه برای ساختن یک ساختمان ضروری وحیاتی است. بسیاری از شاخه‌های مهندسی، توصیف چگونگی محصولاتی که باید ساخته شوند را ترسیم می‌کنند و همچنین دقت زیادی می‌کنند که محصولاتشان طبق این مدل‌ها و توصیف‌ها ساخته شوند. مدل‌های خوب و دقیق در برقراری یک ارتباط کامل بین افراد پروژه، نقش زیادی می‌توانند داشته باشند. علت اصلی مدل کردن سیستم‌های پیچیده این است که نمی‌توان به یکباره کل سیستم را تجسم کرد و ممکن است سیستم دارای ابهامات بسیاری باشد. لذا برای رفع این ابهامات و نیز برای فهم کامل سیستم و یافتن و نمایش ارتباط بین قسمت‌های مختلف، از مدل‌سازی استفاده می‌شود. فعالیت مدل‌سازی خود شامل دو مرحله‌ی مدل تحلیل و مدل طراحی می‌باشد. مدل تحلیل پس از فعالیت ارتباطات (جمع‌آوری نیازمندی‌ها) و قبل از مدل طراحی انجام می‌شود، در واقع خروجی مدل تحلیل، ورودی مدل طراحی می‌باشد.

پس از جمع‌آوری نیازمندی‌ها در فعالیت ارتباطات نوبت به مدل تحلیل (مدل‌سازی لیست نیازمندی‌ها) می‌رسد. مدل‌سازی که فعالیتی فنی به شمار می‌رود نیازمندی‌ها را باید به گونه‌ای مدل نماید که برای سازنده و مشتری قابل فهم باشد.

مدل‌هایی که در فعالیت مدل‌سازی (بخش مدل تحلیل)، تهیه می‌شوند، سه هدف مهم را در تولید برنامه‌های کامپیوتری دنبال می‌کنند:

۱- توصیف نمادین نیازهای مشتری، توسط مدل‌سازی لیست نیازمندی‌ها، براساس نیازمندی‌های وظیفه‌مندی و هم نیازمندی‌های غیروظیفه مندی.
۲- مدل تحلیل، بستری مناسب را برای فعالیت طراحی نرم‌افزار ایجاد می‌کند.
۳- در فعالیت ساخت (بخش تست)، هنگامیکه اعتبارسنجی (validation) نرم‌افزار انجام می‌شود، مدل تحلیل، می‌تواند به عنوان یک تصویر از نرم‌افزار مورد انتظار، جهت فعالیت تست مورد استفاده قرار گیرد.

دقت کنید که، تهیه و مدل‌سازی لیست نیازمندی‌ها مستقل از نحوه پیاده‌سازی، انجام می‌گردد. نیازمندی‌های واقعی کاربران که مشخص شد، برای پیاده‌سازی آن نیز چاره اندیشیده می‌شود.

۱۱- گزینه (۲) صحیح است.

مدل تحلیل به روش ساخت‌یافته از سه بخش مدل‌سازی داده‌ای، مدل‌سازی عملکردی و مدل‌سازی رفتاری تشکیل شده است، مدل‌سازی داده‌ای شامل تحلیل موجودیت‌ها و تحلیل پرس و جوها می‌باشد. تحلیل موجودیت‌ها توسط ابزار مدل ER و تحلیل پرس و جوها توسط ابزار حساب رابطه‌ای مدل می‌شوند، مدل‌سازی عملکردی توسط ابزار DFD و مدل‌سازی رفتاری توسط ابزار STD مدل می‌شود.

STD یا State Transition Diagram یا نمودار انتقال حالت، جهت مدل‌سازی رفتاری در سیستم‌های بی‌درنگ مورد استفاده قرار می‌گیرد.

هر نرم‌افزاری که توسط سنسور و حسگر، عالم خارج را شنود و مورد ارزیابی قرار می‌دهد، تا در موقع لزوم و در زمان واقعی، حقیقی و تا دیر نشده عکس‌العمل نشان دهد، یک نرم‌افزار بی درنگ است.

نمودار انتقال حالت، نشان می‌دهد که چگونه سیستم برای پاسخگویی به رویدادها، در بین حالات خود گذر می‌کند. با وقوع هر رویداد، عموما اقدام خاصی نیز صورت گرفته و سیستم به یک حالت دیگر گذر می‌کند. منظور از حالت، هر رفتار قابل مشاهده از یک سیستم بی‌درنگ است، به بیان دیگر STD رفتار یک سیستم بی‌درنگ را با تصویر کردن حالات آن و روال‌ها و رویدادهایی که باعث تغییر حالت سیستم می‌شود، به نمایش می‌گذارد.

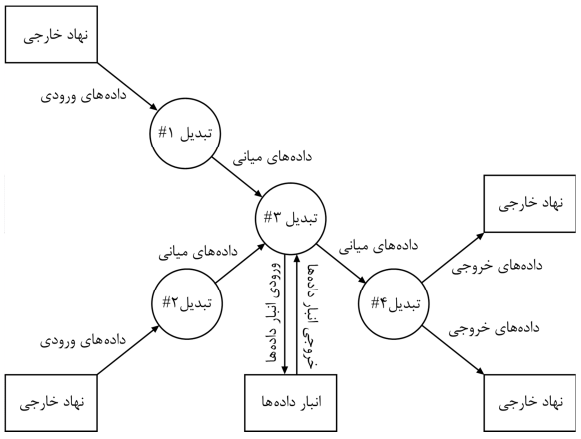
۱۲- گزینه (۳) صحیح است.

مدل‌سازی عملکردی یا مدل‌سازی جریان‌گرا، مجموعه‌ای از مدل‌های به نسبت قدیمی هستند که یکی از مهم‌ترین آن‌ها DFD است.

برای اینکه داده‌های خام ورودی به اطلاعات مفید خروجی تبدیل شوند، این داده‌ها در درون نرم‌افزار جریان پیدا می‌کنند و مجموعه‌ای از پردازش‌ها بر روی آن‌ها انجام می‌شود. هر پردازش یا باعث تغییر شکل اطلاعات شده یا اطلاعات را از بخشی به بخش دیگر منتقل می‌کند. بنابراین کارکردها و توابع نرم‌افزار، پردازش‌هایی هستند که انتقال یا تغییر شکل داده‌ها را پشتیبانی می‌کنند. مدل عملکردی یا کارکردی نرم‌افزار، ضمن نمایش پردازش‌های نرم‌افزار، جریان داده‌های نرم‌افزار را نیز نمایش می‌دهد.

به بیان دقیق‌تر، تمامی نرم‌افزارها مجموعه‌ای از داده‌ها را به عنوان ورودی پذیرفته، سپس مجموعه‌ای از پردازش‌ها را بر روی آن‌ها انجام داده و سپس آن‌ها را به عنوان اطلاعات خروجی در اختیار موجودیت‌های خارج از سیستم قرار می‌دهند. در واقع در این مدل، سیستم کامپیوتری به عنوان یک **مبدل اطلاعات** نمایش داده می‌شود.

DFD یا Data Flow Diagram یا نمودار جریان داده جهت مدل‌سازی عملکردی مورد استفاده قرار می‌گیرد. مدل شکل زیر، نمودار جریان داده نامیده می‌شود.



نمودار جریان داده

در این مدل برای نمایش نهادهای خارجی از یک مستطیل استفاده می‌شود (منظور از نهادهای خارجی، ورودی‌ها یا خروجی‌های داده‌ای هستند که می‌تواند شامل سخت‌افزار، افراد، برنامه‌های دیگر و یا سیستم‌های تبدیل اطلاعات دیگر باشد.) نهاد خارجی، جزیی از سیستم یا سیستمی دیگر است که اطلاعات مورد نیاز نرم‌افزار را تولید نموده یا اطلاعات تولید شده توسط نرم‌افزار را دریافت می‌کند.

برای نمایش حباب‌ها از یک دایره نام‌دار و معنادار استفاده می‌شود. دایره (فعل یا حباب)، پردازش و یا تبدیلی را که روی داده اعمال می‌شود، نشان می‌دهد. در واقع وظیفه تبدیل داده‌های ورودی به اطلاعات خروجی، توسط مجموعه‌ای از پردازش‌ها انجام می‌شود.

در نمودار جریان‌داده ترتیب حباب‌ها مشخص نیست و فاقد هر گونه نمایش صریح برای توالی پردازش است. توالی را می‌توان به صورت ضمنی در این نمودار مشخص نمود. بنابراین گزینه سوم درست است.

شرح جزییات صریح منطقی (توالی، ساختارهای تکرار و ساختارهای شرط) هر حباب به معنی بیان روال انجام کار هر حباب توسط شبه کد یا فلوچارت تا زمان طراحی مولفه به عنوان بخشی از مدل طراحی نرم‌افزار به تعویق می‌افتد.

برای نمایش جریان داده از خطوطی که دارای برچسبی نام‌دار و معنادار بر روی خود هستند استفاده می‌شود. خطوط برچسب‌دار با نام داده‌های مورد نظر نام‌گذاری می‌شوند. داده‌هایی که بر روی خطوط برچسب‌دار از حبابی به حباب دیگر به شکلی سیال در حال حرکت و جریان هستند همان داده‌هایی هستند که در بخش مدل‌سازی داده‌ای شناسایی شده‌اند. این داده‌ها یا جریان داده باعث انتقال کنترل از وضعیتی به وضعیتی دیگر نمی‌شود. تغییر وضعیت، مختص سیستم‌های بی-درنگ است. ضمن اینکه STD یا State Transition Diagram یا نمودار انتقال حالت، جهت مدل سازی رفتاری در سیستم‌های بی‌درنگ مورد استفاده قرار می‌گیرد و نه DFD. بنابراین گزینه اول نادرست است.

برای نمایش ورودی و خروجی مابین مخزن داده یا منبع ذخیره‌سازی یا انبار داده یا پایگاه داده و حباب (پردازش یا پردازه یا فرآیند) از خطوط مضاعف دو طرفه و بدون نام استفاده می‌شود. بنابراین گزینه دوم نیز نادرست است.

با استفاده از DFD که گراف جریان داده یا چارت حبابی نیز نامیده می‌شود، می‌توان یک سیستم نرم‌افزاری را در سطوح مختلفی از انتزاع نمایش داد. در حقیقت این نمودار قابلیت افراز شدن به چند سطح را دارا است که با پیشروی در این سطوح، جریان اطلاعات و حباب‌ها را با جزییات بیشتر و انتزاع کمتری نشان می‌دهد.

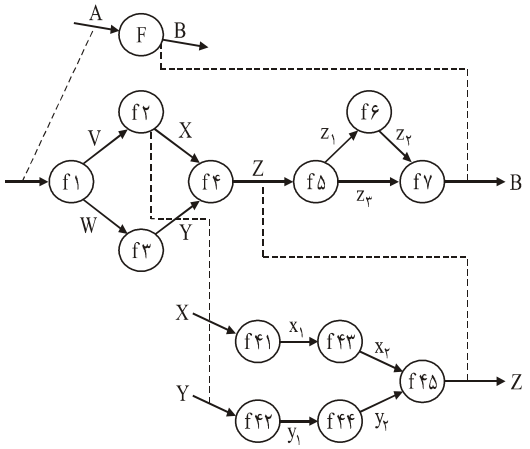
اولین نمودار جریان داده، DFD سطح صفر نام دارد که مدل بنیادی سیستم یا مدل بستر (Context Model) نیز نامیده می‌شود. در این سطح، کل سیستم به صورت فقط یک حباب همراه با ورودی و خروجی‌های آن و هم نام با نام برنامه کامپیوتری نشان داده می‌شود.

در سطح بعدی نمودار، هر کدام از حباب‌ها ممکن است به چندین حباب شکسته شود تا جزییات بیشتری را نشان دهد. هر کدام از حباب‌های نشان داده شده در سطح یک، یک زیر

عملیات از سیستم کلی نشان داده شده در سطح صفر است.

برای تهیه DFD باید از پارسینگ گرامر استفاده شود. برای این منظور، تمامی فعل‌ها در داخل متن توضیحی مربوط به لیست نیازمندی‌ها به صورت حباب‌های برچسب‌دار در نظر گرفته می شوند. تمامی اسم‌ها نیز در داخل متن توضیحی مربوط به لیست نیازمندی‌ها به صورت نهادهای خارجی (مستطیل) یا جریان داده‌ای (خطوط برچسب‌دار) در نظر گرفته می‌شوند. علاوه بر این، اسامی و افعال ممکن است به هم وابسته باشند، بنابراین می‌توان از این اطلاعات برای پالایش بهتر یک حباب در DFD سطح بعد استفاده نمود. در تهیه DFD سطوح پایین‌تر باید پیوستگی اطلاعات (Information Continuity) حفظ گردد. بدین معنا که ورودی‌ها و خروجی‌های هر حباب در سطح فعلی و سطح بعدی یکسان باشد. به این خصوصیت بالانس کردن نیز گفته می‌شود و برای ساخت مدل‌های سازگار ضروری است. نتیجه این‌که هرچه در سطوح DFD پایین‌تر می‌رویم، جزییات حباب‌ها (پردازش‌ها) بیشتر مشخص می‌شوند. اما ورودی‌ها و خروجی‌های کل این مجموعه ثابت باقی می‌مانند.

مثال:



این شکل یک مدل بنیادی برای سیستم F را نشان می‌دهد که دارای ورودی A و خروجی B به عنوان DFD سطح صفر می‌باشد.

این شکل در تبدیلات بعدی به F۱ تا F۷ افراز شده است در حالی که همچنان پیوستگی اطلاعات (Information Continuity) حفظ گشته است. یعنی ورودی و خروجی هر افراز بدون تغییر باقی مانده‌است. در این تبدیلات که نمودارهای DFD سطح یک هستند جزییات بیشتری از حباب‌ها نمایان شده است. در مرحله بعدی حباب F۴ در DFD سطح یک خود به چند حباب دیگر به نام‌های F۴۱ تا F۴۵ افراز گشته است. این نمودار DFD سطح دو نامیده می‌شود و روند تبدیلات داده‌ای را به صورت جزیی‌تری بررسی می‌کند. با توجه به مطالب فوق، گزینه چهارم نیز نادرست است.

۴ مفاهیم طراحی ساخت‌یافته

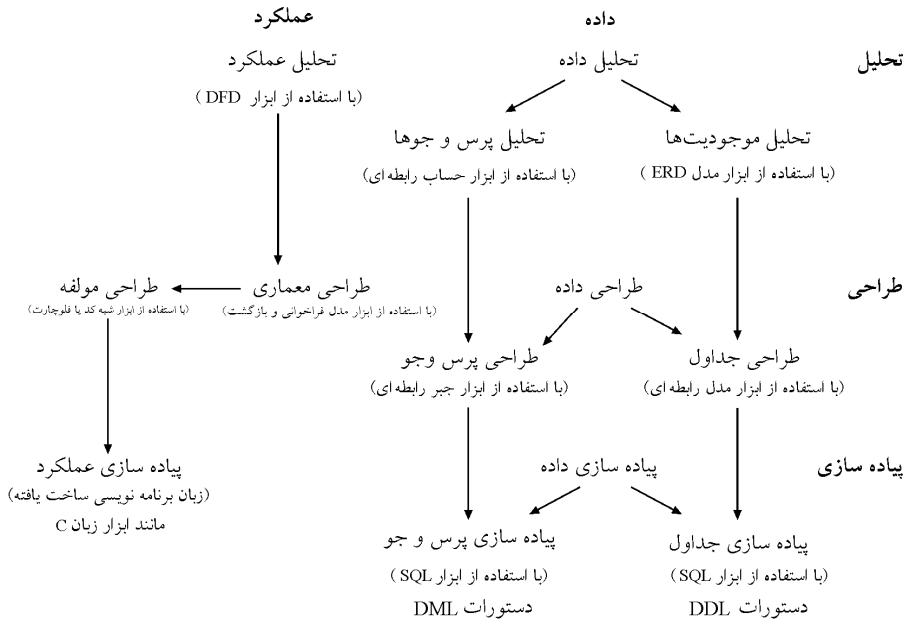
فعالیت‌های چارچوبی فرآیند تولید نرم‌افزار

به طورکلی فعالیت‌های مرتبط با فرآیند تولید نرم‌افزار صرف‌نظر از اندازه، پیچیدگی پروژه و زمینه‌ی کاربردی آن و مستقل از متدولوژی ساخت‌یافته و شیءگرا به پنج فعالیت ارتباط، برنامه‌ریزی، مدل‌سازی، ساخت و استقرار تقسیم می‌شود، به بیان دیگر فعالیت‌ها در هر دو دسته‌ی متدولوژی ساخت‌یافته و شیءگرا همین‌ها خواهند بود، اما کارهایی که در هر فعالیت در متدولوژی ساخت‌یافته و شیءگرا انجام می‌شود شباهت‌ها و تفاوت‌هایی خواهد داشت.

در ادامه فعالیت چارچوبی مدل‌سازی (مدل طراحی) از فرآیند تولید نرم‌افزار براساس متدولوژی ساخت‌یافته بیان خواهد شد:

مدل‌سازی (تحلیل و طراحی)

یک مدل، ساده شده یک واقعیت است. ایجاد یک مدل برای سیستم‌های نرم‌افزاری قبل از ساخت یا بازساخت آن، به اندازه داشتن نقشه برای ساختن یک ساختمان ضروری وحیاتی است. بسیاری از شاخه‌های مهندسی، توصیف چگونگی محصولاتی که باید ساخته شوند را ترسیم می‌کنند و همچنین دقت زیادی می‌کنند که محصولاتشان طبق این مدل‌ها و توصیف‌ها ساخته شوند. مدل‌های خوب و دقیق در برقراری یک ارتباط کامل بین افراد پروژه، نقش زیادی می‌توانند داشته باشند. علت اصلی مدل کردن سیستم‌های پیچیده این است که نمی‌توان به یکباره کل سیستم را تجسم کرد و ممکن است سیستم دارای ابهامات بسیاری باشد. لذا برای رفع این ابهامات و نیز برای فهم کامل سیستم و یافتن و نمایش ارتباط بین قسمت‌های مختلف آن، از مدل‌سازی استفاده می‌شود. فعالیت مدل‌سازی خود شامل دو مرحله‌ی مدل تحلیل و مدل طراحی می‌باشد. مدل تحلیل پس از فعالیت ارتباطات (جمع‌آوری نیازمندی‌ها) و قبل از مدل طراحی انجام می‌شود، در واقع خروجی مدل تحلیل، ورودی مدل طراحی می‌باشد. شکل زیر گویای این مطلب می‌باشد:



مدل طراحی

پس از مدل تحلیل، نوبت به مدل طراحی می‌رسد، پس از آنکه نیازهای نرم‌افزار شناخته شد و توسط مدل تحلیل، لیست نیازمندی‌های مشتری مدل‌سازی شد، باید برای پیاده‌سازی نرم‌افزار آماده شد. واضح است که گذر مستقیم به مرحله تولید کد و پیاده‌سازی عاقلانه نیست.

مانند معمار ساختمان‌سازی که پس از تهیه لیست نیازمندی‌های مشتری، بی درنگ به ساخت و پیاده‌سازی ساختمان بپردازد، بدون آنکه طرح و نقشه‌ای که به ادبیات زمین و ساختمان آشنا باشد، در دست داشته باشد!

مدل تحلیل، مدل‌سازی عالم خارج به زبانی شبیه انسان است، اما مدل طراحی مدل‌سازی عالم داخل ماشین به زبانی شبیه زبان ماشین است. بنابراین برای اینکه ساخت برنامه کامپیوتری که به زبان ماشین است، امکان‌پذیر باشد، باید مدل تحلیل به مدل طراحی نگاشت شود تا مدل طراحی بتواند به عنوان نقشه راهی شبیه به زبان ماشین، راهگشا باشد. مدل طراحی مانند دوربین عکاسی می‌باشد که شرایطی را فراهم می‌آورد که تصویر عالم خارج را در عالم دوربین و نگاتیو ذخیره‌سازی نماید. هر چه در عالم خارج باشد، عینا، اما در ابعادی کوچکتر یا بزرگتر در دوربین ذخیره می‌شود، در این نگاشت چیزی جا نمی‌ماند، مدل تحلیل خوب به مدل طرحی خوب و مدل تحلیل بد به مدل طراحی بد نگاشت می‌شود، یک مدل طراحی خوب از یک مدل تحلیل خوب و یک مدل طراحی بد از یک مدل تحلیل بد نشات گرفته است. اما این خیلی مهم است که بتوانید یک مدل تحلیل خوب را به یک مدل طراحی خوب نگاشت کنید. که این دیگر هنر طراح است!

کاپر، تعبیر جالبی را از مدل طراحی بیان نموده است، «مدل طراحی دقیقاً جایی است که

همزمان بر روی دو عالم ایستاده‌اید، عالم انسان و عالم ماشین، و باید تلاش نمود این دو عالم را به هم رساند.»

مدل طراحی به روش ساخت‌یافته شامل چهار بخش **طراحی داده**، **طراحی معماری**، **طراحی مؤلفه** و **طراحی واسط** می‌باشد.

طراحی داده

طراحی داده بر دو بخش طراحی جدول و طراحی پرس و جو می‌باشد. طراحی جدول از بخش طراحی داده، تحلیل موجودیت (ERD) از مدل تحلیل را به عنوان ورودی دریافت کرده و توسط مدل رابطه‌ای، طراحی جدول را انجام می‌دهد. طراحی پرس و جو از بخش طراحی داده، تحلیل پرس و جو از مدل تحلیل را به عنوان ورودی دریافت کرده و توسط جبر رابطه‌ای، طراحی پرس و جو را انجام می‌دهد.

توجه: در طراحی داده، فرهنگ داده‌های موجود در مدل تحلیل، مورد استفاده قرار می‌گیرد.

طراحی معماری

طراحی معماری، تحلیل عملکرد (DFD) از مدل تحلیل را به عنوان ورودی دریافت کرده و توسط سبک ساخت‌یافته (فراخوانی و بازگشت)، طراحی معماری را انجام می‌دهد.

در معنای عام، معماری به معنی نحوه ارتباط بخش‌های مختلف یک سازه است. در حیطه مهندسی نرم‌افزار نیز معماری به معنی نحوه ارتباط بخش‌های مختلف سازه‌ای به نام برنامه کامپیوتری است. طراحی معماری یا معماری نرم‌افزار، ساختار کلی نرم‌افزار و شیوه‌های یکپارچگی یک سیستم را بیان می‌کند. به عبارت دیگر، ساختار سلسله مراتبی **مولفه‌های برنامه (توابع یا پیمانه‌ها)**، شیوه تعامل مولفه‌ها با یکدیگر و **ساختمان داده‌های** مورد نیاز مولفه‌ها را نشان می‌دهد. معماری نرم‌افزار یک مدل قابل درک از چگونگی سازمان‌دهی سیستم است. در واقع نشان‌گر ساختمان داده‌ها و مؤلفه‌های برنامه‌ای است که برای ساختن یک سیستم کامپیوتری لازم است. به طور دقیق‌تر معماری نرم‌افزار شامل دو سطح از طراحی می‌باشد یعنی طراحی داده و طراحی معماری. در واقع این ساختار مانند یک نقشه ساختمان، مبنای ساخت نرم‌افزار قرار می‌گیرد.

توجه: در طراحی معماری، اسکلت، ساختار و چیدمان کلی مولفه‌های (توابع) برنامه به این معنی که چه مولفه‌ای (تابعی) چه مولفه‌ای (تابعی) دیگر را صدا می‌زند، بدون ذکر جزئیات داخلی مولفه‌ها (توابع) مشخص می‌گردد (ساختار درختی برنامه بدون ذکر جزئیات مولفه‌ها (توابع)). مانند اسکلت یک ساختمان که گویای جایگاه مؤلفه‌های ساختمان است اما هنوز آجرچینی نشده است. (اسکلت یک ساختمان بدون آجرچینی).

توجه: به طراحی معماری، طراحی کلی نیز گفته می‌شود.

یک طراحی معماری ایده‌آل باید خصیصه‌های زیر را رعایت نماید:

۱- طراحی معماری باید بر اساس پیمانه‌ای کردن نرم‌افزار ایجاد گردد که این امر منجر به

کنترل بهتر نرم‌افزار، بالا رفتن خوانایی برنامه، قابلیت استفاده مجدد بودن پیمانه‌ها و نگهداری آسان برنامه می‌گردد.

۲- اسکلت، ساختار و نحوه چیدمان پیمانه‌ها در کنار یکدیگر باید مشخص گردد.

۳- توابع و پیمانه‌های موجود در طراحی معماری باید براساس نیازمندی‌های وظیفه‌مندی و هم نیازمندی‌های غیر وظیفه‌مندی ایجاد گردد.

۴- طراحی معماری یک نرم‌افزار باید این قابلیت را داشته باشد تا بتواند با اندکی تغییر برای سیستم‌های آتی که هم دامنه و مشابه با پروژه فعلی هستند به کار گرفته شود. به عبارت دیگر طراحی معماری ایده‌آل باید پیمانه‌ها یا مولفه‌هایی را ایجاد نماید که قابلیت استفاده مجدد را در پرژه‌های آتی دارا باشد.

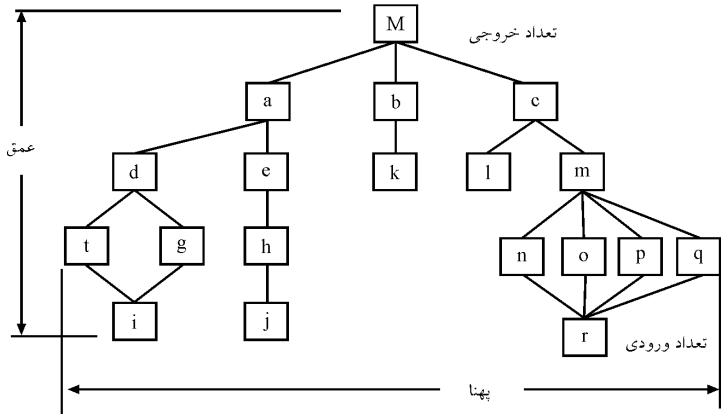
۵- طراحی معماری باید نگهداری نرم‌افزار در آینده را ساده سازد.

توجه: سبک معماری فراخوانی و بازگشت بدین معنی است که یک پیمانه، پیمانه‌ای دیگر را فراخوانی می‌کند و پس از طی روال پیمانه جاری، به دستور بعد از فراخوانی پیمانه حرکت می‌کند.

توجه: روابط میان مؤلفه‌ها می‌تواند فراخوانی یک رویه از پیمانه‌ای به پیمانه دیگر یا دستیابی به توابع پایگاه داده باشد.

سلسله مراتب کنترل در طراحی معماری

مدل ریاضی طراحی معماری، گراف است. بنابراین طراحی معماری خواص گراف را به ارث می‌برد. سبک معماری فراخوانی و بازگشت برای نمایش نحوه چیدمان پیمانه‌ها در طراحی معماری از سلسله مراتب کنترل استفاده می‌نماید. سلسله مراتب کنترل که ساختار برنامه نیز نام دارد در یک ساختار درختی، ترتیب قرارگیری پیمانه‌ها و نحوه سازمان‌دهی پیمانه‌ها (مولفه‌ها) را در طراحی معماری نرم‌افزار به سبک فراخوانی و بازگشت نشان می‌دهد. در سلسله مراتب کنترل یک برنامه، ساختار برنامه به این صورت است که پیمانه اصلی تعدادی پیمانه را فراخوانی می‌کند و آن‌ها نیز پیمانه‌های دیگری را فراخوانی می‌کنند. شکل زیر سلسله مراتب کنترل، در یک معماری به سبک فراخوانی و بازگشت را نشان می‌دهد:



رابطه کنترل بین پیمانه‌ها بدین صورت بیان می‌شود که اگر پیمانه‌ای، پیمانه‌ای دیگر را کنترل کند، بالاتر از آن قرار می‌گیرد و پیمانه «**پدر**» نام دارد و بر عکس اگر پیمانه‌ای، توسط پیمانه‌ای دیگر کنترل شود، پایین‌تر از آن قرار می‌گیرد و پیمانه «**فرزند**» نام دارد.

برای مثال، پیمانه M پدر پیمانه‌های a، b و c است و پیمانه h فرزند پیمانه e، نوه پیمانه a و نتیجه پیمانه M است.

عمق

نشان‌دهنده تعداد سطوح سلسله مراتب کنترل است.

مثال: عمق شکل فوق برابر عدد ۵ است.

پهنا

نشان‌دهنده گستردگی کلی سلسله مراتب کنترل است، به عبارت دیگر بیشترین تعداد پیمانه در یک سطح، پهنای سلسله مراتب کنترل را مشخص می‌کند.

مثال: پهنای شکل فوق برابر عدد ۷ و مربوط به سطح چهارم است.

توان خروجی (Fan out)

تعداد پیمانه‌هایی که به طور مستقیم توسط پیمانه دیگر کنترل می‌شود. به عبارت دیگر تعداد پیمانه‌های فرزند یک پیمانه، توان خروجی آن پیمانه را مشخص می‌کند.

مثال: توان خروجی پیمانه d برابر عدد ۲ است. زیرا پیمانه d، پیمانه‌های t و g را به طور مستقیم کنترل می‌کند.

توان ورودی (Fan in)

تعداد پیمانه‌هایی که به طور مستقیم یک پیمانه مشترک را کنترل می‌کنند. به عبارت دیگر تعداد پیمانه‌های پدر یک پیمانه، توان ورودی آن پیمانه را مشخص می‌کند.

مثال: توان ورودی پیمانه r برابر عدد ۴ است. زیرا پیمانه‌های n، o، p و q، پیمانه r را به طور مستقیم کنترل می‌کنند.

پیمانه‌سازی (modularity)

وقتی با یک مساله برنامه‌نویسی به زبان ساخت‌یافته مواجه می‌شوید، برای بالابردن خوانایی و کنترل بهتر برنامه این پرسش مطرح است که چگونه این مساله به توابع (پیمانه‌ها یا ماژول‌ها) تقسیم شود، که تعیین تعداد توابع به عهده مهندسان نرم‌افزار است. تعیین تعداد توابع از مدل تحلیل توسط حباب‌های موجود در نمودار DFD آغاز می‌شود، سپس این حباب‌ها توسط مدل طراحی به توابع برنامه نگاشت می‌شوند. اما این سوال نیز مطرح است که تعداد این توابع چقدر

باید باشد، به عبارت بهتر تقسیم برنامه به توابع تا کجا باید پیشروی کند.

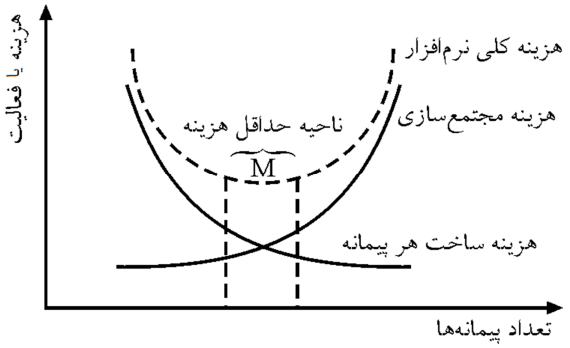
پیمانه‌ای کردن یعنی تقسیم نرم‌افزار به چند مولفه جداگانه و متمایز که این مولفه‌ها به عنوان پیمانه‌هایی هستند که از اجتماع آن‌ها، خواسته‌های مساله برآورده می‌شود. با این عمل مدیریت مفهومی یک برنامه حاصل می‌شود و قابلیت خوانایی نرم‌افزار چندین برابر می‌گردد. درک و فهم نرم‌افزار یکپارچه (monolithic) که تنها از یک پیمانه (ماژول) تشکیل شده است، بسیار مشکل است، زیرا تعداد متغیرها، حجم ارجاعات، تعداد مسیرهای کنترلی و پیچیدگی سراسری آن بسیار زیاد است. بنابراین اگر بتوان نرم‌افزار را به پیمانه‌هایی مناسب تقسیم نمود، پیچیدگی و هزینه کلی آن کاهش خواهد یافت. اما تعداد این پیمانه‌ها نباید بیش از حد باشد زیرا هزینه مجتمع‌سازی یا یکپارچه‌سازی و اتصال آن‌ها افزایش می‌یابد.

در یک نرم‌افزار با کاهش تعداد پیمانه‌ها (ماژول‌ها)، هزینه یکپارچه‌سازی آن‌ها کاهش می‌یابد. اما هزینه ساخت هر پیمانه افزایش می‌یابد، بنابراین هزینه کلی آن نیز که حاصل جمع دو مقدار هزینه یکپارچه‌سازی و هزینه ساخت هر پیمانه است، افزایش می‌یابد.

همچنین در یک نرم‌افزار با افزایش تعداد پیمانه‌ها (ماژول‌ها)، هزینه یکپارچه‌سازی آن‌ها افزایش می‌یابد. اما هزینه ساخت هر پیمانه کاهش می‌یابد، بنابراین هزینه کلی آن نیز که حاصل جمع دو مقدار هزینه یکپارچه‌سازی و هزینه ساخت هر پیمانه است، افزایش می‌یابد.

همچنین در یک نرم‌افزار با داشتن تعداد مناسب پیمانه (ماژول)، همه هزینه‌ها کاهش می‌یابد. زیرا هم هزینه یکپارچه‌سازی و هم هزینه ساخت هر پیمانه، هر دو کاهش می‌یابد، بنابراین هزینه کلی آن نیز که حاصل جمع دو مقدار هزینه یکپارچه‌سازی و هزینه ساخت هر پیمانه است، کاهش می‌یابد.

شکل زیر رابطه تعداد پیمانه‌های نرم‌افزار با هزینه توسعه نرم‌افزار را نشان می‌دهد. در این شکل، یکی از منحنی‌ها هزینه توسعه یک پیمانه را نشان می‌دهد و منحنی دیگر، هزینه یکپارچه‌سازی پیمانه‌ها را نشان می‌دهد. به تبع هزینه کلی نرم‌افزار برابر حاصل جمع این دو منحنی در هر نقطه است، که با منحنی خط‌چین نشان داده شده است.



نمودار فوق نشان می‌دهد که هزینه یا فعالیت لازم برای ساخت پیمانه‌های نرم‌افزاری با افزایش تعداد پیمانه‌ها الزاما کاهش می‌یابد ولی به موازات رشد بیش از حد پیمانه‌ها، هزینه مربوط به

اجتماع و یکپارچه‌سازی پیمانه‌ها نیز رشد می‌کند. در واقع همواره با تعداد مناسبی از پیمانه‌ها که با M نشان داده شده است، می‌توان هزینه و کار را کمینه کرد.

اما سوالی که مطرح است این است که حد بهینه تعداد پیمانه‌ها (M) چه مقدار است؟

در واقع لازم نیست که به طور تخصصی نیرو و هزینه‌ای برای به دست آوردن مقدار دقیق M صرف شود. بلکه با شروع به پیمانه‌ای کردن یک مساله، مهندس نرم‌افزار خود بر حسب دانش و تجربه‌ای که دارد به این نتیجه می‌رسد که تا چه حد پیمانه‌ای کردن لازم است. در واقع در یک مقداری در همسایگی M روند پیمانه‌ای کردن متوقف می‌شود. بنابراین این نتیجه حاصل می‌شود که باید بین تعداد پیمانه‌ها و هزینه‌های یکپارچه‌سازی تعادل برقرار شود.

توجه: وقتی با یک مساله برنامه‌نویسی به زبان شیءگرا مواجه می‌شوید، مانند روش‌های ساخت‌یافته دیگر این پرسش مطرح نیست که چگونه این مساله به چند تابع (پیمانه) تقسیم می‌شود بلکه این پرسش مطرح است که چگونه این مساله به چند کلاس (پیمانه) تقسیم می‌شود.

توجه: مفاهیم شیءگرایی در فصل شیءگرایی تشریح خواهد شد.

چگونه نرم‌افزاری موفق را خلق کنیم؟

برای پاسخ به این پرسش، ابتدا باید به این پرسش مهم پاسخ داد که چرا نرم‌افزار را خلق می‌کنیم؟

پاسخ: بله، برای رفع نیازهای انسان.

اما از نگاه انسان یک پروژه موفق نرم‌افزاری، پروژه‌ای است که بر اساس سه خصوصیت اساسی زیر تولید گردد:

۱- بازه‌ی زمانی از قبل برنامه‌ریزی شده (بازه‌ی زمانی مشخص)

۲- بودجه‌ای از قبل پیش‌بینی شده و با صرف کمترین هزینه (مقرون به صرفه)

۳- دقیقاً مطابق با نیازمندی‌های وظیفه‌مندی و غیروظیفه‌مندی کاربران (کیفیت مطلوب)

بنابراین برای رسیدن به خصوصیات پروژه‌های موفق نرم‌افزاری مطابق آنچه پیش از این نیز گفتیم باید از مهندسی نرم‌افزار استفاده گردد. در یک بیان کامل، مهندسی نرم‌افزار نظامی است یکپارچه شامل فرآیندها، روش‌ها و ابزارها که منجر به ایجاد نرم‌افزاری در بازه‌ی زمانی از قبل برنامه‌ریزی شده، بودجه‌ای از قبل پیش‌بینی شده و دقیقاً مطابق با نیازمندی‌های وظیفه‌مندی و غیروظیفه‌مندی کاربران می‌گردد. این تعریف مطابق آنچه پیش از این در مورد خصوصیات پروژه‌های موفق نرم‌افزاری بیان کردیم، تولید پروژه‌های نرم‌افزاری را به سمت موفقیت سوق می‌دهد.

حال این پرسش مطرح است، که مگر مهندسی نرم‌افزار چه کار می‌کند که پروژه‌های نرم‌افزاری را به سمت موفقیت سوق می‌دهد؟

پاسخ: در یک کلام ساده می‌توان گفت، مهندسی نرم‌افزار به فرآیند ساخت نرم‌افزار نظم و

ترتیب و ساختار می‌دهد. بدین معنی که چه کسی[1]، چه کاری را[2]، چه موقع[3] و چگونه[4] باید انجام دهد. در فعالیت ارتباط به ارتباط‌گر، می‌گوید توسط ابزارهایی همچون گفتگو، مشاهده و مصاحبه به تهیه لیست نیازمندی‌های وظیفه‌مندی و غیروظیفه‌مندی مشتری بپرداز، در گام بعدی یعنی فعالیت مدل تحلیل به تحلیل‌گر می‌گوید به مدل‌سازی لیست نیازمندی‌های مشتری بپرداز، همچنین برای غلبه بر پیچیدگی‌های مساله و در نهایت حل مساله به صورت موفق به عنوان هدف نهایی، ابتدا در این مدل با حذف جزئیات به کلی‌گویی و انتزاعی بودن بپرداز، یعنی مساله را به زبان انسان مدل کن، اما کم‌کم کم‌کم در فعالیت‌های بعدی، مدل تحلیل را به مدل طراحی و در نهایت به پیاده‌سازی و زبان ماشین نزدیک کن. پس از آنکه در مدل تحلیل یک دید کلی از نرم‌افزار ایجاد شد، سپس پالایش بخش‌های مختلف نرم‌افزار همچون داده‌ها و عملکردها آغاز می‌شود. برای بالابردن خوانایی و کنترل بهتر برنامه، مهندسی نرم‌افزار توصیه کرده است که پالایش پیمانه اصلی برنامه را با پیمانه‌ای کردن پیش ببرید. بدین نحو که پیمانه اصلی برنامه به چند پیمانه دیگر تقسیم شود و این روال تا حدی مطلوب ادامه پیدا کند. در مدل تحلیل این کار توسط نمودار DFD آغاز می‌شود.

مدل تحلیل، مدل‌سازی عالم خارج به زبانی شبیه انسان است، اما مدل طراحی مدل‌سازی عالم داخل ماشین به زبانی شبیه زبان ماشین است. بنابراین برای اینکه ساخت برنامه کامپیوتری که به زبان ماشین است، امکان‌پذیر باشد، باید مدل تحلیل به مدل طراحی نگاشت شود تا مدل طراحی بتواند به عنوان نقشه راهی شبیه به زبان ماشین، راهگشا باشد. در مدل طراحی، طراحی معماری به عنوان ساخت اسکلت و ساختار برنامه انجام می‌شود. ساختاری بر پایه پیمانه‌ها، آنچه از مدل تحلیل به مدل طراحی به ارث رسیده است. در مدل طراحی، حباب‌های موجود در مدل تحلیل در نمودار DFD به پیمانه‌ها (توابع) نگاشت می‌شوند.

مستقل از متدولوژی ساخت‌یافته و شیءگرا، جدا از اینکه پیمانه‌ای کردن نرم‌افزار منجر به **خوانایی بالاتر برنامه** و **کنترل بهتر برنامه** در حال ساخت می‌گردد، پیمانه‌ای‌کردن همچنین این خاصیت را فراهم می‌آورد تا در صورتی که پیمانه‌هایی با رعایت **اصول طراحی معماری ایده‌آل** ایجاد گردند، این پیمانه‌ها هم در نرم‌افزار فعلی دارای خاصیت **نگهداری آسان** باشند و هم در نرم‌افزارهای آتی خاصیت **قابلیت استفاده مجدد** را دارا باشند تا ساخت پروژه‌های نرم‌افزاری آتی با صرفه جویی در هزینه‌ها مقرون به صرفه گردد. زمان و هزینه‌ی فرآیند تولید نرم‌افزار با استفاده از قطعات آماده (قابلیت استفاده مجدد) کاهش چشمگیری دارد. در واقع یک قطعه با قابلیت استفاده‌ی مجدد یک بار ساخته می‌شود و بارها و بارها استفاده می‌شود و این یعنی سودآوری! بنابراین مهندسی نرم‌افزار برای ساخت پیمانه‌هایی ایده‌آل و خوب برای معماری نرم‌افزار در حال ساخت و همچنین قطعات و پیمانه‌هایی آماده با قابلیت استفاده مجدد برای نرم‌افزارهای‌های آتی،

[1] Who
[2] What
[3] When
[4] How

اصول **پنهان‌سازی اطلاعات، استقلال عملیاتی، افزایش انسجام** و **کاهش اتصال** را به عنوان **اصول طراحی معماری ایده‌آل** توصیه کرده است. در نهایت این اسکلت و ساختار و پیمانه‌های ایده‌آل که از ابتدا با پیروی از اصول مهندسی نرم‌افزار به صورت درست و اصولی پایه گذاری شده است، پس از حرکت به سمت طراحی مولفه و گذر از آن، در گام آخر یعنی پیاده‌سازی، در نهایت به سمت ساخت یک پروژه موفق نرم‌افزاری حرکت می‌کند و خصوصیات موفق پروژه‌های نرم‌افزاری را با خود به ارمغان می‌آورد. واین یعنی موفقیت. **موفقیت تصادفی نیست!**

پنهان‌سازی اطلاعات

در ساخت‌یافتگی، پنهان‌سازی اطلاعات (information hiding) نتیجه پیمانه‌ای کردن است. به بیان دیگر شرط لازم برای برقراری پنهان‌سازی اطلاعات، پیمانه‌ای کردن است و شرط کافی برای برقراری پنهان‌سازی اطلاعات تعریف متغیرهای محلی و دستورالعمل‌های مرتبط با تابع است. در یک بیان ساده پنهان‌سازی اطلاعات می‌گوید بخشی از نرم‌افزار در داخل یک پیمانه (تابع) محصور شود. هدف از پنهان‌سازی اطلاعات، پنهان کردن روال انجام دستورات تابع و متغیرهای محلی در پس واسط یا بلاک تابع است. استفاده‌کنندگان پیمانه‌ها (توابع) نیازی به دانستن جزئیات داخلی پیمانه‌ها (توابع) ندارند.

استقلال عملیاتی

در ساخت‌یافتگی، استقلال عملیاتی، نتیجه پیمانه‌ای‌کردن و پنهان‌سازی اطلاعات است. به بیان دیگر شرط لازم برای برقراری استقلال عملیاتی، پیمانه‌ای‌کردن و پنهان‌سازی اطلاعات است و شرط کافی برای برقراری استقلال عملیاتی انسجام بالا و اتصال پایین است. در صورتی که پیمانه‌هایی را با عملکرد تک‌منظوره(انسجام بالا) و عدم ارتباط بیش از حد با پیمانه‌های دیگر (اتصال پایین) ایجاد کنیم، استقلال عملیاتی تحقق می‌یابد. که این موارد در ساخت‌یافتگی به طور ذاتی محقق نشده است، و طراحان و برنامه‌نویسان باید در جهت تحقق این موارد کوشا باشند. که این امر منجر به عدم **چابک‌بودن** روند ساخت پروژه‌های ساخت‌یافته می‌شود.

توجه: در مورد مفهوم **چابکی** در فصل متدولوژی‌های چابک صحبت خواهیم کرد.

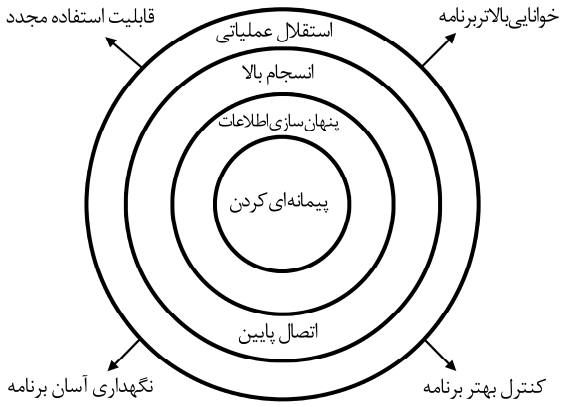
توجه: استقلال عملیاتی خوب، کلید طراحی خوب و طراحی خوب، کلید یک نرم‌افزار با کیفیت می‌باشد. استقلال عملیاتی خوب با دو مفهوم انسجام (Cohesion) و اتصال (Coupling) مورد ارزیابی قرار می‌گیرد. انسجام، معیاری است که توان نسبی کارکردی یک پیمانه را نشان می‌دهد و اتصال، معیاری است که میزان نسبی وابستگی پیمانه‌ها به یکدیگر را نشان می‌دهد.

توجه: در یک طراحی ایده‌آل، هدف، محقق کردن بالاترین سطح انسجام (Cohesion بالا) داخل پیمانه‌های برنامه و کمترین سطح اتصال (Coupling پایین) مابین پیمانه‌های برنامه است.

توجه: پیمانه‌ای‌کردن، برقراری پنهان‌سازی اطلاعات، انسجام بالا، اتصال پایین و برقراری استقلال عملیاتی، به عنوان اصول و معیارهای طراحی معماری مطلوب، منجر به **خوانایی بالاتر برنامه، کنترل بهتر برنامه، قابلیت استفاده مجدد** پیمانه‌های (توابع) برنامه جاری در برنامه‌های آتی

و **نگهداری آسان برنامه** جاری می‌گردد.

شکل زیر گویای مطلب است:



انسجام (Cohesion)

انسجام یا همبستگی، توسعه مفهوم پنهان‌سازی اطلاعات است، یک پیمانه یکپارچه و منسجم، یک وظیفه منفرد را انجام می‌دهد. انسجام، یک خصیصه مثبت در طراحی نرم‌افزار بوده که هرچه میزان آن بیشتر باشد، طراحی معماری از کیفیت بالاتری برخوردار است. انسجام در ساخت‌یافتگی به معنی درجه وظایف مختلف و غیرمرتبط داخل یک پیمانه یا تابع است. در یک گروه ارکستر سمفونیک، سازهای مختلفی نواخته می‌شود اما همه مرتبط با هم و حول یک هدف خاص نواخته می‌شوند. در واقع گروه ارکستر سمفونیک از همبستگی و انسجام بالایی برخوردار است. بنابراین درجه‌ی همبستگی این گروه برابر مقدار یک می‌باشد زیرا همه اعضاء حول یک محور و وظیفه کار می‌کنند. هر چه درجه وظایف مختلف و غیرمرتبط یک پیمانه پایین‌تر باشد، پیمانه از انسجام بالاتری برخوردار است. انسجام، به میزان همبستگی میان محتویات داخل یک پیمانه یا تابع می‌پردازد. به بیان دیگر داخل یک پیمانه یا تابع چه خبر است. هرچه انسجام پیمانه‌ها بیشتر باشد، طراحی معماری بهتری داریم. انسجام در ساخت‌یافتگی را می‌توان در قالب یک طیف از پایین‌ترین سطح انسجام تا بالاترین سطح انسجام طبقه‌بندی نمود. در طراحی معماری، همواره برای بالاترین سطح انسجام تلاش می‌شود. در ادامه سطوح مختلف انسجام در ساخت‌یافتگی از بالاترین سطح انسجام (مطلوب‌ترین) تا پایین‌ترین سطح انسجام (نامطلوب‌ترین) رتبه‌بندی شده است:

۱- انسجام تابعی یا عملیاتی یا کارکردی (Functional Cohesion)

اگر درجه تعداد وظایف یا مسئولیت‌های مختلف یک پیمانه برابر عدد یک باشد، آن پیمانه دارای انسجام عملیاتی است. در این نوع انسجام، پیمانه مورد نظر دقیقا یک عمل و مسئولیت را انجام داده و یک هدف واحد را دنبال می‌کند. این نوع انسجام بهترین سطح انسجام را ارائه می‌کند. به این پیمانه‌ها، پیمانه‌های مصمم (single minded module) نیز می‌گویند. این سطح از انسجام، بالاترین سطح انسجام و سطح مطلوب است.

مثال: تابع جمع دو عدد.

```
#include <stdio.h>
void sum(void);
int main()
{
sum();
return 0;
}
sum(void)
{
int a,b,r;
scanf("%d %d", &a , &b);
r=a+b;
printf("%d", r);
}
```

توجه: پیمانه یا تابع sum() فقط یک وظیفه یا مسئولیت و آن هم عملیات جمع دو عدد را انجام می‌دهد، به عبارت دیگر درجه تعداد وظایف یا مسئولیت‌های مختلف این پیمانه برابر عدد یک می‌باشد، بنابراین این پیمانه دارای انسجام عملیاتی است.

۲- انسجام ترتیبی یا متوالی (Sequential Cohesion)

این نوع از انسجام زمانی رخ می‌دهد که وظایف طوری در یک پیمانه مجتمع شوند که اجرای پشت سرهم آنها ملاک باشد. هدف اصلی این انسجام پشت سرهم انجام شدن وظایف مرتبط به هم است. در این نوع انسجام، خروجی یک وظیفه، ورودی وظیفه دیگر است.

مثال: انجام وظیفه جمع ۱۰ عدد و وظیفه محاسبه میانگین ۱۰ عدد جمع شده از وظیفه قبل در یک تابع یا پیمانه.

```
#include <stdio.h>
void fun(void);
int main()
{
fun();
return 0;
}
fun(void){
int i, n, sum=0, avg;
for(i=1; i<=10; i++)
{
scanf("%d", &n);
sum=sum+n;
}
avg=sum / 10;
printf("%d",avg );
}
```

۳- انسجام رویه‌ای (Procedural Cohesion)

این نوع از انسجام زمانی رخ می‌دهد که وظایف طوری در یک پیمانه مجتمع شوند که اجـرای پشت سرهم آنها ملاک باشد. هدف اصلی این انسجام پشت سـرهم انجـام شـدن وظـایف مـرتبط است. در این نوع انسجام، لزومی ندارد خروجی یک وظیفه، ورودی وظیفه دیگر باشد. به عبـارت دیگر این انسجام زمانی رخ می‌دهد که یک سری وظیفه همواره رویه مشخصی از اجرا را به شکلی مرتبط دنبال کنند.

مثال: انجام وظیفه مقداردهی یک ماتریس ۱۰۰ *۱۰۰ و وظیفـه مقـدار صـفر قـرار دادن قطـر اصـلی ماتریس در یک تابع یا پیمانه.

```
#include <stdio.h>
void fun(void);
int main()
{
fun();
return 0;
}
fun(void)
{
int a[100][100],i,j;
for (i=0; i<100; i++){
      for (j=0; j<100; j++){
           scanf("%d", &a[i][j]);
    }
}
for(i=0; i<100; i++)
{
a[i][i]=0;}
 }
```

۴- انسجام زمانی یا موقتی (Temporal Cohesion)

اگر پیمانه‌ای حاوی وظایف یا کارهای مختلفی باشد که همگی باید در محدوده زمانی خاصـی در پاسخ به وقوع یک رویداد خاص اجرا شوند، آنگاه آن پیمانـه دارای انسـجام زمـانی اسـت. بـه عنوان مثال در مساله کنترل فشار کابین هواپیما، اگر سنسور، اندازه فشار بـیش از حـد را در کـابین خلبان حس کند، آنگاه یک پیمانه در سطح انسجام زمانی می‌تواند هم وظیفه تماس با برج مراقبت، هم وظیفه فعال‌سازی صدای خطر و هم وظیفه فعال‌سازی کاهش ارتفاع را با هـم انجـام دهـد. در این مثال درجه تعداد وظایف مختلف در یک پیمانه برابر مقدار ۳ است، کـه نشـانه انسـجام پـایین است.

۵- انسجام منطقی (Logical Cohesion)

اگر پیمانه‌ای حاوی وظایف یا کارهای مختلفی باشد که همگی به شکل منطقی توسط ساختارهای کنترلی به هم مرتبط شوند، آنگاه آن پیمانه دارای انسجام منطقی است.

مثال:

```
#include <stdio.h>
void fun(int flag);
int main()
{
int x;
scanf("%d", &x);
fun(x);
return 0;
}
fun(int flag )
{
if (flag%2==0)
    even();
else
  odd();
}
```

توجه: عملگر % باقی مانده تقسیم صحیح دو عملوند خود را می‌دهد.

توجه: پیمانه یا تابع ()fun دو وظیفه یا مسئولیت مختلف و غیر مرتبط به هم یعنی وظیفه تعیین زوج یا فرد بودن دو عدد را انجام می‌دهد، به عبارت دیگر درجه تعداد وظایف یا مسئولیت‌های مختلف این پیمانه بیشتر از عدد یک است، مطابق تعریف اگر پیمانه‌ای حاوی وظایف یا کارهای مختلفی باشد که همگی به شکل منطقی توسط ساختارهای کنترلی به هم مرتبط شوند، آنگاه آن پیمانه دارای انسجام منطقی است. بنابراین این پیمانه دارای انسجام منطقی است.

۶- انسجام تصادفی یا سودمندی یا ابزاری (Coincidental Cohesion)

این نوع از انسجام زمانی رخ می‌دهد که وظایف طوری در یک پیمانه مجتمع شوند که هیچگونه ارتباطی بین آن‌ها برقرار نباشد. به بیان دیگر چند وظیفه مختلف نامرتبط در یک پیمانه مجتمع شوند. این سطح از انسجام، پایین‌ترین سطح انسجام و سطح نامطلوب است. به انسجام تصادفی، انسجام سودمندی یا ابزاری (Utility Cohesion) نیز گفته می‌شود.

مثال: انجام وظیفه جمع دو عدد و وظیفه محاسبه فاکتوریل یک عدد در یک تابع یا پیمانه.

```
#include<stdio.h>
void fun(void);
int main()
{
```

```
fun();
return 0;
}
fun(void)
{
int a, b, r, i, f=1;
scanf("%d %d", &a , &b);
r=a+b;
printf("%d", r);
scanf("%d", &n);
int i,f=1;
for(i=1; i<=n; i++)
      {
       f=f*i;
       }
printf("%d",f );
}
```

توجه: پیمانه یا تابع ()fun دو وظیفه یا مسئولیت مختلف و غیر مرتبط به هم یعنی وظیفه جمع دو عدد و محاسبه فاکتوریل یک عدد را انجام می‌دهد، به عبارت دیگر درجه تعداد وظایف یا مسئولیت‌های مختلف این پیمانه بیشتر از عدد یک است، بنابراین این پیمانه دارای انسجام تصادفی است.

توجه: به طورکلی، تمامی انسجام‌های شی‌گراء، شامل انسجام عملیاتی یا کارکردی، انسجام لایه‌ای و انسجام ارتباطی از کلیه‌ی انسجام‌های ساخت‌یافته، در سطح بالاتری(مطلوب‌تری) قرار دارند، زیرا مولفه‌های شی‌گراء یعنی کلاس‌ها، به طور ذاتی از طراحی ایده‌آل و مطلوب برخوردار هستند، انسجام‌های شیءگرا، در فصل شیءگرایی تشریح می‌گردند.

اتصال (Coupling)

کاهش وابستگی میان پیمانه‌ها یا توابع، یک خصیصه مثبت در طراحی نرم‌افزار بوده که هرچه میزان آن کمتر باشد، طراحی معماری از کیفیت بالاتری برخوردار است. اتصال یا وابستگی در ساخت‌یافتگی به معنی میزان اتصال و وابستگی مابین پیمانه‌ها یا توابع است. به بیان دیگر میان پیمانه‌ها یا توابع چه خبر است. میزان اتصال، به پیچیدگی نقطه اتصال دو پیمانه یا تابع برای تبادل اطلاعات بستگی دارد. هر چه میزان اتصال مابین پیمانه‌ها یا توابع پایین‌تر باشد، طراحی معماری بهتری داریم. به بیان دیگر هرچه پیمانه‌ها یا توابع برای تحقق اهدافشان، نیازمند ارتباط کمتری با دیگر پیمانه‌ها یا توابع باشند، اتصال آن برنامه کمتر است. اگر چه ارتباط میان پیمانه‌ها یا توابع برای مجتمع کردن برنامه لازم است، اما باید تا جای ممکن بر کاهش اتصال میان پیمانه‌ها و توابع تاکید کرد. ارتباط ساده میان پیمانه‌ها یا توابع به برنامه‌ای منجر می‌شود که درک آن آسان‌تر بوده و کمتر در معرض خطای منتشرشونده قرار می‌گیرد، ناشی از رخ دادن یک خطا در یک مکان و انتشار آن به نقاط دیگر برنامه. اتصال را می‌توان در قالب یک طیف از کمترین سطح اتصال تا

بیشترین سطح اتصال طبقه‌بندی نمود. در طراحی معماری همواره برای کمترین سطح اتصال تلاش می‌شود. در ادامه سطوح مختلف اتصال در ساخت‌یافتگی از کمترین سطح اتصال (مطلوب‌ترین) تا بیشترین سطح اتصال (نامطلوب‌ترین) رتبه‌بندی شده است:

۱- بدون اتصال (No Coupling)

هنگامی که هیچ نوع ارتباطی میان دو پیمانه وجود ندارد.

۲- اتصال عادی (Routine Call Coupling)

هنگامی که یک پیمانه، پیمانه دیگری را بدون پارامتر ورودی فراخوانی می‌کند.

```
#include <stdio.h>
void fun(void);
int main()
{
fun();
return 0;
}
void fun(void){
printf("*");
}
```

۳- اتصال داده‌ای (Data Coupling)

هرگاه بین دو پیمانه، داده‌های ساده (صحیح، اعشاری و کارکتر) مبادله شود، اتصال داده‌ای رخ داده است.

مثال: تابع جمع دو عدد.

```
#include <stdio.h>
void sum(int a, int b);
int main()
{
int x,y;
scanf("%d %d", &x , &y);
sum(x,y);
return 0;
}
sum(int a , int b)
{
int r;
r=a+b;
printf("%d", r);
}
```

هنگام اجرا، برنامه از تابع main شروع می‌شود و به دستور scanf می‌رسد. دو عدد از ورودی گرفته و در متغیرهای x و y ذخیره می‌کند. خط فراخوانی تابع sum یعنی sum(x,y) باعث می‌شود آرگومان x در پارامتر a و آرگومان y در پارامتر b کپی شود و کنترل به درون تابع sum پرش کند. و عملیات جمع و چاپ انجام گردد. در نهایت، برنامه با بازگشت کنترل به تابع main() و اجرای دستور return 0; پایان می‌یابد. توجه به این نکته مهم است که نام پارامترها کاملا اختیاری است و می‌تواند همنام آرگومان‌ها باشد یا نباشد.

۴- اتصال برچسبی یا مُهری (Stamp Coupling)

هرگاه بین دو پیمانه، داده‌های ساخت‌یافته (رکورد، آرایه و غیره) مبادله شوند. اتصال برچسبی یا اتصال تمبری یا اتصال مُهری رخ داده است.

مثال: همانطور که می‌دانید، نام آرایه در واقع اشاره‌گری است که به اولین عنصر آرایه اشاره می‌کند. بنابراین نحوه صدا زدن تابع با آرایه مشابه اشاره‌گرها از نوع فراخوانی با ارجاع (Call By Reference) است، یعنی فقط آدرس شروع آرایه به تابع فرستاده می‌شود و اگر تغییراتی در تابع به آرایه داده شود. این تغییرات به تابع صدازننده نیز اعمال می‌گردد. در مثال زیر Syntax فرستادن آرایه به تابع نشان داده شده است.

مثال: خروجی این برنامه چیست؟

```
#include < stdio.h >
void modify (int b[ ]);
main ()
{
int a[۳] = {۱,۲,۳};
printf ("\n%d%d%d",a[۰],a[۱],a[۲]);
modify (a);
printf ("\n%d%d%d",a[۰],a[۱],a[۲]);
}
void modify (int b[ ])
{
int i;
for (i = ۰ ; i <= ۲;++i)
b[i] = ۰;
printf ("\n %d %d %d" , b[۰] ,b[۱] , b[۲]) ;
}
```

خروجی:

```
 ۱  ۲  ۳
 ۰  ۰  ۰
 ۰  ۰  ۰
```

توجه: در برنامه فوق دو تابع main و modify به دلیل مبادله داده ساخت‌یافته آرایه، دارای اتصال برچسبی یا اتصال مُهری هستند.

اغلب هنگام تعریف و معرفی، اسم آرایه به همراه براکت خالی نوشته می‌شود و هنگام صدا زدن، جلوی نام تابع فقط نام آرایه بدون براکت آورده می‌شود (modify (a)). البته اگر هنگام تعریف و معرفی، طول آرایه را داخل براکت بنویسید برنامه همچنان درست است.

از آنجا که آرایه و اشاره‌گر یک مفهوم می‌باشند. تعریف یا معرفی تابع modify مثال فوق را می‌توان به صورت زیر نیز انجام داد:

void modify (int b[ ]) ⇔ void modify (int *b)

دقت کنید که اسم آرایه هنگام تعریف و هنگام صدا زدن می‌تواند همنام باشد یا نباشد و این همنامی یا عدم همنامی هیچ اثری در اجرای برنامه ندارد.

۵- اتصال کنترلی (Control Coupling)

هرگاه بین دو پیمانه، داده‌های کنترلی مبادله شود، اتصال کنترلی رخ داده است. منظور از داده‌های کنترلی یا پرچم‌های کنترلی، متغیرهایی هستند که از پیمانه مبدا به داخل پیمانه مقصد (فراخوانی شده) انتقال می‌یابند و حرکت و کنترل پیمانه فراخوانی شده را تحت تاثیر قرار می‌دهند.

مثال:

```
#include <stdio.h>
void fun(int flag);
int main()
{
int x;
scanf("%d", &x);
fun(x);
return 0;
}
fun(int flag )
{
if (flag%2==0)
    even();
else
  odd();
}
```

توجه: عملگر % باقی مانده تقسیم صحیح دو عملوند خود را می‌دهد.

توجه: هنگام اجرا، برنامه از تابع main شروع می‌شود و به دستور scanf می‌رسد. یک عدد از ورودی گرفته و در متغیر x ذخیره می‌کند. خط فراخوانی تابع fun یعنی fun(x) باعث می‌شود آرگومان x در پارامتر flag کپی شود و کنترل به درون تابع fun پرش کند. و عملیات تعیین زوج و فرد بودن عدد خوانده شده انجام گردد. در نهایت، برنامه با بازگشت کنترل به تابع main() و اجرای دستور return 0; پایان می‌یابد.

توجه: به اتصال کنترلی، اتصال حدی یا وابستگی حدی (Constrained dependencies) نیز گفته می‌شود.

۶- اتصال خارجی (External Coupling)

هرگاه بین پیمانه‌های برنامه اصلی و پیمانه‌های خارجی (توابع سیستمی و کتابخانه‌ای و توابع پایگاه داده)، داده مبادله شود، اتصال خارجی رخ داده است.

مثال:

```
#include <stdio.h>
#include <math.h>
void sum(void);
void fun(void);
int main()
{
sum(void);
fun(void);
return 0;
}
sum(void)
{
int a,b,r;
scanf("%d %d", &a , &b);
r=a+b;
printf("%d", r);
}
fun(void)
{
int x,r;
scanf("%d", &x);
r=sin(x);
printf("%d", r);
}
```

توجه: برای تمامی توابع ریاضی مانند تابع ()sin در زبان C می‌بایست فایل math.h را در برنامه include کرد.

توجه: تابع ()sum با توابع سیستمی ورودی و خروجی scanf و printf اتصال خارجی دارد و تابع ()fun با تابع سیستمی sin و با توابع سیستمی ورودی و خروجی scanf و printf اتصال خارجی دارد.

۷- اتصال عمومی یا مشترک یا اشتراکی (Common Coupling)

هرگاه چند پیمانه از طریق خواندن و نوشتن در داده‌های سراسری به هم مرتبط شوند، اتصال عمومی رخ داده است.

مثال: جابه جایی محتویات دو متغیر.

```
#include<stdio.h>
void setab(void);
void swap(void);
void getab(void);
int a,b;
int main()
{
setab();
swap();
getab();
return 0;
}
setab (void){
scanf("%d %d", &a , &b);
}
swap(void){
int temp;
temp=a;
a=b;
b=temp;
}
getab(void)
{
printf("%d %d", a , b);
}
```

توجه: متغیرهای سراسری a و b در دیتاسگمنت ساخته می‌شوند و اگر مقداردهی نشوند مقدار اولیه آنها صفر است. متغیرهای سراسری ابتدای برنامه ساخته شده و تا انتهای برنامه فضای آنها

حفظ می‌گردد. اما متغیرهای محلی در استک ساخته می‌شوند و اگر مقداردهی نشوند مقدار اولیه آنها نامشخص است. متغیرهای محلی به محض ورود به زیربرنامه ساخته شده و هنگام اتمام زیربرنامه از بین می‌روند.

توجه: هنگام اجرا، برنامه از تابع main شروع می‌شود و به دستور setab می‌رسد. در ادامه کنترل به درون تابع setab پرش کند. در این تابع دو عدد از ورودی گرفته شده و در متغیرهای سراسری a و b ذخیره می‌شود. در ادامه با اتمام تابع setab کنترل به دستور بعد از فراخوانی تابع یعنی دستور ;()swap می‌رسد، در تابع ()swap مطابق الگوریتم مطرح شده محتویات دو متغیر سراسری a و b جابه جا می‌شود. در ادامه با اتمام تابع ()swap نیز کنترل به دستور بعد از فراخوانی تابع یعنی دستور ;()getab می‌رسد، در تابع ()getab محتویات دو متغیر سراسری a و b در خروجی نمایش داده می‌شوند. در نهایت، برنامه با بازگشت کنترل به تابع ()main و اجرای دستور ;0 return پایان می‌یابد.

توجه: در مثال فوق، ابتدا پیمانه setab، مقدار متغیرهای سراسری a و b را مشخص می‌کند، سپس در ادامه پیمانه swap محتویات متغیرهای سراسری a و b را جابه جامی کند و در نهایت پیمانه getab محتویات متغیرهای سراسری a و b را در خروجی قرار می‌دهد.

توجه: در این شرایط این پیمانه ها به دلیل استفاده از متغیرهای سراسری پتانسیل وقوع خطای منتشرشونده را دارند، در صورتی که پیمانه‌ای در انجام وظایفش کوتاهی کند و دارای خطای منطقی گردد، این خطا به پیمانه‌های بعدی به شکلی منتشر شونده، انتشار می‌یابد. و در صورت مشاهده خطا، باید پیمانه فعلی، پیمانه قبلی و قبل‌ترها مورد وارسی قرار گیرند.

۸- اتصال محتوایی (Content Coupling)

هرگاه خروجی نهایی یک پیمانه به عنوان ورودی یک پیمانه دیگر مورد استفاده قرار گیرد، بین این دو پیمانه اتصال محتوایی رخ داده است. این سطح از اتصال، بیشترین سطح اتصال و سطح نامطلوب است.

مثال: برنامه‌ای که طول و عرض مستطیلی را گرفته و در دو تابع مجزا ابتدا حاصل جمع طول و عرض را حساب می‌کند و در تابعی دیگر با استفاده از حاصل تابع قبل، این بار محیط مستطیل را محاسبه می‌کند.

```
#include<stdio.h>
int sum(int a, int b);
void rectangle(int p);
int main(){
int x,y,s;
scanf("%d %d", &x , &y);
s=sum(x,y);
rectangle(s);
return 0;
```

```
}
sum(int a , int b){
int r;
r=a+b;
return(r);
}
void rectangle(int p){
int m;
m=2*p;
printff("%d" , m);
}
```

توجه: هنگام اجرا، برنامه از تابع main شروع می‌شود و به دستور scanf می‌رسد. دو عدد را به عنوان طول و عرض مستطیل از ورودی گرفته و در متغیرهای x و y ذخیره می‌کند. خط فراخوانی تابع sum یعنی sum(x,y) باعث می‌شود آرگومان x در پارامتر a و آرگومان y در پارامتر b کپی شود و کنترل به درون تابع sum پرش کند. و عملیات جمع طول و عرض مستطیل انجام گردد. در ادامه با اتمام تابع sum ابتدا خروجی تابع sum توسط دستور return(r); داخل متغیر s قرار داده می‌شود و سپس کنترل به دستور بعد از فراخوانی تابع یعنی دستور rectangle(s); می‌رسد، در تابع rectangle() مطابق الگوریتم مطرح شده با استفاده از خروجی تابع sum()، محیط مستطیل محاسبه می‌گردد. و مقدار حاصل در خروجی نمایش داده می‌شود. در نهایت، برنامه با بازگشت کنترل به تابع main() و اجرای دستور return 0; پایان می‌یابد.

توجه: در این شرایط دو پیمانه sum() و rectangle() به دلیل وجود اتصال محتوایی پتانسیل وقوع خطای منتشرشونده را دارند، در صورتی که پیمانه sum() در انجام وظایفش کوتاهی کند و دارای خطای منطقی گردد، این خطا به پیمانه بعدی یعنی rectangle() به شکلی منتشر شونده، انتشار می‌یابد. و در صورت مشاهده خطا، باید پیمانه فعلی و قبلی مورد وارسی قرار گیرند.

توجه: به اتصال محتوایی، اتصال جریانی یا وابستگی جریانی (Flow Dependencies) نیز گفته می‌شود.

توجه: اتصال محتوایی، بیشترین سطح اتصال و نامطلوب‌ترین است. «**اتصال محتوایی بدترین نوع اتصال و بیشترین میزان در نقض پنهان‌سازی اطلاعات را دارا است.**» تعریف متغیر سراسری در تابع و ایجاد اتصال عمومی یا مشترک یا اشتراکی با میزان کمتری نسبت به اتصال محتوایی می‌تواند در شرایط خاص ناقض اصل پنهان‌سازی اطلاعات باشد.

تست‌های فصل چهارم

۱- Common Coupling برای زیربرنامه‌ها به چه معنی است؟ (مهندسی کامپیوتر – آزاد ۷۱)

۱) ایجاد روشی که زیربرنامه‌ها بتوانند داده‌های خود را به زیربرنامه دیگر منتقل کنند.
۲) ایجاد یک محیط سراسری برای برنامه‌هایی که دارای متغیرهای مشترک می‌باشند.
۳) ایجاد یک ساختار مشترک برای زیربرنامه‌ها به طوری که متغیرها دارای نام‌های متفاوت از نظر زیربرنامه‌ها باشند.
۴) اتصال مجموعه‌ای از زیربرنامه‌ها به گونه‌ای که یک زیربرنامه برای اجرای زیربرنامه دیگر نظارت دارد.

۲- دستور زیر مبین چه نوع Coupling بین برنامه صدازننده و صداشونده می‌باشد؟
(مهندسی کامپیوتر – آزاد ۷۱)

CALL PRINT _ LABEL USING MAILING _ ADDRESS.

(فرض کنید MAILING _ ADDRESS به صورت COMPOSITE باشد).

۱) یک Stamp Coupling
۲) یک Data Coupling
۳) یک Content Coupling
۴) یک Common Coupling

۳- در طراحی پیمانه‌ای (modular) سیستم نرم‌افزاری با توجه به معیارهای کیفی cohesion و coupling طراحی خوب دارای کدام ویژگی است؟ (مهندسی IT – دولتی ۸۳)

۱) cohesion بالا و coupling بالا
۲) cohesion پایین و coupling پایین
۳) cohesion پایین و coupling بالا
۴) cohesion بالا و coupling پایین

۴- پیمانه‌های c، g و k هر کدام به داده‌ای (مانند فایل یا ناحیه‌ای از حافظه که دستیابی سراسری به آن وجود دارد)، در ناحیه سراسری داده‌ها دستیابی دارند. پیمانه c داده را مقداردهی اولیه می‌نماید. سپس پیمانه g مقدار آن را دوباره محاسبه و آن را به هنگام‌سازی می‌کند. در این حالت کدام یک از سطوح coupling وجود دارد؟ (مهندسی IT – دولتی ۸۳)

۱) data coupling
۲) external coupling
۳) common coupling
۴) content coupling

۵- کدام‌یک از جملات زیر صحیح است؟ (مهندسی IT – آزاد ۸۴)

۱) در یک نرم‌افزار با افزایش تعداد ماژول‌ها، هزینه افزایش می‌یابد.
۲) در یک نرم‌افزار با افزایش تعداد ماژول‌ها، هزینه یکپارچه نمودن کاهش می‌یابد.
۳) در یک نرم‌افزار با کاهش تعداد ماژول‌ها، هزینه کلی نرم‌افزار کاهش می‌یابد.
۴) در یک نرم‌افزار با داشتن تعداد مناسب ماژول، هزینه کاهش می‌یابد.

۶- کدام‌یک از گزینه‌های زیر صحیح است؟ (مهندسی IT – آزاد ۸۶)

۱) تحلیل به بیان نیازهای وظیفه‌مندی (functionality) بدون توجه به چگونگی پیاده‌سازی آنها می‌پردازد.

۲) در تحلیل هر دو دسته نیازمندی‌های وظیفه‌مندی و غیروظیفه‌مندی مورد بررسی قرار می‌گیرد.
۳) در طراحی پیمانه‌ای (modular) سیستم نرم‌افزاری، طراحی خوب دارای cohesion بالا و coupling بالا می‌باشد.
۴) در طراحی پیمانه‌ای (modular) سیستم نرم‌افزاری، طراحی خوب دارای cohesion پایین و coupling بالا می‌باشد.

۷- کدام‌یک از انواع اشتراک‌داده‌ها در میان دو پیمانه امکان انتشار خطا را کمتر می‌کند؟
(مهندسی IT – دولتی ۸۶)

۱) وجود داده سراسری
۲) تبادل پارامتر به صورت داده ساده
۳) تبادل پارامتر به صورت ساختمان داده
۴) تبادل پارامتر به صورت داده‌ی کنترلی

۸- کدام عبارت صحیح نیست؟ **(مهندسی IT – دولتی ۸۷)**

۱) طراحی نرم‌افزار با در نظر گرفتن امکان استفاده مجدد نیاز به تلاش بیشتری دارد.
۲) هنگامی که نرم‌افزار به سفارش مشتری ساخته می‌شود از قابلیت اطمینان بالاتری برخوردار است.
۳) هنوز مزایای استفاده از مؤلفه‌های نرم‌افزاری آماده به خوبی و به طور کامل روشن نشده است.
۴) فشاری که همواره به علت وجود مهلت تحویل پروژه وجود دارد از توجه لازم به سرمایه‌گذاری برای آینده جلوگیری می‌کند.

۹- اگر پیمانه c از طریق پارامترهای داده‌ای با پیمانه a ارتباط برقرار نماید، در این صورت کدام یک از سطوح اتصال (coupling) بین آنها وجود دارد؟ **(مهندسی IT – دولتی ۸۹)**

۱) برچسبی (Stamp Coupling)
۲) داده‌ای (Data Coupling)
۳) کنترلی (Control Coupling)
۴) مشترک (Common Coupling)

۱۰- در بین انواع وابستگی (Coupling) که در زیر فهرست شده‌اند، کدام‌یک از همه مطلوب‌تر است؟
(مهندسی IT – دولتی ۹۱)

۱) وابستگی محتوایی ۲) وابستگی داده‌ای ۳) وابستگی خارجی ۴) وابستگی کنترلی

۱۱- دو پیمانه از طریق تبادل یک رکورد با یکدیگر در ارتباطند. این نوع اتصال (coupling) از نوع ..... است. **(مهندسی IT – دولتی ۹۳)**

۱) Data Coupling (اتصال داده‌ای)
۲) Stamp Coupling (اتصال تمبری)
۳) Control Coupling (اتصال کنترلی)
۴) Common Coupling (اتصال اشتراکی)

۱۲- کدام‌یک از انواع اتصال (Coupling) زیر، مخفی‌سازی اطلاعات را نقض می‌کند؟
(مهندسی IT – دولتی ۱۴۰۰)

۱) اتصال محتوایی (Content Coupling)
۲) اتصال مشترک (Common Coupling)
۳) اتصال خارجی (External Coupling)
۴) اتصال کنترلی (Control Coupling)

پاسخ تست‌های فصل چهارم

۱- گزینه (۲) صحیح است.

«ایجاد روشی که زیربرنامه‌ها بتوانند داده‌های خود را به زیر برنامه دیگر منتقل کنند»، روش‌های مختلفی همچون اتصال داده‌ای، اتصال برچسبی، اتصال کنترلی، اتصال محتوایی و اتصال عمومی دارد. که این تبادل داده فقط مختص به اتصال عمومی نیست. بنابراین گزینه اول نادرست است.

مطابق تعریف اتصال عمومی، هرگاه چند پیمانه از طریق خواندن و نوشتن در داده‌های سراسری به هم مرتبط شوند، اتصال عمومی رخ داده است. بنابراین گزینه دوم درست است.

هرگاه یک متغیر محلی، داخل یک تابع، همنام با متغیر سراسری تعریف گردد، آنگاه متغیر سراسری در داخل تابع مورد نظر خاصیت سراسری بودن خود را از دست می‌دهد و متغیر مورد نظر به صورت محلی رفتار می‌کند. جهت استفاده از متغیر سراسری در یک تابع مورد نظر کافیست نام متغیر سراسری صدا زده شود، متغیری که در یک تابع مورد نظر همنام با متغیر سراسری نیست، متغیر محلی است، و نه متغیر سراسری. بنابراین گزینه سوم نیز نادرست است.

مطابق تعریف اتصال کنترلی، هرگاه بین دو پیمانه، داده‌های کنترلی مبادله شود، اتصال کنترلی رخ داده است. منظور از داده‌های کنترلی یا پرچم‌های کنترلی، متغیرهایی هستند که از پیمانه مبدا به داخل پیمانه مقصد (فراخوانی شده) انتقال می‌یابند و حرکت و کنترل پیمانه فراخوانی شده را تحت تاثیر قرار می‌دهند. بنابراین عبارت مطرح شده در گزینه چهارم یعنی «اتصال مجموعه‌ای از زیربرنامه‌ها به گونه‌ای که یک زیربرنامه برای اجرای زیربرنامه دیگر نظارت دارد» مطابق تعریف اتصال کنترلی است و نه اتصال عمومی، بنابراین گزینه چهارم نیز نادرست است.

۲- گزینه (۱) صحیح است.

متغیر MAILING_ADDRESS مطابق خط فراخوانی(CALL) تابع PRINT_LABLE، به صورت COMPOSITE یعنی مرکب، برچسبی و مُهری در نظر گرفته شده است. بنابراین در خط فراخوانی تابع PRINT_LABLE با اتصال برچسبی یا مُهری (Stamp Coupling) مابین برنامه صدا زننده و برنامه صدا شونده مواجه هستیم. بنابراین گزینه اول درست است.

۳- گزینه (۴) صحیح است.

در یک طراحی ایده‌آل، هدف، محقق کردن بالاترین سطح انسجام (Cohesion بالا) داخل پیمانه‌های برنامه و کمترین سطح اتصال (Coupling پایین) مابین پیمانه‌های برنامه است.

۴- گزینه (۳) صحیح است.

هرگاه چند پیمانه از طریق خواندن و نوشتن در داده‌های سراسری به هم مرتبط شوند، اتصال عمومی یا اتصال مشترک (Common Coupling) رخ داده است.

۵- گزینه (۴) صحیح است.

وقتی با یک مساله برنامه‌نویسی به زبان ساخت‌یافته مواجه می‌شوید، برای بالابردن خوانایی و کنترل بهتر برنامه این پرسش مطرح است که چگونه این مساله به توابع (پیمانه‌ها یا ماژول‌ها) تقسیم شود، که تعیین تعداد توابع به عهده مهندسان نرم‌افزار است. تعیین تعداد توابع از مدل تحلیل توسط حباب‌های موجود در نمودار DFD آغاز می‌شود، سپس این حباب‌ها توسط مدل طراحی به توابع برنامه نگاشت می‌شوند. اما این سوال نیز مطرح است که تعداد این توابع چقدر باید باشد، به عبارت بهتر تقسیم برنامه به توابع تا کجا باید پیشروی کند.

پیمانه‌ای کردن یعنی تقسیم نرم‌افزار به چند مولفه جداگانه و متمایز که این مولفه‌ها به عنوان پیمانه‌هایی هستند که از اجتماع آن‌ها، خواسته‌های مساله برآورده می‌شود. با این عمل مدیریت مفهومی یک برنامه حاصل می‌شود و قابلیت خوانایی نرم‌افزار چندین برابر می‌گردد. درک و فهم نرم‌افزار یکپارچه (monolithic) که تنها از یک پیمانه (ماژول) تشکیل شده است، بسیار مشکل است، زیرا تعداد متغیرها، حجم ارجاعات، تعداد مسیرهای کنترلی و پیچیدگی سراسری آن بسیار زیاد است. بنابراین اگر بتوان نرم‌افزار را به پیمانه‌هایی مناسب تقسیم نمود، پیچیدگی و هزینه کلی آن کاهش خواهد یافت. اما تعداد این پیمانه‌ها نباید بیش از حد باشد زیرا هزینه مجتمع‌سازی یا یکپارچه‌سازی و اتصال آن‌ها افزایش می‌یابد.

در یک نرم‌افزار با کاهش تعداد پیمانه‌ها (ماژول ها)، هزینه یکپارچه‌سازی آن‌ها کاهش می‌یابد. اما هزینه ساخت هر پیمانه افزایش می‌یابد، بنابراین هزینه کلی آن نیز که حاصل جمع دو مقدار هزینه یکپارچه‌سازی و هزینه ساخت هر پیمانه است، افزایش می‌یابد. بنابراین گزینه سوم نادرست است.

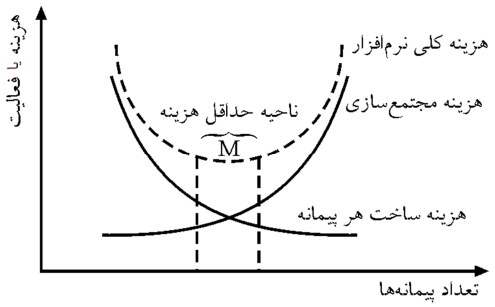
همچنین در یک نرم‌افزار با افزایش تعداد پیمانه‌ها (ماژول ها)، هزینه یکپارچه‌سازی آن‌ها افزایش می‌یابد. اما هزینه ساخت هر پیمانه کاهش می‌یابد، بنابراین هزینه کلی آن نیز که حاصل جمع دو مقدار هزینه یکپارچه‌سازی و هزینه ساخت هر پیمانه است، افزایش می‌یابد. بنابراین گزینه اول نیز نادرست است. زیرا نوع هزینه را مشخص نکرده است. همچنین گزینه دوم نیز نادرست است. زیرا، در شرایط مطرح شده، هزینه یکپارچه‌سازی افزایش می‌یابد.

همچنین در یک نرم‌افزار با داشتن تعداد مناسب پیمانه (ماژول)، همه هزینه‌ها کاهش می‌یابد. زیرا هم هزینه یکپارچه‌سازی و هم هزینه ساخت هر پیمانه، هر دو کاهش می‌یابد، بنابراین هزینه کلی آن نیز که حاصل جمع دو مقدار هزینه یکپارچه‌سازی و هزینه ساخت هر پیمانه است، کاهش می‌یابد.

شکل زیر رابطه تعداد پیمانه‌های نرم‌افزار با هزینه توسعه نرم‌افزار را نشان می‌دهد. در این شکل، یکی از منحنی‌ها هزینه توسعه یک پیمانه را نشان می‌دهد و منحنی دیگر، هزینه یکپارچه‌سازی پیمانه‌ها را نشان می‌دهد. به تبع هزینه کلی نرم‌افزار برابر حاصل جمع این دو منحنی در هر نقطه است، که با منحنی خط چین نشان داده شده است.

در یک نرم‌افزار با داشتن تعداد مناسب ماژول (پیمانه)، هزینه کاهش می‌یابد. زیرا هم هزینه یکپارچه‌سازی و هم هزینه ساخت هر پیمانه، هر دو کاهش می‌یابد، بنابراین هزینه کلی آن نیز که

حاصل جمع دو مقدار هزینه یکپارچه‌سازی و هزینه ساخت هر پیمانه است، کاهش می‌یابد. بنابراین گزینه چهارم درست است.



نمودار فوق نشان می‌دهد که هزینه یا فعالیت لازم برای ساخت پیمانه‌های نرم‌افزاری با افزایش تعداد پیمانه‌ها الزاما کاهش می‌یابد ولی به موازات رشد بیش از حد پیمانه‌ها، هزینه مربوط به اجتماع و یکپارچه‌سازی پیمانه‌ها نیز رشد می‌کند. در واقع همواره با تعداد مناسبی از پیمانه‌ها که با M نشان داده شده است، می‌توان هزینه و کار را کمینه کرد.

۶- گزینه (۲) صحیح است.

یک مدل، ساده شده یک واقعیت است. ایجاد یک مدل برای سیستم‌های نرم‌افزاری قبل از ساخت یا بازساخت آن، به اندازه داشتن نقشه برای ساختن یک ساختمان ضروری وحیاتی است. بسیاری از شاخه‌های مهندسی، توصیف چگونگی محصولاتی که باید ساخته شوند را ترسیم می‌کنند و همچنین دقت زیادی می‌کنند که محصولاتشان طبق این مدل‌ها و توصیف‌ها ساخته شوند. مدل‌های خوب و دقیق در برقراری یک ارتباط کامل بین افراد پروژه، نقش زیادی می‌توانند داشته باشند. علت اصلی مدل‌کردن سیستم‌های پیچیده این است که نمی‌توان به یکباره کل سیستم را تجسم کرد و ممکن است سیستم دارای ابهامات بسیاری باشد. لذا برای رفع این ابهامات و نیز برای فهم کامل سیستم و یافتن و نمایش ارتباط بین قسمت‌های مختلف، از مدل‌سازی استفاده می‌شود. فعالیت مدل‌سازی خود شامل دو مرحله‌ی مدل تحلیل و مدل طراحی می‌باشد. مدل تحلیل پس از فعالیت ارتباطات (جمع‌آوری نیازمندی‌ها) و قبل از مدل طراحی انجام می‌شود، در واقع خروجی مدل تحلیل، ورودی مدل طراحی می‌باشد.

پس از جمع‌آوری نیازمندی‌ها در فعالیت ارتباطات نوبت به مدل تحلیل (مدل‌سازی لیست نیازمندی‌ها) می‌رسد. مدل‌سازی که فعالیتی فنی به شمار می‌رود نیازمندی‌ها را باید به گونه‌ای مدل نماید که برای سازنده و مشتری قابل فهم باشد.

مدل‌هایی که در فعالیت مدل‌سازی (بخش مدل تحلیل)، تهیه می‌شوند، سه هدف مهم را در تولید برنامه‌های کامپیوتری دنبال می‌کنند:

۱- توصیف نمادین نیازهای مشتری، توسط مدل‌سازی لیست نیازمندی‌ها، براساس نیازمندی‌های وظیفه‌مندی و هم نیازمندی‌های غیروظیفه‌مندی.

۲- مدل تحلیل، بستری مناسب را برای فعالیت طراحی نرم‌افزار ایجاد می‌کند.

۳- در فعالیت ساخت (بخش تست)، هنگامیکه اعتبارسنجی (validation) نرم‌افزار انجام می‌شود، مدل تحلیل، می‌تواند به عنوان یک تصویر از نرم‌افزار مورد انتظار، جهت فعالیت تست مورد استفاده قرار گیرد.

دقت کنید که، تهیه و مدل‌سازی لیست نیازمندی‌ها مستقل از نحوه پیاده‌سازی، انجام می‌گردد. نیازمندی های واقعی کاربران که مشخص شد، برای پیاده‌سازی آن نیز چاره اندیشیده می‌شود. بنابراین گزینه اول نادرست است و گزینه دوم درست است.

در یک طراحی ایده‌آل، هدف، محقق کردن بالاترین سطح انسجام (Cohesion بالا) داخل پیمانه‌های برنامه و کمترین سطح اتصال (Coupling پایین) مابین پیمانه‌های برنامه است. بنابراین گزینه سوم و چهارم نیز نادرست هستند.

۷- گزینه (۲) صحیح است.

هرگاه بین دو پیمانه، داده‌های ساده (صحیح، اعشاری و کارکتر) مبادله شود، اتصال داده‌ای رخ داده است. اتصال داده‌ای در کمترین سطح اتصال (مطلوب‌ترین) قرار دارد.

۸- گزینه (۴) صحیح است.

مستقل از متدولوژی ساخت‌یافته و شیءگرا، جدا از اینکه پیمانه‌ای کردن نرم‌افزار منجر به **خوانایی بالاتر برنامه** و **کنترل بهتر برنامه** در حال ساخت می‌گردد، پیمانه‌ای کردن همچنین این خاصیت را فراهم می‌آورد تا در صورتی که پیمانه‌هایی با رعایت **اصول طراحی معماری ایده‌آل** ایجاد گردند، این پیمانه‌ها هم در نرم‌افزار فعلی دارای خاصیت **نگهداری آسان** باشند و هم در نرم‌افزارهای آتی خاصیت **قابلیت استفاده مجدد** را دارا باشند تا ساخت پروژه‌های نرم‌افزاری آتی با صرفه‌جویی در هزینه‌ها مقرون به صرفه گردد. زمان و هزینه‌ی فرآیند تولید نرم‌افزار با استفاده از قطعات آماده (قابلیت استفاده مجدد) کاهش چشمگیری دارد. در واقع یک قطعه با قابلیت استفاده‌ی مجدد یک بار ساخته می‌شود و بارها و بارها استفاده می‌شود و این یعنی سودآوری! بنابراین مهندسی نرم‌افزار برای ساخت پیمانه‌هایی ایده‌آل و خوب برای معماری نرم‌افزار در حال ساخت و همچنین قطعات و پیمانه‌هایی آماده با قابلیت استفاده مجدد برای نرم‌افزارهای‌های آتی، اصول **پنهان‌سازی اطلاعات**، **استقلال عملیاتی**، **افزایش انسجام** و **کاهش اتصال** را به عنوان **اصول طراحی معماری ایده‌آل** توصیه کرده است. در نهایت این اسکلت و ساختار و پیمانه‌های ایده‌آل که از ابتدا با پیروی از اصول مهندسی نرم‌افزار به صورت درست و اصولی پایه گذاری شده است، پس از حرکت به سمت طراحی مولفه و گذر از آن، در گام آخر یعنی پیاده‌سازی، در نهایت به سمت ساخت یک پروژه موفق نرم‌افزاری حرکت می‌کند و خصوصیات موفق پروژه‌های نرم‌افزاری را با خود به ارمغان می‌آورد. واین یعنی موفقیت. **موفقیت تصادفی نیست!**

صنعت در حال حرکت به سمت مونتاژ قطعات است، اما نرم‌افزارها بیشتر بر اساس نیاز مشتریان و به صورت سفارشی ساخته می‌شوند و در دنیای سخت‌افزار، استفاده مجدد از قطعات، بخش طبیعی از فرآیند مهندسی است در واقع ماهیت سخت‌افزار این امکان را می‌دهد، تا به طور

جداگانه هر یک از اجزای محصول (حتی در مکان‌های متفاوت) به صورت جداگانه ساخته شود و در نهایت با یکدیگر مونتاژ گردند.

اما در مهندسی نرم‌افزار این امربه تازگی مورد توجه قرار گرفته است و استفاده از مؤلفه‌های آماده جهت ساخت نرم‌افزار به تازگی مرسوم شده است. به عبارت دیگر هنوز مزایای استفاده از مولفه‌های نرم‌افزاری آماده به خوبی و به طور کامل روشن نشده است. امروزه، ایده‌ی استفاده مجدد نه تنها الگوریتم‌ها، بلکه ساختار داده‌ها را نیز در بر می‌گیرد. اجزاء مدرن قابل استفاده مجدد (کلاس)، هم دارای داده می‌باشند و هم شیوه پردازش مخصوص آن داده‌ها را شامل هستند که مهندسی نرم‌افزار را قادر می‌سازد تا برنامه‌های کاربردی جدیدی را از روی قطعات قابل استفاده مجدد بسازد. به طور مثال، امروزه رابطه‌های گرافیکی کاربر با استفاده از اجزای قابل استفاده مجدد ساخته می‌شوند که ایجاد پنجره‌های گرافیکی، منوهای بازشونده، جعبه متن‌ها و دکمه‌ها را میسر می‌سازد. بنابراین گزینه سوم درست است.

اگرچه همواره فشار زمانی ناشی از مهلت تحویل پروژه، بر روی پروژه وجود دارد. ولی از توجه لازم به سرمایه گذاری برای آینده جلوگیری نمی‌کند. زیرا در صورتی که برای آینده سرمایه‌گذاری نکنیم، مجبور هستیم، در آینده هزینه‌های بیشتری را متحمل شویم. برای مثال اگر به دلیل فشار زمانی ناشی از مهلت تحویل پروژه که بر روی پروژه قرار دارد، ساخت قطعات، با خاصیت قابل استفاده مجدد بودن برای پروژه‌های آتی، نادیده گرفته شود، آنگاه هزینه‌های ساخت مجدد در آینده باید پرداخت گردد.

مؤلفه یک قطعه‌ی آماده، کاربردی و پیاده‌سازی شده است که دارای واسط لازم جهت اتصال با سایر قطعات و استقرار در بخشی از یک سیستم عملیاتی می‌باشد.

زمان و هزینه‌ی فرآیند تولید نرم‌افزار براساس مدل توسعه‌ی مبتنی بر مؤلفه به دلیل استفاده از قطعات آماده (قابلیت استفاده مجدد) کاهش چشمگیری دارد. در واقع یک قطعه با قابلیت استفاده‌ی مجدد یک بار ساخته می‌شود و بارها و بارها استفاده می‌شود و این یعنی سودآوری!

بنابراین در یک شرایط ایده‌آل و مدیریت کارآمد، فشار زمانی ناشی از مهلت تحویل پروژه که بر روی پروژه قرار دارد، نبایستی از توجه لازم به سرمایه‌گذاری برای آینده جلوگیری کند. بنابراین گزینه چهارم نادرست است.

۹- گزینه (۲) صحیح است.

هرگاه بین دو پیمانه، داده‌های ساده (صحیح، اعشاری و کارکتر) مبادله شود، اتصال داده‌ای رخ داده است. اتصال داده‌ای در کمترین سطح اتصال (مطلوب‌ترین) قرار دارد.

۱۰- گزینه (۲) صحیح است.

هرگاه بین دو پیمانه، داده‌های ساده (صحیح، اعشاری و کارکتر) مبادله شود، اتصال داده‌ای رخ داده است. اتصال داده‌ای در کمترین سطح اتصال (مطلوب‌ترین) قرار دارد.

۱۱- گزینه (۲) صحیح است.

هرگاه بین دو پیمانه، داده‌های ساخت‌یافته (رکورد، آرایه و غیره) مبادله شوند. اتصال برچسبی

یا اتصال تمبری یا اتصال مُهری رخ داده است.

مثال: همانطور که می‌دانید، نام آرایه در واقع اشاره‌گری است که به اولین عنصر آرایه اشاره می‌کند. بنابراین نحوه صدا زدن تابع با آرایه مشابه اشاره‌گرها از نوع فراخوانی با ارجاع (Call By Reference) است، یعنی فقط آدرس شروع آرایه به تابع فرستاده می‌شود و اگر تغییراتی در تابع به آرایه داده شود. این تغییرات به تابع صدازننده نیز اعمال می‌گردد. در مثال زیر Syntax فرستادن آرایه به تابع نشان داده شده است.

مثال: خروجی این برنامه چیست؟

```
#include < stdio.h >
void modify (int b[ ]);
main ()
{
int a[۳] = {۱,۲,۳};
printf ("\n%d%d%d",a[۰],a[۱],a[۲]);
modify (a);
printf ("\n%d%d%d",a[۰],a[۱],a[۲]);
}
void modify (int b[ ])
{
int i;
for (i = ۰ ; i <= ۲;++i)
b[i] = ۰;
printf ("\n %d %d %d" , b[۰] ,b[۱] , b[۲]) ;
}
```

خروجی:

```
۱ ۲ ۳
۰ ۰ ۰
۰ ۰ ۰
```

در برنامه فوق دو تابع main و modify به دلیل مبادله داده ساخت‌یافته آرایه، دارای اتصال برچسبی یا اتصال مُهری هستند.

اغلب هنگام تعریف و معرفی، اسم آرایه به همراه براکت خالی نوشته می‌شود و هنگام صدا زدن، جلوی نام تابع فقط نام آرایه بدون براکت آورده می‌شود ((modify (a)). البته اگر هنگام تعریف و معرفی، طول آرایه را داخل براکت بنویسید برنامه همچنان درست است.

از آنجا که آرایه و اشاره‌گر یک مفهوم می‌باشند. تعریف یا معرفی تابع modify مثال فوق را می‌توان به صورت زیر نیز انجام داد:

void modify (int b[ ]) ⇔ void modify (int * b)

دقت کنید که اسم آرایه هنگام تعریف و هنگام صدا زدن می‌تواند همنام باشد یا نباشد و این همنامی یا عدم همنامی هیچ اثری در اجرای برنامه ندارد.

۱۲- گزینه (۱) صحیح است.

هرگاه خروجی نهایی یک پیمانه به عنوان ورودی یک پیمانه دیگر مورد استفاده قرار گیرد، بین این دو پیمانه اتصال محتوایی (Content Coupling) رخ داده است. این سطح از اتصال، بیشترین سطح اتصال و سطح نامطلوب است. اتصال محتوایی بدترین نوع اتصال و بیشترین میزان در نقض پنهان‌سازی اطلاعات را دارا است.

۵ مدل طراحی ساخت‌یافته

سبک معماری

نرم‌افزار نیز همانند یک ساختمان، دارای سبک‌های متفاوتی برای ساخت می‌باشد. به طور کلی برای طراحی معماری و ساخت نرم‌افزارها دو سبک ساخت‌یافته (مبتنی بر فراخوانی و بازگشت توابع) و سبک شیءگرا (مبتنی بر ارسال پیام مابین اشیاء) مورد استفاده قرار می‌گیرد.

توجه: سبک معماری، در واقع شیوه طراحی معماری را مشخص می‌کند.

توجه: طراحی معماری ساخت‌یافته بر اساس سبک ساخت‌یافته (مبتنی بر فراخوانی و بازگشت توابع) انجام می‌گردد.

توجه: طراحی معماری شیءگرا بر اساس سبک شیءگرا (مبتنی بر ارسال پیام مابین اشیاء) انجام می‌گردد.

توجه: طراحی معماری شیءگرا و سبک شیءگرا در فصل شیءگرایی تشریح خواهد شد.

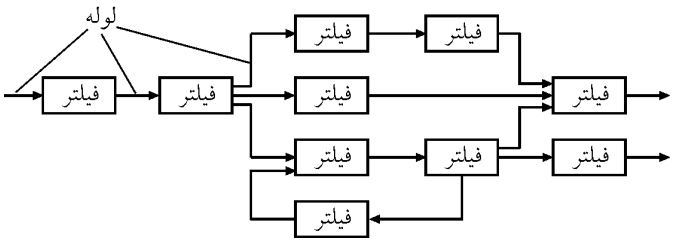
توجه: سبک ساخت‌یافته (مبتنی بر فراخوانی و بازگشت توابع) به دو طبقه کلی معماری تابع‌گرا (جریان داده یا تجزیه عملیات) و معماری داده‌گرا تقسیم می‌گردد.

۱- معماری تابع‌گرا (جریان داده)

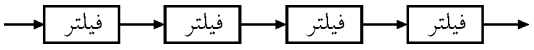
معماری جریان داده یا تابع‌گرا یا تجزیه عملیات هنگامی به کار برده می‌شود که قرار است داده‌های ورودی از طریق یک سری مؤلفه‌های محاسباتی یا دست کاری کننده، به داده‌های خروجی تبدیل شوند، الگوی «لوله و فیلتر» (شکل الف) دارای مجموعه‌ای از قطعات، موسوم به فیلتر است که توسط لوله‌هایی به هم متصل می‌شوند، وظیفه‌ی این لوله‌ها انتقال داده‌ها از مؤلفه‌ای به مؤلفه‌ی بعدی است. هر فیلتری مستقل از مؤلفه‌های دیگر عمل می‌کند و برای پذیرش داده‌های ورودی به شکلی خاص طراحی شده است، بنابراین داده‌های خروجی (برای فیلتر بعدی) را به شکلی خاص تولید می‌کند. ولی این فیلتر از نحوه کار فیلترهای مجاور خود هیچ اطلاعی ندارد.

اگر جریان داده‌ها تا حد یک خط تبدیل تنزل یابد (شکل ب)، آن را دنباله‌ای دسته‌ای (Batch Sequential) می‌نامند. این الگو یک دسته از داده‌ها را پذیرفته سپس، از یک سری

مؤلفه‌های ترتیبی (فیلتر) برای تبدیل آن استفاده می‌کند.



الف) لوله‌ها و فیلترها



ب) دنباله‌ای دسته‌ای

توجه: معماری تابع‌گرا، برای کاربردهایی با محاسبات پیچیده و سنگین مورد استفاده قرار می‌گیرند، مانند نرم‌افزارهای محاسبات ریاضی. در این نوع کاربردها حجم داده‌ها پایین و حجم محاسبات بالا است. برای مثال نرم‌افزار ماشین حساب از این مدل معماری استفاده می‌کند.

توجه: نوع داده در این مدل معماری، متغیر ساده است.

۲- معماری داده‌گرا

توجه: معماری داده‌گرا، برای کاربردهای بانک اطلاعات، مورد استفاده قرار می‌گیرد. در این نوع کاربردها حجم داده‌ها بالا و حجم محاسبات پایین است.

توجه: نوع داده در این مدل معماری، جداول و رکوردهای بانک اطلاعات است.

توجه: معماری داده‌گرا به دو مدل متمرکز و نامتمرکز (client/server) تقسیم می‌گردد.

مدل متمرکز

در این معماری یک بانک اطلاعات روی یک سیستم کامپیوتری و بدون ارتباط با سیستم های کامپیوتری دیگر ایجاد می‌شود. و برای کاربردهای کوچک و با امکانات محدود است. سخت‌افزار این سیستم می‌تواند در حد یک کامپیوتر شخصی و یا بالاتر باشد. در این مدل، بانک اطلاعات و برنامه کاربردی آن، هر دو بر روی یک کامپیوتر قرار دارند. برای مثال به نرم‌افزار دفترچه تلفن بر روی یک کامپیوتر شخصی می‌توان اشاره نمود.

توجه: به مدل متمرکز، مدل برنامه اصلی و برنامه فرعی نیز گفته می‌شود.

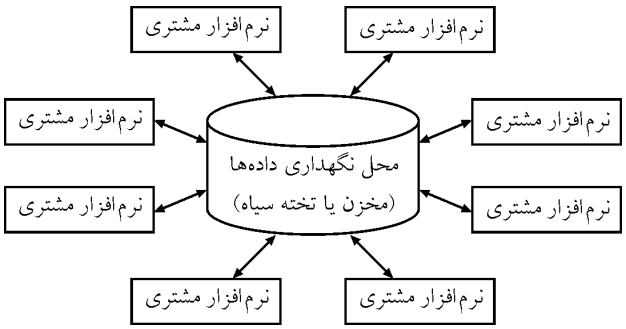
مدل نامتمرکز (client/server)

در این مدل، بانک اطلاعات و برنامه کاربردی آن، بر روی کامپیوترهای مختلفی قرار دارند. در مدل نامتمرکز مجموعه‌هایی از serverها، مجموعه‌ای از سرویس‌ها را به clientها از طریق شبکه‌های کامپیوتری ارئه می‌دهند.

توجه: به مدل نامتمرکز، مدل فراخوانی روال از راه دور نیز گفته می‌شود.
توجه: مدل نامتمرکز (client/server) به دو صورت زیر می‌باشد:
۱- مدل داده محور (اشتراکی یا مخزنی)
۲- مدل توزیع شده

مدل داده محور یا اشتراکی یا مخزنی (Data Repository)

یک مخزن داده‌ای یا بانک اطلاعات (Server) در مرکز این معماری قرار دارد و مؤلفه‌های دیگری (Client) که داده‌های آن را بروزرسانی، درج یا حذف می‌کنند، به دفعات به آن دستیابی دارند. در برخی موارد، مخزن داده‌ها انفعالی است. یعنی، نرم‌افزار مشتری، مستقل از هر گونه عملیاتی که توسط نرم‌افزارهای مشتری دیگری انجام می‌شود یا تغییرات اعمال شده روی داده‌ها، به آن‌ها دستیابی دارد. در شکل دیگری از این روش، مخزن به یک تخته سیاه تبدیل می‌شود که در صورت تغییر داده‌های موردنظر نرم‌افزار مشتری، پیامی به آن ارسال می‌کند.



معماری داده محوری، باعث بهبود قابلیت جامعیت می‌شود. یعنی، مؤلفه‌های موجود را می‌توان تغییر داد یا مؤلفه‌های مشتری جدید را به معماری افزود، بدون آنکه نیازی به در نظر گرفتن مشتریان دیگر باشد (زیرا مؤلفه‌های مشتری مستقل از هم کار می‌کنند) به علاوه، داده‌ها را می‌توان با استفاده از راهکار تخته سیاه در میان مشتریان مبادله کرد (یعنی مؤلفه تخته سیاه به هماهنگ سازی انتقال اطلاعات بین مشتریان کمک می‌کند) مؤلفه‌های مشتری پردازش‌هایی هستند که مستقل از هم اجرا می‌شوند.

برای مثال به clientهای موجود در شعبه‌های یک بانک سرمایه که با server مرکزی در ارتباط هستند می‌توان اشاره نمود.

توزیع شده (Distributed)

اصطلاح پردازش توزیعی بدین مفهوم است که ماشین‌های مجزا می‌توانند در یک شبکه ارتباطی به یکدیگر متصل شوند. به گونه‌ای که یک عملکرد پردازش داده منفرد می‌تواند بر روی چندین ماشین شبکه پراکنده شود. در سیستم پایگاه داده‌های توزیع شده، پایگاه داده‌ها بر روی چند کامپیوتر ذخیره می‌شود. کامپیوترها در یک سیستم توزیع شده از طریق شبکه‌های کامپیوتری

به یکدیگر مرتبط هستند. می‌توان گفت که در این مدل، تعدادی پایگاه داده‌های ذخیره شده روی کامپیوترهای مختلف وجود دارد، که از نظر کاربران، پایگاه داده واحدی هستند. به بیان دیگر داده‌ها در محل‌های مختلف ذخیره شده‌اند، اما کاربران به هنگام استفاده از داده‌ها به ظاهر پایگاه داده‌ای یکپارچه را مشاهده می‌کنند.

برای مثال به clientهای موجود در شعبه‌های یک بانک سرمایه که با server توزیع شده در ارتباط هستند می‌توان اشاره نمود.

توجه: سیستم‌های توزیع شده، بر دو دسته، فرآیندهای توزیع شده توسط مکانیزم نخ، و داده‌های توزیع شده، طبقه‌بندی می‌گردد، که مورد اخیر مورد بحث ما بود. مورد اول یعنی فرآیندهای توزیع شده توسط مکانیزم نخ نیز شاخه‌ای از مباحث ساخت برنامه‌های کامپیوتری است. که در مبحث برنامه‌نویسی مبتنی بر نخ به آن پرداخته می‌شود.

نگاشت مدل تحلیل عملکردی به مدل طراحی معماری

همانطور که گفتیم، مدل تحلیل، مدل‌سازی عالم خارج به زبانی شبیه انسان است، اما مدل طراحی مدل‌سازی عالم داخل ماشین به زبانی شبیه زبان ماشین است. بنابراین برای اینکه ساخت برنامه کامپیوتری که به زبان ماشین است، امکان‌پذیر باشد، باید مدل تحلیل به مدل طراحی نگاشت شود تا مدل طراحی بتواند به عنوان نقشه راهی شبیه به زبان ماشین، راهگشا باشد. طراحی معماری، تحلیل عملکرد (DFD) از مدل تحلیل را به عنوان ورودی دریافت کرده و توسط سبک ساخت‌یافته (فراخوانی و بازگشت)، طراحی معماری را انجام می‌دهد.

برای نگاشت مدل تحلیل عملکردی که در مرحله مدل تحلیل به صورت نمودار جریان داده (DFD) نشان داده می‌شود به طراحی معماری (ساختار برنامه) از شش قدم زیر استفاده می‌شود:

۱- تعیین نوع نمودار جریان داده (تعیین نوع DFD)
۲- تعیین محدودیت‌های جریان
۳- نگاشت DFD به طراحی معماری (ساختار برنامه)
۴- تعیین سلسله مراتب کنترل
۵- پالایش ساختار حاصل با استفاده از اصول طراحی معماری
۶- پالایش معماری با ساختار نهایی

توجه: تعیین نوع DFD باید بر روی آخرین سطح DFD انجام گیرد.

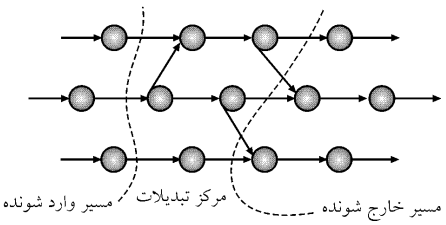
توجه: نوع نمودار جریان داده که در قدم ۱ تعیین می‌شود، نوع نگاشت لازم برای قدم ۳ را مشخص می‌کند.

انواع نمودار جریان داده

۱- جریان تبدیل (Transform Flow)

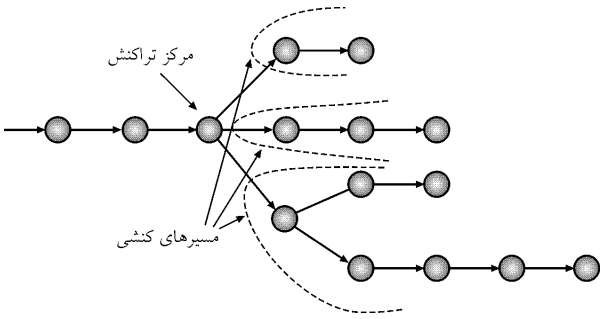
با توجه به مدل پایه‌ای سیستم (سطح صفر DFD)، ابتدا اطلاعات وارد نرم‌افزار شده، سپس عملیات پردازش بر روی داده‌ها انجام می‌گیرد و در آخر داده‌های حاصل به خروجی منتقل می‌شود. در جریان تبدیل، داده‌ها از یک یا چند مسیر وارد شده و طی یک یا چند تبدیل به

خروجی منتقل می‌شوند. وقتی بخشی از یک دیاگرام جریان داده دارای چنین خاصیتی باشد در واقع یک جریان تبدیل داریم. محدوده‌ای که طی آن داده‌ها از شکل خارجی به شکل داخلی تغییر می‌کنند، به نام مسیر واردشونده و محدوده‌ای که داده‌ها از شکل داخلی به شکل خارجی منتقل می‌شوند مسیر خارج‌شونده می‌نامند. حد بین این دو جریان نیز به عنوان «**مرکز تبدیل**» شناخته می‌شود.



۲- جریان تراکنش (Transaction Flow)

اگر در جریان تراکنش، یک جریان ورودی به یک حباب مرکزی وارد شده و همه خروجی‌ها به صورت مشخص از یک حباب مرکزی سرچشمه گیرند، اصطلاحاً به چنین حبابی، «**مرکز تراکنش**» می‌گویند و به تمامی مسیرهایی که از مرکز تراکنش آغاز می‌گردند، مسیر کنش گفته می‌شود.



توجه: در صورتی که مراکز تبدیل و تراکنش چندین خروجی داشته باشند، برای مرکز تبدیل همه مسیرهای خروجی همزمان اتفاق می‌افتند، در حالی که برای مرکز تراکنش فقط یکی از مسیرهای کنش انتخاب می‌شود. به بیان دیگر انتخاب مسیر کنش در جریان تراکنش بر عهده مرکز تراکنش یا حباب مرکزی یا حباب تصمیم گیرنده است.

توجه: بدیهی است که یک DFD می‌تواند هم جریان تبدیل و هم جریان تراکنش داشته باشد. به عنوان مثال در یک جریان تراکنش، جریان اطلاعات در راستای یک مسیر کنش، ممکن است دارای خصوصیات جریان تبدیل باشد.

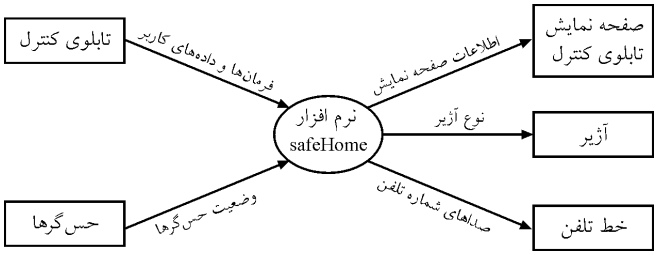
نگاشت تبدیل

نگاشت تبدیل، بخشی از فعالیت طراحی است که یک DFD با خواص جریان تبدیل را به یک ساختار یا طراحی معماری تبدیل می‌کند.

مراحل انجام این کار را برای بخشی از «نرم‌افزار SafeHome» که وظیفه آن حفاظت خودکار از منزل است، بررسی می‌کنیم.

۱- بازنگری مدل پایه‌ای سیستم

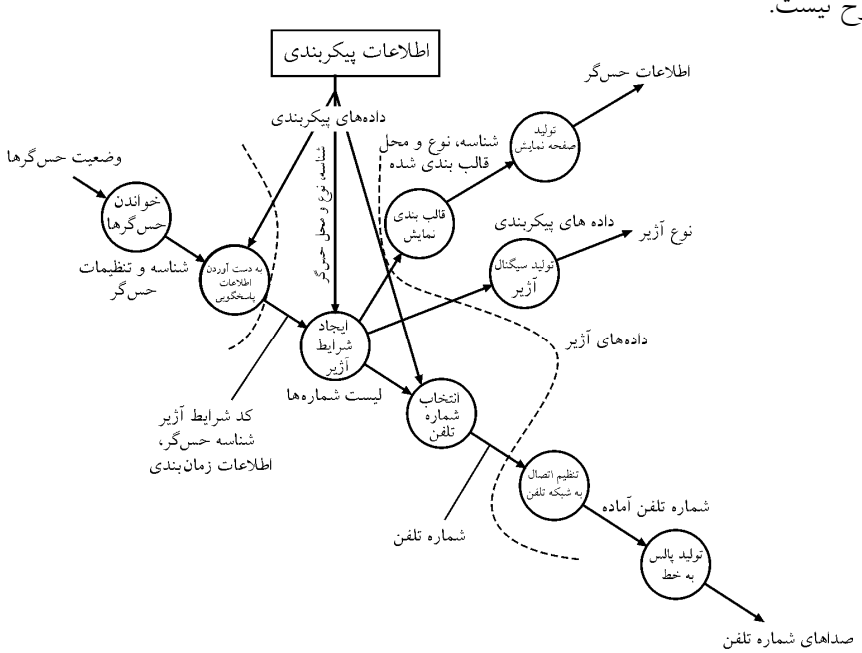
این مدل شامل DFD سطح صفر و اطلاعات مرتبط با آن می‌باشد.



DFD سطح صفر برای نرم‌افزار SafeHome

۲- بازنگری و پالایش نمودارهای جریان داده نرم‌افزار در سطوح پایین‌تر

در این مرحله اطلاعات قبل، جهت تولید جزییات بیشتر، پالایش و بازنگری می‌شود. یعنی با استفاده از اطلاعات حاصل از مدل‌های تحلیل موجود در مشخصات نیازهای نرم‌افزار، جزییات بیشتری به DFD افزوده شده و DFD سطوح پایین‌تر ترسیم می‌گردد. برای مثال فوق، پالایش تا DFD سطح سوم، جزییات کافی را در اختیار قرار می‌دهد و بیش از آن نیاز به پایین رفتن در سطوح نیست.



DFD سطح سوم برای نرم‌افزار SafeHome

۳- تعیین نوع جریان (جریان تبدیل یا جریان تراکنش) در DFD

با دقت در DFD سطح سه نرم‌افزار SafeHome مشخص می‌شود که هیچ «مرکز تراکنش» واضحی وجود ندارد. زیرا همه خروجی‌ها به صورت مشخص از یک پردازش مرکزی سرچشمه نمی‌گیرند. پس جریان از نوع تبدیل است.

۴- تفکیک مرکز تبدیل به وسیله مرزهای جریان ورودی و خروجی

مرزهای ورودی و خروجی برای نرم‌افزار SafeHome در DFD سطح سوم آن با منحنی‌های خط‌چین نشان داده شده است و بین آنها «مرکز تبدیل» قرار گرفته است.

۵- فاکتورگیری سطح اول

در این حالت DFD به معماری فراخوانی و بازگشت تبدیل می‌شود.

برای انجام فاکتورگیری از قرارداد زیر استفاده می‌شود:

قرارداد:

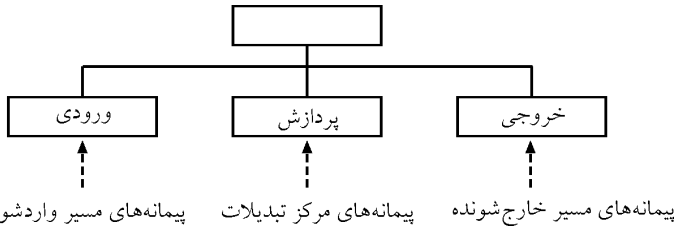
طبق قرارداد در افراز افقی برای جریان‌های تبدیل همواره سه واحد کنترل وجود دارد:

۱- کنترل‌کننده ورودی

۲- کنترل‌کننده خروجی

۳- کنترل‌کننده پردازش

به طور کلی ساختار فاکتورگیری سطح اول به صورت زیر است:



توجه: فاکتورگیری باعث ایجاد ساختاری می‌شود که در آن پیمانه‌های سطح بالا کارهای تصمیم‌گیری و پیمانه‌های سطح پایین نیز اکثراً کارهای ورودی، پردازش و خروجی را انجام می‌دهند. پیمانه‌های میانی قدری کنترل و قدری کار انجام می‌دهند. در واقع ساختار برنامه، توزیع بالا به پایین از کنترل را نشان می‌دهد.

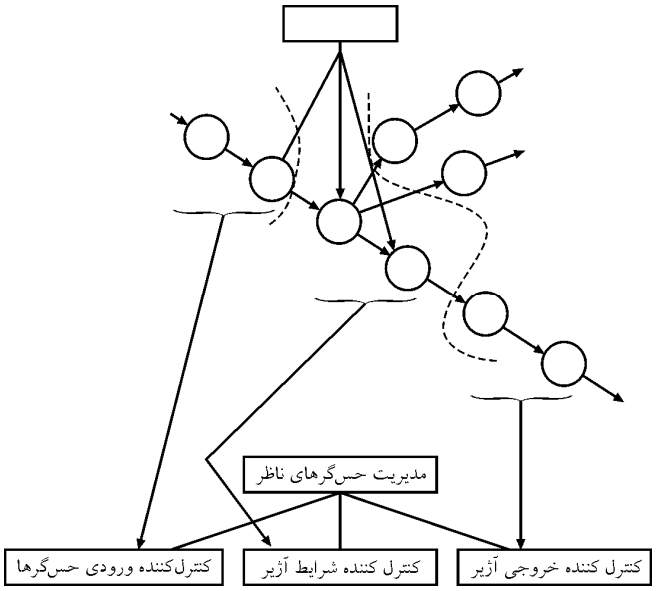
توجه: یک کنترل‌کننده اصلی موسوم به «**مجری نظارت بر سنسورها**» در بالای ساختار برنامه قرار دارد و به هماهنگ کردن عملیات کنترلی زیردستان کمک می‌کند.

الف) یک کنترل‌کننده پردازش اطلاعات ورودی موسوم به «**کنترل‌کننده ورودی حسگرها**» دریافت کلیه داده‌های ورودی را هماهنگ می‌کند.

ب) یک کنترل‌کننده جریان تبدیلی موسوم به «**کنترل‌کننده شرایط آژیر**» برای انجام کلیه عملیات بر روی داده‌ها به شکل داخلی آن نظارت دارند (برای مثال پیمانه‌ای که روال‌های

گوناگون تبدیل داده‌ها را فراخوانی می‌کنند).

ج) یک کنترل‌کننده پردازش اطلاعات خروجی، که «**کنترل‌کننده خروجی آژیر**» نامیده می‌شود، تولید اطلاعات خروجی را هماهنگ می‌کند.



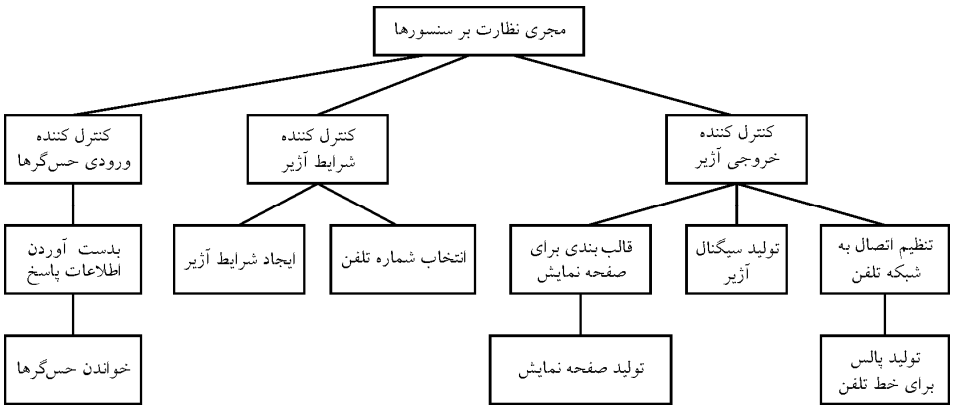
فاکتورگیری سطح اول برای DFD سطح سوم نرم‌افزار SafeHome

۶- فاکتورگیری سطح دوم

روند نگاشت حباب‌ها به محل مناسب در ساختار معماری، در شاخه کنترل‌کننده ورودی، بدین نحو است که باید از مرز جریان ورودی DFD به سمت خارج DFD حرکت نمود و حباب‌هایی که در هر شاخه دیده می‌شود را بعنوان مولفه فرزند به ساختار معماری اضافه نمود، برای مثال در شاخه کنترل‌کننده ورودی در DFD مطرح شده، با اولین حرکت از مرز ورودی DFD به سمت خارج DFD، یک حباب «به دست آوردن اطلاعات پاسخ» دیده می‌شود. این حباب به عنوان مولفه فرزند مولفه «کنترل‌کننده ورودی حس‌گرها» به ساختار معماری اضافه می‌گردد. با ادامه این حرکت به سمت خارج، حباب «خواندن حسگرها» دیده می‌شود. که در شاخه حباب «به دست آوردن اطلاعات پاسخ» قرار دارد. این حباب نیز به عنوان مولفه فرزند مولفه «به دست آوردن اطلاعات پاسخ» به معماری اضافه می‌گردد. روند نگاشت حباب‌ها به محل مناسب در ساختار معماری، در شاخه کنترل‌کننده خروجی نیز، بدین نحو است که باید از مرز جریان خروجی DFD به سمت خارج DFD حرکت نمود و حباب‌هایی که در هر شاخه دیده می‌شود را بعنوان مولفه فرزند به ساختار معماری اضافه نمود. برای مثال در شاخه کنترل‌کننده خروجی در DFD مطرح شده، با اولین حرکت از مرز خروجی DFD به سمت خارج DFD، سه حباب «قالب‌بندی صفحه نمایش»، «تولید سیگنال آژیر» و «تنظیم اتصال به شبکه تلفن» دیده می‌شود. این حباب‌ها به عنوان

مولفه‌های فرزند مولفه «کنترل‌کننده خروجی آژیر» به ساختار معماری اضافه می‌گردد. با ادامه این حرکت به سمت خارج، حباب «تولید صفحه نمایش» دیده می‌شود. که در شاخه حباب «قالب‌بندی برای صفحه نمایش» قرار دارد. این حباب نیز به عنوان مولفه فرزند مولفه «قالب‌بندی برای صفحه نمایش» به معماری اضافه می‌گردد. همچنین حباب «تولید پالس برای خط تلفن» دیده می‌شود. که در شاخه حباب «تنظیم اتصال به شبکه تلفن» قرار دارد. این حباب نیز به عنوان مولفه فرزند مولفه «تنظیم اتصال به شبکه تلفن» به معماری اضافه می‌گردد.

روند نگاشت حباب‌ها به محل مناسب در ساختار معماری، در شاخه کنترل‌کننده پردازش با کنترل‌کننده ورودی و خروجی متفاوت است، بدین نحو است که مستقل از سریال یا موازی بودن حباب‌های مرکز تبدیل، همه حباب‌های مرکز تبدیل، به عنوان مولفه‌های فرزند مولفه کنترل‌کننده پردازش به ساختار معماری اضافه می‌گردند. برای مثال در شاخه کنترل‌کننده پردازش در DFD مطرح شده، مستقل از موازی یا سریالی بودن حباب‌ها، حباب «انتخاب شماره تلفن» و «ایجاد شرایط آژیر» به عنوان مولفه‌های فرزند مولفه «کنترل‌کننده شرایط آژیر» به ساختار معماری اضافه می‌گردند. در یک بیان کلی، در این مرحله هر کدام از حباب‌های DFD در سطح مورد نظر، توسط یک مؤلفه مرتبط در ساختار معماری مشخص می‌شود. به بیان دیگر فاکتورگیری در بقیه سطوح پایین‌تر ساختار معماری، متناظر با حباب‌های موجود در DFD انجام می‌شود و مولفه‌های مناسب در طراحی معماری به دست می‌آید.



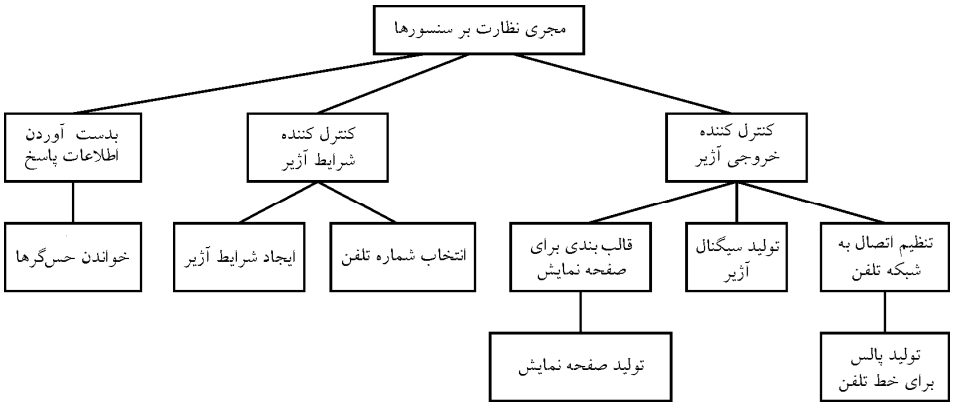
فاکتورگیری سطح دوم برای DFD سطح سوم نرم‌افزارSafeHome

۷- پالایش ساختار معماری برای افزایش کیفیت نرم‌افزار

این کار عموماً از طریق تجزیه و ترکیب پیمانه‌ها در راستای افزایش انسجام و کاهش اتصال انجام می‌گردد. هرگاه در طراحی معماری، یک مولفه کنترل‌کننده (پدر)، فقط یک مولفه تحت کنترل (فرزند) داشته باشد، می‌توان مولفه کنترل‌کننده (پدر) را حذف نمود.

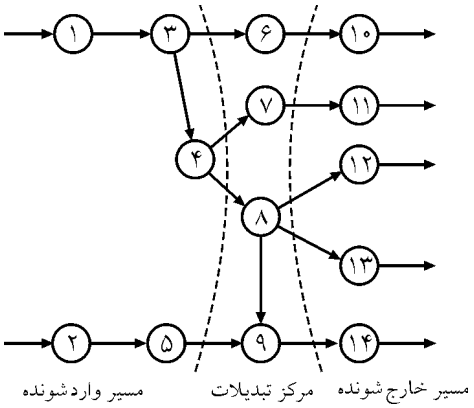
در نخستین پالایش معماری نرم‌افزار SafeHome می‌توان اصلاحاتی انجام داد. برای مثال، پیمانه «کنترل کننده ورودی حس گرها» را می‌توان حذف نمود. شکل زیر، ساختار پالایش شده

طراحی معماری می‌باشد، زیرا کنترل کننده‌های اضافی، حذف شده‌اند.



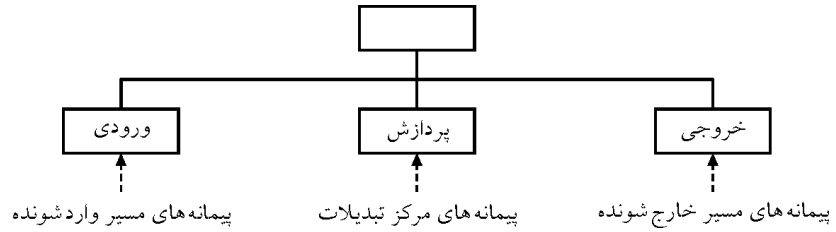
معماری پالایش شده برای نرم‌افزار SafeHome

مثال: طراحی معماری متناظر با DFD زیر را به دست آورید:

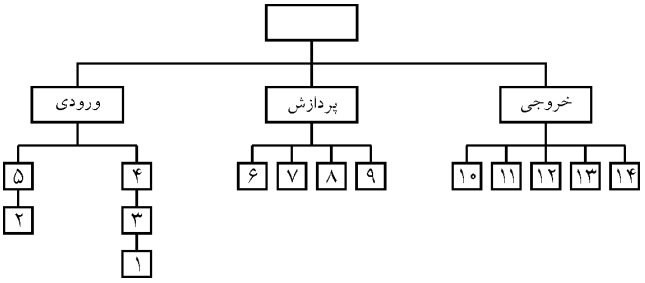


با توجه به قرارداد بیان شده، داریم:

فاکتورگیری سطح اول:

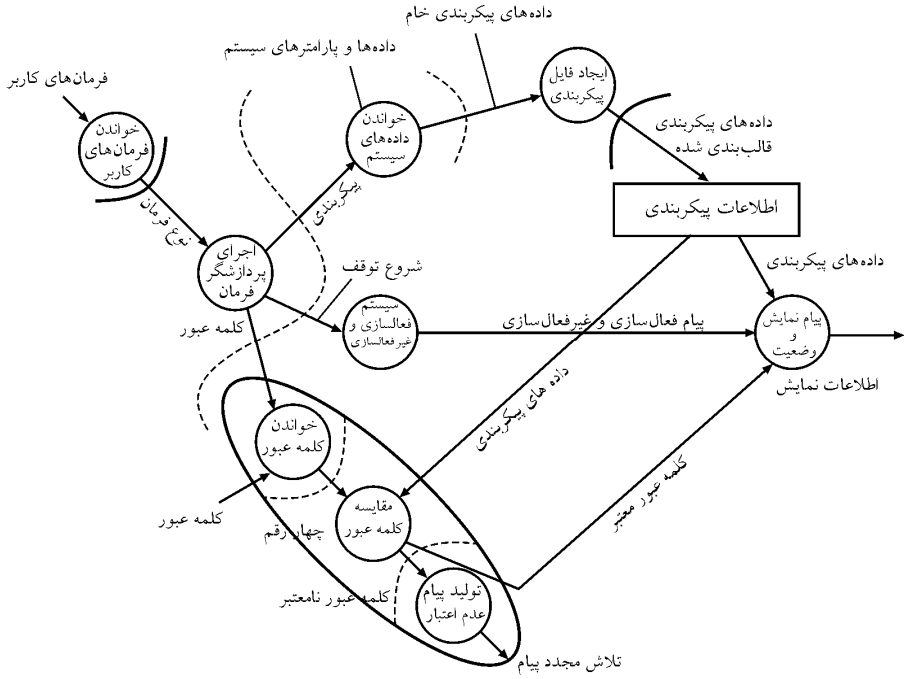


فاکتورگیری سطح دوم:



نگاشت تراکنش

برای بررسی نگاشت تراکنش، زیر سیستم محاوره با کاربر را در سیستم SafeHome در نظر می‌گیریم. نمودار زیر، DFD سطح دوم آن را نشان می‌دهد.



DFD سطح دوم برای زیر سیستم محاوره با کاربر در نرم افزار SafeHome

حال مراحل نگاشت تراکنش را برای زیر سیستم فوق مورد بررسی قرار می‌دهیم:

مراحل نگاشت تراکنش همانند نگاشت تبدیل است، بنابراین تنها به تشریح مراحل مورد تفاوت می‌پردازیم:

۱- بازنگری مدل پایه‌ای سیستم

۲- بازنگری و پالایش نمودارهای جریان داده نرم‌افزار در سطوح پایین‌تر

۳- تعیین نوع جریان (جریان تبدیل یا جریان تراکنش) در DFD

با دقت در DFD سطح دوم برای زیر سیستم محاوره با کاربر نرم‌افزار SafeHome مشخص می‌شود که یک «مرکز تراکنش» با نام حباب «اجرای پردازشگر فرمان» وجود دارد. زیرا همه خروجی‌ها به صورت مشخص از این پردازش مرکزی سرچشمه می‌گیرند. پس جریان از نوع تراکنش است. همچنین مسیرهای بالایی و پایینی از نوع جریان تبدیل هستند. بنابراین DFD مورد نظر از هر دو نوع جریان، یعنی تراکنش و تبدیل است.

۴- تفکیک مرکز تراکنش و تبدیل به وسیله مرزهای جریان ورودی و خروجی

هر مسیر فعالیت باید جداگانه بررسی شود تا خواص آن نظیر تبدیلی یا تراکنشی بودن آن معین شود. در نمودار فوق، حباب «اجرای پردازشگر فرمان» به عنوان «مرکز تراکنش» بوده و دارای سه مسیر تراکنش است، مسیر بالایی و پایینی دارای «جریان تبدیل» نیز هستند که مرکز تبدیل آن‌ها به ترتیب «ایجاد فایل پیکربندی» و «مقایسه کلمه عبور» می‌باشد. مرزبندی این دو جریان تبدیل در شکل فوق مشخص شده است.

توجه: حباب «اجرای پردازشگر فرمان» مرکز تراکنش است، زیرا همه خروجی‌ها به صورت مشخصی از این پردازش مرکزی سرچشمه می‌گیرند.

۵- فاکتورگیری سطح اول

در این حالت DFD به معماری فراخوانی و بازگشت تبدیل می‌شود.

برای انجام فاکتورگیری از قرارداد زیر استفاده می‌شود:

قرارداد:

طبق قرارداد، در افراز افقی برای جریان‌های تراکنشی همواره دو شاخه وجود دارد:

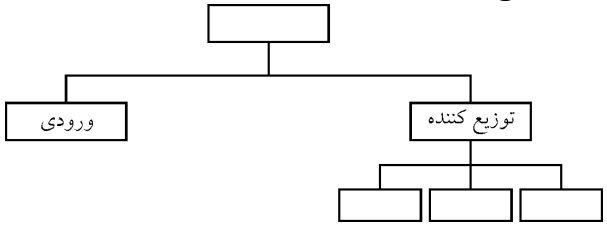
۱- شاخه ورودی (کنترل‌کننده دریافت)

۲- شاخه توزیع‌کننده (کنترل‌کننده ارسال)

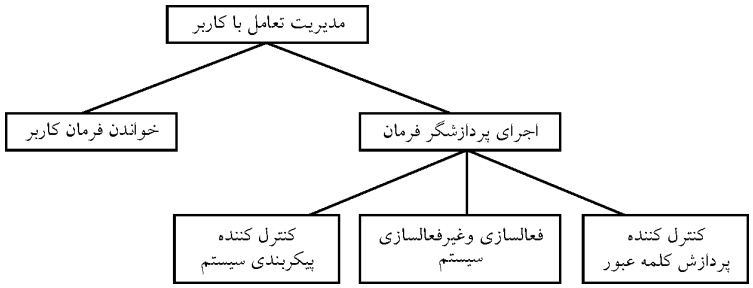
توجه: شاخه توزیع کننده (کنترل‌کننده ارسال)، همان حباب مرکز تراکنش است.

شاخه توزیع، یک پیمانه توزیع‌کننده است که تمامی پیمانه‌های فعالیت زیرین را کنترل می‌کند. هر کدام از مسیرهای فعالیت نیز به ساختاری متناظر خواص خودشان تصویر می‌شوند.

ساختار فاکتورگیری سطح اول به صورت زیر است:



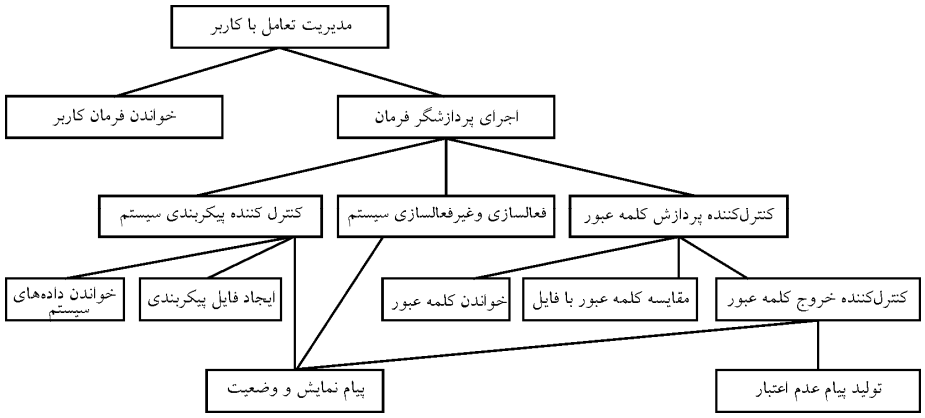
همان‌طور که در DFD سطح دوم زیر سیستم محاوره با کاربر در نرم افزار SafeHome مشاهده می‌شود، مسیر بالایی و پایینی دارای جریان تبدیل هستند، بنابراین در ساختار فاکتورگیری سطح اول نگاشت تراکنش، مطابق آنچه پیش از این در مورد نگاشت جریان تبدیل گفتیم، یک کنترل‌کننده برای هر یک از مسیرهای مذکور در نظر گرفته می‌شود.



فاکتورگیری سطح اول برای زیرسیستم محاوره با کاربر

۶- فاکتورگیری سطح دوم

هر مسیر فعالیت براساس خواص خود به ساختار متناظرش تصویر می‌شود. روند نگاشت حباب‌ها به محل مناسب در ساختار معماری، کاملا مشابه نگاشت تبدیل است، یعنی حرکت از مرزهای DFD، به سمت خارج DFD است، در بخش ورود اطلاعات حباب «خواندن فرمان‌های کاربر» دیده می‌شود. همچنین در بخش مسیرهای کنشی سه مسیر بالایی، میانی و پایینی وجود دارد. مطابق شکل مسیرهای بالایی و پایینی از جنس جریان تبدیل و مسیر میانی مسیر ساده است. بنابراین نگاشت مسیر میانی به شکل عادی انجام می‌گردد، یعنی حرکت از مرزهای DFD، به سمت خارج DFD.. اما مسیرهای بالایی و پایینی از جنس جریان تبدیل هستند. و مطابق آنچه پیش از این در مورد نگاشت تبدیل گفتیم، نگاشت جریان تبدیل انجام می‌گردد. نتیجه این کار در شکل زیر نشان داده شده است.



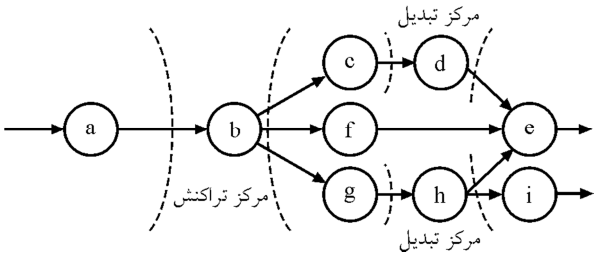
معماری زیرسیستم محاوره با کاربر

۷- پالایش ساختار معماری برای افزایش کیفیت نرم‌افزار

این کار عموماً از طریق تجزیه و ترکیب پیمانه‌ها در راستای افزایش انسجام و کاهش اتصال انجام می‌گردد. هرگاه در طراحی معماری، یک مولفه کنترل‌کننده (پدر)، فقط یک مولفه تحت کنترل (فرزند) داشته باشد، می‌توان مولفه کنترل‌کننده (پدر) را حذف نمود.

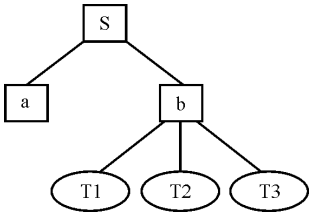
شکل فوق، ساختار پالایش شده طراحی معماری می‌باشد، زیرا کنترل کننده‌های اضافی، حذف شده‌اند.

مثال: طراحی معماری متناظر با DFD داده شده را به دست آورید:



پاسخ: درDFD فوق، دو مسیر بالایی و پایینی داری جریان تبدیل هستند، بنابراین در ساختار کلی طراحی معماری، باید نگاشت تبدیل نیز مورد استفاده قرار گیرد.

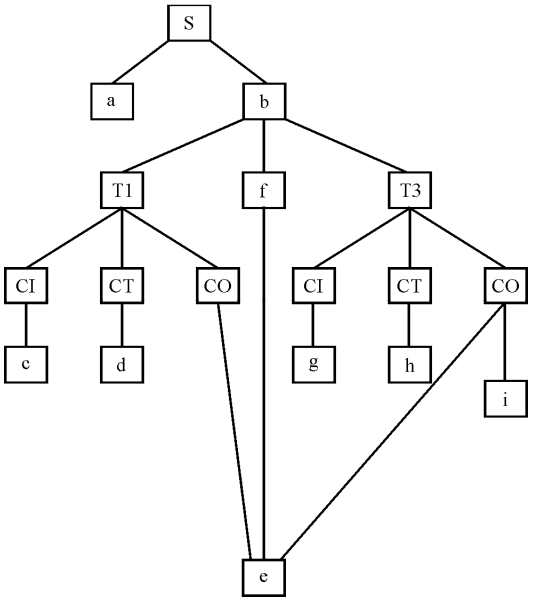
در گام اول، مطابق قوانین مطرح شده برای نگاشت تراکنش بر اساس DFD مطرح شده، داریم:



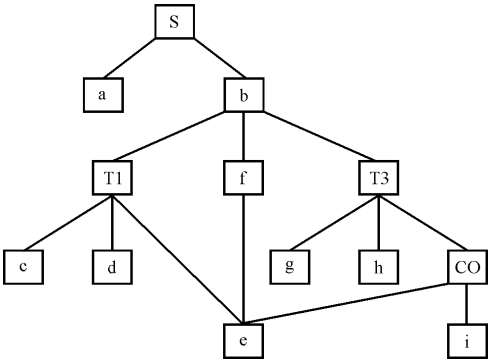
توجه: در ساختار فوق، سه مسیر T1، T2 و T3 به ترتیب سه مسیر کنشی، بالایی، میانی و پایینی را نشان می‌دهند.

توجه: مسیرهای T1 و T3، مسیرهایی از جنس جریان تبدیل و مسیر T2 مسیر ساده است. و نگاشت آن به شکل عادی انجام می‌گردد، یعنی حرکت از مرزهای DFD، به سمت خارج DFD است.

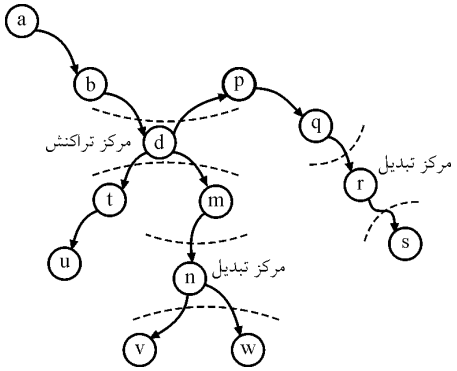
مطابق آنچه پیش از این در مورد نگاشت تبدیل گفتیم، طراحی معماری زیر ایجاد می‌گردد:



همچنین پس از پالایش، طراحی معماری زیر ایجاد می‌گردد:

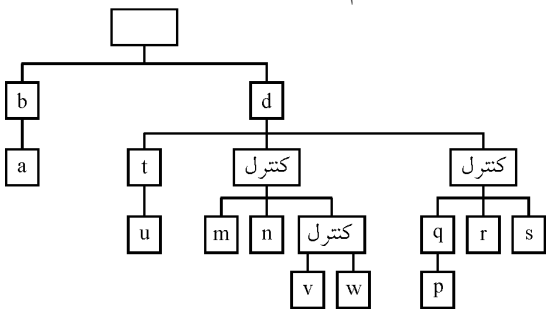


مثال: طراحی معماری متناظر با DFD داده شده را به دست آورید:



پاسخ: از آنجا که همه خروجی‌ها به صورت مشخص از پیمانه d سرچشمه می‌گیرند، لذا پیمانه d «مرکز تراکنش» است و در مسیر فعالیت qrs، پیمانه r «مرکز تبدیل» است. همچنین در مسیر میانی پیمانه n «مرکز تبدیل» است.

با توجه به قراردادهای بیان شده داریم:



شاخص گزینش طراحی

نیازمندی‌های عملکردی مشتری در مدل تحلیل توسط DFD، مدل‌سازی عملکردی می‌شود، سپس توسط نگاشتی که بر اساس سبک فراخوانی و بازگشت انجام می‌گردد، هر کدام از حباب‌های DFD در سطح مورد نظر، توسط یک مؤلفه مرتبط در ساختار معماری مشخص می شود. به بیان دیگر طراحی معماری، متناظر با حباب‌های موجود در DFD انجام می‌شود و مولفه‌های مناسب در طراحی معماری به دست می‌آید. به عبارت دیگر هر حباب موجود در DFD، به یک مولفه در ساختار طراحی معماری نگاشت می‌گردد. بنابراین هرچه نیازمندی‌های عملکردی، بهتر شناخته شوند، حباب‌های موجود در DFD کامل‌تر خواهد بود، و به تبع در نگاشت DFD به ساختار معماری، طراحی معماری نیز ایده‌آل‌تر خواهد بود. میزان ایده‌آل بودن یک طراحی معماری، از رابطه «شاخص گزینش طراحی» به صورت زیر محاسبه می‌گردد:

$$d = \left(\frac{N_S}{N_a}\right)^{*۱۰۰}$$

که در آن N_S تعداد ابعاد (مولفه‌های) به کار رفته در یک معماری پیشنهادی است، که توسط سازنده شناسایی شده‌اند.

و N_a تعداد کل ابعاد (مولفه‌های) مورد نظر مشتری در فضای طراحی است، یعنی آنچه مطلوب مشتری است تا توسط سازنده شناخته شود.

مقدار شاخص گزینش طراحی (d)، مقداری بین ۰ و ۱۰۰ برحسب درصد است. هرچه این مقدار، به عدد ۱۰۰ نزدیکتر باشد، معماری پیشنهادی به معماری ایده‌آل نزدیکتر است.

مثال: اگر مقدار N_a برابر ۲۰ و مقدار N_S برابر ۱۵ باشد، مقدار شاخص گزینش طراحی (d)، برابر ۷۵٪ خواهد بود.

مثال: اگر مقدار N_a برابر ۲۰ و مقدار N_S برابر ۲۰ باشد، مقدار شاخص گزینش طراحی (d)، برابر ۱۰۰٪ خواهد بود.

همان‌طور که قبلاً بیان کردیم، فعالیت مدل طراحی شامل طراحی داده، طراحی معماری، طراحی واسط کاربر و طراحی مؤلفه می‌باشد. تا به اینجا طراحی داده و طراحی معماری را به طور مفصل مورد بررسی قرار دادیم. حال در ادامه به بررسی طراحی مؤلفه و طراحی واسط کاربر می‌پردازیم:

طراحی مولفه

طراحی مؤلفه، طراحی معماری از همان فعالیت مدل طراحی را به عنوان ورودی دریافت کرده و طراحی مؤلفه را توسط ابزارهایی همچون شبه کد یا فلوچارت ایجاد می‌کند.

طراحی مولفه، فعالیت تبدیل طراحی معماری به نرم‌افزار است. در این مرحله، سطح انتزاع طراحی معماری به سطح انتزاع نرم‌افزار کاربردی نزدیک می‌گردد. طراحی در سطح مؤلفه‌ها، نرم افزار را در سطحی از انتزاع تصویر می‌کند که به کد نزدیک است. طراحی مولفه، به عنوان نقشه راهی دقیق، و نزدیک به زبان پیاده‌سازی، در فعالیت پیاده‌سازی نرم‌افزار، منجر به صرفه جویی در زمان و هزینه‌های تولید می‌گردد. در طراحی مولفه، مهندس نرم‌افزار باید ساختمان داده‌ها، واسط‌ها و الگوریتم‌ها را با جزییات کافی به نمایش در آورد تا راهنمای تولید کد منبع زبان برنامه‌نویسی باشد.

توجه: در طراحی مؤلفه، اسکلت، ساختار و چیدمان کلی مولفه‌های (توابع) برنامه به این معنی که چه مولفه‌ای (تابعی) چه مولفه‌ای (تابعی) دیگر را صدا می‌زند، با ذکر جزئیات داخلی مولفه‌ها (توابع) مشخص می‌گردد (ساختار درختی برنامه با ذکر جزئیات مولفه‌ها (توابع)). مانند اسکلت یک ساختمان که گویای جایگاه مؤلفه‌های ساختمان است و آجرچینی هم شده است. (اسکلت یک ساختمان به همراه آجرچینی).

توجه: به طراحی مولفه، طراحی جزئی، طراحی تفصیلی و طراحی رویه‌ای نیز گفته می‌شود.

توجه: مؤلفه یک قطعه‌ی آماده، کاربردی و پیاده‌سازی شده است که دارای واسط لازم جهت اتصال با سایر قطعات و استقرار در بخشی از یک سیستم عملیاتی می‌باشد.

در دیدگاه ساخت‌یافته به مؤلفه پیمانه نیز گفته می‌شود. در حیطه‌ی مهندسی نرم‌افزار ساخت‌یافته، مولفه یک قطعه عملیاتی مبتنی بر تابع است که بر دو نوع می‌باشد:

۱) تابع کاربردی: مانند تابع جمع که برنامه‌نویس آن را می‌نویسد. اگر تابع کاربردی، شرایط قابل حمل بودن (مانند تعریف متغیر درون تابع به صورت محلی و صریح و عدم استفاده از متغیرهای سراسری) را داشته باشد می‌تواند به عنوان یک قطعه‌ی آماده‌ی قابل استفاده‌ی مجدد در پروژه‌های بعدی مورد استفاده مجدد قرار گیرد.

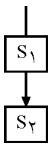
۲) تابع سیستمی: مانند تابع سینوس که کامپایلر تعاریف آن را فراهم می‌کند و می‌تواند به عنوان یک قطعه‌ی آماده و قابل استفاده‌ی مجدد، در پروژه‌ها مورد استفاده مجدد قرار گیرد.

توجه: در حیطه مهندسی نرم‌افزار شیءگرا، مؤلفه (پیمانه) یک قطعه‌ی عملیاتی مبتنی بر کلاس

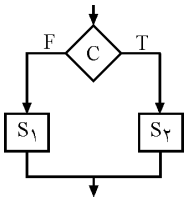
است. در فصل شیءگرایی در این مورد بیشتر صحبت خواهیم کرد.

توجه: در برنامه‌های ساخت‌یافته، هر مولفه یک ساختار منطقی مشخص دارد و پس از طی یک روال از شروع، به پایان مولفه می‌رسد، جزئیات صریح منطقی توابع به صورت ساختارهای زیر است:

ترتیب یا توالی(Sequence): اجرای پشت سر هم دستورات بدون پرش.



ساختارهای شرط (Condition): امکان ایجاد انشعاب بر اساس مقدار شرط مورد نظر.



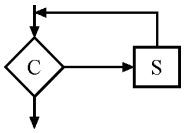
ساختارهای تکرار(Repetition): امکان ایجاد تکرار برای یک مجموعه خاص از دستوارات بر اساس مقدار شرط تکرار مورد نظر.

حلقه while-do :

ابتدا شرط جلوی while بررسی می‌شود، اگر این شرط درست باشد، دستورات حلقه اجرا شده و دوباره کنترل به اول حلقه منتقل می‌گردد. در این حالت دوباره شرط آزمایش می‌شود و عمل مذکور مرتبا تکرار می‌شود. هنگامی که شرط جلوی while نادرست شود، کنترل به دستور بعد از بلوک while منتقل می‌گردد.

فرم کلی این حلقه به صورت زیر است:

```
while condition Do
begin
.
.
.
end;
```



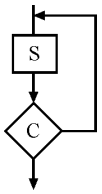
حلقه repeat-until :

در اینجا برعکس حلقه while-do، ابتدا دستورات داخل حلقه اجرا شده و سپس شرط جلوی until آزمایش می‌شود. اگر این شرط نادرست باشد، دوباره دستورات حلقه اجرا می‌گردد. پس حلقه repeat-until حداقل یکبار اجرا می‌شود، ولی حلقه while ممکن است اصلا اجرا نشود. نکته مهم آن است که حلقه repeat-until آنقدر تکرار می‌شود تا جلوی until درست شود. هنگامی که این شرط درست شد، حلقه تمام می‌شود، پس شرط جلوی until عکس شرط جلوی while

است.

فرم کلی این حلقه به صورت زیر است:

repeat
:
until condition



توجه: شرح جزییات داخلی (توالی، ساختارهای شرط و ساختارهای تکرار) برای هر پیمانه یا تابع به معنی بیان روال انجام کار هر تابع توسط **شبه کد** یا **فلوچارت** بیان می‌شود.

توجه: در طراحی مولفه، فرهنگ داده‌های موجود در مدل تحلیل، مورد استفاده قرار می‌گیرد.

ابزارهای طراحی مولفه

برای نمایش جزئیات مولفه‌ها (توابع)، از ابزارهای زیر استفاده می‌گردد:

۱- فلوچارت (Flowchart)

فلوچارت ابزاری است که نحوه اجرای یک برنامه را به صورت گرافیکی در اختیار برنامه‌نویس قرار می‌دهد. فلوچارت یک ابزار تصویری و بصری ساده برای شرح جزئیات داخلی (توالی، ساختارهای شرط و ساختارهای تکرار) برای هر پیمانه یا تابع است. فلوچارت از نمادهای بیضی، مستطیل، لوزی و فلش جهت مدل‌سازی خود استفاده می‌کند. بیضی نشانه شروع و پایان اجرای برنامه، مستطیل نشانه عملیات محاسبات، متوازی الاضلاع نشانه عملیات ورودی و خروجی، لوزی نشانه شرط و تکرار و فلش نشانه توالی است.

«یک تصویر، گویاتر از هزار حرف است» ولی این که کدام تصویر و کدام هزار حرف، قدری اهمیت دارد! (ضرب المثل چینی).

مثال: طراحی مولفه یا بیان جزئیات مربوط به مولفه (تابع) جمع دو عدد توسط فلوچارت.

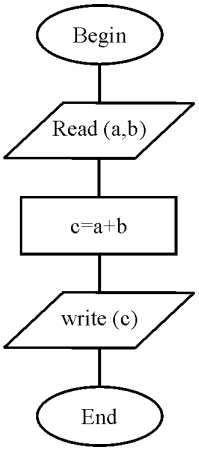
توجه: تابع یا مولفه جمع دو عدد در مدل‌سازی عملکردی فعالیت مدل تحلیل، در ابتدا یکی از حباب‌های DFD، با نام حباب جمع دو عدد بوده است، که پس از نگاشت DFD به معماری نرم‌افزار، به مولفه (تابع) جمع دو عدد تبدیل شده است.

توجه: در طراحی مولفه، PSPEC یا مشخصات پردازش هر حباب، مربوط به مدل‌سازی عملکردی مدل تحلیل، جهت رسم فلوچارت هر تابع معادل با حباب موجود در مدل‌سازی عملکردی مدل تحلیل، مورد استفاده قرار می‌گیرد.

توجه: عمل نوشتن PSPEC یا مشخصات پردازش هر حباب، توسط یک زبان مادری همچون فارسی یا انگلیسی انجام می‌گردد.

PSPEC یا مشخصات پردازش جمع دو عدد مطابق آنچه در مدل‌سازی عملکردی مدل تحلیل

برای حباب جمع دو عدد، در DFD مربوطه نوشتیم، به صورت زیر است:



۱- شروع
۲- سه متغیر a، b و c را تعریف کن.
۳- دو عدد از ورودی خوانده و مقادیر آنها را در متغیرهای a و b قرار بده.
۴- مقادیر دو متغیر a و b را جمع کن و مقدار حاصل را در متغیر c قرار بده.
۵- مقدار متغیر c را در خروجی نمایش بده.
۶- پایان

طراحی مولفه یا بیان جزئیات مربوط به مولفه (تابع) جمع دو عدد توسط فلوچارت، به صورت مقابل است:

۲- شبه کد یا زبان طراحی برنامه

زبان طراحی برنامه یا PDL یا Program Design Language که به آن شبه کد یا سود و کد نیز می‌گویند، یک زبان ترکیبی است که در آن از واژگان یک زبان مثل انگلیسی و نحوی شبیه به زبان برنامه‌نویسی (یعنی یک زبان برنامه‌نویسی ساخت‌یافته) جهت طراحی مولفه، استفاده می‌شود. در نگاه اول، PDL مانند یک زبان برنامه‌نویسی به نظر می‌رسد. شباهت میان PDL و زبان برنامه‌نویسی واقعی در استفاده از متون بیانی (برای مثال زبان انگلیسی) است که مستقیماً در دستورات PDL مورد استفاده قرار می‌گیرد. بنابراین PDL را نمی‌توان کامپایل کرد. ولی، هم‌اکنون ابزارهای PDL ای موجود هستند که PDL را به اسکلت‌بندی یک زبان برنامه‌نویسی، ترجمه می‌کنند.

PDL ساخت یافته ممکن است تبدیل ساده‌ای از یک زبان برنامه‌نویسی ساخت یافته نظیر C باشد. یا ممکن است محصولی باشد که اختصاصاً برای طراحی مولفه ایجاد شده است. نحو اصلی در PDL باید شامل ساختارهایی برای تعریف زیربرنامه‌ها، داده‌ها (ساده و ساخت‌یافته) و ساختارهای برنامه‌نویسی ساخت‌یافته (توالی، شرط و تکرار) باشد. بیانی بودن زبان‌های طبیعی در کنار منطق زبان‌های برنامه‌نویسی، PDL را به یکی از موثرترین و مفیدترین ابزارهای طراحی مولفه بدل کرده است. PDL نسبت به زبان‌های برنامه‌نویسی در سطح بالاتری از انتزاع قرار دارد، بنابراین PDL می‌تواند به عنوان یک ابزار مناسب، قبل از تولید کد مورد استفاده قرارگیرد. PDL را می‌توان از نظر منطقی بررسی کرد و اگر خطای منطقی در آن نهفته باشد، آن خطاها را قبل از فعالیت پیاده‌سازی مرتفع نمود. در عین حال، خطاهای ساده نحوی نیز مشابه آنچه در زبان‌های برنامه‌نویسی وجود دارد، PDL را تحت تاثر قرار نمی‌دهد، برای مثال اگر در انتهای دستورات از کاراکتر "; " استفاده نشده باشد، مساله خاصی ایجاد نمی‌گردد، زیرا PDL شرط کامپایل را ندارد.

توجه: به خاطر بسپارید که PDL، زبان برنامه‌نویسی نیست. طراح می‌تواند بنا به نیاز خود عمل کند و نگران نحو نباشد.

توجه: استفاده از زبان C بدون درج کاراکتر "; "، در انتهای دستورات، نـوعی PDL محسـوب می‌گردد.

مثال: طراحی مولفه یا بیان جزئیات مربوط به مولفه(تابع) جمع دو عدد توسط شبه کد.

توجه: تابع یا مولفه جمع دو عدد در مدل‌سازی عملکردی فعالیت مدل تحلیل، در ابتدا یکی از حباب‌های DFD، با نام حباب جمع دو عدد بوده اسـت، کـه پـس از نگاشـت DFD بـه معمـاری نرم‌افزار، به مولفه (تابع) جمع دو عدد تبدیل شده است.

توجه: در طراحی مولفه، PSPEC یا مشخصات پردازش هـر حبـاب، مربـوط بـه مـدل‌سـازی عملکردی مدل تحلیل، جهت نوشتن شبه کد هـر تـابع معـادل بـا حبـاب موجـود در مـدل‌سـازی عملکردی مدل تحلیل، مورد استفاده قرار می‌گیرد.

توجه: عمل نوشتن PSPEC یا مشخصات پردازش هر حباب، توسط یک زبان مادری همچـون فارسی یا انگلیسی انجام می‌گردد.

PSPEC یا مشخصات پردازش جمع دو عدد مطابق آنچه در مدل‌سازی عملکردی مدل تحلیل برای حباب جمع دو عدد، در DFD مربوطه نوشتیم، به صورت زیر است:

۱- شروع
۲- سه متغیر a، b و c را تعریف کن.
۳- دو عدد از ورودی خوانده و مقادیر آنها را در متغیرهای a و b قرار بده.
۴- مقادیر دو متغیر a و b را جمع کن و مقدار حاصل را در متغیر c قرار بده.
۵- مقدار متغیر c را در خروجی نمایش بده.
۶- پایان

طراحی مولفه یا بیان جزئیات مربوط به مولفه (تابع) جمع دو عدد توسط شبه کد، بـه صـورت زیر است:

```
int main()
{
sum()
return 0
}
sum(void)
{
int a,b,c
scanf("%d %d", &a , &b)
c=a+b
printf("%d", c)
}
```

توجه: در PDL فوق کاراکتر "; " در انتهای دستورات استفاده نشده است.

۳- جدول تصمیم‌گیری

جدول تصمیم‌گیری را به عنوان یک ابزار مکمل در طراحی مولفه می‌توان مورد استفاده قرار داد. در بسیاری از کاربردهای نرم‌افزاری، برخی از برنامه‌ها دارای پیچیدگی‌های شرطی زیادی، برای وقوع یک فعالیت است. به بیان دیگر در شرایط و حالات مختلف یک برنامه، باید عکس‌العمل مناسب ارائه شود. در واقع هرگاه تعداد شرط‌های برنامه زیاد باشد، و هر ترکیب منجر به فراخوانی یک تابع خاص شود، می‌توان از جدول تصمیم گیری استفاده نمود.

مثال: در یک فروشگاه کتاب، مشتریان می‌توانند برحسب سابقه و میزان خریدشان، در سه گروه، عادی، نقره‌ای و طلایی جهت دریافت تخفیفات طبقه‌بندی شوند.

شرایط	۱	۲	۳	۴	۵	۶
مشتری معمولی	T	T	–	–	–	–
مشتری نقره‌ای	–	–	T	T	–	–
مشتری طلایی	–	–	–	–	T	T
دوره تخفیفات ویژه	F	T	F	T	F	T
اقدامات						
تخفیف اعمال نشود	✓	–	–	–	–	–
۸ ٪ تخفیف اعمال شود	–	–	✓	✓	–	–
۱۵ ٪ تخفیف اعمال شود	–	–	–	–	✓	✓
n ٪ تخفیف اعمال شود	–	✓	–	✓	–	✓

قانون ستون ۱: یعنی مشتری معمولی اگر در دوره تخفیفات ویژه نباشد (F)، شامل هیچ تخفیفی نمی‌گردد.

قانون ستون ۲: یعنی مشتری معمولی اگر در دوره تخفیفات ویژه باشد (T)، فقط شامل n٪ تخفیف دوره تخفیفات ویژه می‌گردد.

قانون ستون ۳: یعنی مشتری نقره‌ای اگر در دوره تخفیفات ویژه نباشد (F)، فقط شامل ۸٪ تخفیف می‌گردد.

قانون ستون ۴: یعنی مشتری نقره‌ای اگر در دوره تخفیفات ویژه باشد (T)، علاوه بر تخفیف ۸٪ ویژه مشتریان نقره‌ای، شامل n ٪ تخفیف دوره تخفیفات ویژه نیز می‌گردد.

قانون ستون ۵: یعنی مشتری طلایی اگر در دوره تخفیفات ویژه نباشد (F)، فقط شامل ۱۵٪ تخفیف می‌گردد.

قانون ستون ۶: یعنی مشتری طلایی اگر در دوره تخفیفات ویژه باشد (T)، علاوه بر تخفیف ۱۵٪ ویژه مشتریان طلایی، شامل n ٪ تخفیف دوره تخفیفات ویژه نیز می‌گردد.

تا به اینجا طراحی داده، طراحی معماری و طراحی مولفه از مدل طراحی، تشریح شد، حال در ادامه به بررسی آخرین بخش از مدل طراحی یعنی طراحی واسط می‌پردازیم:

طراحی واسط

طراحی واسط یا همان واسط کاربر، براساس ورودی‌ها و خروجی‌های مورد نیاز کاربران نهایی به شکل نقشی بر روی کاغذ یا طرحی بر روی کامپیوتر ایجاد می‌گردد. مانند نحوه چیدمان منوها و فرم‌ها.

واسط کاربر

یکی از بخش‌های ضروری هر سیستم کامپیوتری، واسط کاربر آن سیستم است که جهت ارتباط سیستم با محیط اطرافش استفاده می‌شود. کیفیت این ارتباط تأثیر چشمگیری در کارایی سیستم و همینطور رضایت‌مندی مشتری دارد، چه بسا سیستم‌هایی که علی‌رغم کارکردهای مناسبشان، به دلیل ضعف در طراحی واسط کاربرشان، از بازار تجاری حذف شدند.

واسط کاربر چه برای یک دستگاه پخش موسیقی دیجیتال طراحی شده باشد و چه برای سیستم کنترل یک هواپیما، آنچه که اهمیت دارد، قابلیت استفاده است. اگر سازوکارهای واسط، از طراحی خوبی برخوردار باشد، کاربر با استفاده از ریتمی ملایم به تعامل با دستگاه می‌پردازد که به او امکان می‌دهد تا بدون هیچ‌گونه تلاش زیادی به اهداف خود دست پیدا کند. ولی اگر واسط کاربر، خوب طراحی نشده باشد، کاربر سردرگم می‌شود و نتیجه نهایی، چیزی جز ناراحتی و سرخوردگی کاربر نخواهد بود. واسط کاربر، پنجره‌ای فراروی نرم‌افزار است. واسط در بسیاری موارد، ادراک کاربر از کیفیت سیستم را شکل می‌دهد. اگر این پنجره غبار گرفته شود، موج‌دار شود یا شکسته شود، کاربر ممکن است، سیستمی پرقدرت را پس بزند.

طی سه دهه‌ی نخست عصر کامپیوترها، قابلیت استفاده در میان سازندگان نرم‌افزار دغدغه اصلی به شمار نمی‌رفت. اما امروزه صنعت نرم‌افزار، اهمیت ویژه‌ای را برای طراحی واسط کاربر قائل است.

این حرکت در دهه‌ی ۸۰ میلادی به شکل جدی آغاز شد، به طوری که نورمن (Norman) در سال ۱۹۸۸، در این باره چنین می‌گوید:

برای ساخت فناوری‌هایی که برازنده انسان باشد، مطالعه انسان ضروری است. ولی اکنون ما فقط تمایل داریم، فناوری را مطالعه کنیم. در نتیجه انسان‌ها ناگزیرند از فناوری پیروی کنند. زمان آن فرا رسیده است که این روند وارونه شود، وقت آن رسیده است، که فناوری را وادار به دنباله روی از انسان کنیم.

قوانین طایی ماندل

تئوماندل در کتاب «طراحی واسط» خود، سه قانون طلایی، برای طراحی واسط کاربر مناسب، بیان نموده است:

۱- سپردن کنترل برنامه به کاربر

در یک نرم‌افزار با واسط مناسب، نرم‌افزار در اختیار کاربر است و نه کاربر در اختیار نرم‌افزار. بنابراین فرد طراح باید نرم‌افزاری طراحی کند که در آن کنترل نرم‌افزار به طور کامل در اختیار کاربر باشد. برای رسیدن به این هدف، طراح باید نکاتی را رعایت کند که در زیر به آن‌ها اشاره شده است:

الف) حذف تعاملات غیر ضروری بین کاربر و نرم‌افزار

تعاملات سازنده، مناسب و ضروری باشد، اما تعاملات غیر ضروری نباشد. برای مثال، اگر در یکی از منوهای یک واژه‌پرداز، گزینه چک کردن املای کلمات انتخاب شود، نرم‌افزار به حالت چک کردن املا می‌رود و دلیلی وجود ندارد که اگر کاربر طی این کار مایل به انجام یک کار دیگر مثل ویرایش متن باشد، مجبور باشد کماکان در حالت چک کردن املا باقی بماند و یا به سختی به کار دیگر نقل مکان کند. در واقع کاربر باید به سادگی و بدون انجام کار زیاد، از حالتی به حالت دیگر برود.

ب) انعطاف پذیری در تعامل

این اصل، به معنای قائل شدن حق انتخاب برای کاربر است. در واقع بهتر است نرم‌افزار، امکان استفاده از انواع دستگاه‌های ورودی را برای کاربر فراهم نماید تا کاربر بتواند بنا به شرایط از دستگاه ورودی مناسب، برای انجام کارهای خود استفاده نماید. برای مثال برای تایپ از صفحه کلید، برای انتخاب منوها، ماوس و صفحه کلید، برای نقاشی، ماوس و قلم نوری و ... .

ج) امکان توقف تعامل و برگشت‌پذیری تعاملات

وقتی کاربر عملیاتی را انجام داد، باید بتواند این عملیات را به منظور انجام عمل دیگری متوقف نماید، اما بدون از دست رفتن نتیجه عملیات، مانند متوقف کردن عمل تایپ به منظور رنگی نمودن متن تایپ شده. همچنین برگشت‌پذیری عملیات انجام شده باید موجود باشد. مانند Ctrl+Z در واژه‌پرداز ورد.

د) سفارشی‌کردن تعاملات

کاربران اغلب، تعدادی عملیات را به طور مکرر باید انجام دهند. بنابراین بهتر است یک راهکار (ماکرو) طراحی شود که کاربران ماهر بتوانند واسط را سفارشی کنند تا تعامل آسان شود. مانند ماکرونویسی در نرم‌افزار اکسل.

برخی کارها، روال مشخص و از قبل تعیین شده‌ای دارد، بنابراین بهتر است با تعریف این روال‌های روتین در قالب ماکرو و تابع، انجام این وظایف به نرم‌افزار محول گردد. مانند تعریف توابع مربوط به عملیات شست‌وشوی لباس در دستگاه ماشین لباس‌شویی به جای انجام کارهای تکراری در هر روز و هر روز. یکبار تعریف تابع و استفاده از تابع، بارها و بارها.

ه) پنهان‌سازی جزئیات فنی از دید کاربران

واسط کاربر، نباید کاربر را وادار به تعامل در سطح جزئیات مربوط به ماشین نماید. به بیان

دیگر کاربر، منوها و فرم‌ها را در نرم‌افزار ببیند و این وظیفه نرم‌افزار باشد که با جزئیات سخت افزاری ماشین ارتباط بر قرار کند.

ی) تعامل با اشیاء روی صفحه نمایش

وقتی کاربر اشیای لازم برای انجام یک وظیفه را به شیوه‌ای فیزیکی بتواند دستکاری کند، احساس می‌کند، کنترل بیشتری بر روی نرم‌افزار دارد. برای مثال واسطی که به کاربر امکان می‌دهد تا شیء را حرکت دهد یا اندازه آن را تغییر دهد، نمونه‌هایی از دستکاری مستقیم است.

۲- کاهش بار حافظه کاربر

هر چه کاربر مجبور به حفظ جزئیات بیشتر در تعامل با نرم‌افزار باشد، احتمال خطای او در تعامل با نرم‌افزار بالاتر است. به همین دلیل است که در یک طراحی واسط کاربر خوب، به ماندگاری اطلاعات در حافظه کاربر توجه نمی‌شود. در واقع، در صورت امکان، این نرم‌افزار است که باید اطلاعات مربوطه را به شیوه‌ای شگفت‌انگیز از طریق نشانه‌ها به یاد کاربر بیاورد و برای کاربر یادآوری کند.

الف) کاهش دسترسی به حافظه کاربر

واسط باید طوری طراحی شود تا تقاضا به حافظه کاربر کاهش یابد. این عمل با ایجاد علائم بصری و نشانه‌گذاری مسیر است که به کاربر، امکان می‌دهد تا مسیرهای مورد نظر خود را شناسایی کند. برای مثال وقتی پشت کامپیوتر هستید، روال حرکت به یک بخش مورد نظر را می‌دانید، اما اگر پشت کامپیوتر نباشید، و از طریق تلفن همان مسیر و روال حرکت را از شما مورد سؤال قرار دهند، شاید به دشواری بتوانید، راهنمایی کنید! این مورد بارها پیش آمده است ...

ب) ایجاد پیش فرض

در هر نرم‌افزاری باید مجموعه‌ای از پیش فرض‌ها با مقادیر اولیه و ثابت وجود داشته باشد تا علاوه بر اینکه کاربر ترجیحات و نوع خواسته‌های خود را اعمال می‌کند، پیش فرض‌های اولیه نیز بتوانند در صورت لزوم حالت نرم‌افزار را به حالت اولیه خود برگرداند. مانند وجود پیش فرض‌های رنگ‌آمیزی و چیدمان منوها در یک نرم‌افزار.

ج) ایجاد میانبرهای هوشمند

هنگامی که برای دستیابی به عملکردهای سیستم از کلیدهای میانبر استفاده می‌شود. (مثل Ctrl+P برای عمل چاپ)، بین کلید میانبر و آن عمل باید ارتباطی منطقی وجود داشته باشد که به خاطر آوردن آن آسان باشد، برای مثال می‌توان از حرف اول آن عمل استفاده نمود.

د) استفاده از شناسه‌های واقعی در فرم‌ها و منوها

برای نام‌گذاری شناسه‌های مورد استفاده در فرم‌ها و منوها، بهتر است از شناسه‌های واقعی در محیط عملیاتی واقعی استفاده گردد. مانند شناسه‌های استاد، دانشجو و درس در محیط عملیاتی دانشگاه.

هـ) سازماندهی سلسله‌مراتبی

واسط کاربر باید دارای سازماندهی سلسله مراتبی باشد، در واقع راهنمایی کاربر، توسط واسط نرم‌افزار باید به صورت سلسه مراتبی و قدم به قدم باشد. واسط کاربر نرم‌افزار باید بتواند به شیوه‌ای سلسله مراتبی، کاربر را به مقصد مورد نظر هدایت و راهنمایی نماید. و نه اینکه واسط کاربر، به یکباره، کاربر را در معرض مسیرهای مختلف قرار دهد و باعث سردرگمی کاربر گردد. مانند منوهای موجود در یک سایت خبرگزاری که از بین گروه‌های مختلف خبری، یک گروه خبری مورد علاقه خود را انتخاب می‌کنید و در بخش مورد نظر، بخش بعدی، بعدی و بعدی را انتخاب می‌کنید.

۳- سازگاری واسط کاربر

واسط‌های کاربر از نظر شکل و شمایل باید مطابق قوانین و استانداردهای رسمی و غیررسمی مقبول جامعه فناوری باشد، به عبارت دیگر با این استانداردها سازگار باشد. برای رسیدن به این هدف رعایت موارد زیر توصیه می‌گردد:

الف) استفاده از نمادها و اشکال سازگار با محیط عملیاتی.

برای مثال، استفاده از نماد کتاب، برای ورود به بخش کتاب در یک فروشگاه اینترنتی.

ب) استفاده از نمادها و اشکال سازگار با محیط نرم‌افزارهای هم خانواده.

برای مثال، نماد ►، در نرم‌افزارهای پخش صدا و فیلم، فایل مورد نظر را اجرا می‌کند.

ج) سازگاری با اخلاق عمومی نرم‌افزارها.

برای‌مثال، Ctrl+S، به طور غیررسمی، در اغلب نرم‌افزارها، شیوه‌ای عمومی برای ذخیره فایل محسوب می‌گردد. تغییر این پارادایم‌های عمومی، در یک واسط کاربر نرم‌افزار، از نگاه کاربر، ناخوشایند است.

توجه: مشتری و کاربر چیزی را می‌خواهد که برق بزند و به شیوه‌ای کارآمد کار کند، شیوه‌های خلق این خواسته‌ها از نگاه مشتری اهمیتی ندارد!

تحلیل و طراحی واسط کابر

فرایند تحلیل و طراحی واسط کابر، یک فرآیند تکرارشونده است. که با ایجاد مدل‌هایی مختلف، از عملکرد سیستم (آن طور که از خارج به نظر می‌رسد) آغاز می‌شود. روند تکرارشونده از تحلیل، طراحی، پیاده‌سازی و ارزیابی، آنقدر تکرار می‌شود تا واسطی ایجاد گردد که مورد رضایت مشتری واقع گردد.

مدل‌های تحلیل و طراحی واسط

مدل‌های تحلیل و طراحی، واسط کابر را از زوایای مختلف مورد بررسی قرار می‌دهد. چهار مدل کاربر، مدل طراحی، مدل ذهنی کاربر و مدل پیاده‌سازی برای این امر مورد استفاده قرار می‌گیرند. هر کدام از این مدل‌ها ممکن است با دیگری متفاوت باشد، نقش طراح واسط، آشتی دادن این مدل‌ها، و به دست آوردن نمایشی سازگار از واسط است.

مدل کاربر

مدل کاربر خصوصیات کاربران نهایی را مدل‌سازی می‌کند. یک واسط کاربر باید مطابق خصوصیات کاربران نهایی آن ایجاد گردد. شناخت کاربران و خصوصیات آن‌ها اغلب توسط مهندس فاکتورهای انسانی (human engineer) و گاهی نیز توسط مهندس نرم‌افزار انجام می‌گردد.

جف پاتون در خصوص طراحی واسط کاربر چنین می‌گوید:

حقیقت این است که، طراحان و سازندگان، زیاد به کاربران فکر می‌کنند. ولی در غیاب یک مدل ذهنی قوی از کاربران خاص، ما خودمان را جای کاربران می‌گذاریم و این کاربر محوری نیست، خود محوری است.

مدل طراحی

مدل طراحی، خصوصیات داخلی نرم‌افزار را مدل‌سازی می‌کند. مدل طراحی توسط مهندس نرم‌افزار انجام می‌گردد.

مدل ذهنی کاربر

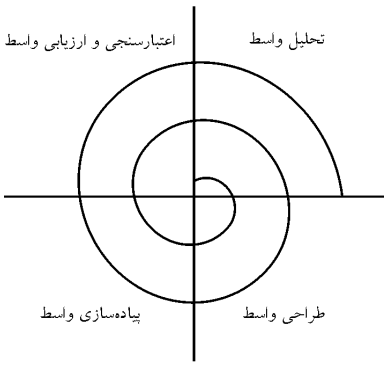
مدل ذهنی کاربر، تصورات و انتظارات کاربران نهایی از واسط نرم‌افزار را مدل‌سازی می‌کند. طراح واسط باید تلاش کند، تا این تصویر ذهنی را هرچه بیش‌تر به واقعیت نزدیک کند.

مدل پیاده‌سازی

مدل پیاده‌سازی شکل ظاهری و خارجی نرم‌افزار را مدل‌سازی می‌کند. مدل پیاده‌سازی توسط برنامه‌نویسان واسط انجام می‌گردد. همچنین مدل پیاده‌سازی، شامل فایل‌های کمکی، کتابچه‌ها و فایل‌های آموزشی می‌باشد. هنگامی که مدل ذهنی کاربر، با مدل پیاده‌سازی شده، شباهت بیشتری داشته باشد، کاربر از واسط راضی بوده و به شکل بهتری با نرم‌افزار ارتباط برقرار نموده و از آن استفاده می‌کند.

فرایند تولید واسط کاربر

همانطور که پیش از این نیز گفتیم، فرآیند تولید واسط کاربر، یک فرآیند تکاملی مبتنی بر رویکرد تکرار و تکامل است.



تحلیل واسط

تحلیل واسط از چهار بخش، تحلیل کاربران، تحلیل وظایف، تحلیل شیوه نمایش محتوی و تحلیل محیط تشکیل شده است.

تحلیل کاربران: شناخت سطح آگاهی و مهارت‌های کاربران برای استفاده از نرم‌افزار، برای ایجاد واسطی کاربرپسند که مورد استفاده کابران قرار گیرد.

تحلیل وظایف: شناخت کارکردها و سرویس‌های مورد انتظار کاربران از نرم‌افزار، برای ایجاد نرم‌افزاری ایده‌آل که با واسطی کاربرپسند مورد استفاده کابران قرار گیرد. مدل‌سازی لیست نیازمندی‌های مشتری می‌تواند در این بخش مورد استفاده قرار بگیرد.

تحلیل شیوه نمایش محتوی: شناخت نحوه نمایش خروجی‌ها و گزارش‌های نرم‌افزار از نگاه کاربران، برای ایجاد واسطی کاربرپسند که مورد استفاده کاربران قرار گیرد.

تحلیل محیط: شناخت شرایط محیطی کاربران، برای ایجاد واسطی کاربرپسند که مورد استفاده کابران قرار گیرد. برای مثال در کابین خلبان هواپیما ماوس مورد استفاده قرار نمی‌گیرد..

طراحی واسط

مدل‌سازی نحوه چیدمان بخش‌های مختلف واسط کاربر، همچون ترتیب پنجره‌ها، مشخص کردن آیتم‌های منوهای اصلی و فرعی، جداول، شیوه‌های تعامل و محل قرار گرفتن آیکن‌ها، بر اساس مدل تحلیل واسط در این مرحله انجام می‌گردد.

چالش‌های طراحی واسط

در حین انجام طراحی واسط نرم‌افزار، چالش‌های زیر مطرح هستند:

زمان پاسخ

به حد فاصل زمانی، بین ارسال دستور از سوی کاربر تا دریافت پاسخ از سوی نرم‌افزار زمان پاسخ گفته می‌شود. زمان پاسخ با دو شاخص زیر از سوی کاربران مورد ارزیابی قرار می‌گیرد:

طول زمان پاسخ: به اندازه زمان پاسخ، طول زمان پاسخ گفته می‌شود.

تغییرپذیری زمان پاسخ: به میزان انحراف، از متوسط زمان پاسخ، تغییرپذیری زمان پاسخ گفته می‌شود. هرچه قدر این انحراف کمتر باشد، واسط کاربر، برای کاربر، رضایت بخش‌تر خواهد بود. کم بودن تغییرپذیری سبب می‌شود تا کاربر شاهد ریتم ملایمی در ارتباط با نرم‌افزار باشد. زیرا در این شرایط ریتم انجام کار و مدت زمان اجرای وظایف، مشخص است و کاربر با نوسانات زیاد آزرده خاطر نمی‌گردد. در سیستم‌های مشابه با زمان پاسخ یکسان، اولویت با سیستمی است که تغییرپذیری کمتری را دارا باشد.

تسهیلات کمکی نرم‌افزار

تقریباً همه کاربران یک سیستم تعاملی و مبتنی بر کامپیوتر هر از گاهی به راهنمایی و کمک نیاز دارند. در برخی موارد، یک پرسش ساده که از همکاری پرسیده می‌شود، راهگشا خواهد بود. در سایر موارد، جستجوی مفصل در مجموعه‌ای چند جلدی از جزوات راهنمای کاربران، ممکن

است تنها گزینه پیش رو باشد. به هر حال، در اکثر موارد، نرم‌افزارهای مدرن، امکانات راهنمای آنلاینی فراهم می‌سازند که به کاربر این امکان را می‌دهند تا بدون ترک واسط، پاسخ پرسش خود را بگیرد یا مشکلی را حل کند.

مدیریت خطاها

پیام‌های خطا و هشدار، خبرهای ناگواری هستند که هنگام خراب شدن اوضاع، تحویل کاربران سیستم‌های تعاملی می‌شود. پیام‌های خطا و هشدارها در بدترین حالت خود، اطلاعات بیهوده و گمراه‌کننده‌ای می‌دهند که فقط به افزایش ناراحتی کاربر کمک می‌کنند. اما یک مدیریت خطای مناسب، می‌تواند تا حد زیادی کاربر را دلگرم کرده و به او اطمینان دهد که نرم‌افزار همچنان تحت کنترل وی عمل می‌نماید. به طور کلی، هر پیام خطا یا هشدار تولید شده توسط یک سیستم تعامل باید دارای ویژگی‌های زیر باشد:

۱- پیام باید مشکل را به زبانی شرح دهد که کاربر قادر به درک آن باشد.

۲- پیام خطا باید علل وقوع خطا را بیان کند، برای مثال، فایل‌های داده احتمالا آسیب دیده‌اند، به گونه‌ای که کاربر بتواند علت وقوع خطا را مرتفع نماید.

۳- پیام باید حاوی یک توصیه سازنده برای رهایی از وضعیت خطا باشد.

۴- پیام باید با یک نشانه سمعی و بصری همراه باشد. یعنی یک صدای بوق با نمایش پیام همراه شود، یا حالت چشمک زدن داشته باشد، یا به رنگی ظاهر شود که به آسانی به عنوان رنگ خطا، قابل تشخیص باشد.

۵- پیام نباید قضاوت گونه باشد. یعنی لحن آن نباید طوری باشد که کاربر را مورد شماتت قرار دهد.

توجه: از آنجا که هیچ کس واقعا از خبرهای ناگوار خوشش، نمی‌آید، معدود کاربرانی هستند که پیام‌های خطا را دوست داشته باشند، هر چند که این پیام‌ها خیلی خوب طراحی شده باشند. ولی یک فلسفه موثر برای پیام‌های خطا می‌تواند تاثیر بسزایی در کیفیت یک سیستم تعاملی داشته باشد و هنگام رخ دادن مشکل، تا حد زیادی از ناراحتی کاربر بکاهد.

نام‌گذاری منوها و دستورات

زمانی متداول‌ترین شیوه تعامل میان کابر و سیستم نرم‌افزاری، تایپ فرمان‌ها بود و از آنها برای انواع برنامه‌های کاربردی استفاده می‌شد. امروزه استفاده از واسط‌های پنجره‌ای، تکیه بر فرمان‌های تایپی را کاهش داده است، ولی بسیاری از کابران قدرتمند همچنان شیوه تایپ فرمان‌ها را ترجیح می‌دهند. هنگام طراحی محیط تعامل از طریق تایپ فرمان‌ها، به مسائل زیر باید توجه داشت:

۱- به ازای هر گزینه انتخابی در منوها، معادل فرمان تایپی آن موجود باشد. برای مثال برای گزینه پرینت در منو، معادل فرمان تایپی آن هم باشد.

۲- فرمان‌های تایپی باید دارای ساختاری متناسب با دستور مورد نظر باشند. برای مثال ساختار Ctrl+P، برای اجرای دستور پرینت.

۳- امکان تغییر ساختار فرمان‌های تایپی، برای مثال Ctrl+space، برای درج نیم فاصله در

متون.

دسترس‌پذیری در برنامه کاربردی

با همگانی شدن برنامه‌های کاربردی کامپیوتری، مهندسان نرم‌افزار، باید اطمینان حاصل کنند که طراحی واسط شامل سازوکارهایی است که برای افرادی با نیازهای خاص، دستیابی آسان را میسر می‌سازد. به بیان دیگر استفاده از برنامه‌های کاربردی، برای افرادی با نارسایی‌های بینایی، شنوایی، حرکتی، گفتاری و یادگیری، بنا به دلایل اخلاقی و حقوق انسانی و حتی تجاری، امکان‌پذیر باشد.

جهانی‌سازی

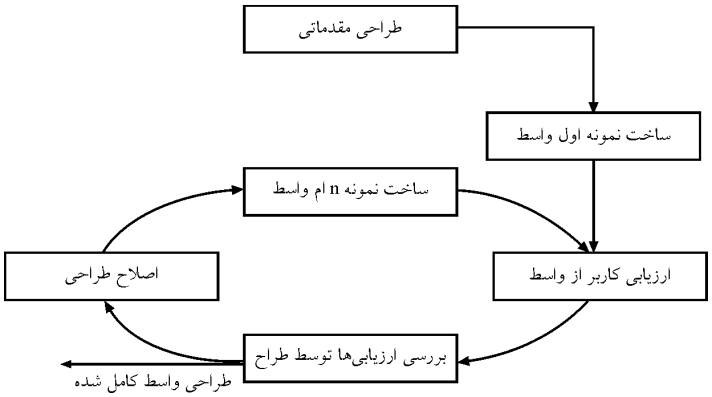
اغلب مهندسان نرم‌افزار و مدیران آنها، مهارت‌ها و تلاش لازم برای ایجاد واسط‌هایی را که پاسخگوی نیازهای زبان‌ها و مناطق متفاوت باشد، کوچک می‌شمارند. به وفور پیش می‌آید که ابتدا واسط‌ها برای یک زبان و منطقه خاص طراحی می‌شوند و سپس در کشورهای دیگر از آنها استفاده می‌شود. چالش اصلی ساخت واسط برنامه‌های کاربردی جهانی، امکان محلی‌سازی واسط کاربر، برای کاربران در نقاط مختلف جهان است. استاندارد یونیکد، برای پرداختن به چالش مدیریت ده‌ها زبان طبیعی با صدها کاراکتر و نماد، توسعه یافته است.

پیاده‌سازی

پس از طراحی واسط، نوبت به پیاده‌سازی می‌رسد. در این مرحله نمونه‌ای اولیه جهت ارزیابی مشتری توسط ابزارهای برنامه‌نویسی ایجاد می‌گردد. تا طی روندی مبتنی بر تکرار و تکامل واسط نهایی کاربر ایجاد گردد.

اعتبارسنجی یا ارزیابی طراحی

پس از ایجاد نمونه عملیاتی واسط کاربر، این نمونه باید بر اساس خواسته‌های مشتری، مورد ارزیابی قرار گیرد، تا مشخص شود که نیازهای کاربر برآورده می‌شود یا خیر. این روند، مبتنی بر تکرار و تکامل تا ایجاد واسطی کاربر پسند ادامه پیدا می‌کند. به بیان دیگر چرخه ارزیابی آنقدر ادامه می‌یابد که دیگر نیازی به اصلاحات بیشتر در ساخت واسط نباشد. شکل زیر گویای مطلب است:



در هنگام ارزیابی نمونه‌های اولیه واسط کاربر، معیارهای زیر در نظر گرفته می‌شوند:
۱- یادگیری آسان و قابل استفاده بودن
۲- برآورده ساختن تمامی سرویس‌های مورد نظر کاربران
۳- جلب رضایت مندی کاربران

برای بررسی میزان رضایت مندی کاربران، می‌توان از روش‌های رسمی همچون پرسشنامه و یا روش‌های غیررسمی همچون مصاحبه استفاده نمود. همچنین می‌توان از روش‌های کمی همچون متوسط زمان پاسخ، نحوه استفاده از راهنماها و تعداد خطاها در واحد زمان برای ارزیابی واسط کاربر استفاده نمود.

تا به اینجا فعالیت ارتباطات، فعالیت برنامه‌ریزی و فعالیت مدل‌سازی شامل مدل تحلیل و مدل طراحی بررسی گردید، حال در ادامه به بررسی فعالیت ساخت و فعالیت استقرار از فعالیت‌های باقی‌مانده فرآیند تولید نرم افزار(فعالیت‌های چارچوبی) می‌پردازیم.

فعالیت ساخت (پیاده‌سازی و تست)

فعالیت پیاده سازی

پس از مدل طراحی نوبت به پیاده‌سازی و تست می‌رسد. پیاده‌سازی جدول از بخش پیاده‌سازی داده، طراحی جدول از فعالیت مدل طراحی را به عنوان ورودی دریافت کرده و توسط دستورات DDL در SQL، پیاده‌سازی جداول را انجام می‌دهد.

پیاده‌سازی پرس‌وجو از بخش پیاده‌سازی داده، طراحی پرس و جو از فعالیت مدل طراحی را به عنوان ورودی دریافت کرده و توسط دستورات DML در SQL پیاده‌سازی پرس‌وجو را انجام می‌دهد.

پیاده‌سازی عملکرد، طراحی مؤلفه از فعالیت مدل طراحی را به عنوان ورودی دریافت کرده و توسط یک زبان برنامه‌نویسی (ساخت‌یافته یا شی‌گرا) پیاده‌سازی عملکرد را انجام می‌دهد.

توجه: دقت کنید که می‌توان فعالیت تحلیل و طراحی را به روش و ابزارهای ساخت‌یافته انجام داد ولی فعالیت پیاده‌سازی را توسط یک زبان برنامه‌نویسی شی‌گرا انجام داد و از امکانات شی‌گرایی زبان استفاده نکرد اما عکس این مطلب امکان‌پذیر نیست، یعنی نمی‌توان فعالیت تحلیل و طراحی را به روش شیءگرا انجام داد ولی فعالیت پیاده‌سازی را توسط یک زبان ساخت‌یافته انجام داد زیرا زبان ساخت‌یافته امکانات شی‌گرایی (همچون کلاس، وارثت و چندریختی) را پشتیبانی نمی‌کند.

این نوع برنامه‌نویسی در دهه ۱۹۶۰ مطرح شد. در این شیوه برنامه به قطعات کوچکتر، یعنی بلوک تابع تقسیم می‌شود. همچنین در هر تابع می‌توان بلوک‌هایی داشت که در واقع به صورت زیر تابع عمل کرده و از دید تابع به صورت یک جمله دیده می‌شوند. هر یک از این بلوک‌ها نیز می‌توانند به نوبه خود شامل بلوک‌های دیگری باشند. متغیرها به صورت global یا local هستند. جهش و پرش نداریم و کنترل برنامه بهتر صورت می‌گیرد.

مثال:

```
{
if(x>0) {
        x=x*x;
        printf("The value of square x is %d",x);
   }
else  {
       x=x-x;
       printf ("x is negative number");
   }
}
```

برنامه‌نویسی ساخت‌یافته روی تعریف خوب ساختارهای کنترلی، یا ساختارهای کنترلی خوش تعریف (well-defined)، بلاک‌های کد (code-block) و عدم استفاده (یا حداقل استفاده) از دستور go to و توابع مستقل و جداگانه که در آنها از متغیرهای محلی و تکنیک‌های بازگشتی (recursive) استفاده می‌شود، تأکید دارد. با این تکنیک برنامه‌نویسی، می‌توان برنامه‌هایی به طول تقریبی ۵۰۰۰۰ خط تولید کرد. اولین زبان از نوع ساخت‌یافته PL/1 بوده است، ولی معروفترین آنها C و پاسکال می‌باشد. وجود متغیرهای local در توابع امکان تداخل ناخواسته بین قسمت‌های مختلف برنامه را کاهش می‌دهد. اساس برنامه‌نویسی ساخت‌یافته تجزیه یک برنامه به مجموعه‌ای از توابع تشکیل‌دهنده آن است. اگرچه نتایج حاصل از به کارگیری برنامه‌نویسی ساخت‌یافته برای برنامه‌های متعارف و معمولی بسیار مناسب است، اما در مواقعی که اندازه برنامه از حد معینی بزرگتر شود، با اشکال مواجه می‌شود. لذا به منظور امکان ایجاد برنامه‌های پیچیده‌تر، نیاز به نگرشی نوین، در برنامه‌نویسی بود. مقارن با این نتیجه، برنامه‌نویسی شیءگرا ابداع شد. برنامه‌نویسی شیءگرا با بهره گیری از ایده‌های به دست آمده در برنامه نیسی ساخت‌یافته و ترکیب آنها با مفاهیم توانمند جدید، این امکان را فراهم می‌آورد تا سازماندهی برنامه خود را به شیوه دیگری انجام دهید. برنامه‌نویسی شیءگرا شما را ترغیب می‌کند تا یک مساله را به زیر گروه‌های مربوطه تجزیه کنید. هر زیر گروه خود کلاسی است حاوی دستورالعمل‌ها (متدها) و داده‌های (صفات) آن کلاس. با این روش پیچیدگی برنامه کاهش یافته و برنامه نویس می‌تواند برنامه‌های بزرگ را کنترل کند.

توجه: در ساخت یافتگی پیمانه‌ای کردن برنامه مبتنی بر تابع است. در حالی که در شیءگرایی پیمانه‌ای کردن برنامه مبتنی بر کلاس انجام می‌گردد.

ساختارهای برنامه‌نویسی ساخته‌یافته عبارتند از: توالی، شرط، تکرار

این سه ساختار در برنامه‌نویسی ساخت‌یافته که به عنوان ابزارهای برنامه‌نویسی رویه‌ای می‌باشند، اساسی و ضروری به شمار می‌روند.

فعالیت تست

پس از پیاده‌سازی نوبت به تست می‌رسد، در این مرحله کلیه‌ی موارد پیاده‌سازی شده از نظر

خطاهای نحوی و خطاهای معنایی براساس لیست نیازمندی‌های مشتری (چک لیست) که در فعالیت ارتباطات تهیه شده بود مورد وارسی قرار می‌گیرد تا مشخص شود نرم‌افزار بر اساس ورودی‌های مورد نظر مشتری، خروجی‌های مورد انتظار مشتری را برآورده می‌سازد یا خیر.

توجه: این فعالیت توسط برنامه‌نویسان و در گام بعدی در صورت لزوم توسط یک گروه تست مستقل (ITG) انجام می‌گردد.

توجه: فعالیت تست در فصل فعالیت تست، مفصل‌تر، تشریح خواهد شد.

فعالیت استقرار

پس از تست نوبت به فعالیت استقرار می‌رسد. در این مرحله، نرم‌افزار به مشتری تحویل داده می‌شود و مشتری با بررسی محصول دریافتی، باز خوردهای به دست آمده براساس همین ارزیابی‌ها را به تیم نرم‌افزاری ارائه می‌دهد. این بازخوردها می‌توانند مبنایی برای ارتقاء و یا تصحیح نسخه‌ی بعدی نرم‌افزار باشد.

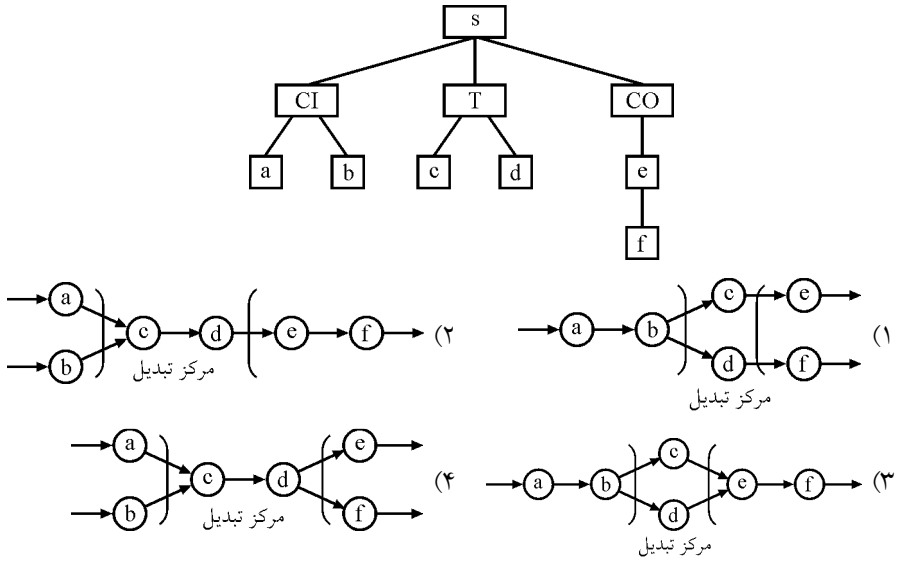
تست‌های فصل پنجم

۱- معماری فراخوانی روال‌های راه دور مربوط به کدام‌یک از سبک‌های معماری نرم‌افزار است؟

(مهندسی IT – آزاد ۸۵)

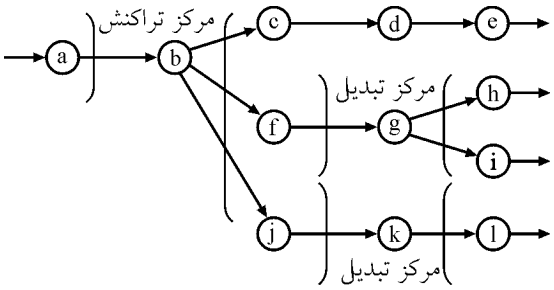
۱) سبک جریان داده
۲) سبک شیءگرا
۳) سبک لایه‌ای
۴) سبک فراخوانی و برگشت

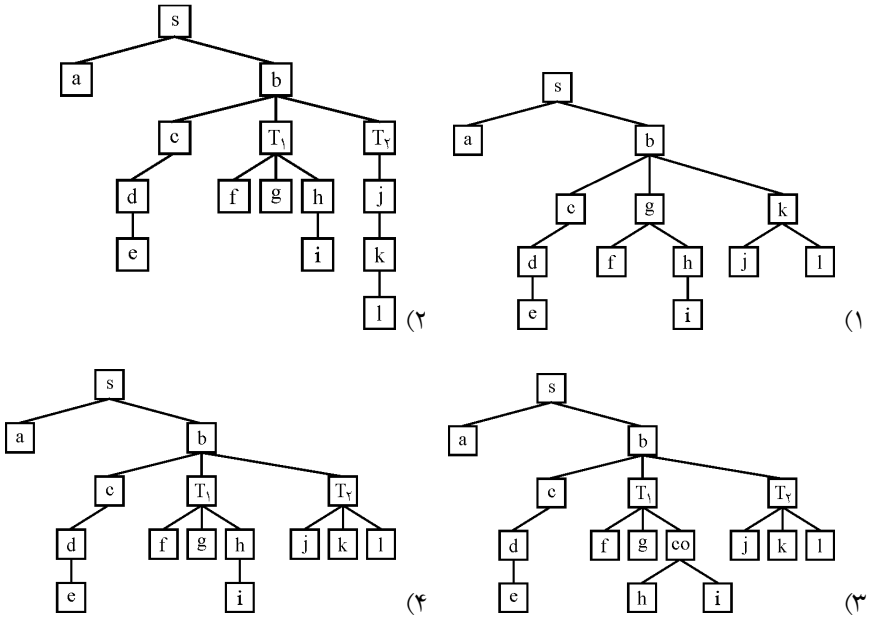
۲- نمودار ساختار زیر، برگرفته از کدام نمودار جریان داده‌هاست؟ **(مهندسی IT – دولتی ۸۶)**



۳- کدام نمودار ساختار برنامه از نمودار جریان داده‌ای زیر به دست آمده است؟

(مهندسی IT – دولتی ۸۶)





۴- کدام عبارت در مورد طراحی نرم‌افزار از طریق روش تجزیه عملیات (Functional Decomposition) و روش طراحی داده‌گرا (Data Structure Design) صحیح است؟ (مهندسی IT – دولتی ۸۷)

۱) روش طراحی تجزیه عملیات و روش طراحی داده‌گرا، ساختمان داده متفاوت و همچنین ساختار برنامه متفاوتی تولید می‌کنند.

۲) روش طراحی تجزیه عملیات و روش طراحی داده‌گرا، ساختمان داده یکسان لیکن ساختار برنامه متفاوتی تولید می‌کنند.

۳) روش طراحی تجزیه عملیات و روش طراحی داده‌گرا، ساختمان داده متفاوت لیکن ساختار برنامه یکسانی تولید می‌کنند.

۴) روش طراحی تجزیه عملیات و روش طراحی داده‌گرا، ساختمان داده یکسان و همچنین ساختار برنامه یکسانی تولید می‌کنند.

۵- شاخص گزینش طراحی با رابطه $d = \left(\frac{N_s}{N_a}\right)^{*}۱۰۰$ تعریف می‌شود که در آن N_s تعداد ابعاد به کار رفته در یک معماری پیشنهادی و N_a تعداد کل ابعاد در فضای طراحی است. (مهندسی IT – دولتی ۸۹)

۱) اگر $۰ \le d \le ۱$ باشد، معماری ایده‌ال است.

۲) اگر $d \ge ۱$ باشد، معماری ایده‌ال است.

۳) هر چه d بزرگتر باشد، معماری پیشنهادی به معماری ایده‌آل نزدیک‌تر است.

۴) هر چه d کوچک‌تر باشد، معماری پیشنهادی به معماری ایده‌ال نزدیک‌تر است.

۶- کدام‌یک از جملات زیر نادرست است؟ (مهندسی IT – دولتی ۹۰)

۱) در معماری مخزنی (Repository) تکامل داده‌ها مشکل و هزینه‌بر است.
۲) معماری سرویس‌دهنده / سرویس‌گیرنده برای داده‌های توزیع شده مناسب است.
۳) معماری مخزنی برای به اشتراک گذاشتن حجم زیادی از داده‌ها مناسب است.
۴) در معماری سرویس‌دهنده / سرویس‌گیرنده (Clien/Server) مدل داده‌های مشترکی مورد استفاده قرار می‌گیرد.

۷- کدام‌یک از فعالیت‌های زیر معمولاً از فعالیت‌های اصلی در طراحی سیستم‌های تجاری نیست؟
(مهندسی IT – دولتی ۹۰)

۱) طراحی واسط کاربر
۲) تعیین محدودیت‌های زمانی
۳) تعیین معماری بستر (Platform)
۴) طراحی شمای پایگاه داده

۸- فرض کنید که یک تیم مهندسی در حال ایجاد یک سامانه‌ی ثبت نام برای یک سازمان است. کدام یک از تصمیمات زیر به احتمال قوی‌تر در جریان طراحی سامانه اتخاذ می‌شود؟ (مهندسی IT – دولتی ۹۱)

۱) سامانه خواسته‌ها را طبق نیازهای سازمان برآورده کرده است.
۲) سامانه از استانداردهای خاص سازمان برای ایجاد سیستم‌ها تبعیت خواهد نمود.
۳) سامانه به صورت هفتگی گزارشاتی را برای مدیریت سازمان تولید خواهد نمود.
۴) زیرسامانه‌ی واسط کاربر شامل دو زیرسامانه مجزا برای تعامل با انواع مختلف کاربران خواهد بود.

۹- کدام یک از موارد زیر از قواعد طلایی طراحی واسط کاربری نیست؟
(مهندسی IT – دولتی ۹۴)

۱) کاربر باید حس کند که برنامه را تحت کنترل دارد
۲) کاهش بار حافظه کاربر
۳) زمان پاسخ مناسب
۴) سازگاری واسط

پاسخ تست‌های فصل پنجم

۱- گزینه (۴) صحیح است.

نرم‌افزار نیز همانند یک ساختمان، دارای سبک‌های متفاوتی برای ساخت می‌باشد. به طور کلی برای طراحی معماری و ساخت نرم‌افزارها دو سبک ساخت‌یافته (مبتنی بر فراخوانی و بازگشت توابع) و سبک شیءگرا (مبتنی بر ارسال پیام مابین اشیاء) مورد استفاده قرار می‌گیرد.

سبک ساخت‌یافته (مبتنی بر فراخوانی و بازگشت توابع) به دو طبقه کلی معماری تابع‌گرا (جریان داده) و معماری داده‌گرا تقسیم می‌گردد.

معماری داده‌گرا به دو مدل متمرکز و نامتمرکز (client/server) تقسیم می‌گردد.

به مدل متمرکز، مدل برنامه اصلی و برنامه فرعی نیز گفته می‌شود.

به مدل نامتمرکز، مدل فراخوانی روال از راه دور نیز گفته می‌شود.

مدل نامتمرکز (client/server) به دو صورت زیر می‌باشد:

۱- مدل داده محور (اشتراکی یا مخزنی)

۲- مدل توزیع شده

۲- گزینه (۲) صحیح است.

با دقت در DFD موجود درگزینه‌ها و وجود «مرکز تبدیل» مشخص می‌شود که جریان DFDهای موجود در گزینه‌ها از نوع تبدیل است.

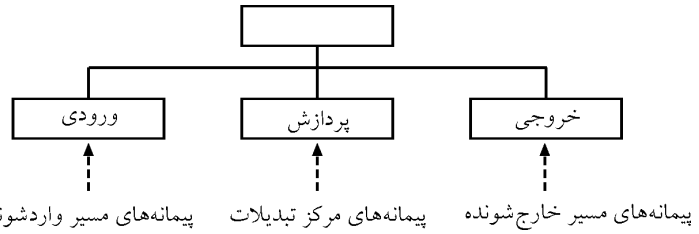
طبق قرارداد در افراز افقی برای جریان‌های تبدیل همواره سه واحد کنترل وجود دارد:

۱- کنترل‌کننده ورودی

۲- کنترل‌کننده خروجی

۳- کنترل‌کننده پردازش

به طور کلی ساختار فاکتورگیری سطح اول به صورت زیر است:



مطابق طراحی معماری مطرح شده در صورت سوال، یک کنترل‌کننده اصلی موسوم به «**S**» در بالای ساختار برنامه قرار دارد و به هماهنگ کردن عملیات کنترلی زیردستان کمک می‌کند.

الف) یک کنترل‌کننده پردازش اطلاعات ورودی موسوم به «**CI**» دریافت کلیه داده‌های ورودی را هماهنگ می‌کند.

ب) یک کنترل‌کننده جریان تبدیلی موسوم به «**T**» برای انجام کلیه عملیات بر روی داده‌ها به

شکل داخلی آن نظارت دارند (برای مثال پیمانه‌ای که روال‌های گوناگون تبدیل داده‌ها را فراخوانی می‌کنند).

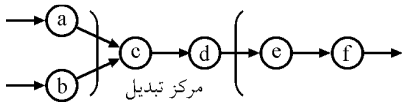
ج) یک کنترل‌کننده پردازش اطلاعات خروجی، که «**CO**» نامیده می‌شود، تولید اطلاعات خروجی را هماهنگ می‌کند.

روند نگاشت حباب‌ها به محل مناسب در ساختار معماری، در شاخه کنترل‌کننده ورودی، بدین نحو است که باید از مرز جریان ورودی DFD به سمت خارج DFD حرکت نمود و حباب‌هایی که در هر شاخه دیده می‌شود را بعنوان مولفه فرزند به ساختار معماری اضافه نمود، برای مثال در شاخه کنترل‌کننده ورودی در DFD مطرح شده در گزینه دوم، با اولین حرکت از مرز ورودی DFD به سمت خارج DFD، دو حباب «a» و «b» دیده می‌شود. این حباب‌ها به عنوان مولفه‌های فرزند مولفه «CI» به ساختار معماری اضافه می‌گردد.

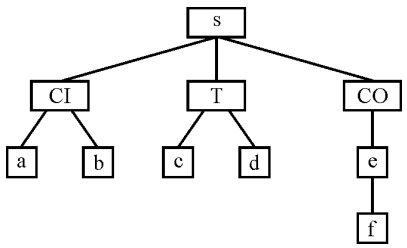
روند نگاشت حباب‌ها به محل مناسب در ساختار معماری، در شاخه کنترل‌کننده خروجی نیز، بدین نحو است که باید از مرز جریان خروجی DFD به سمت خارج DFD حرکت نمود و حباب‌هایی که در هر شاخه دیده می‌شود را بعنوان مولفه فرزند به ساختار معماری اضافه نمود. برای مثال در شاخه کنترل‌کننده خروجی در DFD مطرح شده در گزینه دوم، با اولین حرکت از مرز خروجی DFD به سمت خارج DFD، حباب «e» دیده می‌شود. این حباب به عنوان مولفه فرزند مولفه «CO» به ساختار معماری اضافه می‌گردد. با ادامه این حرکت به سمت خارج، حباب «f» دیده می‌شود. که در شاخه حباب «e» قرار دارد. این حباب نیز به عنوان مولفه فرزند مولفه «e» به معماری اضافه می‌گردد.

روند نگاشت حباب‌ها به محل مناسب در ساختار معماری، در شاخه کنترل‌کننده پردازش با کنترل‌کننده ورودی و خروجی متفاوت است، بدین نحو است که مستقل از سریال یا موازی بودن حباب‌های مرکز تبدیل، همه حباب‌های مرکز تبدیل، به عنوان مولفه‌های فرزند مولفه کنترل‌کننده پردازش به ساختار معماری اضافه می‌گردند. برای مثال در شاخه کنترل‌کننده پردازش در DFD مطرح شده در گزینه دوم، مستقل از موازی یا سریالی بودن حباب‌ها، حباب «c» و «d» به عنوان مولفه‌های فرزند مولفه «T» به ساختار معماری اضافه می‌گردند. شکل زیر، گویای مطلب است:

مدل تحلیل: (گزینه دوم)



مدل طراحی:



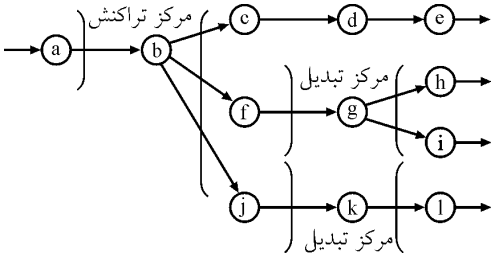
گزینه اول نادرست است. زیرا بخش ورودی و بخش خروجی DFD با طراحی معماری مطرح شده در سوال تطابق ندارد.

گزینه سوم نادرست است. زیرا بخش ورودی DFD با طراحی معماری مطرح شده در سوال تطابق ندارد.

گزینه چهارم نادرست است. زیرا بخش خروجی DFD با طراحی معماری مطرح شده در سوال تطابق ندارد.

۳- گزینه (۳) صحیح است.

مطابق DFD مطرح شده در صورت سوال داریم:



درDFD فوق، دو مسیر میانی و پایینی دارای جریان تبدیل هستند، بنابراین در ساختار کلی طراحی معماری، باید نگاشت تبدیل نیز مورد استفاده قرار گیرد.

طبق قرارداد، در افراز افقی برای جریان‌های تراکنش همواره دو شاخه وجود دارد:

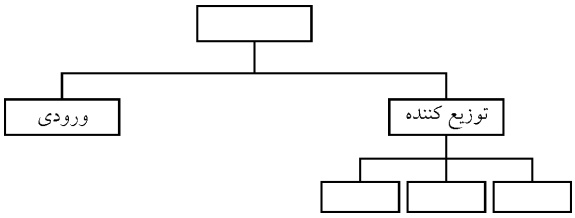
۱- شاخه ورودی (کنترل‌کننده دریافت)

۲- شاخه توزیع‌کننده (کنترل‌کننده ارسال)

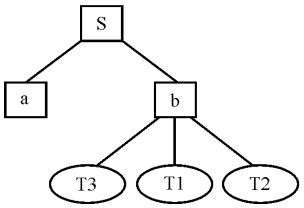
شاخه توزیع کننده (کنترل‌کننده ارسال)، همان حباب مرکز تراکنش است.

شاخه توزیع، یک پیمانه توزیع‌کننده است که تمامی پیمانه‌های فعالیت زیرین را کنترل می‌کند. هر کدام از مسیرهای فعالیت نیز به ساختاری متناظر خواص خودشان تصویر می‌شوند.

ساختار فاکتورگیری سطح اول برای جریان تراکنش به صورت زیر است:



بنابراین در گام اول، مطابق قوانین مطرح شده برای نگاشت تراکنش بر اساس DFD مطرح شده، داریم:



در ساختار فوق، سه مسیر T3، T1 و T2 به ترتیب سه مسیر کنشی، بالایی، میانی و پایینی را نشان می‌دهند.

مسیرهای T1 و T2، مسیرهایی از جنس جریان تبدیل و مسیر T3 مسیر ساده است. و نگاشت آن به شکل عادی انجام می‌گردد، یعنی حرکت از مرزهای DFD، به سمت خارج DFD است.

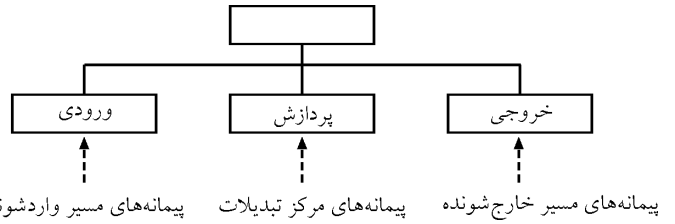
طبق قرارداد در افراز افقی برای جریان‌های تبدیل همواره سه واحد کنترل وجود دارد:

۱- کنترل‌کننده ورودی

۲- کنترل‌کننده خروجی

۳- کنترل‌کننده پردازش

به طور کلی ساختار فاکتورگیری سطح اول جریان تبدیل به صورت زیر است:



برای مسیر T1، یک کنترل‌کننده اصلی موسوم به «T1» در بالای ساختار معماری قرار می‌گیرد و به هماهنگ کردن عملیات کنترلی زیردستان کمک می‌کند.

الف) یک کنترل‌کننده پردازش اطلاعات ورودی موسوم به «**CI**» دریافت کلیه داده‌های ورودی را هماهنگ می‌کند.

ب) یک کنترل‌کننده جریان تبدیلی موسوم به «**CT**» برای انجام کلیه عملیات بر روی داده‌ها به شکل داخلی آن نظارت دارند (برای مثال پیمانه‌ای که روال‌های گوناگون تبدیل داده‌ها را فراخوانی می‌کنند).

ج) یک کنترل‌کننده پردازش اطلاعات خروجی، که «**CO**» نامیده می‌شود، تولید اطلاعات خروجی را هماهنگ می‌کند.

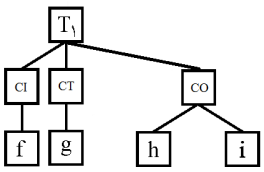
روند نگاشت حباب‌ها به محل مناسب در ساختار معماری، در شاخه کنترل‌کننده ورودی، بدین نحو است که باید از مرز جریان ورودی DFD به سمت خارج DFD حرکت نمود و حباب‌هایی که در هر شاخه دیده می‌شود را بعنوان مولفه فرزند به ساختار معماری اضافه نمود، برای مثال در شاخه کنترل‌کننده ورودی در مسیر T1، با اولین حرکت از مرز ورودی DFD به سمت خارج DFD، حباب «f» دیده می‌شود. این حباب به عنوان مولفه فرزند مولفه «CI» به ساختار معماری

اضافه می‌گردد.

روند نگاشت حباب‌ها به محل مناسب در ساختار معماری، در شاخه کنترل‌کننده خروجی نیز، بدین نحو است که باید از مرز جریان خروجی DFD به سمت خارج DFD حرکت نمود و حباب‌هایی که در هر شاخه دیده می‌شود را بعنوان مولفه فرزند به ساختار معماری اضافه نمود. برای مثال در شاخه کنترل‌کننده خروجی در مسیر T1، با اولین حرکت از مرز خروجی DFD به سمت خارج DFD، حباب های «i» و «h» دیده می‌شود. این حباب‌ها به عنوان مولفه‌های فرزند مولفه «CO» به ساختار معماری اضافه می‌گردد.

روند نگاشت حباب‌ها به محل مناسب در ساختار معماری، در شاخه کنترل‌کننده پردازش با کنترل‌کننده ورودی و خروجی متفاوت است، بدین نحو است که مستقل از سریال یا موازی بودن حباب‌های مرکز تبدیل، همه حباب‌های مرکز تبدیل، به عنوان مولفه‌های فرزند مولفه کنترل‌کننده پردازش به ساختار معماری اضافه می‌گردند.

برای مثال در شاخه کنترل‌کننده پردازش در مسیر T1، مستقل از موازی یا سریالی بودن حباب‌ها، حباب «g» به عنوان مولفه فرزند مولفه «CT» به ساختار معماری اضافه می‌گردد. شکل مقابل گویای مطلب است:



برای مسیرT2، یک کنترل‌کننده اصلی موسوم به «T2» در بالای ساختار معماری قرار می‌گیرد و به هماهنگ کردن عملیات کنترلی زیردستان کمک می‌کند.

الف) یک کنترل‌کننده پردازش اطلاعات ورودی موسوم به «**CI**» دریافت کلیه داده‌های ورودی را هماهنگ می‌کند.

ب) یک کنترل‌کننده جریان تبدیلی موسوم به «**CT**» برای انجام کلیه عملیات بر روی داده‌ها به شکل داخلی آن نظارت دارند (برای مثال پیمانه‌ای که روال‌های گوناگون تبدیل داده‌ها را فراخوانی می‌کنند).

ج) یک کنترل‌کننده پردازش اطلاعات خروجی، که «**CO**» نامیده می‌شود، تولید اطلاعات خروجی را هماهنگ می‌کند.

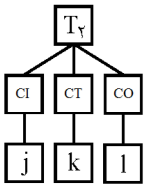
روند نگاشت حباب‌ها به محل مناسب در ساختار معماری، در شاخه کنترل‌کننده ورودی، بدین نحو است که باید از مرز جریان ورودی DFD به سمت خارج DFD حرکت نمود و حباب‌هایی که در هر شاخه دیده می‌شود را بعنوان مولفه فرزند به ساختار معماری اضافه نمود، برای مثال در شاخه کنترل‌کننده ورودی در مسیر T2، با اولین حرکت از مرز ورودی DFD به سمت خارج DFD، حباب «j» دیده می‌شود. این حباب به عنوان مولفه فرزند مولفه «CI» به ساختار معماری اضافه می‌گردد.

روند نگاشت حباب‌ها به محل مناسب در ساختار معماری، در شاخه کنترل‌کننده خروجی نیز، بدین نحو است که باید از مرز جریان خروجی DFD به سمت خارج DFD حرکت نمود و حباب‌هایی که در هر شاخه دیده می‌شود را بعنوان مولفه فرزند به ساختار معماری اضافه نمود.

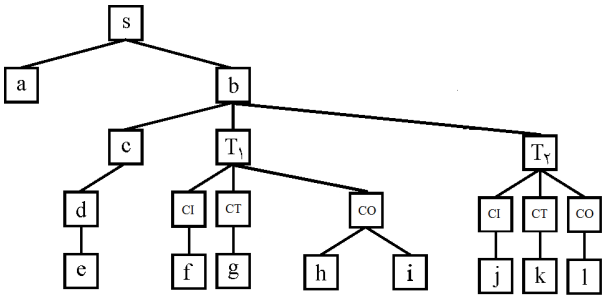
برای مثال در شاخه کنترل‌کننده خروجی در مسیر T2، با اولین حرکت از مرز خروجی DFD به سمت خارج DFD، حباب «l» دیده می‌شود. این حباب به عنوان مولفه‌های فرزند مولفه «CO» به ساختار معماری اضافه می‌گردد.

روند نگاشت حباب‌ها به محل مناسب در ساختار معماری، در شاخه کنترل‌کننده پردازش با کنترل‌کننده ورودی و خروجی متفاوت است، بدین نحو است که مستقل از سریال یا موازی بودن حباب‌های مرکز تبدیل، همه حباب‌های مرکز تبدیل، به عنوان مولفه‌های فرزند مولفه کنترل‌کننده پردازش به ساختار معماری اضافه می‌گردند.

برای مثال در شاخه کنترل‌کننده پردازش در مسیر T2، مستقل از موازی یا سریالی بودن حباب‌ها، حباب «k» به عنوان مولفه فرزند مولفه «CT» به ساختار معماری اضافه می‌گردد. شکل مقابل گویای مطلب است:

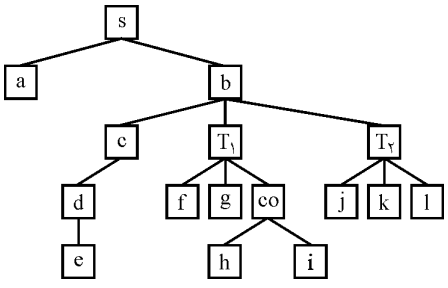


در حالت کلی ساختار طراحی معماری به صورت زیر است:



هرگاه در طراحی معماری، یک مولفه کنترل‌کننده (پدر)، فقط یک مولفه تحت کنترل (فرزند) داشته باشد، می‌توان مولفه کنترل‌کننده (پدر) را حذف نمود.

شکل مقابل، ساختار پالایش شده طراحی معماری می‌باشد، زیرا کنترل کننده‌های اضافی، حذف شده‌اند.



۴- گزینه (۱) صحیح است.

نرم‌افزار نیز همانند یک ساختمان، دارای سبک‌های متفاوتی برای ساخت می‌باشد. به طور کلی برای طراحی معماری و ساخت نرم‌افزارها دو سبک ساخت‌یافته (مبتنی بر فراخوانی و بازگشت

توابع) و سبک شیءگرا (مبتنی بر ارسال پیام مابین اشیاء) مورد استفاده قرار می‌گیرد.

سبک ساخت‌یافته (مبتنی بر فراخوانی و بازگشت توابع) به دو طبقه کلی معماری تابع‌گرا (جریان داده یا تجزیه عملیات) و معماری داده‌گرا تقسیم می‌گردد.

معماری جریان داده یا تابع‌گرا یا تجزیه عملیات هنگامی به کار برده می‌شود که قرار است داده‌های ورودی از طریق یک سری مؤلفه‌های محاسباتی یا دست کاری کننده، به داده‌های خروجی تبدیل شوند.

معماری تابع‌گرا، برای کاربردهایی با محاسبات پیچیده و سنگین مورد استفاده قرار می‌گیرند، مانند نرم‌افزارهای محاسبات ریاضی. در این نوع کاربردها حجم داده‌ها پایین و حجم محاسبات بالا است. برای مثال نرم‌افزار ماشین حساب از این مدل معماری استفاده می‌کند. نوع داده در این مدل معماری، متغیر ساده است.

معماری داده‌گرا، برای کاربردهای بانک اطلاعات، مورد استفاده قرار می‌گیرد. در این نوع کاربردها حجم داده‌ها بالا و حجم محاسبات پایین است. نوع داده در این مدل معماری، جداول و رکوردهای بانک اطلاعات است.

معماری داده‌گرا به دو مدل متمرکز و نامتمرکز (client/server) تقسیم می‌گردد.

مدل نامتمرکز (client/server) به دو صورت زیر می‌باشد:

۱- مدل داده محور (اشتراکی یا مخزنی)

۲- مدل توزیع شده

بنابراین معماری تابع‌گرا و داده‌گرا از نظر ماهیت داده‌ای و ساختمان داده با هم متفاوت هستند، یکی برای ذخیره داده‌های خود از متغیرهای ساده استفاده می‌کند و دیگری از جداول و رکوردها. از آنجا که نحوه دسترسی به این داده‌ها یعنی متغیرهای ساده و جداول و رکوردها متفاوت است، بنابراین ساختار برنامه‌نویسی معماری تابع‌گرا و داده‌گرا نیز متفاوت خواهد بود.

۵- گزینه (۳) صحیح است.

نیازمندی‌های عملکردی مشتری در مدل تحلیل توسط DFD، مدل‌سازی عملکردی می‌شود، سپس توسط نگاشتی که بر اساس سبک فراخوانی و بازگشت انجام می‌گردد، هر کدام از حباب‌های DFD در سطح مورد نظر، توسط یک مؤلفه مرتبط در ساختار معماری مشخص می‌شود. به بیان دیگر طراحی معماری، متناظر با حباب‌های موجود در DFD انجام می‌شود و مولفه‌های مناسب در طراحی معماری به دست می‌آید. به عبارت دیگر هر حباب موجود در DFD، به یک مولفه در ساختار طراحی معماری نگاشت می‌گردد. بنابراین هرچه نیازمندی‌های عملکردی، بهتر شناخته شوند، حباب‌های موجود در DFD کامل‌تر خواهد بود، و به تبع در نگاشت DFD به ساختار معماری، طراحی معماری نیز ایده‌آل‌تر خواهد بود. میزان ایده‌آل بودن یک طراحی معماری، از رابطه «شاخص گزینش طراحی» به صورت زیر محاسبه می‌گردد:

$$d = \left(\frac{N_s}{N_a}\right) * ۱۰۰$$

که در آن N_s تعداد ابعاد (مولفه‌های) به کار رفته در یک معماری پیشنهادی است، که توسط سازنده شناسایی شده‌اند.

و N_a تعداد کل ابعاد (مولفه‌های) مورد نظر مشتری در فضای طراحی است، یعنی آنچه مطلوب مشتری است تا توسط سازنده شناخته شود.

مقدار شاخص گزینش طراحی (d)، مقداری بین ۰ و ۱۰۰ برحسب درصد است. هرچه این مقدار، به عدد ۱۰۰ نزدیکتر باشد، معماری پیشنهادی به معماری ایده‌آل نزدیکتر است.

مثال: اگر مقدار N_a برابر ۲۰ و مقدار N_s برابر ۱۵ باشد، مقدار شاخص گزینش طراحی (d)، برابر ۷۵٪ خواهد بود.

مثال: اگر مقدار N_a برابر ۲۰ و مقدار N_s برابر ۲۰ باشد، مقدار شاخص گزینش طراحی (d)، برابر ۱۰۰٪ خواهد بود.

۶- گزینه (۴) صحیح است.

نرم‌افزار نیز همانند یک ساختمان، دارای سبک‌های متفاوتی برای ساخت می‌باشد. به طور کلی برای طراحی معماری و ساخت نرم‌افزارها دو سبک ساخت‌یافته (مبتنی بر فراخوانی و بازگشت توابع) و سبک شیءگرا (مبتنی بر ارسال پیام مابین اشیاء) مورد استفاده قرار می‌گیرد.

سبک ساخت‌یافته (مبتنی بر فراخوانی و بازگشت توابع) به دو طبقه کلی معماری تابع‌گرا (جریان داده یا تجزیه عملیات) و معماری داده‌گرا تقسیم می‌گردد.

معماری جریان داده یا تابع‌گرا یا تجزیه عملیات هنگامی به کار برده می‌شود که قرار است داده‌های ورودی از طریق یک سری مؤلفه‌های محاسباتی یا دست کاری کننده، به داده‌های خروجی تبدیل شوند.

معماری تابع‌گرا، برای کاربردهایی با محاسبات پیچیده و سنگین مورد استفاده قرار می‌گیرند، مانند نرم‌افزارهای محاسبات ریاضی. در این نوع کاربردها حجم داده‌ها پایین و حجم محاسبات بالا است. برای مثال نرم‌افزار ماشین حساب از این مدل معماری استفاده می‌کند.

نوع داده در این مدل معماری، متغیر ساده است.

معماری داده‌گرا، برای کاربردهای بانک اطلاعات، مورد استفاده قرار می‌گیرد. در این نوع کاربردها حجم داده‌ها بالا و حجم محاسبات پایین است.

نوع داده در این مدل معماری، جداول و رکوردهای بانک اطلاعات است.

معماری داده‌گرا به دو مدل متمرکز و نامتمرکز (client/server) تقسیم می‌گردد.

در مدل متمرکز معماری یک بانک اطلاعات روی یک سیستم کامپیوتری و بدون ارتباط با سیستم های کامپیوتری دیگر ایجاد می‌شود. و برای کاربردهای کوچک و با امکانات محدود است. سخت‌افزار این سیستم می‌تواند در حد یک کامپیوتر شخصی و یا بالاتر باشد. در این مدل، بانک اطلاعات و برنامه کاربردی آن، هر دو بر روی یک کامپیوتر قرار دارند. برای مثال به نرم‌افزار دفترچه تلفن بر روی یک کامپیوتر شخصی می‌توان اشاره نمود.

در مدل نامتمرکز (client/server) مدل، بانک اطلاعات و برنامه کاربردی آن، بر روی

کامپیوترهای مختلفی قرار دارند. در مدل نامتمرکز مجموعه‌هایی از serverها، مجموعه‌ای از سرویس‌ها را به clientها از طریق شبکه‌های کامپیوتری ارئه می‌دهند.

مدل نامتمرکز (client/server) به دو صورت زیر می‌باشد:

۱- مدل داده محور (اشتراکی یا مخزنی)

۲- مدل توزیع شده

در مدل داده محوری یا اشتراکی یا مخزنی(Data Repository) یک مخزن داده‌ای یا بانک اطلاعات (Server) در مرکز این معماری قرار دارد و مؤلفه‌های دیگری (Client) که داده‌های آن را بروزرسانی، درج یا حذف می‌کنند، به دفعات به آن دستیابی دارند.

برای مثال به clientهای موجود در شعبه‌های یک بانک سرمایه که با server مرکزی در ارتباط هستند می‌توان اشاره نمود.

در مدل توزیع‌شده (Distributed)، اصطلاح پردازش توزیعی بدین مفهوم است که ماشین‌های مجزا می‌توانند در یک شبکه ارتباطی به یکدیگر متصل شوند. به گونه‌ای که یک عملکرد پردازش داده منفرد می‌تواند بر روی چندین ماشین شبکه پراکنده شود. در سیستم پایگاه داده‌های توزیع شده، پایگاه داده‌ها بر روی چند کامپیوتر ذخیره می‌شود. کامپیوترها در یک سیستم توزیع شده از طریق شبکه‌های کامپیوتری به یکدیگر مرتبط هستند. می‌توان گفت که در این مدل، تعدادی پایگاه داده‌های ذخیره شده روی کامپیوترهای مختلف وجود دارد، که از نظر کاربران، پایگاه داده واحدی هستند. به بیان دیگر داده‌ها در محل‌های مختلف ذخیره شده‌اند، اما کاربران به هنگام استفاده از داده‌ها به ظاهر پایگاه داده‌ای یکپارچه را مشاهده می‌کنند.

برای مثال به clientهای موجود در شعبه‌های یک بانک سرمایه که با server توزیع شده در ارتباط هستند می‌توان اشاره نمود.

عبارت موجود در گزینه چهارم «در معماری سرویس‌دهنده / سرویس‌گیرنده (Clien/Server) مدل داده‌های مشترکی مورد استفاده قرار می‌گیرد.» نادرست است. زیرا، در معماری سرویس‌دهنده / سرویس‌گیرنده (Clien/Server) علاوه بر مدل داده‌های مشترکی یا مدل داده محور (اشتراکی یا مخزنی) مدل توزیع شده نیز می‌تواند مورد استفاده قرار گیرد.

۷- گزینه (۲) صحیح است.

مدل طراحی سیستم‌های تجاری شامل مراحل زیر است:

۱) طراحی داده

۲) طراحی معماری (ارتباط مؤلفه‌ها)

۳) طراحی واسط (واسط کاربر و برنامه)

۴) طراحی مؤلفه (جزییات دقیق‌تر مولفه‌ها)

طراحی شمای پایگاه داده در طراحی داده انجام می‌گردد.

تعیین معماری بستر(Platform) در طراحی معماری انجام می‌گردد.

طراحی واسط کاربر، در طراحی واسط انجام می‌گردد.

اگر منظور از عبارت «تعیین محدودیت‌های زمانی» در گزینه دوم، محدودیت‌های زمانی در پاسخ‌گویی سیستم به کاربر باشد، این محدودیت‌های زمانی در پاسخگویی سیستم مربوط سیستم‌های بی‌درنگ است و مرتبط با سیستم‌های تجاری نیست، البته این محدودیت‌های زمانی به عنوان خواسته و نیازمندی مشتری در فعالیت ارتباطات به عنوان مرحله تهیه لیست نیازمندی‌های مشتری باید توسط مشتری بیان گردد و در طراحی مولفه با جزئیات بیشتر تشریح گردد. بنابراین گزینه دوم نادرست است.

همچنین اگر منظور از عبارت «تعیین محدودیت‌های زمانی» در گزینه دوم، محدودیت‌های زمانی مربوط به خصوصیات و ریسک‌های پروژه‌های موفق نرم‌افزاری که شامل بازه‌ی زمانی از قبل برنامه‌ریزی شده (بازهٔ زمانی مشخص یا محدودیت‌های زمانی)، بودجه‌ای از قبل پیشبینی شده و با صرف کمترین هزینه (مقرون به صرفه) و دقیقاً مطابق با نیازمندی‌های واقعی کاربران (کیفیت مطلوب) باشد، آنگاه کنترل و مدیریت این محدودیت‌های زمانی مرتبط با مدیریت پروژه و فعالیت‌های چتری است، که ارتباطی به فعالیت مدل طراحی سیستم‌های تجاری ندارد. بنابراین از این نگاه نیز مجددا گزینه دوم نادرست است.

۸- گزینه (۴) صحیح است.

از آنجاکه گزینه‌های اول، دوم و سوم نیازمندی‌هایی را به صورت کلی مطرح نموده است. بنابراین این نیازمندی‌ها مربوط به نیازمندی‌های مشتری بوده و در فعالیت ارتباطات توسط ارتباط‌گر تهیه و جمع‌آوری می‌گردد.

مدل طراحی شامل مراحل زیر است:

۱) طراحی داده

۲) طراحی معماری (ارتباط مؤلفه‌ها)

۳) طراحی واسط (واسط کاربر و برنامه)

۴) طراحی مؤلفه (جزییات دقیق‌تر مولفه‌ها)

عبارت «زیرسامانه‌ی واسط کاربر شامل دو زیرسامانه مجزا برای تعامل با انواع مختلف کاربران خواهد بود.» در گزینه چهارم، مربوط به مدل طراحی، بخش طراحی واسط است. بنابراین گزینه چهارم درست است.

۹- گزینه (۳) صحیح است.

تئوماندل در کتاب «طراحی واسط» خود، سه قانون طلایی، برای طراحی واسط کاربر مناسب، بیان نموده است:

۱- سپردن کنترل برنامه به کاربر (کاربر باید حس کند که برنامه را تحت کنترل دارد)

۲- کاهش بار حافظه کاربر

۳- سازگاری واسط کاربر

توجه: زمان پاسخ مناسب موجود در گزینه سوم، مربوط به نیازمندهای غیروظیفه‌مندی است و نه طراحی واسط کاربر.

۶ مفاهیم شیءگرایی

مقدمه

عالم انسان‌ها، عالمی مبتنی بر شیءگرایی است، و انسان‌ها در کنار یکدیگر از طریق همکاری و گفتگو مسائل خود را حل می‌کنند. از آنجا که برنامه‌های کامپیوتری برای حل مسائل انسان‌ها نوشته می‌شوند، می‌توان همان رویکرد شیءگرایی عالم انسان‌ها را به عالم برنامه‌های کامپیوتری نیز وارد کرد و به همان شیوه‌ای در نوشتن برنامه‌های کامپیوتری فکر کرد که در عالم انسان‌ها فکر می‌شود. همانطور که انسان‌ها در اجتماع انسانی خود برای حل مسائل به همکاری و گفتگو می‌پردازند، اشیا نیز می‌توانند در اجتماع برنامه‌های کامپیوتری برای حل مسائل به همکاری و گفتگو بپردازند. در عالم شیءگرایی انسان‌ها، هر شی پدیده‌ای قابل لمس است که دارای صفات و عملکردهایی است. بنابراین می‌توان گفت خود انسان، یک شی است که دارای صفاتی همچون نام، نام خانوادگی و غیره و عملکردهایی همچون تکلم، دیدن، شنیدن و پمپاژ خون است. برای مثال شما با فراخونی عملکرد تکلم خود و سپس فراخوانی عملکرد شنیداری دوستتان از طریق ارسال پیام، با دوستتان برای حل یک مساله به گفتگو می‌پردازید. بنابراین دنیای انسانی ما نیز دنیای شیءگرایی و اشیا است.

متدولوژی شیءگرا (مهندسی نرم‌افزار شیءگرا)

متدولوژی شیءگرا یا مهندسی نرم‌افزار شیءگرا نظامی است یکپارچه شامل مدل فرآیند شیءگرا (مدرن)، روش شیءگرا (مبتنی بر مفاهیم کلاس، وراثت و چندریختی) و ابزارهای شیءگرا که منجر به ایجاد نرم‌افزاری در بازه‌ی زمانی از قبل برنامه‌ریزی شده، بودجه‌ای از قبل پیش‌بینی شده و دقیقاً مطابق با نیازمندی‌های واقعی مشتری می‌گردد.

در متدولوژی شیءگرا، در اولین مرحله‌ی مدل‌سازی (مدل تحلیل) سیستم در قالب کلاس‌ها (شامل داده (صفت) و عملکرد (متد)) نشان داده می‌شود. سپس طی روندی سلسله مراتبی و مطابق با روش بالا به پایین، کلاس‌ها با جزئیات بیشتری مشخص می‌شوند. این روند تا به جایی

ادامه می‌یابد که جزئیات کلاس‌های برنامه جهت پیاده‌سازی مشخص شوند.

توجه: متدولوژی RUP متداول‌ترین نمونه از متدولوژی شیءگرا براساس روش شیءگرا (مبتنی بر مفاهیم کلاس، وراثت و چندریختی)، مدل فرآیند مبتنی بر مؤلفه‌ی شیءگرا با رویکرد تکرار و تکامل و ابزارهای شیءگرا (مثل ابزار مدل‌سازی UML و زبان برنامه‌نویسی C++) می‌باشد.

توجه: نسبت نمونه متدولوژی شیءگرای RUP به متدولوژی شیءگرا مثل نسبت سیستم عامل ویندوز مدرن به مفاهیم مدرن سیستم عامل است.

توجه: متدولوژی RUP به تفصیل در فصل مربوطه شرح داده خواهد شد.

توجه: UML در فصل بعد شرح داده می‌شود.

توجه: مدل فرآیند مبتنی بر مؤلفه‌ی شیءگرا و روش شیءگرا (مبتنی بر مفاهیم کلاس، وراثت و چندریختی) جلوتر شرح داده می‌شود.

مدل فرآیند تولید نرم‌افزار مدرن (شیءگرا)

مدل فرآیند تولید نرم‌افزار مدرن، به صورت تکاملی مدرن می‌باشد.

مدل تکاملی مدرن

مدل تکاملی مدرن، مبتنی بر مؤلفه شیءگرا نام دارد و براساس رویکرد تکرار و تکامل می‌باشد.

مدل توسعه‌ی مبتنی بر مؤلفه شیءگرا

مؤلفه یک قطعه‌ی آماده، کاربردی و پیاده‌سازی شده است که دارای واسط لازم جهت اتصال با سایر قطعات و استقرار در بخشی از یک سیستم عملیاتی می‌باشد.

در حیطه مهندسی نرم‌افزار شیءگرا، **واحد مؤلفه**، یک قطعه‌ی عملیاتی مبتنی بر کلاس است که بر دو نوع می‌باشد:

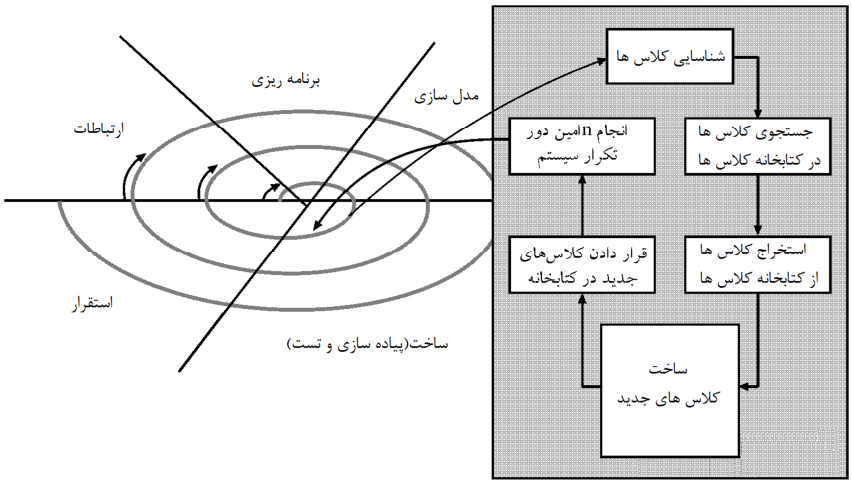
۱) کلاس کاربردی: مانند کلاس دانشجو که برنامه‌نویس آن را می‌نویسد و به دلیل داشتن شرایط قابل حمل به شکل ذاتی، می‌تواند به عنوان یک قطعه‌ی آماده و قابل استفاده‌ی مجدد در پروژه‌های بعدی مورد استفاده مجدد قرار گیرد.

۲) کلاس سیستمی: مانند کلاس جعبه‌ی متن که کامپایلر تعاریف آن را فراهم می‌کند و می‌تواند به عنوان یک قطعه‌ی آماده و قابل استفاده‌ی مجدد، در پروژه‌ها مورد استفاده مجدد قرار گیرد.

این مدل از نظر ماهیت، مدلی تکاملی است و ساختاری تکرارشونده را برای توسعه‌ی نرم‌افزار در پیش می‌گیرد. اما در تولید برنامه‌های کاربردی از مؤلفه‌های آماده استفاده می‌کند، در واقع این مدل، برنامه را با استفاده از ترکیب و سرهم کردن قطعات نرم‌افزاری از پیش ساخته شده می‌سازد.

روال استقرار کلاس در معماری نرم‌افزار (ساختار برنامه یا اسکلت برنامه) براساس مدل توسعه‌ی مبتنی بر مؤلفه شیءگرا به صورت زیر است:

۱- شناسایی کلاس‌های مورد نیاز برنامه
۲- جستجوی کلاس‌ها در کتابخانه‌ی کلاس‌های سیستمی یا کتابخانه‌ی کلاس‌های کاربردی
توجه: کلاس‌های کاربردی ایجاد شده در پروژه‌های قبلی در کتابخانه‌ی کلاس‌های کاربردی، نگهداری می‌شوند.
۳- در صورت وجود کلاس‌های مورد نیاز در کتابخانه‌ی کلاس‌های سیستمی یا کتابخانه‌ی کلاس‌های کاربردی، کلاس استخراج و دوباره استفاده می‌شود. در غیراینصورت کلاس مورد نیاز ایجاد می‌گردد.
۴- کلاس در معماری نرم‌افزار (اسکلت برنامه) استقرار می‌یابد.
۵- انجام تست برای اطمینان از صحت کار انجام می‌شود.
توجه: دقت کنید که مراحل فوق مبتنی بر تکرار و تکامل صورت می‌گیرد، یعنی پروژه به تدریج و براساس تکرار، تکامل می‌یابد.
توجه: زمان و هزینه‌ی فرآیند تولید نرم‌افزار براساس مدل توسعه‌ی مبتنی بر مؤلفه شیءگرا به دلیل استفاده از قطعات آماده (قابلیت استفاده مجدد) کاهش چشمگیری دارد. در واقع یک قطعه با قابلیت استفاده‌ی مجدد یک بار ساخته می‌شود و بارها و بارها استفاده می‌شود و این یعنی سودآوری!
شکل زیر ساختار تکاملی مدل توسعه‌ی مبتنی بر مؤلفه شیءگرا را نشان می‌دهد:



مدل توسعه‌ی مبتنی بر مؤلفه شیءگرا

همان‌طوری که در شکل فوق مشخص است، فرآیند در راستای یک چرخش تکاملی حرکت می‌کند که شروع آن، ارتباط با مشتری برای تشخیص نیازمندی‌ها و سپس فعالیت مدل سازی که منجر به تعیین کلاس‌های پایه می‌شود. فرآیند تولید شیءگرایی بر استفاده مجدد تأکید دارد. بنابراین پیش از اقدام به ساخت یک کلاس جدید، کتابخانه کلاس‌های موجود مورد جستجو قرار

می‌گیرد. هنگامی که کلاسی در کتابخانه پیدا نشود، مهندسی نرم‌افزار با اجرای تحلیل شیءگرا (OOA)، طراحی شیءگرا (OOD)، برنامه‌نویسی شیءگرا (OOP) و تست شیءگرا (OOT) مبادرت به ایجاد آن کلاس و اشیای مشتق از آن می‌کند. سپس کلاس جدید در کتابخانه قرار داده می‌شود به طوری که در آینده بتوان دوباره از آن استفاده مجدد نمود.

توجه: دیدگاه شیءگرا مستلزم استفاده از مدل فرآیند تکاملی در مهندسی نرم‌افزار است، بدیهی است که تعیین کلیه کلاس‌های لازم برای یک سیستم نرم‌افزاری در یک دور تکرار، کاری بسیار دشوار است. به موازات تکامل نرم افزار نیاز به کلاس‌های بیشتر حس می‌شود. بنابراین الگوی تکاملی برای فرآیند تولید شیءگرایی بسیار مناسب است.

شیءگرایی

شیءگرایی پیش از آنکه یک متدولوژی مشخص برای توسعه سیستم‌ها باشد یک قالب فکری برای مدل‌سازی و ساخت سیستم‌ها است.

دیدگاه شیءگرایی از اواسط دهه‌ی ۱۹۷۰ مطرح شد. در این دوران تلاش‌های زیادی برای ایجاد نمونه متدولوژی‌های شیءگرا صورت پذیرفت. در نتیجه این تلاش‌ها بود که در طول ۵ سال یعنی ۱۹۸۹ تا ۱۹۹۴، تعداد نمونه متدولوژی‌های شیءگرا از کمتر از ۱۰ نمونه متدولوژی به بیش از ۵۰ نمونه متدولوژی رسید. تکثیر نمونه متدولوژی‌های شیءگرا و زبان‌های شیءگرا و رقابت بین آنها به حدی بود که این دوران به عنوان «جنگ نمونه متدولوژی‌ها» لقب گرفت. از جمله نمونه متدولوژی‌های شیءگرای معروف و پرکاربرد آن زمان می‌توان به OSE، OMT و BOOCH اشاره نمود.

فراوانی و اشباع نمونه متدولوژی‌های شیءگرا و نیز نبودن یک زبان مدل‌سازی استاندارد باعث مشکلات فراوانی شده بود.

بسیاری از این نمونه متدولوژی‌های شیءگرا، مفاهیم مشترک شیءگرایی را در قالب و مدل‌های مختلف بیان می‌کردند که این واگرایی و نبودن توافق میان نحوه مدل‌سازی، کاربران تازه کار را از دنیای شیءگرایی زده می‌کرد.

سرانجام در سال ۲۰۰۰ شرکت رشنال، نمونه متدولوژی شیءگرایی را تحت عنوان RUP یا Rational Unified Process به معنی فرآیند یکپارچه رشنال را مطرح ساخت که امروزه از طرفداران بسیاری برخوردار است. شرکت رشنال بعدها توسط شرکت IBM خریداری شد.

توجه: RUP ، یک نمونه متدولوژی شیءگرا است که در جهت کنترل و انجام پروژه‌های نرم‌افزاری ایجاد شده است. در واقع چارچوبی در جهت هدایت صحیح و موفق پروژه‌های نرم‌افزاری می‌باشد که کلیه فعالیت‌های چارچوبی انجام پروژه یعنی ارتباط، تحلیل، طراحی، پیاده‌سازی، تست و استقرار را تحت کنترل دارد. RUP، یک چارچوب عمومی است که برای کلیه پروژه‌ها صرف‌نظر از اندازه و میزان پیچیدگی آنها امکاناتی برای ایجاد نرم‌افزار فراهم می‌کند.

توجه: از آنجا که RUP از یکپارچه‌سازی سه نمونه متدولوژی شیءگرای معروف OMT، OSE

و BOOCH ایجاد شده است به آن فرآیند یکپارچه رشنال می‌گویند.

توجه: RUP یک نمونه متدولوژی بزرگ صنعتی، برای تولید سیستم‌های نرم‌افزاری است که برای سهولت درک آن، کلیاتی از آن بدون نام شرکت رشنال و بدون محرز کردن جریان‌های کاری مربوط به فعالیت‌های چتری تولید نرم‌افزار و مدل‌سازی کسب و کار و بدون اشاره به قدرت RUP که همان ابزارهای حمایت‌کننده آن می‌باشد، در قالب فرآیند یکپارچه توسعه نرم‌افزارUSDP یا Unifed Software Development Process در دانشگاه‌های معتبر جهان ظهور کرده است. در واقع می‌توان گفت که RUP نسخه پیاده‌سازی شده‌ای از USDP است. بنابراین USDP به عنوان یک فرآیند شیءگرای تولید و توسعه سیستم‌ها، دارای مدل فرآیندی است که روند کلی تولید نرم‌افزار را مشخص می‌کند. به بیان دیگر شکل خلاصه شده RUP، USDP است.

توجه: زبان مدل‌سازی یکپارچه UML یا Unified Modeling Language یک متدولوژی نیست بلکه یک زبان مدل‌سازی می‌باشد که صرفاً برای مدل‌سازی مراحل تحلیل و طراحی شیءگرایی توسط علایم ویژه و استاندارد که برای همه قابل فهم باشد به کار می‌رود.

توجه: RUP در فصل مربوطه تشریح خواهد شد.

توجه: UML در فصل بعد تشریح خواهد شد.

در دنیای واقعی، انسان‌ها اشیاء را دسته‌بندی می‌کنند و از این طریق بهتر پی به خصوصیات اشیاء و وابستگی میان آنها می‌برند. در دیدگاه شیءگرا، دامنه مسئله به عنوان مجموعه‌ای از کلاس‌ها توصیف می‌شود که دارای صفات و متدهای معین هستند. داده‌های نمونه‌هایی از این کلاس‌ها، به عنوان اشیاء، توسط توابع که موسوم به متد هستند دستکاری می‌شوند و همچنین اشیاء این امکان را دارند تا در اجتماع برنامه از طریق مکانیزم پیام‌رسانی با یکدیگر ارتباط برقرار کنند. اشیاء در دسته‌بندی‌هایی با نام کلاس قرار می‌گیرند.

عمل تعریف کلاس‌های کاربردی، شامل تعریف کردن صفات و متدها می‌شود که این کار توسط مهندس نرم‌افزار انجام می‌گیرد. یک کلاس هم داده (صفت) و هم عملکرد (متد) را که باید روی داده انجام شود بسته‌بندی (کپسوله‌سازی) می‌کند. این خاصیت مهم منجر به ساخت کلاس‌هایی می‌گردد که می‌توان از آنها استفاده مجدد نمود.

قابلیت استفاده مجدد، امروزه به عنوان یکی از صفات حائز اهمیت در تولید محصولات نرم‌افزاری است و این ویژگی در متدولوژی‌های شیءگرا بسیار نمود یافته است. یعنی متدولوژی شیءگرا، قطعات نرم‌افزاری با کیفیت بالایی را تولید می‌کند که قابلیت استفاده مجدد دارند.

در متدولوژی شیءگرا نیز همانند متدولوژی ساخت‌یافته، مراحل ارتباط، تحلیل، طراحی، پیاده‌سازی، تست و استقرار وجود دارد. با این تفاوت که در فعالیت مدل تحلیل، کلاس‌های موجود در دامنه مسئله شناسایی و مدل سازی می‌گردند. در مرحله طراحی، معماری، واسط کاربر و مؤلفه‌های نرم‌افزار تعیین می‌شوند و در پیاده‌سازی، طراحی انجام شده توسط یک زبان شیءگرا به کد تبدیل می‌شود.

آنچه که در طی فرآیند مهندسی نرم‌افزار شیءگرا (متدولوژی شیءگرا) تولید می‌شود

مجموعه‌ای از مدل‌های شیءگرا است که نیازمندی‌ها، تحلیل، طراحی، پیاده‌سازی و تست محصول را تشریح می‌کنند.

ویژگی‌های اصلی شیءگرایی

سه ویژگی اصلی شیءگرایی عبارتند از: کپسوله‌سازی، وراثت و پلی‌مورفیسم (چندریختی)

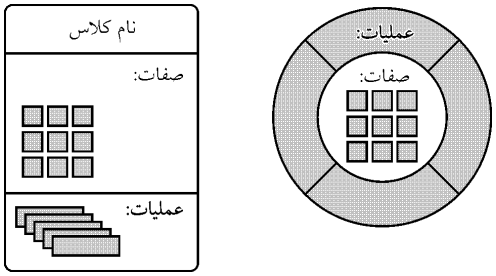
الف) بسته‌بندی یا کپسوله‌سازی (Encapsulation)

در شیءگرایی، بسته‌بندی نتیجه پیمانه‌ای کردن است. کنار هم قرار دادن صفات (داده‌ها) و متدهای (دستورالعمل یا رفتارها یا عملکردها یا توابع) مرتبط به هم، در یک بسته را بسته‌بندی می‌گویند. در واقع متدهایی که بر روی یک ساختار داده‌ای عمل می‌کنند، با ساختار داده‌ای مربوطه در یک بسته قرار می‌گیرند. کپسوله‌سازی امکان می‌دهد جعبه سیاه‌هایی به نام کلاس‌ها ساخته شوند که کاملاً مستقل از کلاس‌های دیگر باشد، درون این جعبه تمام داده‌ها و عملکردهای لازم وجود دارد.

کلاس (Class)

کلاس مکانیزمی است که توسط آن، مفهوم کپسوله‌سازی پیاده‌سازی می‌شود.

بنابراین مطابق تعریف بسته‌بندی، کلاس نیز صفات و متدهای مرتبط به هم را در یک بسته، بسته‌بندی می‌کند. درون یک کلاس، صفات و متدها یا هردو می‌توانند به صورت اختصاصی (private) و یا عمومی (public) باشند. صفات و متدهای اختصاصی، تنها توسط عناصر همان کلاس، شناخته شده و قابل دسترس هستند. بدین معنا که بخش‌های دیگر برنامه که خارج از محدوده کلاس هستند نمی‌توانند به صفات و متدهای کلاس دسترسی داشته باشند. اما زمانی که صفات و متدهای یک کلاس به صورت عمومی تعریف شوند، سایر بخش‌های برنامه می‌توانند به آنها دسترسی داشته باشند. شکل زیر نحوه نمایش کلاس‌ها را نشان می‌دهد:



توجه: وجه تمایز کلاس‌ها در تفاوت در صفات آن‌ها است. مانند تفاوت در صفات کلاس استاد و کلاس دانشجو.

کلاس‌ها دو چهره دارند:

الف) چهره داخلی یا خصوصی (Internal)

که شامل صفات و متدهای خصوصی کلاس است. دسترسی به این بخش فقط توسط خود

کلاس امکان‌پذیر است.

ب) چهره رابط یا عمومی (Interface)

که شامل متدهای عمومی یا واسط کلاس است. تنها وسیله ارتباطی اشیاء چهره رابط آنهاست. اشیاء درون یک سیستم با فراخوانی چهره رابط یکدیگر (که همان پیام‌رسانی است) کارها را انجام می‌دهند. بنابراین یک شیء که پیامی را ارسال می‌کند نیازی ندارد تا از جزئیات ساختار داخلی شیء مقصد اطلاعاتی داشته باشد.

توجه: مکانیزم پیام را جلوتر شرح می‌دهیم.

شیء (Object)

به نمونه‌ای از یک کلاس، شیء گفته می‌شود. مانند دانشجو اکبری از کلاس دانشجو.

توجه: معرفی یک کلاس یک انتزاع یا تجرید **منطقی** (logical abstraction) است که یک نوع داده جدید را معرفی می‌کند و نشان می‌دهد که یک شی از این نوع، چه شکل و شمایلی دارد. اما معرفی یک شیء از یک کلاس، یک موجودیت **فیزیکی** (physical entity) از نوع یک کلاس را ایجاد می‌کند، بدین معنی که با معرفی یک شیء، فضای حافظه‌ای به شیء اختصاص داده می‌شود ولی در معرفی یک کلاس فضای حافظه‌ای تخصیص نمی‌یابد.

توجه: وجه تمایز اشیاء در مقادیر صفات اشیاء است. مانند تفاوت در مقادیر دانشجو اکبری و دانشجو احمدی.

صفات (Attributes)

کلاس‌ها توسط صفات از یکدیگر متمایز و شناخته می‌شوند. صفات یک کلاس توسط متدهای همان کلاس قابل دستکاری هستند. صفات یک کلاس می‌تواند بر اساس محدوده دامنه آن مقداردهی شود. مانند صفات وزن و رنگ مربوط به یک شیء.

متد یا عملکرد یا رفتار یا سرویس (Methods)

مجموعه عملیات مربوط به دستکاری یا دسترسی به صفات یک کلاس را متدهای آن کلاس می‌گویند.

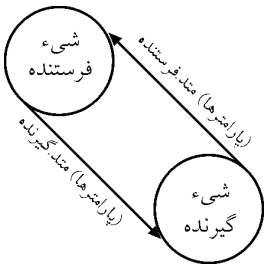
پیام (Message)

راه ایجاد تعاملات پویا مابین اشیاء همکار، ارسال پیام است، در واقع پیام مکانیزمی است که اشیاء به وسیله آن با هم ارتباط برقرار می‌کنند. یک پیام باعث می‌شود رفتاری در شیء گیرنده انجام شود. بنابراین ارسال یک پیام از یک شیء مبدا به شیء مقصد به معنی صدا زدن یکی از متدهای شیء مقصد است. یک پیام شامل شیء مقصد (شیء گیرنده). متد (متدی که در شیء مقصد پیام را دریافت می‌کند) و پارامترهای مربوطه می‌باشد.

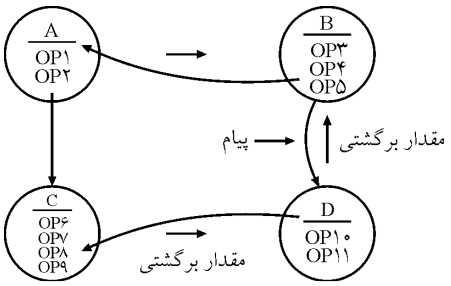
تعامل میان اشیاء در شکل زیر نشان داده شده است، یک عمل در داخل شیء فرستنده پیامی به شکل زیر تولید می‌کند:

(پارامترها) متد شیء مقصد. نام شیء مقصد

که در آن، مقصد، شیء گیرنده‌ای را تعریف می‌کند که توسط پیام صدا زده می‌شود.



مثال: چهار شیء A، B، C و D با مبادله پیام با یکدیگر ارتباط برقرار می‌کنند.



مبادله پیام‌های بین اشیاء

اگر شیء B بخواهد متد OP10 از شیء D را اجرا کند، پیامی به شکل زیر به D ارسال می‌کند:

D.OP10(data)

شیء D نیز به عنوان بخشی از اجرای OP10 ممکن است، پیامی به شکل زیر به C بفرستد:

C.OP8(data)

C عمل OP8 را می‌یابد، آن را اجرا می‌کند و سپس یک مقدار بازگشتی مناسب به D ارسال می‌کند. عمل OP10 کامل می‌شود و مقداری را به B بازمی‌گرداند.

توجه: برای آن که از یک شیء درخواست شود تا یکی از متدهای خود را انجام دهد، باید پیامی به آن داده شود تا معلوم شود چه باید بکند. شیء گیرنده ابتدا با انجام متد درخواست شده و سپس با بازگرداندن کنترل به شیء مبدا، این پیام را پاسخ می‌دهد.

توجه: وقتی با یک مساله برنامه‌نویسی یک زبان شیءگرا مواجه می‌شوید، مانند روش‌های ساخت‌یافته دیگر این پرسش مطرح نیست که چگونه این مساله به چند تابع تقسیم می‌شود بلکه این پرسش مطرح است که چگونه این مساله به چند کلاس تقسیم می‌شود.

توجه: در پیاده‌سازی برنامه‌های شیءگرا، ابتدا کلاس‌های مورد نیاز ایجاد می‌شوند، سپس در طول برنامه و بسته به نیاز، نمونه‌هایی (intance) از این کلاس‌ها ساخته می‌شوند که در واقع همان

اشیاء برنامه را تشکیل می‌دهند. زیرا در عالم برنامه‌های شیءگرا، در نهایت این اشیاء هستند که برای حل مساله، به همکاری و گفتگو با یکدیگر می‌پردازند.

تعریف کلاس و اشیاء در C++

برای تعریف کلاس از کلمه کلیدی class به صورت زیر استفاده می‌شود:

```
class نام کلاس
{
    private:
     داده‌ها و توابع اختصاصی
    public:
     داده‌ها و توابع عمومی
    protected:
      داده‌ها و توابع محافظت شده
; اشیایی از کلاس {
```

توجه: هر عضو کلاس می‌تواند دارای یکی از خصوصیات private، public و protected باشد.

توجه: هر کلاس می‌تواند یک یا چند بخش از private، public و protected باشد.

اعضاء private : این اعضاء فقط از داخل کلاس قابل دسترسی است و از خارج از کلاس قابل دسترسی نمی‌باشد. بیان کلمه private اختیاری است، اعضا کلاس به صورت پیش فرض خصوصی هستند.

اعضاء public : این اعضاء از داخل و خارج از کلاس قابل دسترسی می‌باشد.

اعضاء protected : این اعضاء فقط از داخل کلاس و یا توسط کلاس‌هایی که از این کلاس به طور مستقیم ارث‌بری دارند قابل دسترسی می‌باشد.

توجه: هنگام تعریف یک کلاس، داخل آن کلمات کلیدی private، public و protected را به هر ترتیب و تقدم می‌توان به کار برد.

تعریف اشیاء

به دو روش می‌توان اشیایی از نوع کلاس تعریف کرد:

۱- در انتهای تعریف کلاس، اسامی اشیاء کلاس ذکر شوند.

۲- پس از تعریف کلاس در هر جای برنامه به صورت زیر تعریف شود:

```
; نام شیء    نام کلاس
```

توجه: نامگذاری اشیاء همانند نامگذاری متغیرهاست و برای تعریف چند شیء از یک نوع کلاس باید آن‌ها را با کاما از یکدیگر جدا نمود.

مثال: کد زیر تعریف یک کلاس را نشان می‌دهد.

```
class Ratio {
    private:
     int  num, den;
```

```
  public:
    void assign (int, int);
    void print ( );
  } ;
```

در بلوک این کلاس دو متغییر و دو تابع تعریف شده است. توابع ( ) assign و ( ) print، توابع عضو هستند.

توجه: به توابع عضو، «متد» یا «سرویس» نیز گفته می‌شود.

مثال: برنامه ساده زیر نحوه تعریف کلاس و شیء را به شما نشان می‌دهد:

```
#include <iostream.h>
class k{
    int a;
public:
    void seta (int n );
     int geta ();
};
void k :: seta (int n)
{
    a=n;
}
int k:: geta ()
{
   return a;
}
main()
{
   k ob1,ob2;
   ob1.seta(53);
   ob2.seta(78);
   cout << ob1.geta() << "\n";      // عدد ۵۳ نمایش داده می‌شود
   cout << ob2.geta() << "\n";   // عدد ۷۸ نمایش داده می‌شود
    return 0;
}
```

در کلاس k، صفت a خصوصی و ()seta و ()geta متدهای عمومی هستند. فقط متدهای عضو کلاس k می‌توانند به صفت a دسترسی مستقیم داشته باشند و متدهای خارج از کلاس k نمی‌توانند به آن دسترسی داشته باشند. همچنین از آنجایی که متدهای ()seta و ()geta به عنوان متدهای عمومی کلاس k معرفی شده‌اند، می‌توانند توسط بخش‌های دیگر برنامه که شامل کلاس k

نمی‌باشند، فراخوانده شود. اگر در برنامه فوق، در تابع main دستور ob1.a=53; یـا cout<<ob1.a; نوشته می‌شد خطای کامپایلری صادر می‌شد، چون در تابع main متغیر a شناخته شده نیست. ولـی اگر صفت a در قسمت public تعریف می‌شد، باعث می‌شد صفت a توسط سایر بخش‌های برنامه در دسترس قرار گیرد. بنابراین، دستورات ob1.a=53; یا cout<< ob1.a; در تابع main بـه درسـتی اجرا می‌شدند، ولی اصل پنهان‌سازی اطلاعات نقض می‌شد. زیرا تعریف صفات عمومی در کلاس، ناقض اصل پنهان‌سازی اطلاعات است. به اپراتور نقطه جهت دستیابی به a توجه داشته باشـید، بـه طور کلی اپراتور نقطه، زمان فراخوانی توسط یک تابع عضو یا دستیابی به یک متغیر به کار می‌رود به طوری که ابتدا نام شیء و پس از به کارگیری اپراتور نقطه، نام تابع یا متغیر عضو آورده می‌شود.

به طور کلی هنگام دستیابی به یک صفت یا متد از نام شیء بـه همـراه عملگـر نقطـه اسـتفاده می‌شود. دقت کنید که با تعریف کلاس k هنوز هیچ فضایی در حافظه گرفته نشده است و تنها فرم کلی معرفی شده است. با اجرای دستور k ob1,ob2; در تابع main است که دو شیء متفـاوت از k ساخته می‌شود و در این لحظه برای ob1 و ob2 فضاهایی جداگانه در حافظه کنار گذاشته می‌شود. توجه داشته باشید که صفت a داخل شی ob1 کاملا جدا از صفت a داخل شی ob2 است.

توجه: دسـتورات ob1.seta(53); ،ob2.seta(78); ،ob1.geta(); و ob2.geta(); سـاختار چهـار پیام را نشان می‌دهد که از تابع main به اشیاء مورد نظر در مقصد ارسال می‌شوند.

توجه: به تعریف بدنه متدها بعد از بدنه کلاس مدل outline گفته مـی‌شـود، بـرای مثـال متـد seta() و geta() به صورت outline تعریف شده است. در این حالت برای تعریف بدنه متـد، بایـد اعلام کرد که این متد، متعلق به کدام کلاس است. برای این کار نام کلاس مربوطه باید قبل از نـام متـد بـه همـراه اپراتـور :: (two colon) آورده شـود. نـام علامـت :: در ایـن ترکیـب موسـوم بـه اپراتور scope resolution operator یا عملگر تعیین میدان دید است.

توجه: صفات و متدهایی که داخل کلاس معرفی می‌شوند، در واقع اعضای آن کلاس خوانـده می‌شوند. بنا به تعریف چنانچه صفات و متدهای تعریف شـده در یـک کـلاس صـرفا مخـتص آن کلاس باشند، اصطلاحا به آنها اعضای اختصاصی یا private گفته می‌شود. بدین معنا که آنهـا تنهـا می‌توانند توسط اعضای همان کلاس مورد استفاده قرار گیرند. برای معرفی اعضـای عمـومی یـک کلاس از کلمه کلیدی public که به دنبال علامت : (colon) آورده می‌شود، اسـتفاده مـی‌شـود. بـه همین ترتیب تمامی صفات و متدهایی که بعد از کلمه کلیدی public در کلاس تعریف می‌شـوند، اعضای عمومی تلقی شده و توسط سایر اعضای کلاس و هریک از بخش‌های برنامه قابل دسترسی هستند.

توجه: از آنجا که متدولوژی شیءگرایی به طـور ذاتـی اصـول طراحـی مطلـوب و کارآمـد را رعایت می‌کند، بنابراین، هرگاه از متدولوژی شیءگرایی در خلق برنامه‌هـای کـامپیوتری خـود کـه همان رسیدن به هدف و حل مساله هست استفاده نماییم، در مقصد این متـدولوژی مزایـایی را بـه ارمغان می‌آورد که از ارزش بالایی برخوردار است. مزایایی همچون رعایت «**اصـل پنهـان‌سـازی اطلاعات**»، «**استقلال عملیاتی**»، «**قابلیت استفاده مجدد**»، «**انسجام بالا**» و «**اتصال پایین**».

پنهان‌سازی اطلاعات (information hiding)

در شیءگرایی، پنهان‌سازی اطلاعات، نتیجه پیمانه‌ای‌کردن و بسته‌بندی است. به بیان دیگر شرط لازم برای برقراری پنهان‌سازی اطلاعات، پیمانه‌ای کردن و بسته‌بندی است و شرط کافی برای برقراری پنهان‌سازی اطلاعات تعریف صفات خصوصی و متدهای مرتبط با صفات در ساختار کلاس است. مکانیزم کلاس، مفهوم بسته‌بندی را پیاده‌سازی می‌کند. در یک بیان ساده پنهان‌سازی اطلاعات می‌گوید بخشی از نرم‌افزار در داخل یک پیمانه (کلاس) بسته‌بندی و محصور شود. هدف از پنهان‌سازی اطلاعات، پنهان کردن روال انجام دستورات توابع کلاس‌ها (متدها) و صفات خصوصی کلاس در پس واسط یا بلاک کلاس است. استفاده‌کنندگان پیمانه‌ها (کلاس‌ها) نیازی به دانستن جزئیات داخلی پیمانه‌ها (کلاس‌ها) را ندارند.

در این دیدگاه اشیایی که با یک شیء خاص در ارتباطند، به جزئیات داخلی آن یعنی داده‌ها و متدهای خصوصی یا داخلی، دسترسی نداشته و تنها اجازه دسترسی به داده‌ها و متدهای عمومی یا واسط را خواهند داشت. برای مثال هنگامی که شما برای حل یک مساله با دوستتان به همکاری و گفتگو می‌پردازید فقط می‌توانید به متدهای عمومی دوستتان مانند متد شنیداری پیام ارسال کنید و شما نمی‌توانید به صفات و متدهای خصوصی دوستتان مانند متد پمپاژ خون پیام ارسال کنید.

توجه: در عالم شیءگرایی بهتر است، داده‌ها همواره خصوصی تعریف شوند و اگر قرار بر تغییر مقادیر داده‌های شیء باشد، این تغییرات از طریق متدهای همان شیء اعمال گردد. در واقع در برنامه‌نویسی شیءگرا دسترسی مستقیم به داده‌های یک شیء از طریق اشیاء دیگر وجود ندارد و می‌بایست این کار از طریق فراخوانی متدهای شیء مقصد صورت پذیرد. متدهای یک شیء به دستیابی‌های دیگر اینطور وانمود می‌کنند که داده‌های شیء را تغییر می‌دهم اما آنطور که من می‌گویم، نه آنطور که شما می‌خواهید، برای مثال داده‌های سلول‌های خاکستری همه جانداران هنگام فرآیند مشاهده تغییر می‌کند، اما هر یک بر اساس متد شخصی خودش این دادها را تغییر می‌دهد، و این چنین می‌شود که هر یک از جانداران یک پدیده واحد را به شیوه‌های متفاوت و با رنگ‌های متفاوت می‌بینند.

استقلال عملیاتی (functional independence)

در شیءگرایی، استقلال عملیاتی، نتیجه پیمانه‌ای‌کردن، بسته‌بندی و پنهان‌سازی اطلاعات است. به بیان دیگر شرط لازم برای برقراری استقلال عملیاتی، پیمانه‌ای کردن، بسته‌بندی و پنهان‌سازی اطلاعات است و شرط کافی برای برقراری استقلال عملیاتی، انسجام بالا و اتصال پایین است. مکانیزم کلاس، مفهوم بسته‌بندی را پیاده‌سازی می‌کند. در صورتی که پیمانه‌هایی را با عملکرد تک منظوره (انسجام بالا) و عدم ارتباط بیش از حد با پیمانه‌های دیگر (اتصال پایین) ایجاد کنیم، استقلال عملیاتی تحقق می‌یابد. که این موارد در شیءگرایی به طور ذاتی توسط مکانیزم کلاس، محقق شده است. که این امر منجر به **چابک‌بودن** روند ساخت پروژه‌های شیءگرا می‌شود.

توجه: در مورد مفهوم **چابکی** در فصل متدولوژی‌های چابک صحبت خواهیم کرد.

یک کلاس، خودکفا، قابل انتقال و قابل استفاده مجدد می‌باشد، زیرا هرآنچه لازم دارد همراه

خود دارد. پس از ایجاد، نوشتن و رفع اشکال یک کلاس، می‌توان آن را به برنامه‌نویس‌های دیگر ارائه داد تا در برنامه‌های خود از آن استفاده کنند. به این عمل، قابلیت استفاده مجدد می‌گویند.

در برنامه‌نویسی شیءگرا، ارث‌بری تعمیم قابل توجهی از ایده قابلیت استفاده مجدد است. برنامه‌نویس می‌تواند کلاس موجود را در اختیار بگیرد بدون آنکه مجبور باشد آن را تغییر دهد و نیز می‌تواند ویژگی‌ها و قابلیت‌هایی به آن اضافه کند. این عمل با ایجاد و ارث‌بری کلاس فرزند از کلاس پدر انجام می‌شود. کلاس فرزند، تمام قابلیت‌های عمومی کلاس پدر را به ارث می‌برد اما ممکن است ویژگی‌های جدیدی نیز داشته باشد. برای مثال، ممکن است کلاسی بنویسید (یا آن را از یک برنامه نویس بخرید) که منوی سیستم ایجاد می‌کند که می‌تواند شبیه ویندوز یا واسطه‌های گرافیکی کاربر (GUI) باشد. این کلاس، درست کار می‌کند و لازم نیست آن را تغییر دهید اما می‌خواهید قابلیت‌هایی را نیز به آن اضافه کنید به طوری که تمام قابلیت‌های عمومی منوی موجود را به ارث ببرد. برای این منظور، کافیست یک کلاس فرزند ایجاد کنید به طوری که تمام قابلیت‌های عمومی منوی موجود در کلاس پدر را به ارث ببرد و همچنین قابلیت‌های اضافه شده شما را نیز برآورده سازد. در این باره گفتنی زیاد است که در مورد آنها جلوتر در بخش ارث‌بری بیشتر توضیح می‌دهیم.

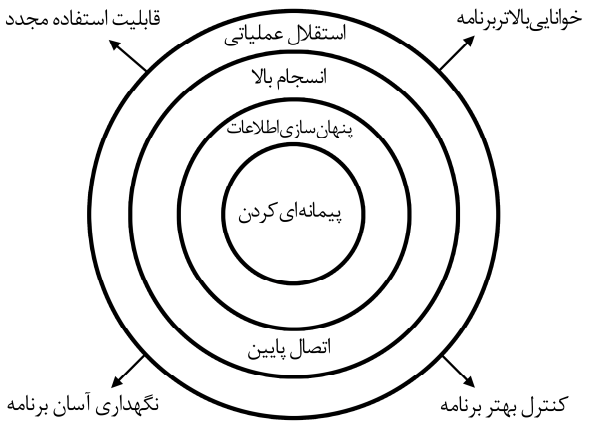
یکی از مزایای اصی و ذاتی برنامه‌نویسی شیءگرا، سهولت استفاده مجدد از بخش‌های مختلف برنامه فعلی برای برنامه‌های آتی است. شرکت‌های متعدد نرم‌افزاری دریافته‌اند که استفاده مجدد از کلاس‌ها در برنامه‌های آتی، برای صرفه‌جویی در وقت و هزینه‌های برنامه‌نویسی آنها حائز اهمیت فراوان است.

توجه: استقلال عملیاتی خوب، کلید طراحی خوب و طراحی خوب، کلید یک نرم‌افزار با کیفیت می‌باشد. استقلال عملیاتی خوب با دو مفهوم انسجام (Cohesion) و اتصال (Coupling) مورد ارزیابی قرار می‌گیرد. انسجام، معیاری است که توان نسبی کارکردی یک پیمانه را نشان می‌دهد و اتصال، معیاری است که میزان نسبی وابستگی پیمانه‌ها به یکدیگر را نشان می‌دهد.

توجه: در یک طراحی ایده‌آل، هدف، محقق کردن بالاترین سطح انسجام (Cohesion بالا) داخل پیمانه‌های برنامه و کمترین سطح اتصال (Coupling پایین) مابین پیمانه‌های برنامه است.

توجه: پیمانه‌ای‌کردن، بسته‌بندی، برقراری پنهان‌سازی اطلاعات، انسجام بالا، اتصال پایین و برقراری استقلال عملیاتی، به عنوان اصول و معیارهای طراحی معماری مطلوب و ایده‌آل یعنی **خوش تعریف** (Well Formed)، منجر به **خوانایی بالاتر برنامه، کنترل بهتر برنامه، قابلیت استفاده مجدد** پیمانه‌های(کلاس‌های) برنامه جاری در برنامه‌های آتی و **نگهداری آسان برنامه** جاری می‌گردد.

شکل زیر گویای مطلب است:



توجه: اصول و معیارهای طراحی معماری مطلوب یعنی پیمانه‌ای‌کردن (modularity)، بسته‌بندی (Encapsulation)، برقراری پنهان‌سازی اطلاعات (information hiding)، انسجام بالا (Cohesion بالا)، اتصال پایین (Coupling پایین) و برقراری استقلال عملیاتی (functional independence)، به طور ذاتی در شیءگرایی برقرار است. بنابراین، پیمانه‌ای‌کردن، بسته‌بندی، برقراری پنهان‌سازی اطلاعات، انسجام بالا، اتصال پایین و برقراری استقلال عملیاتی، به عنوان اصول و معیارهای طراحی معماری مطلوب که منجر به خوانایی بالاتر برنامه، کنترل بهتر برنامه، قابلیت استفاده مجدد (reusability) پیمانه‌های (کلاس‌های) برنامه جاری در برنامه‌های آتی و نگهداری آسان برنامه جاری می‌گردد، به عنوان نتایج استفاده از شیءگرایی محسوب می‌گردند.

انسجام (Cohesion)

انسجام یا همبستگی، توسعه مفهوم پنهان‌سازی اطلاعات است، یک پیمانه یکپارچه و منسجم، یک وظیفه منفرد را انجام می‌دهد. انسجام، یک خصیصه مثبت در طراحی نرم‌افزار بوده که هر چه میزان آن بیشتر باشد، طراحی معماری از کیفیت بالاتری برخوردار است. انسجام در شیءگرایی به معنی درجه وظایف مختلف و غیر مرتبط داخل یک پیمانه یا کلاس است. در یک گروه ارکستر سمفونیک، سازهای مختلفی نواخته می‌شود اما همه مرتبط با هم و حول یک هدف خاص نواخته می‌شوند. در واقع گروه ارکستر سمفونیک از همبستگی و انسجام بالایی برخوردار است. بنابراین درجه‌ی همبستگی این گروه برابر مقدار یک می‌باشد زیرا همه اعضاء حول یک محور و وظیفه کار می‌کنند. هرچه درجه وظایف مختلف و غیرمرتبط یک پیمانه پایین‌تر باشد، پیمانه از انسجام بالاتری برخوردار است. انسجام، به میزان همبستگی میان محتویات داخل یک پیمانه یا کلاس می‌پردازد. به بیان دیگر داخل یک پیمانه یا کلاس چه خبر است. هرچه انسجام پیمانه‌ها بیشتر باشد، طراحی معماری بهتری داریم. انسجام در شیءگرایی را می‌توان در قالب یک طیف از پایین‌ترین سطح انسجام تا بالاترین سطح انسجام طبقه‌بندی نمود. در طراحی معماری، همواره برای بالاترین سطح انسجام تلاش می‌شود. در ادامه سطوح مختلف انسجام در شیءگرایی از بالاترین سطح انسجام (مطلوب‌ترین) تا پایین‌ترین سطح انسجام (نامطلوب‌ترین) رتبه‌بندی شده

است:

۱- انسجام عملیاتی یا کارکردی (Functional Cohesion)

اگر درجه تعداد وظایف یا مسئولیت‌های مختلف یک پیمانه برابر عدد یک باشد، آن پیمانه دارای انسجام عملیاتی است. در این نوع انسجام، پیمانه دقیقا یک مسئولیت را انجام داده و یک هدف واحد را دنبال می‌کند. این نوع انسجام بهترین سطح انسجام را ارائه می‌کند. به این پیمانه‌ها، پیمانه‌های مصمم (single minded module) نیز می‌گویند. در شیءگرایی، یک کلاس به طور ذاتی دارای انسجام عملیاتی می‌باشد، زیرا تمام متدهای یک کلاس حول یک محور، و آن هم، حول محور مسئولیت کلاس مورد نظر مورد استفاده قرار می‌گیرند. مانند کلاس دانشجو که متدهای آن مسئولیت دانشجو بودن را به انجام می‌رسانند.

۲- انسجام لایه‌ای (Layer Cohesion)

این نوع انسجام زمانی رخ می‌دهد که لایه بالاتر (کلاس فرزند یا سطح پایین) به سرویس‌های لایه پایین‌تر (کلاس پدر یا سطح بالا) دسترسی دارد، اما لایه‌های پایین‌تر (کلاس پدر یا سطح بالا) به لایه‌های بالاتر (کلاس فرزند یا سطح پایین) دسترسی ندارد.

در مفهوم وراثت در شیءگرایی، هر زیرکلاس (کلاس فرزند یا سطح پایین) به متدهای ابرکلاس (کلاس پدر یا سطح پایین) دسترسی دارد، در حالی که عکس این دسترسی مجاز نمی‌باشد.

توجه: انسجام لایه‌ای، مزایای زیادی مثل قابلیت استفاده مجدد از متدها یا سرویس‌های کلاس پدر را توسط کلاس فرزند، فراهم می‌آورد.

۳- انسجام ارتباطی (Communication Cohesion)

این نوع انسجام زمانی رخ می‌دهد که تمامی عملیات یک پیمانه به داده‌هایی یکسان و مشترک دسترسی دارند. در شیءگرایی، یک کلاس به طور ذاتی دارای انسجام ارتباطی می‌باشد، زیرا در مکانیزم کلاس در شیءگرایی، که مفهوم بسته‌بندی را پیاده‌سازی می‌کند، همه متدهای کلاس حول محور صفات کلاس به عنوان داده‌هایی یکسان و مشترک، فعالیت می‌کنند. در مفهوم کلاس، تمامی متدهای کلاس بر روی یک ساختمان داده که همان صفات کلاس است، عمل می‌کنند. در این ساختار، داده‌ها یا صفات فقط توسط متدهای همان کلاس دستکاری می‌شوند.

توجه: پیاده‌سازی، تست و نگهداری سیستم‌هایی که انسجام عملیاتی، لایه‌ای و ارتباطی را رعایت می‌کنند، سهل و آسان است، از آنجا که سیستم‌های تولید شده توسط متدولوژی شیءگرایی در مراحل تحلیل و طراحی این اصول رو به طور ذاتی و بنیادی رعایت می‌کنند، بنابراین، می‌توان اینگونه نتیجه گرفت که پیاده‌سازی، تست و نگهداری سیستم‌هایی که توسط متدولوژی شیءگرا خلق می‌شوند، سهل و آسان است. همچنین از این مطالب می‌توان اینگونه استنباط کرد که سیستم‌های تولید شده توسط متدولوژی شیءگرا، ذاتی چابک و سریع دارند.

توجه: در مورد مفهوم **چابکی** در فصل متدولوژی‌های چابک صحبت خواهیم کرد.

توجه: به طورکلی، تمامی انسجام‌های شی‌گراء، شامل انسجام عملیاتی یا کارکردی، انسجام لایه‌ای و انسجام ارتباطی از کلیه‌ی انسجام‌های ساخت‌یافته، در سطح بالاتری(مطلوب‌تری) قرار دارند، زیرا مولفه‌های شی‌گراء یعنی کلاس‌ها، به طور ذاتی از طراحی ایده‌آل و مطلوب برخوردار هستند، انسجام‌های ساخت‌یافته، در فصل مفاهیم طراحی ساخت‌یافته تشریح شدند.

اتصال (Coupling)

کاهش وابستگی میان پیمانه‌ها یا کلاس‌ها، یک خصیصه مثبت در طراحی نرم‌افزار بوده که هر چه میزان آن کمتر باشد، طراحی معماری از کیفیت بالاتری برخوردار است. اتصال در شیءگرایی به معنی میزان اتصال ما بین پیمانه‌ها یا کلاس‌ها است. به بیان دیگر میان پیمانه‌ها یا کلاس‌ها چه خبر است. میزان اتصال، به پیچیدگی نقطه اتصال دو پیمانه یا کلاس برای تبادل اطلاعات بستگی دارد. هر چه میزان اتصال مابین پیمانه‌ها یا کلاس‌ها پایین‌تر باشد، طراحی معماری بهتری داریم. به بیان دیگر هرچه پیمانه‌ها یا کلاس‌ها برای تحقق اهدافشان، نیازمند ارتباط کمتری با دیگر پیمانه‌ها یا کلاس‌ها باشند، اتصال آن برنامه کمتر است. اگرچه ارتباط میان پیمانه‌ها یا کلاس‌ها برای مجتمع کردن برنامه لازم است، اما باید تا جای ممکن بر کاهش اتصال میان پیمانه‌ها یا کلاس‌ها تاکید کرد. ارتباط ساده میان پیمانه‌ها یا کلاس‌ها به برنامه‌ای منجر می‌شود که درک آن آسان‌تر بوده و کمتر در معرض خطای منتشرشونده قرار می‌گیرد، ناشی از رخ دادن یک خطا در یک مکان و انتشار آن به نقاط دیگر برنامه. اتصال را می‌توان در قالب یک طیف از کمترین سطح اتصال تا بیشترین سطح اتصال طبقه‌بندی نمود. در طراحی معماری همواره برای کمترین سطح اتصال تلاش می‌شود. در ادامه سطوح مختلف اتصال در شیءگرایی از کمترین سطح اتصال (مطلوب‌ترین) تا بیشترین سطح اتصال (نامطلوب‌ترین) رتبه‌بندی شده است:

۱- بدون اتصال (No Coupling)

هنگامی که هیچ نوع ارتباطی میان دو پیمانه یا کلاس وجود ندارد.

۲- اتصال انجمنی

هر گاه مابین دو پیمانه یا کلاس را**بطه انجمنی** مطرح باشد، یعنی تبادل پیام مابین برخی متدهای دو کلاس صورت گیرد. مانند ارسال پیام از شیء بازیکن به شیء توپ پس از انجام متد شوت زدن مربوط به شیء بازیکن، برای تغییر مختصات توپ توسط صدا زدن متد تغییر مختصات توپ. یعنی شیء بازیکن، باید در هنگام شوت زدن، مختصات توپ را توسط صدا زدن متد تغییر مختصات توپ، تغییر دهد.

۳- اتصال محتوایی

اتصال محتوایی، در بالاترین سطح اتصال یعنی بدترین نوع اتصال، قرار دارد. در شیءگرایی اتصال محتوایی زمانی رخ می‌دهد که یک پیمانه یا کلاس توانایی این را داشته باشد که داده‌های داخلی پیمانه یا کلاس دیگر را تغییر دهد. اگر کلاسی صفات داخلی خود را، عمومی تعریف کند که در این صورت آن کلاس ناقض اصل پنهان‌سازی اطلاعات نیز شده است، در این شرایط این

امکان برای پیمانه‌ها و کلاس‌های دیگر فراهم می‌گردد که بدون دخالت متدهای کلاس مورد نظر، داده‌های آن را تغییر دهند و این یعنی اتصال محتوایی میان کلاس‌ها.

مثال:

```
#include <iostream.h>
class k{
    int a;
public:
     void seta (int n );
     int geta ();
};
void k :: seta (int n)
{
    a=n;
}
int k:: geta ()
{
   return a;
}
main()
{
   k ob1,ob2;
   ob1.a=53;
   ob2.a=78;
    cout <<ob1.a << "\n";      // عدد ۵۳ نمایش داده می‌شود
    cout <<ob2.a << "\n";   // عدد ۷۸ نمایش داده می‌شود
     return 0;
}
```

توجه: دستورات ob1.a=53; ،ob2.a=78; ،cout<< ob1.a; و cout<< ob2.a; در تابع main به دلیل public بودن صفت a، امکان دسترسی مستقیم از پیمانه main به صفت a در کلاس k را دارند، در اینجا به دلیل نقض اصل پنهان‌سازی اطلاعات، اتصال محتوایی میان پیمانه main و کلاس k، ایجاد گردیده است.

توجه: اتصال محتوایی، بیشترین سطح اتصال و نامطلوب‌ترین است. «**اتصال محتوایی بدترین نوع اتصال و بیشترین میزان در نقض پنهان‌سازی اطلاعات را دارا است.**» تعریف صفت عمومی در کلاس با میزان کمتری نسبت به اتصال محتوایی می‌تواند در شرایط خاص ناقض اصل پنهان‌سازی اطلاعات باشد.

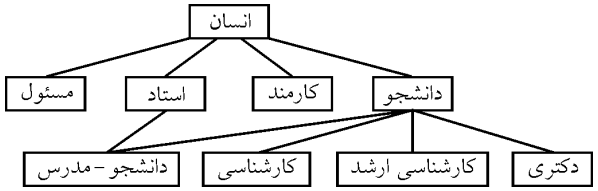
ب) وراثت (Inheritance)

وراثت فرآیندی است که به وسیله آن یک کلاس فرزند می‌تواند صفات و متدهای عمومی کلاس پدر را کسب کند. به عبارت کلی‌تر یک کلاس فرزند ضمن به ارث بردن مجموعه‌ای از صفات و متدهای عمومی کلاس پدر، می‌تواند ویژگی‌های خاص و مختص خود را نیز به آنها اضافه کند. به این ترتیب، سلسله مراتبی از کلاس‌ها (Class Hierarchy) ایجاد می‌شود که قدرت انعطاف‌پذیری زیادی به برنامه‌ساز می‌بخشد. سلسله مراتب کلاس‌ها به صورت درخت است. به عبارت دیگر مدل ریاضی وراثت، درخت است. در این درخت، کلاس‌ها به صورت «ابرکلاس» (Super Class) و «زیرکلاس» (Sub Class) در رده‌های مختلف قرار می‌گیرند.

در بالاترین سطح، کلاسی قرار دارد که خصوصیات مشترک همه کلاس‌های موردنظر را در خود جای می‌دهد. این کلاس از هیچ کلاس دیگری مشتق نمی‌شود (ارث‌بری ندارد) برگ‌های این درخت را کلاس‌هایی تشکیل می‌دهند که هیچ کلاسی از آنها مشتق نمی‌شود. در سطح‌های میانی، کلاس‌هایی قرار می‌گیرند که هم‌ارث می‌برند و هم کلاس‌های دیگر از آنها مشتق می‌شوند.

هر زیرکلاس، داده‌ها و متدهای کلاس‌های رده بالاتر را به ارث می‌برد و نیازهای ویژه خود را به آن می‌افزاید. کلاس ممکن است بعضی از صفات و متدهای کلاس پدر را به ارث نبرد (صفات و متدهای خصوصی) و یا آنها را تغییر دهد(دوباره‌نویسی کند). شکل زیر سلسله مراتب ساده‌ای از کلاس‌های «**دانشگاهیان**» را نشان می‌دهد. خصوصیات کلی همه انسان‌ها که در این کاربرد خاص مطرح می‌شود. در کلاس انسان می‌آید و ویژگی‌های هر کدام (مثل نمره دانشجو، تخصص استاد، نوع و محل مسئولیت برای مسئول و اضافه کاری برای کارمند) در کلاس مربوط به آنها ذکر می‌شود.

کلاس (دانشجو- مدرس) حالت خاصی دارد. این کلاس مربوط به افرادی است که هم درس می‌خوانند و هم درس می‌دهند. این گونه افراد هم از کلاس دانشجو ارث می‌برند، هم از کلاس استاد. این نوع ارث‌بری را «**ارث‌بری چندگانه**» (Multiple Inheritance) می‌نامند.



سلسله مراتب ساده‌ای از کلاس‌های «دانشگاهیان»

زمانی که نیاز به ایجاد یک کلاس جدید باشد مهندس نرم‌افزار چند انتخاب دارد:

۱) کلاس جدید بدون استفاده از وراثت از ابتدا طراحی و ساخته شود.

۲) سلسله مراتب کلاس‌ها بررسی شود تا یک کلاس بالاتر که حاوی اکثر صفات و متدهای موردنیاز است پیدا شود. سپس کلاس جدید از کلاس بالاتر ارث برده و باقی صفات و متدهای موردنظر به آن اضافه گردد.

توجه: وراثت در شیءگرایی، نه تنها، برای مفهوم ساختار سلسله مراتبی کلاس‌ها به کار می‌رود، بلکه برای ایجاد نوعی پلی‌مورفیسم (چندشکلی) به نام overriding (دوباره نویسی) که یکی دیگر از ویژگی‌های اساسی زبان شیءگرا است، مورد استفاده قرار می‌گیرد. این موضوع در ادامه بررسی می‌شود.

نحوه ارث بری در زبان C++ به صورت زیر است:

```
class      نحوه ارث‌بری: نام کلاس مشتق شده      نام کلاس مبنا
{
.
.
.
}
```

توجه: نحوه ارث‌بری می‌تواند public، private و یا protected باشد، چنانچه ذکر نشود، private در نظر گرفته می‌شود.

نحوه ارث‌بری public (عمومی)

اگر نحوه ارث‌بری public باشد قواعد زیر را خواهیم داشت:

۱- کلیه اعضاء عمومی کلاس مبنا، در کلاس مشتق شده نیز اعضاء عمومی خواهند بود.

۲- کلیه اعضاء خصوصی کلاس مبنا، توسط کلاس مشتق شده غیر قابل دسترسی خواهند بود.

۳- کلیه اعضاء حفاظت شده کلاس مبنا، در کلاس مشتق شده نیز اعضاء حفاظت شده خواهند بود.

نحوه ارث‌بری private (خصوصی)

اگر نحوه ارث‌بری private باشد قواعد زیر را خواهیم داشت:

۱- کلیه اعضاء عمومی کلاس مبنا، در کلاس مشتق شده اعضاء خصوصی خواهند بود.

۲- کلیه اعضاء خصوصی کلاس مبنا، توسط کلاس مشتق شده غیر قابل دسترسی خواهند بود.

۳- کلیه اعضاء حفاظت شده کلاس مبنا، در کلاس مشتق شده اعضاء خصوصی خواهند بود.

نحوه ارث‌بری protected (حفاظت شده)

اگر نحوه ارث‌بری protected باشد قواعد زیر را خواهیم داشت:

۱- کلیه اعضاء عمومی کلاس مبنا، در کلاس مشتق شده اعضاء محافظت شده خواهند بود.

۲- کلیه اعضاء خصوصی کلاس مبنا، توسط کلاس مشتق شده غیر قابل دسترسی خواهند بود.

۳- کلیه اعضاء حفاظت شده کلاس مبنا، در کلاس مشتق شده اعضاء حفاظت شده خواهند بود.

مثال: در برنامه زیر به کلاس B، کلاس مبنا (base class) یا ابرکلاس (super class) یا کلاس پدر و به کلاس D که کلاس B را به ارث می‌برد، کلاس مشتق شده (derived class) یا زیرکلاس (sub class) یا کلاس فرزند گفته می‌شود. عملیات ارث‌بری با تعریف کلاس مبنا شروع می‌شود. کلاس مبنا

بیان‌کننده توصیفاتی کلی از مجموعه خصوصیات است. پس از آن یک کلاس مشتق شده این خصوصیات کلی را به ارث برده و خواصی که ویژه خودش می‌باشد را به آن اضافه می‌کند.

```
#include <iostream.h>
class B{
    int i;
public:
    void set_i (int n ){ i= n;}
     int get_i (){return i;}

};
```

نحوه تعریف کلاس مشتق شده//

```
class  D: public B{
    int j;
public:
    void set_j(int n) {j=n;}
    int mul();
};
int D:: mul()
{
   return j*get_i();
}
main()
{
   D ob;
    ob.st_i(8);   // عدد ۸ را در متغیر i می‌ریزد
   ob.st_j(3);    //عدد ۳ را در متغیر j می‌ریزد
   cout << ob.mul();    //عدد ۲۴ را نشان می‌دهد
   return 0;
}
```

به نحوه تعریف کلاس D که از کلاس B ارث می‌برد، توجه کنید:

```
class  D: public B{
    int j;
  public:
void set_j(int n) {j=n;}
int mul();
};
```

با دستورات فوق، تمام **اعضای عمومی** کلاس B یعنی متدهای set_i و get_i جزو **اعضای**

عمومی کلاس D می‌شوند (توسط D به ارث برده می‌شوند) ولی عضو خصوصی i همچنان از نظر کلاس D مخفی بوده و تنها توسط توابع set_i و get_i قابل دسترسی می‌باشد.

در واقع کلاس D تمام اعضای عمومی کلاس B را به ارث برده و به آنها اعضای set_j و mul() و j را اضافه می‌کند. با توجه به تعریف متد mul() مشاهده می‌شود این متد، متد get_i() را که عضو کلاس B است، فراخوانده است. این امکان به لحاظ آنکه اعضای عمومی کلاس B به عنوان اعضای عمومی کلاس D تلقی می‌شوند، امکان‌پذیر می‌باشد. دقت کنید که اگر در تابع D::mul() به جای دستور return j*get_i(); دستور return j*i; نوشته می‌شد، خطای کامپایلری رخ می‌داد، چرا که تابع mul() عنصر خصوصی i که متعلق به کلاس مبنای B است، را نمی‌شناسد.

توجه: اگر خط class D: public B به class D: private B تغییر کند، آنگاه تمام **اعضای عمومی** کلاس B یعنی متدهای set_i و get_i جزو **اعضای خصوصی** کلاس D می‌شوند (توسط D به ارث برده می‌شوند) ولی عضو خصوصی i همچنان از نظر کلاس D مخفی بوده و تنها توسط توابع set_i و get_i قابل دسترسی می‌باشد. و همچنان اگر در تابع D::mul() به جای دستور return j*get_i(); دستور return j*i; نوشته می‌شد، خطای کامپایلری رخ می‌داد، چرا که تابع mul() عنصر خصوصی i که متعلق به کلاس مبنای B است، را نمی‌شناسد.

توجه: اگر خط class D: public B به class D: protected B تغییر کند، آنگاه تمام **اعضای عمومی** کلاس B یعنی متدهای set_i و get_i جزو **اعضای محافظت شده** کلاس D می‌شوند (توسط D به ارث برده می‌شوند) ولی عضو خصوصی i همچنان از نظر کلاس D مخفی بوده و تنها توسط توابع set_i و get_i قابل دسترسی می‌باشد. و همچنان اگر در تابع D::mul() به جای دستور return j*get_i(); دستور return j*i; نوشته می‌شد، خطای کامپایلری رخ می‌داد، چرا که تابع mul() عنصر خصوصی i که متعلق به کلاس مبنای B است، را نمی‌شناسد.

توجه: به تعریف بدنه متدها جلوی خود متدها مدل inline گفته می‌شود. و به تعریف بدنه متدها بعد از بدنه کلاس مدل outline گفته می‌شود، برای مثال متد set_j(int n) {j=n;} به صورت inline و متد mul() به صورت outline تعریف شده است.

۳- چندریختی یا پلی‌مورفیسم (polymorphism)

پلی‌مورفیسم (که در زبان یونانی به معنی «many forms» است) کیفیتی است که این امکان را به وجود می‌آورد تا یک نام، برای یک، دو، یا چندین تکنیک با منظورهای متفاوت ولی در یک راستا و در ارتباط با یک مفهوم، مورد استفاده قرار گیرد.

چندریختی به دو صورت **سربارگذاری (overloading)** و **دوباره‌نویسی (overriding)** وجود دارد.

سربارگذاری (overloading)

سربارگذاری به دو نوع **سربارگذاری تابع** و **سربارگذاری عملگر** طبقه‌بندی می‌گردد.

سربارگذاری تابع

پلی‌مورفیسم (سر بارگذاری تابع) آن گونه که در برنامه‌نویسی شیءگرا به کار گرفته شده است، این امکان را فراهم می‌کند تا بتوان برای مشخص کردن یک تابع عمومی حاوی عملیات مختلف، تنها از یک نام استفاده کرد. در این تابع عمومی نحوه انتخاب هر یک از عملیات مورد نظر صرفاً به وسیله نوع داده‌ای که مورد استفاده قرار می‌گیرد، تعیین می‌شود.

برای مثال در زبان C که در آن پلی‌مورفیسم مورد تاکید و اهمیت قرار نگرفته است، برای محاسبه قدرمطلق یک مقدار، سه تابع ()abs، ()labs و ()fabs وجود دارد. این توابع به ترتیب مقدار قدر مطلق مقادیر عدد صحیح، عدد صحیح از نوع long و ممیز شناور را بر می‌گردانند. در حالی که در C++ به لحاظ تاکید روی پلی‌مورفیسم هر یک از توابع فوق می‌توانند با نام ()abs مورد استفاده قرارگیرند، نوع داده‌ای که به عنوان آرگومان تابع ()abs مورد استفاده قرار می‌گیرد، گونه خاصی از تابع را انتخاب می‌کند (نحوه عمل در ادامه شرح داده می‌شود). همانگونه که در C++ خواهید دید این امکان فراهم شده است تا بتوان از یک نام برای چندین منظور مختلف استفاده کرد. این عمل **سربارگذاری تابع** نامیده شده است.

به بیان کلی‌تر مفهوم پلی مورفیسم همان ایده «**one interface, multiple methods**» است. بدین ترتیب امکان طراحی یک واسط کاربر کلی (generic interface) برای گروهی از فعالیت‌های مرتبط فراهم می‌شود، در حالی که انتخاب نحوه اجرای هریک از فعالیت‌ها صرفاً بستگی به نوع داده مورد استفاده خواهد داشت.

یکی از مزیت‌های پلی‌مورفیسم، کمک در کاهش پیچیدگی برنامه است، این عمل با استفاده از بکارگیری یک واسط کاربر یکسان برای مجموعه‌ای از عملیات تحت یک نام عمومی فراهم شده است. انتخاب عمل مناسب در شرایط مختلف وظیفه‌ای است که به عهده کامپایلر گذاشته شده است. برنامه‌نویس نیازی به انتخاب تابع مورد نظر خود نخواهد داشت و تنها نیاز او چگونگی به کارگیری واسط کاربر کلی در برنامه است. همانگونه که در مثال‌های بعدی مشاهده خواهید کرد. داشتن سه اسم برای تابع قدر مطلق به جای تنها یک نام که جنبه کلی مورد نظر را می‌رساند، کار را تا حد زیادی پیچیده خواهد کرد.

در C++ دو یا چند تابع می‌توانند مادامی که نوع داده‌ها یا تعداد آرگومان‌های آنها یا هر دو متفاوت باشند، به‌طور مشترک از یک نام استفاده کنند. بدین ترتیب هرگاه توابعی هم‌نام دارای شرایط فوق باشند، اصطلاحا به آنها **توابع سربارگذاری شده (overloaded function)** و به این فرآیند، **سربارگذاری توابع (function overloading)** گفته می‌شود. استفاده از چنین امکاناتی پیچیدگی برنامه را کاهش می‌دهد، زیرا عملیات مشابه و مربوط، تنها با یک نام فراخوانده خواهند شد.

نحوه سربارگذاری توابع بسیار ساده است، تنها کافی است که هر یک از گونه‌های مختلف تابع را به طور جداگانه معرفی و تعریف کنید، این وظیفه کامپایلر است تا به طور اتوماتیک گونه مناسب تابع از نظر تعداد و نوع آرگومان‌ها را انتخاب، فراخوانی و مورد استفاده قرار دهد.

توجه: سربارگذاری توابع، یکی از روش‌های پیاده‌سازی پلی مورفیسم است. سربارگذاری توابع با توجه به نوع و تعداد آرگومان‌ها صورت می‌گیرد.

مثال: در برنامه‌نویسی C بسیاری از توابع هم‌شکل و مربوط، تنها در نوع داده‌هایشان متفاوتند که موارد متعددی از آنها در کتابخانه استاندارد C یافت می‌شود. برای مثال در توابع abs()، labs() و fabs() مقدار برگشتی به ترتیب قدرمطلق یک عدد صحیح معمولی، صحیح بلند و ممیز شناور است و استفاده از سه نام مختلف، تنها به لحاظ تفاوت نوع داده‌های آنهاست. در C++ این مشکل را می‌توان با معرفی یک نام مشترک برای هر سه تابع با بکارگیری قابلیت سربارگذاری توابع همانگونه که در مثال زیر آورده شده است، حل کرد.

```
#include <iostream.h>
int abs(int n);
long abs(long n);
double abs(double n);
main()
{
abs(-10);
abs(-10L);
abs(-10.01);
return 0;
}
int abs(int n)
{
cout<< n<0 ? –n : n; << "\n";
}
long abs(long n)
{
cout<< n<0 ? –n : n; << "\n";
}
double abs(double n)
{
cout<< n<0 ? –n : n; << "\n";
}
```

همانگونه که مشاهده می‌کنید، این برنامه سه تابع مختلف را با یک نام مشترک یعنی abs()، تعریف کرده است، در صورتی که هریک برای نوع داده مشخصی به کار می‌رود. در داخل main() فراخوانی تابع abs() برای سه نوع آرگومان انجام می‌پذیرد و کامپایلر به طور اتوماتیک گونه مناسب abs() را بر حسب نوع داده آرگومان استفاده شده، انتخاب و فرا می‌خواند.

اگرچه مثال ارائه شده بسیار ساده است، اما به‌خوبی اهمیت سربارگذاری توابع را نشان می‌دهد. زیرا فقط با یک نام می‌توان به مجموعه‌ای از فعالیت‌های مشابه دست یافت و پیچیدگی ساختگی برای سه نامی که به هم نزدیک هستند، یعنی abs()، fabs() و labs()، حذف شده و تنها کافی است یک نام را به خاطر داشته باشید. نامی که نشان‌دهنده رفتار و عمل عمومی تابع است و زحمت انتخاب گونه مناسب به عهده کامپایلر خواهد بود (در برنامه‌نویسی شیءگرا به تابع انتخابی، متد گفته می‌شود) و از این رو به طور موثری پیچیدگی برنامه کاهش خواهد یافت. بدین ترتیب با استفاده از خاصیت چندشکلی یا پلی‌مورفیسم، سه نام به یک نام کاهش یافته است. علی رغم این که استفاده از پلی‌مورفیسم در این مثال ساده به نظر می‌رسد، اما نشان‌دهنده آن است که در برنامه‌های بسیار بزرگ نگرش «**یک واسط کاربر برای چندین متد**» می‌تواند به طور موثری مورد استفاده قرار گیرد.

مثال:

```
#include <iostream.h>
void f(int i);
void f(int i, int j);
void f(double k);
main()
{
  f(10);  //تابع f(int i) را صدا می‌زند
  f(10 , 20);  //تابع f(int i , int j) را صدا می‌زند
  f(23.67);  //تابع f(double k) را صدا می‌زند
return 0;
}
void f(int i)
{
   cout<<  i+3 << "\n";
}
void f(int i , int j)
{
    cout<< i+j  << "\n";
}

void f(double k)
{
    cout<< k   << "\n";
}
```

خروجی به صورت زیر است:

```
13
30
23.67
```

در مثال فوق تابع f() سه بار سربارگذاری شده است. کامپایلر با توجه به تعداد پارامترها و یا نوع پارامتر ورودی به راحتی تشخیص می‌دهد که کدام تابع f() را در هر بار بایستی صدا بزند.

مثال: محاسبه مساحت مستطیل و مربع به کمک سربارگذاری توابع.

```
#include <iostream.h>
double area(double length);
double area(double length , double width);
main()
{
 area(10); // محاسبه مساحت مربع area(double length) را صدا می‌زند
 area(10,20); // محاسبه مساحت مستطیل area(double length , double width) را صدا می‌زند
return 0;
}
double area(double length)
{
   cout<< length * length ; << "\n";
}
double area(double length , double width)
{
   cout<< length * width ; << "\n";
}
```

توجه: اگر سربارگذاری نبود برنامه‌نویس باید نام دو تابع مختلف برای محاسبه مساحت مربع و مستطیل را به خاطر می‌سپرد ولی حالا تنها لازم است نام یک تابع را در ذهن داشته باشد و این نوشتن برنامه‌های بزرگ و پیچیده را ساده‌تر می‌کند. سربارگذاری با توجه به نوع و تعداد آرگومان‌ها صورت می‌گیرد.

سربارگذاری عملگر

تمامی زبان‌های برنامه‌نویسی در مورد عملگرهای حسابی، به طور مجازی کاربردهای محدودی از پلی‌مورفیسم را به کار بسته‌اند. برای مثال در C علامت + برای جمع مقادیر عدد صحیح معمولی، صحیح بلند، حروف، رشته و اعداد ممیز شناور، مورد استفاده قرار می‌گیرد. در این حالت‌ها کامپایلر به طور اتوماتیک می‌داند کدام یک از انواع عملیات حسابی را اجرا کند.

در C++ این مفهوم برای انواع داده‌هایی که توسط برنامه‌نویس تعریف می‌شود، بسط یافته و این نوع پلی‌مورفیسم موسوم به سربارگذاری اپراتور است.

دوباره نویسی (overriding)

پلی‌مورفیسم به دو طریق ایجاد می‌گردد، یکی در زمان کامپایل‌کردن به همان شکلی که در

سربارگذاری توابع و عملگرها مورد استفاده قرار می‌گیرند و راه دیگر با استفاده از توابع مجازی که پیاده‌سازی پلی‌مورفیسم را در هنگام اجرای برنامه (نه زمان کامپایل) فراهم می‌سازد.

یک تابع مجازی تابعی است که درون یک کلاس پایه معرفی شده و مجددا توسط کلاسی که از آن کلاس پایه مشتق خواهد شد، بازتعریف (دوباره‌نویسی) می‌شود. برای ایجاد یک تابع مجازی در ابتدای معرفی آن کلمه کلیدی virtual قرار داده می‌شود. وقتی کلاس پایه‌ای که شامل یک تابع مجازی است به ارث برده می‌شود، کلاس مشتق شده از آن کلاس پایه این تابع مجازی را با توجه به نیازهای خود مجددا بازتعریف می‌کند. مانند فرزندی که عملکردی به ارث برده از پدر خود را تغییر می‌دهد، در این شرایط عملکرد پدر و برادر تغییر نمی‌کند. روال کار برای پیاده‌سازی تابع مجازی، دسترسی توسط اشاره‌گرها به نام تابع مجازی موجود در کلاس پایه (نام عملکرد مورد نظر در کلاس پدر)، سپس نادیده گرفتن آن، با استفاده از کلمه virtual و در انتها بازتعریف تابع مجازی (بازتعریف عملکرد پدر) داخل کلاس فرزند می‌باشد.

از اشاره‌گری که به عنوان اشاره‌گر به یک کلاس پایه معرفی شده باشد، می‌توان برای اشاره به هر کلاسی که از آن کلاس پایه مشتق شده است نیز استفاده کرد. برای مثال اگر فرض کنید derived کلاس مشتق شده از کلاس پایه base باشد، دستورات زیر صحیح است:

```
base  *p;
base  x;
derived  y;
p=&x;        // p به شیء پایه x اشاره می‌کند که عملی درست است.
p=&y;        // p به شیء مشتق شده y اشاره می‌کند که عملی درست است.
```

توجه: هر چند می‌توان از یک اشاره‌گر پایه جهت اشاره به یک شیء مشتق شده استفاده کرد، اما از این طریق فقط به آن اعضایی از این شیء مشتق شده می‌توان دسترسی داشت که از آن کلاس پایه به ارث برده شده باشند. به بیان دیگر اشاره‌گر به یک کلاس پایه می‌تواند به هر شیء از کلاس مشتق شده از آن کلاس، بدون اینکه خطای مربوط به عدم تطبیق داده‌ها پیش آید، اشاره کند. اما باید توجه داشت که امکان استفاده از اشاره‌گر کلاس پایه برای اشاره کردن به شیء مشتق شده از آن، تنها می‌تواند برای اعضایی از شیء مشتق شده که از کلاس پایه به ارث رسیده‌اند، وجود داشته باشد. زیرا دانش اشاره‌گر پایه تنها منحصرا در چارچوب کلاس پایه و اجزای تعریف شده در آن است و این اشاره‌گر هیچ گونه اطلاعاتی راجع به اعضای تعریف شده در کلاس مشتق شده ندارد. که این موضوع راه دسترسی به تابع مجازی موجود در کلاس پایه را مهیا می‌کند.

توجه: هرگاه اشاره‌گر کلاس پایه، به یک شیء از کلاس مشتق شده اشاره کند، چنانچه کلاس مشتق شده، دارای تابع بازتعریف شده باشد، و فراخوانی تابع مجازی، توسط اشاره‌گر مذکور انجام پذیرد (اشاره‌گر به توابع به ارث رسیده در کلاس مشتق شده دسترسی دارد)، تابع بازتعریف شده مورد استفاده قرار می‌گیرد، یعنی شکل جدید و بازتعریف شده آن و نه شکل قدیم آن. به بیان دیگر، در زمان اجرای برنامه، بر حسب اینکه اشاره‌گر به چه نوع شیءای اشاره می‌کند، تابع بازتعریف شده مربوط به آن شیء انتخاب و مورد استفاده قرار می‌گیرد. بنابراین چنانچه کلاس پایه

دارای تابع مجازی بوده و بیش از یک کلاس از آن مشتق شود، اشیاء مختلفی از نـوع کـلاس‌هـای مشتق شده، می‌توانند توسط اشاره‌گر پایه، به توابع بازتعریف شده خود دسترسـی پیـدا کننـد. ایـن فرآیند راهی است که به کمک آن می‌توان به قابلیت پلی مورفیسم در زمان اجرا دست یافت.

توجه: اگر دو یا چند کلاس مختلف همگی از کلاس پایه واحدی مشتق شوند کـه دارای یـک تابع مجازی است در آن صورت وقتی یک اشاره‌گر پایه به اشیا مختلفی از این کلاس اشاره می‌کند با توجه به هر کدام از این کلاس‌ها، نسخه‌های متفاوتی از آن تابع مجازی به اجرا در خواهد آمـد. این فرآیند همان نحوه به دست آمدن **پلی‌مورفیسم در زمان اجرا** است.

مثال:

```
#include <iostream.h>
class base{
 public:
     virtual void func()
    {
     cout <<"Base version of func()"<<"\n";
     }
};
class derived1: public base{
   public:
   void func()
   {
    cout <<"Derived1 version of func()" << "\n";
    }
};
class derived2 : public base{
    public:
   void func()
   {
    cout <<"Derived2 version of func()" << "\n";
    }

main()
{
base  *p;
base   a;
derived1  b;
derived2  c;
```

```
p=&a;
 p→func();      //نسخه ()fun در کلاس base صدا زده می‌شود
p=&b;
 p→func();      //نسخه ()fun در کلاس derived1 صدا زده می‌شود
p=&c;
 p→func();      //نسخه ()fun در کلاس derived2 صدا زده می‌شود
return 0;
}
```

خروجی برنامه فوق به صورت زیر است:

```
Base version of func()
Derived1 version of func()
Derived2 version of func()
```

توجه: همانگونه که در مثال فوق مشاهده کردید، کلاس base با تابع مجازی ()func، و دو کلاس مشتق شده derived1 و derived2 که هر دو تابع ()func را **override** کرده‌اند با پیاده‌سازی‌های مستقلی معرفی شده‌اند.

توجه: ایجاد خاصیت «**بازنویسی**» در کلاس فرزند باعث می‌شود ویژگی وراثت در کلاس‌ها از نوع رابطه‌ای نباشد، یعنی نمی‌توان به طور قطع گفت که کلاس فرزند همواره دارای متدهای کلاس پدر است. زیرا بعضی متدهای کلاس پدر، ممکن است توسط کلاس فرزند بازنویسی (بازتعریف یا دوباره‌نویسی) شوند.

توجه: یک تابع بازتعریف شده، باید دقیقا دارای همان تعداد پارامتر، از همان نوع و همان مقدار برگشتی باشد. به بیان دیگر، تابع مجازی تعریف شده در کلاس پایه فرم رابط کاربر آن تابع را معرفی می‌کند و هر یک از توابع بازتعریف شده، در کلاس‌های مشتق شده، نحوه پیاده‌سازی آن تابع در هریک از کلاس‌های مشتق شده را تعیین می‌کنند.

توجه: اگر یک کلاس مشتق شده تابع مجازی خود را override نکند، برای آن از همان تابع تعریف شده در نسخه کلاس پایه استفاده خواهد شد. برای مثال اگر در برنامه فوق، در کلاس derived2 تعریف تابع ()fun کامل حذف شود، آنگاه خروجی به صورت زیر می‌شد:

```
Base version of func()
Derived1 version of func()
Base version of func()
```

بنابراین در این حالت یک ابرکلاس وجود دارد که سایر کلاس‌ها از آن ارث می‌برند. ابرکلاس دارای سرویس‌های پایه‌ای است که کلاس‌های فرزند برخی از آنها را بازتعریف می‌کنند و به این عمل سربارگذاری توابع گفته می‌شود.

مثال: در برنامه زیر یک نمونه کاربردی از توابع مجازی، مورد استفاده قرار گرفته است، در این برنامه یک کلاس پایه کلی موسوم به area ایجاد شده است، که دو بعد هر شکلی را در خود نگه داشته

است. علاوه بر آن در این کلاس تابعی مجازی موسوم به **getarea()** تعریف شده است که وقتی توسط کلاس‌های مشتق شده از این کلاس پایه بازتعریف یا **override** می‌شود، مساحت شکل تعریف شده از سوی این کلاس‌های مشتق شده را بر می‌گرداند. در برنامه زیر مساحت یک مثلث و یک مستطیل محاسبه شده است:

```
#include <iostream.h>
class area{
    float dim1,dim2;
public:
    void setarea(float d1, float d2)
    {
      dim1=d1;
      dim2=d2;
    }
    void  getdim( folat &d1, float &d2)
     {
       d1=dim1;
       d2=dim2;
    }
    virtual   float   getarea()
    {
    cout << "you must override this function\n";
     return 0;
     }
};
class  rectangle: public area {
      public:
              float getarea()
              {
               float d1,d2;
               getdim (d1,d2);
               return   d1*d2;
               }
};
class  triangle: public area {
      public:
              float getare()
              {
               float d1,d2;
               getdim (d1,d2);
               return   0.5*d1*d2;
               }
};
main()
{
   area  *p;
   rectangle  r;
```

```
    triangle   t;
    r.setarea(3.0,4.0);
    t.setarea(4.0,5.0);
    p=&r;
   cout<<"area of rectangle:  "<<p→getarea() <<"\n";
   p=&t;
   cout<<"area of triangle:  "<<p→getarea() <<"\n";
return 0;
}
```

خروجی برنامه فوق به صورت زیر است:

```
area of rectangle: 12.00
area of triangle: 10.00
```

توجه: همانگونه که در مثال فوق مشاهده می‌شود، تعریف تابع getarea() در کلاس area عملاً هیچ کاری انجام نمی‌دهد و صرفا فضایی را به خود اختصاص داده است، زیرا کلاس area به هیچ شکل هندسی خاصی ارتباط ندارد و به عبارت دیگر هیچ تعریف کلی برای محاسبه مساحت شکل‌ها در area پیش بینی نشده است. در واقع تابع getarea() باید توسط کلاس‌های مشتق شده override شود، تا بتوان از آن برای محاسبه مساحت یک شکل به خصوص استفاده کرد.

توجه: بازتعریف یا تعریف مجدد یا دوباره‌نویسی تابع مجازی درون کلاس مشتق شده، در وهله نخست چیزی شبیه سربارگذاری تابع به نظر می‌رسد، در حالی که این دو فرآیند با یکدیگر کاملا متفاوتند. یک تابع سربارگذاری شده باید در نوع داده یا تعداد پارامترها تفاوت داشته باشد. در حالی که در تعریف مجدد یک تابع مجازی تعداد و نوع پارامترهای تابع و نوع مقدار بازگشتی حتما باید یکسان باشند. به دلیل این تفاوت‌ها به تعریف توابع سربارگذاری شده عمل **overloading** و به تعریف توابع مجازی عمل **overriding** می‌گویند تا از همدیگر مجزا شوند.

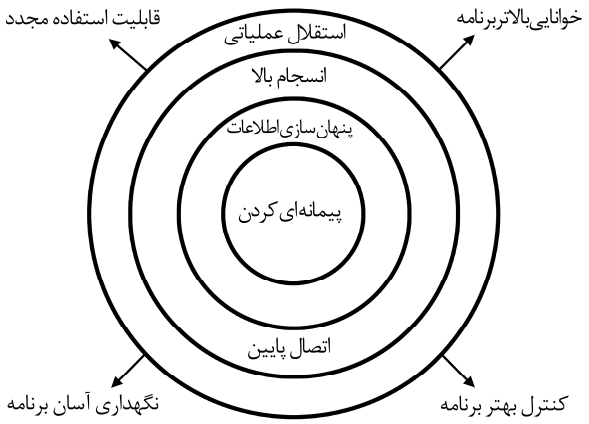
اصول سالید (S.O.L.I.D)

همانطور که گفتیم سه ویژگی اصلی شیءگرایی عبارتند از: کپسوله‌سازی، وراثت و پلی مورفیسم (چندریختی)

توجه: از آنجا که متدولوژی شیءگرایی به طور ذاتی اصول طراحی مطلوب و کارآمد را رعایت می‌کند، بنابراین، هرگاه از متدولوژی شیءگرایی در خلق برنامه‌های کامپیوتری خود که همان رسیدن به هدف و حل مساله هست استفاده نماییم، در مقصد این متدولوژی مزایایی را به ارمغان می‌آورد که از ارزش بالایی برخوردار است. مزایایی همچون رعایت «**اصل پنهان‌سازی اطلاعات**»، «**استقلال عملیاتی**»، «**قابلیت استفاده مجدد**»، «**انسجام بالا**» و «**اتصال پایین**».

توجه: پیمانه‌ای‌کردن، بسته‌بندی، برقراری پنهان‌سازی اطلاعات، انسجام بالا، اتصال پایین و برقراری استقلال عملیاتی، به عنوان اصول و معیارهای طراحی معماری مطلوب، منجر به **خوانایی بالاتر برنامه**، **کنترل بهتر برنامه**، **قابلیت استفاده مجدد** پیمانه‌های(کلاس‌های) برنامه جاری در

برنامه‌های آتی و **نگهداری آسان برنامه** جاری می‌گردد.
شکل زیر گویای مطلب است:



توجه: اصول و معیارهای طراحی معماری مطلوب یعنی پیمانه‌ای‌کردن (modularity)، بسته‌بندی (Encapsulation)، برقراری پنهان‌سازی اطلاعات (information hiding)، انسجام بالا (Cohesion بالا)، اتصال پایین (Coupling پایین) و برقراری استقلال عملیاتی (functional independence)، به طور ذاتی در شیءگرایی برقرار است. بنابراین، پیمانه‌ای‌کردن، بسته‌بندی، برقراری پنهان‌سازی اطلاعات، انسجام بالا، اتصال پایین و برقراری استقلال عملیاتی، به عنوان اصول و معیارهای طراحی معماری مطلوب که منجر به خوانایی بالاتر برنامه، کنترل بهتر برنامه، قابلیت استفاده مجدد (reusability) پیمانه‌های (کلاس‌های) برنامه جاری در برنامه‌های آتی و نگهداری آسان برنامه جاری می‌گردد، به عنوان نتایج استفاده از شیءگرایی محسوب می‌گردند.

سالید (S.O.L.I.D) مخفف پنج اصل اساسی برنامه‌نویسی و طراحی شیءگرا می‌باشد. وقتی این اصول با هم در طراحی و پیاده سازی یک برنامه اعمال می‌شود، سیستم قابلیت این را خواهد داشت که به آسانی قابل **توسعه‌پذیری** و **نگهداری** باشد. در حقیقت اصول سالید، دستورالعمل‌هایی هستند که می‌توان هنگام کار بر روی یک نرم‌افزار، آن‌ها را برای از بین بردن، عوامل نامطلوب در کد، اعمال کرد. اینکار از طریق فراهم آوردن چارچوبی انجام می‌گیرد که با استفاده از آن برنامه‌نویس می‌تواند متن برنامه را اصلاح و بازسازی کندتا کد برنامه خوانا و توسعه‌پذیر شود. این اصول بخشی از راهبرد توسعه نرم‌افزار چابک و تطبیقی می‌باشند.

اصول سالید به صورت زیر است:

۱- اصل تک مسئولیتی (SRP: Single Responsibility Principle)

یک کلاس باید تنها یک وظیفه داشته باشد (یک کلاس باید تنها یک دلیل برای تغییر و توسعه احتمالی داشته باشد و نه بیشتر)، به عبارت دیگر یک کلاس نباید واسط‌های بزرگ و چندمنظوره در اختیار مشتریانش قرار دهد. **این اصل همان** انسجام عملیاتی یا کارکردی

(Functional Cohesion) **است**. اگر درجه تعداد وظایف یا مسئولیت‌های مختلف یک پیمانه برابر عدد یک باشد، آن پیمانه دارای انسجام عملیاتی است. در این نوع انسجام، پیمانه دقیقا یک مسئولیت را انجام داده و یک هدف واحد را دنبال می‌کند. این نوع انسجام بهترین سطح انسجام را ارائه می‌کند. به این پیمانه‌ها، پیمانه‌های مصمم (single minded module) نیز می‌گویند. در شیءگرایی، یک کلاس به طور ذاتی دارای انسجام عملیاتی می‌باشد، زیرا تمام متدهای یک کلاس حول یک محور، و آن هم، حول محور مسئولیت کلاس مورد نظر مورد استفاده قرار می‌گیرند. مانند کلاس دانشجو که متدهای آن مسئولیت دانشجو بودن را به انجام می‌رسانند. این جمله را همه شنیدیم که یک کار انجام بده ولی درست انجام بده!

۲- اصل باز - بسته (OCP: Open/Closed Principle)

اجزای نرم‌افزار باید نسبت به «**توسعه باز**» (یعنی پذیرای توسعه باشد) و نسبت به «**اصلاح و تغییر بسته**» باشند (یعنی پذیرای اصلاح نباشد). (یعنی مثلا برای افزودن یک ویژگی جدید به نرم افزار نیاز نباشد که بعضی از قسمت های کد را بازنویسی کرد، بلکه بتوان آن ویژگی را مانند افزونه به راحتی به نرم‌افزار افزود)، به عبارت دیگر «**یک مولفه باید بتواند گسترش داده شود بدون اینکه اجزای داخلی آن تغییر یابند.**» کلاس‌های فرزند با ارث بردن از کلاس والد تا حد زیادی به جزئیات ساختار والد وابسته می‌شدند. این روند باعث ایجاد اتصال تنگاتنگ (Tight Coupling) می‌شود. در برنامه‌های در هم تنیده پس از ایجاد تغییر در یک کلاس مجبوریم کلاس‌های وابسته و مربوط دیگر را هم تغییر دهیم. یعنی تغییر کوچکی در برنامه باعث بر زمین ریختن آوار برنامه و اورهای مکرر می‌شود. بنابراین جهت محقق‌سازی این اصل می‌بایست وراثت کمتر و چندریختی بیشتر مورد استفاده قرار گیرد. به عبارت دیگر واسپاری وظایف (چندریختی) باید جایگزین استفاده از توارث شود.

۳- اصل جایگزینی لیسکوف (LSP: Liskov Substitution Principle)

فرزندان یک کلاس باید بتوانند جایگزین پدر شوند. به عبارت دیگر اشیاءِ یک برنامه که از یک کلاس والد هستند، باید به راحتی و بدون نیاز به تغییر در برنامه، قابل جایگزینی با یکدیگر باشند. اگر S یک زیر کلاس T باشد، آبجکت های نوع T باید بتوانند بدون تغییر دادن کد برنامه با آبجکت‌های نوع S جایگزین بشوند.

فرض کنیم یک کلاس داریم به اسم A:

```
class A{ ... }
```

قرار است از کلاس A آبجکت‌هایی ساخته شود که در بخش‌های مختلف برنامه استفاده شود. فرض کنید کد زیر قسمت‌های مختلف برنامه است که از کلاس A استفاده می‌کند:

```
x A;
y A;
z A;
```

حال قرار است کلاس A را توسعه دهیم. برای همین کلاسی به اسم B را می‌سازیم که از کلاس A مشتق می‌شود:

```
class B : public A { ... }
```

پس کلاس B، یک زیر نوع (فرزند) از کلاس A است.

بالاتر دیدیم که در برنامه، از کلاس A آبجکت هایی ساخته و استفاده شد. چون کلاس B یک زیر نوع از کلاس A است، میخواهیم داخل برنامه و جایی که از کلاس A استفاده کردیم، بجای کلاس A، از کلاس B استفاده کنیم. یعنی:

```
x A B ;
y A B ;
z A  B;
```

در مثال فوق ما جایگزینی انجام دادیم! کلاس B را با کلاس A عوض کردیم. طبق اصل LSP، وقتی جایگزینی انجام می‌دهیم، برنامه نباید بخاطر جایگزینی دچار خطا بشود. همچنین کد برنامه هم نباید تغییر کند. جهت محقق‌سازی این اصل نباید دوباره‌نویسی(overriding) در کلاس فرزند مورد استفاده قرار گیرد. برای مثال اگر کلاس والد یک متد دارد که خروجی اون عددی است، کلاس فرزند نباید این متد رو جوری دوباره‌نویسی کند که خروجی آرایه باشد.

۴- اصل جداسازی اینترفیس‌ها (ISP: Interface Segregation Principle)

استفاده از چند رابط که هر کدام، فقط یک وظیفه را بر عهده دارد بهتر از استفاده از یک رابط چند منظوره است. این اصل شباهت زیادی به اصل اول دارد که می‌گفت یک کلاس باید تنها یک وظیفه داشته باشد اما این اصل می‌گوید یک اینترفیس باید تنها یک وظیفه داشته باشد به عبارت دیگر کلاس‌ها در استفاده از اینترفیس نباید مجبور باشند متدهایی که به اونها احتیاجی ندارند را پیاده‌سازی کنند. در صورتی که یک Interface اصطلاحا چاق باشد (Fat Interface) کلاس‌های پیاده کننده را مجبور به پیاده‌سازی تعداد زیادی متد می‌کند که قرار نیست استفاده‌ای از آن ها بشود. این اصل می‌گوید که ما باید اینترفیس (Interface) ها را جوری بنویسیم که وقتی یک کلاس از آن استفاده می‌کند، مجبور نباشد متدهایی که لازم ندارد را پیاده‌سازی کند. اینترفیس زیر را در نظر بگیرید:

```
interface Animal{
    fly();
    run();
}
```

این اینترفیس دو متد دارد که باید توسط کلاس‌هایی که از آن استفاده می‌کنند پیاده‌سازی شود، کلاس Dolphin (دلفین) را در نظر بگیرید که از این اینترفیس استفاده می‌کند:

```
class Dolphin implements Animal{
    public fly () {
        return false;
    }
    public run() {
```

```
    // Run
  }
}
```

همانطور که میدانید، دلفین ها نمی‌توانند پرواز کنند. پس ما مجبور شدیم داخل متد fly بنویسیم return false. اینجا قانون ISP نقض شد. چون کلاس دلفین مجبور به پیاده‌سازی متدی شد که از آن استفاده نمی‌کند.

اگر بخواهیم این اصل را رعایت کنیم باید جداسازی اینترفیس انجام بدهیم. پس متد fly را به یک اینترفیس جدا منتقل می‌کنیم:

```
interface Animal {
   run();
}
interface FlyableAnimal {
   fly();
}
```

بنابراین کلاس دلفین دیگر مجبور نیست متد fly را پیاده‌سازی کند و کلاس‌هایی که به این متد نیاز دارند، اینترفیس FlyableAnimal را هم پیاده‌سازی می‌کنند:

```
class Dolphin implements Animal{
   public run() {
   // Run
 }
```

۵- اصل وارونگی وابستگی (DIP: Dependency Inversion Principle)

بهتر است که برنامه به تجرید (abstraction) وابسته باشد نه پیاده‌سازی به عبارت دیگر کلاس های سطح بالا نباید به کلاس های سطح پایین وابسته باشند؛ هر دو باید وابسته به انتزاع (Abstractions) باشند. موارد انتزاعی نباید وابسته به جزییات باشند. جزییات باید وابسته به انتزاع باشند.

کلاس‌های سطح پایین به کلاس‌هایی گفته می‌شود که مسئول عملیات اساسی و پایه‌ای در نرم‌افزار هستند. مثل برقراری ارتباط با دیتابیس یا هارد دیسک، کار با ایمیل و ...

کلاس‌های سطح بالا به کلاس‌هایی گفته می‌شود که عملیات پیچیده‌تر و خاص‌تری انجام می‌دهند و برای انجام این کار از کلاس‌های سطح پایین استفاده می‌کنند. برای مثال کلاس گزارش گیری، برای ثبت و خواندن گزارش، به کلاس دیتابیس یا هارد دیسک نیاز دارد. کلاس Users، برای اطلاع‌رسانی به کلاس ایمیل نیاز دارد.

ابتدا کد زیر رو در نظر بگیرید:

```
class MySQL{
   public insert () {}
   public update () {}
   public delete () {}
}
class Log  {
   private database;
```

```
    constructor () {
        this.database = new  MySQL;
    }
}
```

فرض کنیم یک کلاس سطح پایین داریم مثلا دیتابیس MySQL و یک سری کلاس سطح بالا مثلا گزارش‌گیری (Log) از این کلاس استفاده می‌کنند. اگه بخواهیم یک تغییر در کلاس دیتابیس انجام بدهیم، ممکن است بطور مستقیم تاثیر بگذارد روی کلاس‌هایی که از آن استفاده می‌کنند. مثلا اگر در کلاس MySQL اسم متد را تغییر بدهیم و یا پارامترها را کم و زیاد کنیم، نهایتا در کلاس Log این تغییرات را باید اعمال کنیم.

همچنین کلاس‌های سطح بالا قابل استفاده مجدد نیستند. مثلا اگه بخواهیم برای کلاس Log از دیتابیس‌های دیگر مثلا MongoDB یا هارددیسک استفاده کنیم باید کلاس Log را تغییر بدهیم یا یک کلاس جدا براساس هر نوع دیتابیس بسازیم.

همانطور که می‌بینید اگر یک کلاس سطح بالا وابسته به یک کلاس سطح پایین باشد این مشکلات به وجود می‌آید.

برای حل این مشکل باید با اینترفیس، یک لایه انتزاعی درست کنیم. با این کار کلاس Log دیگر وابسته به یک کلاس خاص برای ذخیره‌سازی و خواندن اطلاعات نیست و می‌توانیم هر نوع دیتابیسی را استفاده کنیم و برای کلاس Log اهمیتی ندارد که با چه نوع دیتابیسی دارد کار می‌کند چون وابسته به انتزاع است.

ابتدا یک اینترفیس می‌سازیم برای اینکه کلاس های سطح بالا و سطح پایین را وابسته به این اینترفیس کنیم:

```
interface Database  {
    insert ();
    update ();
    delete ();
}
```

حالا کلاس‌های سطح پایین باید این اینترفیس را پیاده‌سازی کنند تا وابسته به انتزاع باشند:

```
class MySQL implements Database  {
    public insert () {}
    public update () {}
    public delete () {}
}
class FileSystem implements Database  {
    public insert () {}
    public update () {}
    public delete () {}
}
class MongoDB implements Database  {
```

```
    public insert () {}
    public update () {}
    public delete () {}
}
```

و نهایتا در کلاس های سطح بالا، وابستگی به یک کلاس خاص را بـه اینتـرفیس منتقـل مـی‌کنـیم. کلاس های سطح بالا زمانی وابسته به انتزاع می‌شوند که بجای استفاده مستقیم از کلاس های سـطح پایین، از یک اینترفیس (رابط) استفاده کنند:

```
class Log {
    private db: Database;
    public setDatabase(db: Database) {
        this.db = db;
}
    public update () {
        this.db.update() ;
   }
}
```

همانطور که می‌بینید وابستگی به یک کلاس خاص از بین رفت و می‌تـونیم هـر نـوع دیتابیسـی را برای کلاس Log استفاده کنیم:

```
let logger = new  Log;
logger.setDatabase(new MongoDB);
// …
logger.setDatabase(new FileSystem);
// …
logger.setDatabase(new MySQL);
logger.update ();
```

تست‌های فصل ششم

۱- مهم‌ترین مزیت طراحی شیءگرا این است که: (مهندسی IT – آزاد ۸۴)

۱) cohesion را افزایش و coupling را کاهش می‌دهد.
۲) cohesion را کاهش و coupling را افزایش می‌دهد.
۳) cohesion را افزایش و coupling را افزایش می‌دهد.
۴) cohesion را کاهش و coupling را کاهش می‌دهد.

۲- چهار رکن اصلی طراحی شیءگرایی عبارتند از: (مهندسی IT – آزاد ۸۵)

۱) cohesion, modularity, coupling, abstraction
۲) hierarchy, modularity, encapsulation, abstraction
۳) hierarchy, modularity, reusability, abstraction
۴) coupling, modularity, reusability, abstraction

۳- فرآیند یکپارچه توسعه نرم‌افزار (USDP) به عنوان نماینده کدام‌یک از مدل‌های فرآیند نرم‌افزار معرفی می‌گردد؟ (مهندسی IT – آزاد ۸۶)

۱) مدل توسعه هم‌روند
۲) مدل مارپیچی برنده – برنده (win-win)
۳) مدل روش‌های رسمی
۴) مدل توسعه بر مبنای مؤلفه (Component Base Development)

۴- در کدام‌یک از مشی‌های ایجاد نرم‌افزار زیر، قابلیت استفاده‌ی مجدد نوعاً از دغدغه‌های اصلی طراح است؟ (مهندسی IT – دولتی ۹۰)

۱) مشی جنبه‌گرا (Aspect-Oriented)
۲) ایجاد مبتنی بر مؤلفه (Component-Based)
۳) مشی عامل‌گرا (Agent-Oriented)
۴) مشی شیءگرا (Object-Oriented)

۵- در بین انواع انسجام (Cohesion) که در زیر فهرست شده‌اند، کدام‌یک از همه مطلوب‌تر است؟ (مهندسی IT – دولتی ۹۲)

۱) انسجام منطقی (Logical)
۲) انسجام زمانی (Temporal)
۳) انسجام تصادفی (Coincidental)
۴) انسجام ارتباطی (Communicational)

۶- کدام تعریف بیانگر اصل باز – بسته (OCP) است؟ (مهندسی IT – دولتی ۹۸)

۱) فرزندان یک کلاس باید بتوانند جایگزین پدر شوند.
۲) واسپاری وظایف باید جایگزین استفاده از توارث شود.
۳) یک کلاس نباید واسط‌های بزرگ و چندمنظوره در اختیار مشتریانش قرار دهد.
۴) یک مولفه باید بتواند گسترش داده شود بدون اینکه اجزای داخلی آن تغییر یابند.

۷- یک کلاس طراحی خوش – تعریف (Well Formed)، باید کدام خصوصیت را داشته باشد؟
(مهندسی IT – دولتی ۹۹)

۱) کمترین تعداد کلاس‌های داخلی (Inner Classes) را داشته باشد.
۲) حداقل اتصال (Coupling) را با کلاس‌های دیگر داشته باشد.
۳) عضوی از یک سلسله مراتب توارث باشد.
۴) زیر کلاس‌های هم‌پوشان نداشته باشد.

پاسخ تست‌های فصل ششم

۱- گزینه (۱) صحیح است.

در یک طراحی ایده‌آل، هدف، محقق کردن بالاترین سطح انسجام (Cohesion بالا) داخل پیمانه‌های برنامه و کمترین سطح اتصال (Coupling پایین) مابین پیمانه‌های برنامه است.

در شیءگرایی، استقلال عملیاتی نتیجه پیمانه‌ای‌کردن، بسته‌بندی و پنهان‌سازی اطلاعات است. به بیان دیگر شرط لازم برای برقراری استقلال عملیاتی، پیمانه‌ای‌کردن، بسته‌بندی و پنهان‌سازی اطلاعات است و شرط کافی برای برقراری استقلال عملیاتی، انسجام بالا و اتصال پایین است. مکانیزم کلاس، مفهوم بسته‌بندی را پیاده‌سازی می‌کند. در صورتی که پیمانه‌هایی را با عملکرد تک منظوره (انسجام بالا) و عدم ارتباط بیش از حد با پیمانه‌های دیگر (اتصال پایین) ایجاد کنیم، استقلال عملیاتی تحقق می‌یابد. که این موارد در شیءگرایی به طور ذاتی توسط مکانیزم کلاس، محقق شده است. که این امر منجر به **چابک‌بودن** روند ساخت پروژه‌های شیءگرا می‌شود.

۲- گزینه (۲) صحیح است.

طراح محترم: کپسوله‌سازی (Encapsulation)، وراثت (Inheritance) و چندریختی یا پلی مورفیسم (polymorphism) سه ویژگی اصلی شیءگرایی، محسوب می‌گردند. ارکان اصلی شیءگرایی سه رکن است و نه چهار رکن!

پیمانه‌ای‌کردن، بسته‌بندی، برقراری پنهان‌سازی اطلاعات، برقراری استقلال عملیاتی، انسجام بالا و اتصال پایین، به عنوان اصول و معیارهای طراحی معماری مطلوب، منجر به قابلیت استفاده مجدد پیمانه‌های (کلاس‌های) برنامه جاری در برنامه‌های آتی و نگهداری آسان برنامه جاری می‌گردد.

اصول و معیارهای طراحی معماری مطلوب یعنی پیمانه‌ای‌کردن (modularity) ، بسته‌بندی (Encapsulation)، برقراری پنهان‌سازی اطلاعات (information hiding)، برقراری استقلال عملیاتی (functional independence)، انسجام بالا (Cohesion بالا) و اتصال پایین (Coupling پایین) به طور ذاتی در شیءگرایی برقرار است. بنابراین، پیمانه‌ای کردن، بسته‌بندی، برقراری پنهان‌سازی اطلاعات، برقراری استقلال عملیاتی، انسجام بالا و اتصال پایین به عنوان اصول و معیارهای طراحی معماری مطلوب که منجر به قابلیت استفاده مجدد (reusability) پیمانه‌های (کلاس‌های) برنامه جاری در برنامه‌های آتی و نگهداری آسان برنامه جاری می‌گردد، به عنوان نتایج استفاده از شیءگرایی محسوب می گردند.

مدل تحلیل، مدل‌سازی عالم خارج به زبانی شبیه انسان است، اما مدل طراحی مدل‌سازی عالم داخل ماشین به زبانی شبیه زبان ماشین است. بنابراین برای اینکه ساخت برنامه کامپیوتری که به زبان ماشین است، امکان‌پذیر باشد، باید مدل تحلیل به مدل طراحی نگاشت شود تا مدل طراحی بتواند به عنوان نقشه راهی شبیه به زبان ماشین، راهگشا باشد.

مدل‌های تحلیل، کلی‌تر و انتزاعی‌تر هستند، و برای مدل‌سازی عالم خارج مورد استفاده قرار

می‌گیرند، بدون آنکه به جزئیات نحوه پیاده‌سازی بپردازند، در واقع مدل تحلیل به کلی‌گویی به زبان انسان و حذف جزئیات نحوه پیاده‌سازی می‌پردازد. اما مدل‌های طراحی، جزئی‌تر و غیرانتزاعی‌تر هستند، و برای مدل‌سازی عالم داخل ماشین مورد استفاده قرار می‌گیرند، و به بیان جزئیات نحوه پیاده‌سازی می‌پردازند، در واقع مدل طراحی به جزئی‌گویی به زبان شبیه ماشین و درج جزئیات نحوه پیاده‌سازی می‌پردازد.

با نگاهی سطح به سطح، به حل مساله، سطوح مختلفی از انتزاع را خواهیم داشت. در بالاترین سطح انتزاع، راه‌حل به صورت کلی به زبان محیط مساله بیان می‌گردد. در سطوح پایین‌تر، راه حل به جزئیات پیاده‌سازی نزدیک‌تر می‌شود و در پایین‌ترین سطح انتزاع راه حل به صورتی بیان می‌شود تا مستقیما قابل پیاده‌سازی باشد.

تجرید یا انتزاع (abstraction)، روش و نگرشی در ارائه و توضیح است که در آن فقط اطلاعات مهمی که برای حل یک مساله لازم است، گردآوری و نگهداری می‌شود و از بقیه اطلاعات صرف نظر می‌گردد، بنابراین تجرید یا انتزاع، صرفا یک روش و نگرش در ارائه و توضیح برای حل مساله است که ساخت برنامه‌های کامپیوتری را به سمت پیمانه‌ای کردن سوق می‌دهد.

به عنوان مثال، برای تماس الکترونیکی با یک دوست، کافی است آدرس پست الکترونیک و نام او را بدانید. برای برقراری این تماس، دانستن محل کار، قد، سن و علایق او لازم نیست. این اطلاعات اضافی، هرچند که درست و معتبر باشند، اما در برقراری تماس، کمکی نمی‌کنند. بنابراین باید حذف شوند. رعایت تجرید در فرآیند تولید یک نرم‌افزار، باعث کاهش شلوغی و پیچیدگی پروژه می‌گردد، زیرا از ذکر اطلاعات اضافی خودداری شده و افراد تیم توسعه مانند تحلیل‌گر تنها با اطلاعات ضروری سروکار خواهند داشت. برای مثال، در فرآیند تحلیل یک سیستم بانکداری، وقتی که در سطح اول انتزاع هستیم، اموری نظیر برداشت از حساب، واریز به حساب و غیره به صورت کلی مورد بررسی قرار می‌گیرند و باید از پرداختن به جزئیات، صرف نظر نمود. در سطوح پایین‌تر، جزئیات هر کدام توضیح داده می‌شود.

مثال: مکالمه محاوره ای زیر را در نظر بگیرید:

سوال: کجا هستی؟

پاسخ: بیرون

سوال: کی می یای؟

پاسخ: میام

همانطور که مشاهده می کنید مکالمه فوق کاملا انتزاعی و با حذف جزئیات از طرف پاسخ‌دهنده انجام شده است. و برای سوال کننده همچنان سوالاتی نظیر کجای بیرون دقیقا و چه زمانی دقیقا، باقی مانده است، چون پاسخ‌ها کاملا انتزاعی از طرف پاسخ دهنده بیان شده است.

گزینه دوم به دلیل بیان دو ویژگی encapsulation و hierarchy از سه ویژگی اصلی شیءگرایی، گزینه بهتری برای پاسخ این سوال است. کلمه modularity موجود در گزینه دوم به

عنوان یکی از اصول و معیارهای طراحی معماری مطلوب نتیجه استفاده از شیءگرایی محسوب می‌گردد. و نه یکی از ارکان اصلی شیءگرایی!

۳- گزینه (۴) صحیح است.

متدولوژی RUP متداول‌ترین نمونه از متدولوژی شیءگرا براساس روش شیءگرا (مبتنی بر مفاهیم کلاس، وراثت و چندریختی)، مدل فرآیند **مبتنی بر مؤلفه‌ی شیءگرا** با رویکرد تکرار و تکامل و ابزارهای شیءگرا (مثل ابزار مدل‌سازی UML و زبان برنامه‌نویسی C++) می‌باشد.

نسبت نمونه متدولوژی شیء گرای RUP به متدولوژی شیء گرا مثل نسبت سیستم عامل ویندوز مدرن به مفاهیم مدرن سیستم عامل است.

RUP یک نمونه متدولوژی بزرگ صنعتی، برای تولید سیستم‌های نرم‌افزاری است که برای سهولت درک آن، کلیاتی از آن بدون نام شرکت رشنال و بدون محرز کردن جریان‌های کاری مربوط به فعالیت‌های چتری تولید نرم‌افزار و مدل‌سازی کسب و کار و بدون اشاره به قدرت RUP که همان ابزارهای حمایت‌کننده آن می‌باشد، در قالب فرآیند یکپارچه توسعه نرم‌افزارUSDP یا Unifed Software Development Process در دانشگاه‌های معتبر جهان ظهور کرده است. در واقع می‌توان گفت که RUP نسخه پیاده‌سازی شده‌ای از USDP است. بنابراین USDP به عنوان یک فرآیند شیءگرای تولید و توسعه سیستم‌ها، دارای مدل فرآیندی است که روند کلی تولید نرم‌افزار را مشخص می‌کند. به بیان دیگر شکل خلاصه شده RUP، USDP است. بنابراین مدل فرآیند USDP نیز همانند مدل فرآیند RUP، **مبتنی بر مؤلفه‌ی شیءگرا** است. پس گزینه چهارم درست است.

۴- گزینه (۲) صحیح است.

در دنیای برنامه‌نویسی، برنامه‌نویسی جنبه‌گرا، روش جدیدی است که پس از برنامه‌نویسی شیءگرا و برای رفع مشکلات آن به وجود آمده است. هدف از برنامه‌نویسی جنبه‌گرا، مستقل کردن وظایف (در قالب ماژول‌ها) است به طوری که کمترین تداخل را در یکدیگر داشته باشند تا نهایتاً بتوان از آنها برای برنامه‌های دیگر استفاده کرد.

برای‌مثال وظیفه احراز هویت (اعتبارسنجی کاربران) در یک برنامه را می‌توان به عنوان یک جنبه در نظر گرفت و ماژول جداگانه‌ای برای آن ساخت تا از این پس در تمام پروژه‌های مشابه بتوان از آن استفاده کرد. بیشتر زبان‌های برنامه‌نویسی قدرتمند از برنامه‌نویسی جنبه‌گرا پشتیبانی خوبی به عمل می‌آورند، مانند جاوا برنامه‌نویسی جنبه‌گرا، ارتباط تنگاتنگ و نزدیکی با شیءگرایی دارد ولی دو مفهوم متفاوت می‌باشند. برنامه‌نویسی عامل‌گرا، حالت ویژه‌ای از برنامه‌نویسی شیءگرا می‌باشد که در آن کلاس‌ها فقط شامل یک متد و پارامتر می‌باشند.

در مشی جنبه‌گرا، مشی عامل‌گرا و مشی شیءگرا قابلیت استفاده مجدد به طور ذاتی برقرار است، مگر طراح آگاهانه اقدام به نقض قابلیت استفاده مجدد پیمانه‌ها کند.

در مدل مبتنی بر مولفه ساخت‌یافته، قابلیت استفاده مجدد به طور ذاتی برقرار نیست، مگر طراح آگاهانه اقدام به برقراری قابلیت استفاده مجدد پیمانه‌ها کند.

پیمانه‌ای‌کردن، برقراری پنهان‌سازی اطلاعات، برقراری استقلال عملیاتی، انسجام بالا و اتصال پایین، به عنوان اصول و معیارهای طراحی معماری مطلوب، منجر به قابلیت استفاده مجدد پیمانه‌های (توابع) برنامه جاری در برنامه‌های آتی و نگهداری آسان برنامه جاری می گردد. در مدل مبتنی بر مولفه ساخت‌یافته برقراری اصول فوق برای محقق کردن خاصیت قابلیت استفاده مجدد، باید به عنوان دغدغه اصلی طراح، همواره در طول ساخت نرم‌افزار مطرح باشد.

۵- گزینه (۴) صحیح است.

انسجام (Cohesion) یا همبستگی، توسعه مفهوم پنهان‌سازی اطلاعات است، یک پیمانه یکپارچه و منسجم، یک وظیفه منفرد را انجام می‌دهد. انسجام، یک خصیصه مثبت در طراحی نرم‌افزار بوده که هر چه میزان آن بیشتر باشد، طراحی معماری از کیفیت بالاتری برخوردار است. در طراحی معماری، همواره برای بالاترین سطح انسجام تلاش می‌شود.

سطوح مختلف انسجام در ساخت‌یافتگی از بالاترین سطح انسجام (مطلوب‌ترین) تا پایین‌ترین سطح انسجام (نامطلوب‌ترین) به صورت زیر رتبه‌بندی شده است:

۱- انسجام تابعی یا عملیاتی یا کارکردی (Functional Cohesion)

۲- انسجام ترتیبی یا متوالی (Sequential Cohesion)

۳- انسجام رویه‌ای (Procedural Cohesion)

۴- انسجام زمانی یا موقتی (Temporal Cohesion)

۵- انسجام منطقی (Logical Cohesion)

۶- انسجام تصادفی یا سودمندی یا ابزاری (Coincidental Cohesion)

سطوح مختلف انسجام در شیءگرایی از بالاترین سطح انسجام (مطلوب‌ترین) تا پایین‌ترین سطح انسجام (نامطلوب‌ترین) به صورت زیر رتبه‌بندی شده است:

۱- انسجام عملیاتی یا کارکردی(Functional Cohesion)

۲- انسجام لایه‌ای(Layer Cohesion)

۳- انسجام ارتباطی(Communication Cohesion)

انسجام ارتباطی زمانی رخ می‌دهد که تمامی عملیات یک پیمانه به داده‌هایی یکسان و مشترک دسترسی دارند. در شیءگرایی، یک کلاس به طور ذاتی دارای انسجام ارتباطی می‌باشد، زیرا در مکانیزم کلاس در شیءگرایی، که مفهوم بسته‌بندی را پیاده‌سازی می‌کند، همه متدهای کلاس حول محور صفات کلاس به عنوان داده‌هایی یکسان و مشترک، فعالیت می‌کنند. در مفهوم کلاس، تمامی متدهای کلاس بر روی یک ساختمان داده که همان صفات کلاس است، عمل می‌کنند. در این ساختار، داده‌ها یا صفات فقط توسط متدهای همان کلاس دستکاری می‌شوند.

به طورکلی، تمامی انسجام‌های شیءگراء، شامل انسجام عملیاتی یا کارکردی، انسجام لایه‌ای و انسجام ارتباطی از کلیه‌ی انسجام‌های ساخت‌یافته، در سطح بالاتری (مطلوب‌تری) قرار دارند، زیرا مولفه‌های شیءگراء یعنی کلاس‌ها، به طور ذاتی از طراحی ایده‌آل و مطلوب برخوردار هستند. بنابراین گزینه چهارم درست است.

۶- گزینه (۴) صحیح است.

اجزای نرم افزار باید نسبت به «توسعه باز» (یعنی پذیرای توسعه باشد) و نسبت به «اصلاح و تغییر بسته» باشند (یعنی پذیرای اصلاح نباشد). (یعنی مثلا برای افزودن یک ویژگی جدید به نرم‌افزار نیاز نباشد که بعضی از قسمت‌های کد را بازنویسی کرد، بلکه بتوان آن ویژگی را مانند افزونه به راحتی به نرم‌افزار افزود)، به عبارت دیگر «یک مولفه باید بتواند گسترش داده شود بدون اینکه اجزای داخلی آن تغییر یابند.»

۷- گزینه (۲) صحیح است.

صورت سوال به این شکل است:

یک کلاس طراحی خوش - تعریف (Well Formed)، باید کدام خصوصیت را داشته باشد؟

۱) کمترین تعداد کلاس‌های داخلی (Inner Classes) را داشته باشد.

گزینه اول پاسخ سوال نیست، زیرا هرگاه رابطه‌ای مابین یک واحد کل و تعدادی واحد جزء مطرح باشد، یعنی یک کلاس کل از همکاری تعدادی کلاس جزء تشکیل شده باشد، آنگاه در این شرایط رابطه انجمنی پیشرفته یا رابطه بخشی خواهیم داشت. رابطه انجمنی پیشرفته یا رابطه بخشی به دو رابطه تجمع (Aggregation) و ترکیب (Composition) تقسیم می‌گردد. و کمترین تعداد کلاس‌های داخلی (Inner Class) ارتباطی به یک کلاس طراحی خوش – تعریف (Well Formed) ندارد.

۲) حداقل اتصال (Coupling) را با کلاس‌های دیگر داشته باشد.

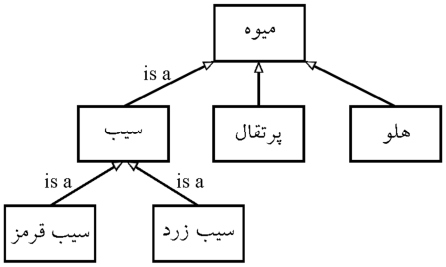
گزینه دوم پاسخ سوال است، زیرا در یک طراحی ایده‌آل یعنی خوش تعریف (Well Formed)، هدف، محقق کردن بالاترین سطح انسجام (Cohesion بالا) داخل پیمانه‌های برنامه و کمترین سطح اتصال (Coupling پایین) مابین پیمانه‌های برنامه است. خوش تعریف بودن یعنی شرایط یک کلاس خیلی وابسته به شرایط کلاس‌های دیگر نباشد.

۳) عضوی از یک سلسله مراتب توارث باشد.

گزینه سوم پاسخ سوال نیست، زیرا وراثت فرآیندی است که به وسیله آن یک کلاس فرزند می‌تواند صفات و متدهای عمومی کلاس پدر را کسب کند. و این موضوع ارتباطی به یک کلاس طراحی خوش – تعریف (Well Formed) ندارد.

۴) زیر کلاس‌های هم‌پوشان نداشته باشد.

گزینه چهارم پاسخ سوال نیست، زیرا رابطه وراثت به عنوان رابطه «is a» یا «kind of» یا «type of» شناخته می‌شود. برای مثال در روابط وراثت می‌گوییم: «A red apple is an apple» و یا «An apple is a fruit». شکل زیر نمونه‌ای از یک سلسله مراتب ارثبری میان «میوه»، «سیب» و «سیب قرمز»، را نشان می‌دهد.



و این موضوع ارتباطی به یک کلاس طراحی خوش – تعریف (Well Formed) ندارد.

۷ مدل تحلیل و مدل طراحی شیءگرا

مقدمه

فرآیند تولید نرم‌افزار در متدولوژی شیءگرا نیز همانند متدولوژی ساخت‌یافته با فعالیت ارتباط آغاز می‌گردد. با این تفاوت که در فعالیت مدل تحلیل شیءگرا به جای استفاده از مفاهیم ساخت‌یافته و نمودارهایی همچون نهاد و رابطه (ERD) و جریان داده (DFD)، از مفاهیم شیءگرا و نمودارهای UML استفاده می‌گردد.

در فعالیت مدل‌سازی شیءگرا برخلاف شیوه ساخت‌یافته، سیستم به عناصر داده و عملکرد تفکیک نمی‌شود بلکه واحدهایی ایجاد می‌گردد که هر یک از آنها شامل عناصر داده و عملکرد (متد) مرتبط با آن می‌باشد (جعبه سیاه). این واحدها را کلاس می‌نامند. از دیدگاه شیءگرا **دامنه‌ی مسئله** از تعدادی کلاس و روابط میان آنها تشکیل شده است.

فعالیت‌های چارچوبی فرآیند تولید نرم‌افزارها بر اساس متدولوژی شیءگرا

۱- فعالیت ارتباط (مهندسی نیازمندی‌ها)

در این مرحله لیست نیازمندی‌های مشتری از طریق ارتباط با مشتری و ابزارهایی همچون گفتگو، مشاهده مصاحبه، پرسش‌نامه، بازدید، نمونه‌سازی دوراندانختنی و نمونه‌سازی تکاملی، جمع‌آوری و آماده می‌گردد.

توجه: مهم‌ترین کار، در فعالیت ارتباطات، مدیریت شناسایی نیازمندی‌ها و پیگیری تغییرات نیازمندی‌های مشتری است.

توجه: لیست نیازمندی‌های مشتری در این مرحله به صورت متن نوشته می‌شود.

مثال: نمایش لیست نیازمندی‌های مشتری برای سیستم ATM.

۱- واریز وجه
۲- انتقال وجه
۳- برداشت وجه
۴- تغییر رمز
۵- پرداخت
۶- نمایش موجودی

۲- فعالیت برنامه‌ریزی

برنامه‌ریزی یعنی هنر حرکت از مبدأ موجود به مقصد مطلوب برای رسیدن به نتیجه‌ای مطلوب بر اساس خواسته‌های مورد نیاز در یک زمان مشخص.

«هر تلاشی منجر به نتیجه‌ای مطلوب نمی‌گردد، بلکه این تلاشی مطلوب است که منجر به نتیجه‌ای مطلوب می‌گردد.»

لازمه‌ی تلاش مطلوب، برنامه‌ریزی است. برنامه‌ریزی می‌تواند اجرای هر کار پیچیده‌ای را ساده‌تر سازد. هر کار مهندسی مستلزم برنامه‌ریزی می‌باشد. مهندسی نرم‌افزار نیز مانند هر فعالیت مهندسی دیگری، نیازمند برنامه‌ریزی است. فعالیت برنامه‌ریزی، برنامه‌ای را برای فعالیت‌های مختلف بخش‌های مختلف فرآیند تولید نرم‌افزار پایه‌ریزی می‌کند. این فعالیت، وظیفه‌های فنی که باید هدایت شوند، ریسک‌هایی که محتمل می‌شوند (مانند عدم شناسایی برخی نیازمندی‌ها، از دست دادن داده‌ها و مدیران)، منابع مورد نیاز، واحدهای کاری که باید ایجاد شوند و برنامه زمان‌بندی برای کارها را تشریح می‌کند. مدیریت، برنامه‌ریزی را به مرحله‌ی اجرا و نظارت می‌برد.

۳- فعالیت مدل‌سازی (تحلیل و طراحی)

یک مدل، ساده شده یک واقعیت است. ایجاد یک مدل برای سیستم‌های نرم‌افزاری قبل از ساخت یا بازساخت آن، به اندازه داشتن نقشه برای ساختن یک ساختمان ضروری وحیاتی است. بسیاری از شاخه‌های مهندسی، توصیف چگونگی محصولاتی که باید ساخته شوند را ترسیم می‌کنند و همچنین دقت زیادی می‌کنند که محصولاتشان طبق این مدل‌ها و توصیف‌ها ساخته شوند. مدل‌های خوب و دقیق در برقراری یک ارتباط کامل بین افراد پروژه، نقش زیادی می‌توانند داشته باشند. علت اصلی مدل کردن سیستم‌های پیچیده این است که نمی‌توان به یکباره کل سیستم را تجسم کرد و ممکن است سیستم دارای ابهامات بسیاری باشد. لذا برای رفع این ابهامات و نیز برای فهم کامل سیستم، یافتن و نمایش ارتباط بین قسمت‌های مختلف آن، از مدل‌سازی استفاده می‌شود. فعالیت مدل‌سازی خود شامل دو مرحله‌ی مدل تحلیل و مدل طراحی می‌باشد. مدل تحلیل پس از فعالیت ارتباطات (جمع‌آوری نیازمندی‌ها) و قبل از مدل طراحی انجام می‌شود، در واقع خروجی مدل تحلیل، ورودی مدل طراحی می‌باشد.

زبان مدل‌سازی یکپارچه (UML)

UML که سرواژه عبارت Unified Modeling Language و به معنی زبان مدل‌سازی یکپارچه است، نمودارهایی را برای مدل‌سازی سیستم‌های شیءگرا ارایه می‌دهد (فعالیت تحلیل تا استقرار).

توجه: از آنجا که UML از ترکیب زبان‌های مدل‌سازی مختلفی همچون Booch، OMT، Rumbough، Jacobson و غیره ایجاد شده است. این زبان، «**زبان مدل‌سازی یکپارچه**» نام‌گذاری شده است.

ادغام مدل‌ها به منظور ایجاد UML از سال ۱۹۹۳ آغاز گردید و تا سال ۱۹۹۵ ادامه داشت. یعنی تا زمانی که نسخه Unified Method معرفی شد.

Unified Method اصلاح شد و در سال ۱۹۹۶ به Unified Modeling Language تغییر نام یافت، نسخه UML 1.0 تأیید شد و در سال ۱۹۹۷ به گروه تکنولوژی آبجکت (Object Technology Group) داده شد و شرکت‌های تولیدکننده نرم‌افزاری شروع به سازگار شدن با آن نمودند. سرانجام در ۱۴ نوامبر ۱۹۹۷، OMG نسخه UML 1.1 را به عنوان یک استاندارد صنعتی معرفی نمود.

توجه: هر نظام مهندسی یک روش استاندارد برای مستندسازی دارد. مهندسی الکترونیک schematic diagrams، مهندسی معماری blueprints diagrams، و مهندسین مکانیک mechanical diagrams را دارند. در حال حاضر صنعت نرم‌افزار UML را در اختیار دارد.

مدل تحلیل: OOA (Object-Oriented Analysis)

پس از جمع‌آوری نیازمندی‌ها در فعالیت ارتباطات، نوبت به مدل تحلیل (مدل‌سازی نیازمندی‌ها) می‌رسد. مدل‌سازی که فعالیتی فنی به شمار می‌رود، نیازمندی‌ها را باید به گونه‌ای مدل نماید که برای سازنده و مشتری قابل فهم باشد.

مدل تحلیل، شامل مراحل زیر می‌باشد:

الف) مدل‌سازی محیط عملیاتی یک کسب و کار

توجه: Business use case یا مورد کاربرد کسب و کار، جهت مدل‌سازی محیط عملیاتی یک کسب و کار مورد استفاده قرار می‌گیرد.

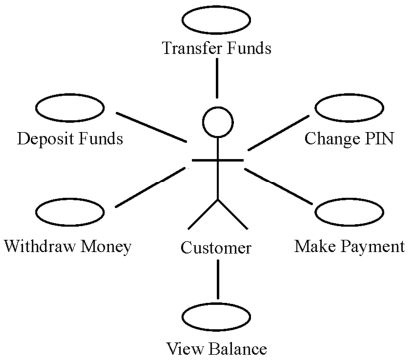
ب) مدل‌سازی لیست نیازمندی‌های مشتری

توجه: Use Case Diagram یا نمودار مورد کاربرد، Requirement Diagram یا نمودار نیاز جهت مدل‌سازی لیست نیازمندی‌های مشتری مورد استفاده قرار می‌گیرد.

توجه: از آنجا که Use Case Diagram یا نمودار مورد کاربرد تعامل بین کاربر نهایی و برنامه کامپیوتری را مدل‌سازی می‌کند، به این مدل، مدل مبتنی بر سناریوی میان کاربر نهایی و برنامه کامپیوتری نیز گفته می‌شود.

توجه: Use Case Diagram یا نمودار مورد کاربرد، می‌تواند به عنوان مدل لیست نیازمندی‌ها، به شکل چک لیست برای فعالیت تست مورد استفاده قرار بگیرد.

مثال: مدل‌سازی لیست نیازمندی‌های مشتری برای سیستم ATM.



توجه: در نمودار فوق، لیست نیازمندی‌های مشتری، مدل‌سازی شده است.

توجه: به هر یک از نیازهای فوق یک use case یا مورد کاربر گفته می‌شود و به اجتماع این use caseها، Use Case Diagram یا نمودار مورد کاربرد گفته می‌شود.

توجه: به یک use case یا مورد کاربرد، نیاز (requirement) یا زیرسیستم (subsystem) نیز گفته می‌شود.

توجه: از آنجا که Use Case Diagram یا نمودار مورد کاربرد یا نمودار نیاز مستقل از مفاهیم شیءگرایی (کلاس، وراثت و چندریختی) است، بنابراین می‌توان مدل‌سازی لیست نیازمندی‌های مشتری در مرحله **مدل تحلیل متدولوژی ساخت‌یافته** (مهندسی‌نرم‌افزار ساخت‌یافته) را توسط این نمودار انجام داد.

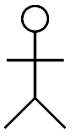
توجه: Use Case Diagram یا نمودار مورد کاربرد، هیچگاه به طور مستقیم، پیاده‌سازی نمی‌شود، بلکه مبنایی برای استخراج دیگر نمودارها قرار می‌گیرد، بنابراین ساختار نمودار کاربرد به پیاده‌سازی نرم‌افزار یا نسخه فیزیکی تبدیل نمی‌گردد.

توجه: Use Caseها و Actorها محدوده سیستم در حال ساخت را مشخص می‌کنند. Use Case شامل تمام آن چیزهایی است که درون سیستم قرار دارد و Actor شامل تمام آن چیزهایی است که خارج از سیستم قرار دارد. هر فرد یا هر چیزی که با سیستم تعامل دارد، Actor یا بازیگر نامیده می‌شود. Use Caseها هر چیز موجود در داخل و محدوده سیستم را توصیف می کنند، در حالی که Actorها هر چیز موجود در خارج از محدوده سیستم را توصیف می‌کنند.

توجه: بازیگران، افراد و یا گاهی نرم‌افزار و یا سخت‌افزارهایی هستند که از سیستم استفاده می‌کنند و یا اطلاعاتی را برای سیستم فراهم می‌کنند. با اینکه همه بازیگران، عناصر خارجی سیستم هستند، اما با این حال هر عنصر خارج سیستم، بازیگر نیست. بازیگران استفاده کنندگان نهایی و یا تولیدکنندگان ابتدایی اطلاعات هستند، بنابراین عناصر خارجی سیستم که تنها وظیفه انتقال اطلاعات را دارند و نقش **رسانه انتقال** را ایفا می‌کنند، نمی‌توانند به عنوان بازیگر در نظر گرفته شوند.

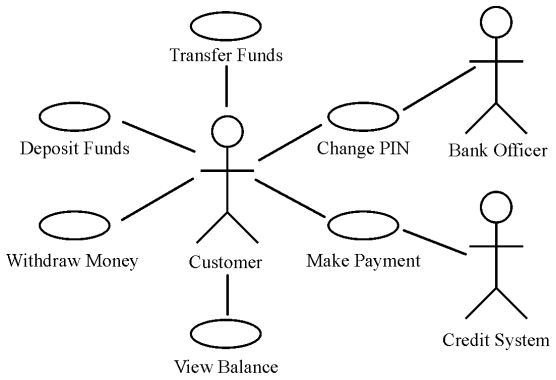
برای‌مثال اگر یک سنسور از طریق رسانه بی‌سیم، اطلاعاتی را به سیستم مرکزی ارسال می‌کند، رسانه بی‌سیم نمی‌تواند بازیگر این سیستم باشد، بلکه سنسور بازیگر آن خواهد بود. بنابراین هر بازیگر اگر استفاده‌کننده باشد نقطه انتها و اگر تولیدکننده اطلاعات باشد نقطه ابتدای آن ارتباط است.

در UML، بازیگران با آدمک‌هایی به شکل مقابل نشان داده می‌شوند:



Actor

نمونه‌ای از استفاده نمودار Use Case در شکل زیر نشان داده شده است: (سیستم ATM)



رابطه انجمنی بین Use Case و Actor

توجه: در نمودار مورد کاربرد، رابطه بازیگران با موارد کاربرد از نوع انجمنی است. خط ممتدی که مابین یک بازیگر و یک مورد کاربرد کشیده می‌شود، نشان‌دهنده رابطه انجمنی میان آن‌ها است و بدین معنا است که بازیگر و مورد کاربرد مذکور با یکدیگر در ارتباط هستند. جهت خط نیز، جهت ارتباط را نشان می‌دهد. بعضی از روابط انجمنی در نمودار مورد کاربرد، به صورت یک طرفه هستند، مانند زمانی که یک بازیگر بدون انتظار برای پاسخ، اطلاعاتی را در اختیار یک مورد کاربرد قرار می‌دهد. اما اغلب روابط انجمنی به صورت دو طرفه هستند، مانند زمانی که یک بازیگر به یک مورد کاربرد، متصل شده و از مورد کاربرد درخواست‌هایی دارد و مورد کاربرد نیز عملیات مورد نیاز را برای بازیگر به انجام رسانده و نتایج را به او بر می‌گرداند. رابطه انجمنی دو طرفه می‌تواند با فلش دو جهته نمایش داده شود، اما از آن جایی که اغلب رابطه‌ها دو طرفه هستند، غالبا از خط بدون فلش استفاده می‌گردد. شکل فوق گویای این مطلب می‌باشد.

توجه: هر بازیگر ممکن است چندین مورد کاربرد را اجرا کند و یک مورد کاربرد نیز می‌تواند توسط چندین بازیگر اجرا یا استفاده شود.

انواع روابط بین Use Caseها

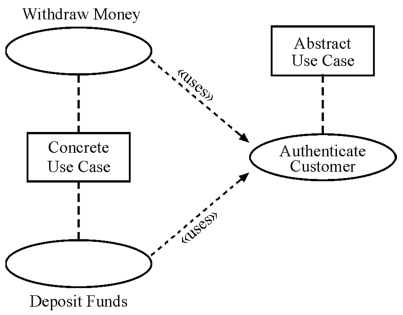
علاوه بر روابطی که بین بازیگرها با موردهای کاربرد وجود دارد ممکن است بین موارد کاربرد با یکدیگر نیز روابطی وجود داشته باشد که عبارتند از:

رابطه «شامل بودن» (Include) یا (Uses)

رابطه Include یا Uses، به یک Use Case، اجازه استفاده از عملیات مهیا شده توسط یک Use Case دیگر را می‌دهد. این نوع رابطه، برای مدل‌سازی برخی عملیاتی که بین دو یا تعداد بیشتری Use Case به طور مشترک استفاده می‌شوند، به کار می‌روند. به بیان دیگر اگر بدنه اصلی چند مورد کاربرد دارای بخش مشترکی باشند که به اندازه کافی بزرگ بوده، به طوری که بتوان آن را به عنوان یک مورد کاربرد مجزا در نظر گرفت، آنگاه می‌توان بخش مشترک مذکور را از درون موارد کاربرد متناظر خارج کرد و آن را به عنوان یک مورد کاربرد مجزا تعریف نمود. در این حالت موارد کاربرد اولیه و جدید به کمک رابطه «Include» به یکدیگر مرتبط شده و موارد کاربرد اولیه شامل مورد کاربرد جدید خواهند شد. بدین ترتیب به جای آن که یک عملیات، چندین بار در موارد مختلف تکرار شود، می‌تواند در قالب یک مورد کاربرد جداگانه، توسط بقیه موارد کاربرد مورد استفاده قرار بگیرد. بدیهی است مورد کاربرد جدید نمی‌تواند به طور مجزا دارای بازیگر باشد. زیرا این مورد کاربرد نمی‌تواند به طور مستقیم توسط یک بازیگر درخواست شود. بلکه یک بازیگر می‌تواند یکی از موارد کاربرد اولیه را درخواست نماید، در اینصورت حتما این مورد کاربرد نیز در حین اجرای آنها اجرا خواهد شد. در مثال ATM، در دو Use Case مربوط به برداشت وجه (Withdraw Money) از حساب و واریز وجه (Deposit Funds) به حساب، نیاز به شناسایی مشتری و PIN آنها، قبل از پیگیری عملیات است. بنابراین به جای تعریف پردازش شناسایی در هردوی این Use Caseها، می‌توان در یک Use Case مجزا به نام Authenticate Customer، این عملیات را تعریف کرد. هر زمان دیگری که یک Use Case نیاز به شناسایی یک کاربر دارد، می‌تواند از عملیات موجود در Use Case موجود به نام Authenticate Customer استفاده کند.

توجه: رابطه «شامل بودن» در UML به صورت فلش نقطه چینی است که از سمت موارد کاربرد اولیه به سمت مورد کاربرد جدید رسم می‌شود و با برچسب «Include» یا «Uses» نشان داده می‌شود.

مثال:



یک رابطه‌ی **Uses**

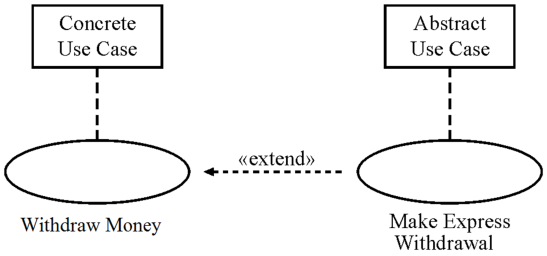
توجه: رابطه Include یا Uses در یک کلیشه «stereotype» نمایش داده می‌شود. کلیشه‌ها در نمودارهای UML توضیحات تکمیلی را ارئه می‌دهند و با نماد «» نمایش داده می‌شوند.

رابطه «توسعه دادن» (Extend)

یک رابطه‌ی extend به یک Use Case اجازه می‌دهد که به طور دلخواه عملیات مهیا شده توسط Use Caseهای دیگر را بسط دهد.

توجه: رابطه «توسعه دادن» در UML به صورت فلش نقطه چینی است که از سمت مورد کاربرد جدید به سمت مورد کاربرد اولیه رسم می‌شود و با برچسب «Extend» نشان داده می‌شود. در واقع، مورد کاربرد جدید، در موارد خاص به موارد کاربرد اولیه اضافه می‌گردد و باعث توسعه و گسترش آنها می‌شود. مورد کاربرد جدید در رابطه «Include» همواره در حین اجرای موارد کاربرد اولیه اجرا می‌شود، اما در رابطه «Extend»، مورد کاربرد جدید ممکن است در اثر برقراری شرایط خاصی اجرا گردد.

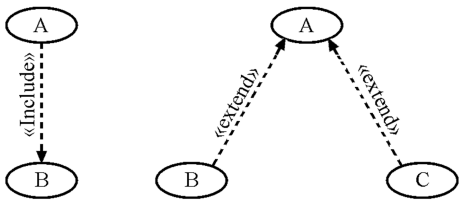
مثال:



یک رابطه‌ی extend

در مثال فوق، use case مربوط به برداشت پول (Withdraw Money) گاهی اوقات از عملیات موجود در use case برداشت سریع از حساب (Make Express Withdrawal) استفاده می‌نماید، اگر و فقط اگر کاربر یک میلیون ریال برداشت سریع را از use case برداشت پول (Withraw Money) درخواست کند.

توجه: در رابطه «Include» وقتی "A" اجرا شود، "B" هم **حتماً** اجرا می‌شود ولی در رابطه «extend» وقتی "A" اجرا شود، **ممکن** است "B" یا "C" یا هیچ‌کدام از Use Caseها اجرا نشوند.



توجه: در رابطه «extend» "B" یا "C" زمانی مورد استفاده "A" قرار می‌گیرد که "A" اجازه‌ی

توسعه، در نقاط قابل توسعه (Extension Point) را فراهم نماید.

ج) سناریونویسی

در این مرحله برای هر use case (مورد کاربرد یا نیاز) سناریو یا شرح حال نوشته می‌شود.

سناریو بر دو طبقه اصلی و فرعی می‌باشد:

سناریوی اصلی: بیان روال حرکت قدم به قدم کارها داخل یک use case از نقطه شروع تا رسیدن به یک نتیجه مطلوب (موفق). (حرکت از خود موجود و رسیدن به خود مطلوب).

مثال: بیان روال حرکت قدم به قدم، برای برداشت وجه موفق از یک حساب، مربوط به use case برداشت وجه در یک سیستم ATM .

سناریوی فرعی: بیان روال حرکت قدم به قدم کارها داخل یک use case از نقطه شروع تا رسیدن به یک نتیجه نامطلوب (ناموفق). (حرکت از خود موجود و رسیدن به خود نامطلوب).

مثال: بیان روال حرکت قدم به قدم، برای برداشت وجه ناموفق از یک حساب، مربوط به use case برداشت وجه در یک سیستم ATM .

توجه: برداشت وجه ناموفق، دلایل مختلفی دارد، همچون رمز عبور نادرست، عدم موجودی کافی در حساب، عدم موجودی اسکناس کافی در صندوق فیزیکی و ... .

توجه: هر use case، فقط و فقط یک سناریوی اصلی دارد و می‌تواند چندین سناریوی فرعی داشته باشد.

نحوه نمایش سناریوی اصلی و فرعی به دو روش زیر می‌باشد:

نوشتاری (متنی): در این روش سناریوی اصلی و فرعی به صورت متنی نوشته می‌شود.

گرافیکی (نموداری): در این روش سناریوی اصلی و فرعی به صورت نمودار، مدل‌سازی می‌شود.

توجه: Activity Diagram یا نمودار فعالیت و Swimlane Diagram یا نمودار خط شنا، جهت مدل‌سازی سناریوی اصلی و فرعی داخل یک use case (نیاز یا مورد کاربرد یا زیرسیستم) مورد استفاده قرار می‌گیرد. به بیان دیگر نمودار فعالیت یا نمودار خط شنا، جهت مدل‌سازی روال انجام کارها داخل یک use case مورد استفاده قرار می‌گیرد.

نمودار فعالیت (Activity Diagram)

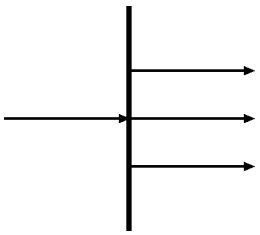
همانطور که پیش از این نیز گفتیم، فلوچارت ابزاری است که نحوه اجرای یک برنامه را به صورت گرافیکی در اختیار برنامه‌نویس قرار می‌دهد. نمودار فعالیت، نسخه توسعه‌یافته فلوچارت است که در UML مورد استفاده قرار می‌گیرد. بنابراین فلوچارت و نمودار فعالیت شباهت زیادی به یکدیگر دارند.

به هر سازمانی که مراجعه کنید با مفهوم روال انجام کار، سر و کار خواهید داشت. روال انجام کار، مراحل و نحوه انجام کارها را در سیستم مشخص می‌کند و به جزئیات آن نمی‌پردازد، به طوری که مشتری یا استفاده‌کننده سیستم، بدون نیاز به پرسش، از مراحلی که باید برای به انجام

رساندن کار خود طی کند، آگاهی می‌یابد. برای مثال، روال انجام کار مربوط به یک سناریوی موفق برای «برداشت وجه»، به این صورت است که: «ابتدا کارت در دستگاه کارت خوان وارد می‌شود، سپس رمز عبور وارد می‌شود، مبلغ درخواستی مشخص می‌شود، مبلغ درخواستی از حساب مورد نظر کسر می‌شود، مبلغ مورد نظر توسط صندوق آماده می‌گردد و در انتها پس از دریافت وجه، رسید تحویل می‌گردد». بنابراین روال انجام کار، یک سری فعالیت متوالی است که در هر سازمان، برای رسیدن به هدف مورد نظر طی می‌شود. از این رو در هر سازمانی می‌توان این سوال را مطرح کرد که «روال انجام کار یک عملیات خاص چگونه است؟». همانطور که گفتیم این روال انجام کار یا سناریو به صورت نوشتاری یا گرافیکی بیان می‌شود. نمودار فعالیت ابزار مناسبی برای نمایش گرافیکی گردش کار است. در نمودار فعالیت هر یک از فعالیت‌های مربوط به روال انجام کار، توسط نماد مستطیل گوشه گرد، جریان کار توسط نماد پیکان، نقاط تصمیم گیری توسط نماد لوزی، نقطه شروع روال انجام کار توسط نماد دایره توپر و نقاط پایان روال انجام کار توسط نماد دایره توپر مضاعف نشان داده می‌شود. نمادهای هم‌روندی و هم‌زمانی نیز در ادامه بررسی می‌گردد.

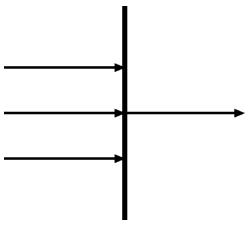
نماد هم‌روندی (Concurrency)

در نمودار فعالیت، هم‌روندی نیز قابل مدل‌سازی است. هم‌روندی در فلوچارت وجود ندارد، اما در نرم‌افزارهای امروزی زیاد مورد استفاده قرار می‌گیرد. برای مثال وقتی شما چند کار را با هم انجام می‌دهید در آنجا هم‌روندی رخ داده است.



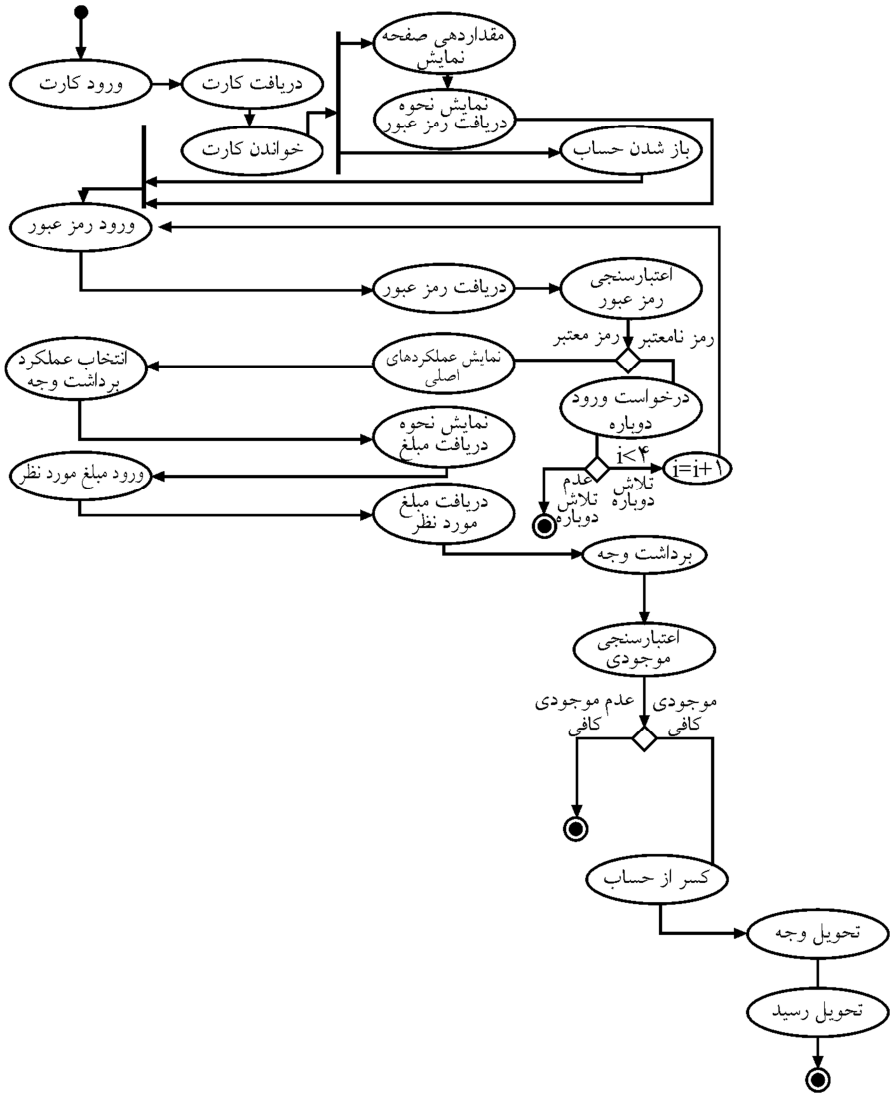
برای مدل‌سازی هم‌روندی از شکل یک چنگال، مطابق شکل مقابل استفاده می‌شود. به طوری که یک مسیر به میله هم‌روندی وارد شده و سپس مسیرهایی که باید به صورت موازی با یکدیگر به اجرا درآیند، از این میله خارج می‌شوند.

نماد هم‌زمانی (Synchronization)



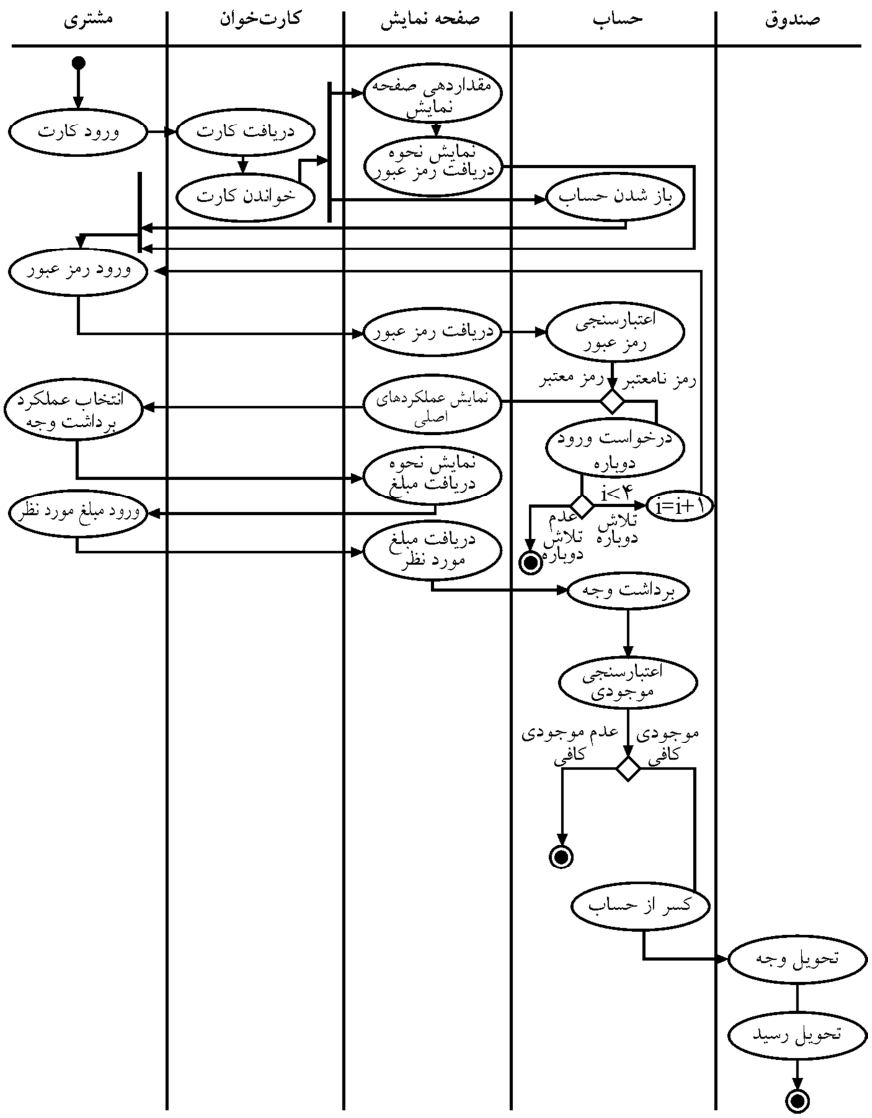
مفهوم هم‌زمانی با هم‌روندی متفاوت است و برای ادغام مسیرهای اجرایی هم‌روند در نقطه انتهایی استفاده می‌شود. در شکل مقابل این نماد نشان داده شده است.

مثال: مدل‌سازی گردش کار مورد کاربرد «برداشت وجه» توسط نمودار فعالیت به صورت زیر است:



نمودار بخش‌بندی یا خط شنا (Swimlane Diagram)

نمودار بخش‌بندی، شکل توسعه یافته نمودار فعالیت است که در UML مورد استفاده قرار می‌گیرد. بنابراین نمودار فعالیت و نمودار بخش‌بندی شباهت زیادی به یکدیگر دارند. در نمودار بخش‌بندی علاوه بر مواردی که نمودار فعالیت نشان می‌دهد، عامل هر فعالیت نیز نشان داده می‌شود.



توجه: در شکل فوق هم‌روندی فعالیت‌های مقداردهی صفحه نمایش، نمایش نحوه دریافت رمز عبور و بازشدن حساب توسط **نماد هم‌روندی** و همچنین هم‌زمانی نقاط انتهایی آنها نیز توسط **نماد هم‌زمانی** نشان داده شده است.

د) کشف کلاس‌های همکار داخل هر use case یا مورد کاربرد

پس از شناسایی موارد کاربرد و سناریونویسی برای هر یک از موارد کاربرد، زمان تعریف کلاس‌ها، صفات، متدها و ارتباطات میان کلاس‌های همکار برای هر یک از موارد کاربرد می‌رسد.

برای شناسایی کلاس‌ها، صفات، متدها و ارتباطات میان کلاس‌ها برای هر یک از موارد کاربرد، از تکنیکی موسوم به CRC که سرواژه‌ی عبارت Class – Responsibility Collaborator و به معنی «مدل همکاری مسئولیت‌های کلاس‌ها» می‌باشد، استفاده می‌شود. مدل‌سازی CRC روشی ساده جهت تعیین و سازماندهی کلاس‌های داخل هر مورد کاربرد است. در این روش به هر کلاس یک کارت CRC اختصاص داده می‌شود که شامل سه بخش کلی زیر است:

نام کلاس

مسئولیت‌های کلاس(Responsibilities)

- صفات کلاس
- متدهای کلاس

همکاران کلاس(Collaborators)

توجه: کلاس‌ها یا به تنهایی از عهده انجام مسئولیت‌های خود بر می‌آیند و یا از طریق همکاری با کلاس‌های همکار خود از عهده انجام مسئولیت‌های خود بر می‌آیند. بنابراین اگر کلاسی همکارانی دارد، باید در بخش مربوط به همکاران نوشته شود.

توجه: برای کشف کلاس‌های همکار داخل هر use case، از سناریوی اصلی (نوشتاری یا نموداری) هر use case استفاده می‌گردد. برای این منظور، **اسامی** موجود داخل هر use case مورد جستجو قرار می‌گیرند. از آنجا که نیازها یا موارد کاربرد مشتری به تدریج و در طی تکرار مشخص می‌شوند و همچنین از آنجا که کلاس‌های همکار داخل هر مورد کاربرد یا نیاز هستند، بنابراین با تکامل و کامل شدن نیازها یا موارد کاربرد، به تبع کلاس‌های همکار هر مورد کاربرد نیز کامل می‌شود. بنابراین تشخیص تمام کلاس‌های برنامه، در همان ابتدای کار تقریبا غیرممکن است، در واقع در طول پروژه و با گذشت زمان تحلیل‌گر متوجه نیازها و به تبع کلاس‌های جدیدی می‌شود که در ابتدای کار نیاز به آن‌ها چندان محسوس نبوده است. بنابراین می‌توان گفت روند تشخیص نیازها یا موارد کاربرد و به تبع کلاس‌های همکار به عنوان مرتفع‌کننده نیازها یا موارد کاربرد، یک فرآیند تکرارشونده است و در طول فرآیند تولید نرم‌افزار کامل و کامل‌تر می‌شوند.

انواع روابط بین کلاس‌های همکار در مدل CRC

سه نوع رابطه مختلف بین کلاس‌های همکار در مدل CRC وجود دارد:

رابطه آگاه است از (has knowledge of)

دو انسانی که همدیگر را می‌شناسند و امکان گفتگو و تبادل پیام با هم دارند، رابطه انجمنی با هم دارند. در رابطه انجمنی یک شیء بطور ساده درباره شیء دیگر می‌داند به همان طریقی که یک فرد ممکن است فرد دیگری را بشناسد. یک برنامه کامپیوتری شیءگرا از اجتماع تعدادی کلاس ایجاد شده است، کلاس‌های همکار **بدون** رابطه جزء و کل و هم‌هدف در یک بخش مشترک از برنامه، کلاس‌های همکار **با** رابطه جزء و کل و هم‌هدف در یک بخش مشترک از برنامه و

کلاس‌های غیرهمکار در دو بخش مختلف از برنامه.

توجه: رابطه‌ای که میان کلاس‌های همکار بدون رابطه جزء و کل جهت گفتگوی میان اشیاء آنها وجود دارد، رابطه انجمنی است.

توجه: کلاس‌های همکار **با** رابطه جزء و کل و هم‌هدف در یک بخش مشترک از برنامه، جلوتر شرح داده می‌شود.

کلاس‌های همکار بدون رابطه جزء و کل، جهت انجام وظایف خود از طریق مکانیزم پیام به گفتگو با یکدیگر می‌پردازند. در یک بیان ساده، هرگاه میان دو شیء بدون رابطه جزء و کل گفتگو باشد، کلاس‌های این دو شیء هم باهم همکار هستند و هم رابطه انجمنی میان آنها برقرار است.

هرگاه مابین دو کلاس را**بطه انجمنی** مطرح باشد، یعنی تبادل پیام مابین برخی متدهای اشیای دو کلاس صورت گیرد، آنگاه در این شرایط **رابطه آگاهی** خواهیم داشت. مانند ارسال پیام از شیء بازیکن به شیء توپ پس از انجام متد شوت زدن مربوط به شیء بازیکن، برای تغییر مختصات توپ توسط صدا زدن متد تغییر مختصات توپ مربوط به شیء توپ. یعنی شیء بازیکن، باید در هنگام شوت‌زدن، مختصات توپ را توسط صدازدن متد تغییر مختصات توپ، تغییر دهد.

توجه: نحوه مدل‌سازی رابطه آگاهی یا رابطه انجمنی جلوتر شرح داده خواهد شد.

رابطه بخشی است از (is part of)

هرگاه رابطه‌ای مابین یک واحد کل و تعدادی واحد جزء مطرح باشد، یعنی یک کلاس کل از همکاری تعدادی کلاس جزء تشکیل شده باشد، آنگاه در این شرایط رابطه بخشی خواهیم داشت. این رابطه خود بر دو نوع **تجمع** و **ترکیب** می‌باشد. نحوه مدل‌سازی رابطه تجمع و رابطه ترکیب جلوتر شرح داده خواهد شد.

رابطه وابسته است به (depends upon)

هر گاه مابین دو کلاس رابطه وابستگی مطرح باشد، رابطه وابستگی خواهیم داشت. در واقع هرگاه تغییرات مقادیر صفات یک شیء، روی مقادیر صفات شیء دیگری تأثیر بگذارد، این دو شیء به هم وابسته هستند. در واقع این رابطه زمانی رخ می‌دهد که مقادیر صفاتی از یک کلاس به مقادیر صفاتی از یک کلاس دیگر وابسته باشد. برای مثال در برنامه کامپیوتری فوتبال، مختصات شیء سر بازیکن، دست بازیکن و پای بازیکن، همواره باید به مختصات شیء بدنه بازیکن وابسته باشد (مگر خشونت در بازی زیاد باشد!). در واقع مختصات سر بازیکن، دست بازیکن و پای بازیکن، همواره باید با تغییرات مختصات بدنه بازیکن تغییر کند. وگرنه در هنگام حرکت بازیکن، سر بازیکن، دست بازیکن و پای بازیکن از بدنه بازیکن جلو یا عقب می‌افتند!

توجه: نحوه مدل‌سازی رابطه وابستگی جلوتر شرح داده خواهد شد.

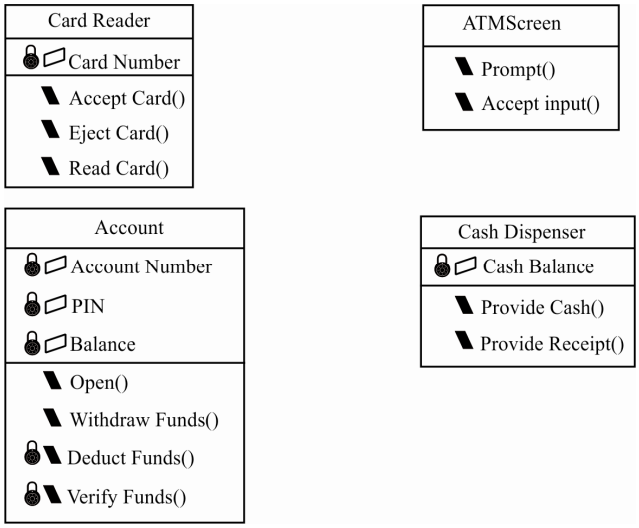
مثال: عمل برداشت وجه در یک سیستم ATM یک نیاز یا مورد کاربرد (use case) است.

«برداشت وجه» یک نیاز یا مورد کاربرد است و این نیاز باید توسط نرم‌افزار برآورده شود.

فرض کنید قرار است کمک کنید تا مراسم عروسی دوستتان به شکلی مطلوب و باشکوه برگزار گردد. بنابراین یک نفر مسئولیت تدارک شام را بر عهده می‌گیرد، شخص دیگری مسئولیت هماهنگی فضای مهمانی و شخص دیگری مسئولیت سفارش کیک را بر عهده می‌گیرد، در واقع این افراد با هم در یک کار گروهی همکاری و همیاری می‌کنند، تا این مراسم به عالی‌ترین شکل ممکن برگزار گردد. حال به وادی نرم‌افزار برگردید، یک نیاز به نام «برداشت وجه» برای مشتری وجود دارد.

در اینجا هم برای مرتفع کردن این نیاز باید کلاس‌هایی مسولیت‌هایی را بر عهده بگیرند و با یکدیگر همکاری کنند تا نیاز «برداشت وجه» بر طرف گردد.

شکل زیر کلاس‌های همکار داخل مورد کاربرد «برداشت وجه» را نمایش می‌دهد:



کلاس‌های همکار داخل مورد کاربرد برداشت وجه

توجه: به کلاس‌های همکار، **کلاس‌های مشارکت‌کننده** نیز گفته می‌شود.

توجه: پس از اتمام مدل‌سازی CRC، کارت‌های CRC با سناریوی اصلی هر use case یا مورد کاربرد تطابق داده می‌شود. تا مشخص شود که آیا تمام کلاس‌های مربوط به use case درست شناسایی شده‌اند یا خیر. که در صورت نیاز، می‌بایست تغییرات لازم بر روی کارت‌های CRC اعمال گردد.

در گام بعدی ارتباطات ایستای میان کلاس‌های همکار توسط نمودار کلاس مدل‌سازی می‌شود.

هـ) مدل‌سازی ارتباطات ایستای میان کلاس‌های همکار

پس از شناسایی کلاس‌ها و کلاس‌های همکار برای هر یک از موارد کاربرد در مدل CRC، زمان مدل‌سازی نموداری و ساختاری توسط نمودار کلاس می‌رسد. در واقع یکی از اهداف

کارت‌های CRC کشف زودهنگام و کم‌هزینه کاستی‌هاست، پاره‌کردن چند تکه کاغذ و کارت CRC به مراتب کم‌هزینه‌تر از سازماندهی مجدد ساختار نمودارهای کلاس است، بنابراین مدل CRC شروع کم‌هزینه مدل‌سازی نموداری و ساختاری توسط نمودار کلاس است.

توجه: Class Diagram یا نمودار کلاس جهت مدل‌سازی ارتباطات ایستای میان کلاس‌های همکار داخل یک case use مورد استفاده قرار می‌گیرد.

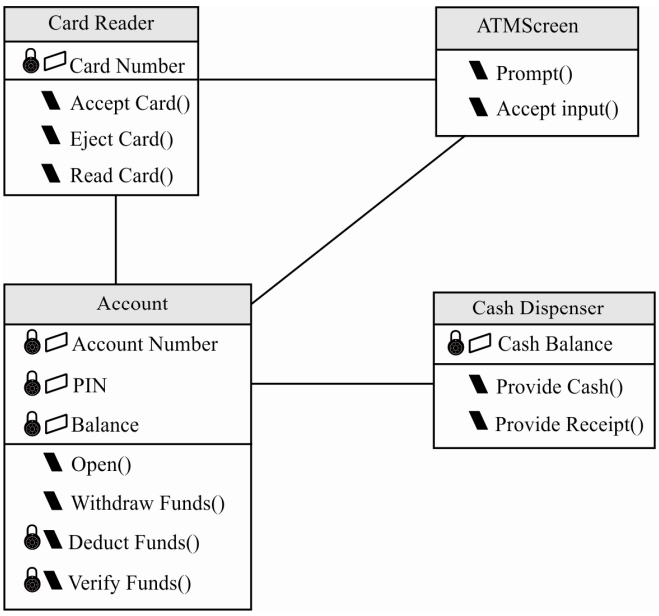
توجه: نمودار کلاس یک مدل‌سازی ساختاری محسوب می‌گردد.

توجه: در فعالیت مدل تحلیل، جزئیات مربوط به کلاس‌ها، شامل جزئیات دقیق صفات و متدها، مشخص نمی‌گردد، بیان این جزئیات تا بخش طراحی مولفه از مدل طراحی به تأخیر می‌افتد.

نمودار کلاس (Class Diagram)

نمودار کلاس به عنوان منبع اصلی تولید کد محسوب می‌گردد. هر کلاس دارای حداقل یک مسئولیت خواهد بود که در سلسله مراحل پالایش کلاس، این مسئولیت‌ها به مجموعه‌ای از صفات و متدها بدل می‌گردند. به نحوی که صفات و متدها بتوانند از عهده مسئولیت‌های کلاس برآیند.

مثال: شکل زیر مدل‌سازی ارتباطات ایستای میان کلاس‌های همکار داخل مورد کاربرد «برداشت وجه» را نمایش می‌دهد:



مدل‌سازی ارتباطات ایستای میان کلاس‌های همکار داخل مورد کاربرد برداشت وجه

توجه: نمودار کلاس فوق، مدل‌سازی ارتباطات ایستای مابین کلاس‌هایی را نشان می‌دهد که مورد کاربرد «برداشت وجه» را به انجام می‌رسانند. این کار با چهار کلاس انجام شده است.

- کارت‌خوان (Card Reader)
- حساب (Account)
- صفحه نمایش ATM (ATM Screen)
- صندوق (Cash Dispenser)

اجزای نمودار کلاس

نمودار کلاس، برای نمایش کلاس‌ها، صفات کلاس‌ها، متدهای کلاس‌ها و نیز روابط مابین کلاس‌ها مورد استفاده قرار می‌گیرد. یک نمودار کلاس از دو جزء اصلی تشکیل می‌شود که عبارتند از: کلاس‌ها و روابط. در ادامه به تشریح جزئیات هر یک از دو جزء مذکور می‌پردازیم.

کلاس‌ها

در یک نمودار کلاس، هر کلاس با مستطیلی نشان داده می‌شود که شامل سه بخش زیر است:

بخش اول: نام کلاس

نام یک کلاس، عنوان منحصر به فردی است که به آن نسبت داده می‌شود و تا حدودی نوع و عملکرد کلاس را توصیف می‌کند. بخش نام کلاس شامل نام کلاس و در صورت نیاز ذکر Stereotype است، برای معرفی بیشتر ماهیت کلاس.

بخش دوم: صفات کلاس (Attributes)

صفات یک کلاس در برگیرنده مشخصات ساختاری کلاس است. به بیان دیگر یک صفت قطعه‌ای از اطلاعاتی است که با یک کلاس مرتبط می‌باشد.

برای تعریف یک صفت لازم است نام و نوع آن مشخص شود. قابلیت رویت یک صفت نیز باید در زمان تعریف آن مشخص گردد. به عبارت دیگر، باید مشخص شود که اشیای کدام کلاس‌ها مجاز به رویت صفت مذکور هستند.

UML چهار نوع قابلیت رویت برای یک صفت متصور است که عبارتند از:

عمومی (Public) با نماد «+»: اگر قابلیت رویت یک صفت از نوع «عمومی» باشد، آنگاه تمام اشیاء(حتی اشیاء کلاس‌های متفاوت) حق دسترسی به صفت مذکور را دارند.

خصوصی (Private) با نماد «-»: اگر قابلیت رویت یک صفت از نوع «خصوصی» باشد، آنگاه حق دسترسی، فقط به صاحب آن صفت، یعنی خود شیء، محدود بوده و اشیای دیگر حق دسترسی به آن را ندارند.

محافظت شده (Protected) با نماد «#»: اگر قابلیت رویت یک صفت از نوع «محافظت شده» باشد، آنگاه علاوه بر شیء صاحب آن صفت، سایر اشیای هم کلاس با شیء مذکور به همراه کلیه

اشیای کلاس‌های فرزند نیز اجازه دسترسی به آن صفت را دارند.

بسته (Package) با نماد «⌐»: اگر قابلیت رویت یک صفت از نوع «بسته» باشد، آنگاه کلیه اشیای کلاس‌های هم بسته با کلاس شیء صاحب آن صفت، اجازه دسترسی به آن ویژگی را دارند.

نکته قابل توجه دیگر این است که تعدادی از صفت‌ها و متدها، علاوه بر نمادهای فوق می‌توانند از قفل‌های کوچکی در سمت چپشان استفاده کنند. نماد قفل، یک صفت یا متد خصوصی (Private) را نشان می‌دهد. صفات و متدهای خصوصی فقط می‌توانند ار طریق شیءای که شامل آنها است قابل دستیابی باشند.

مثال: کلاس حساب (Account) شامل سه صفت زیر است:

- شماره حساب (Account Number)
- پین (PIN)
- موجودی (Balance)

بخش سوم: متدهای کلاس (Operations)

متدهای یک کلاس نیز شامل قابلیت‌های عملکردی کلاس مذکور است. یک کلاس می‌تواند شامل تعدادی متد باشد، که اشیای آن کلاس می‌توانند آن‌ها را انجام دهند. هر متد مانند یک صفت با یک نام، مشخص می‌شود. علاوه بر آن، یک متد می‌تواند دارای یک سری آرگومان ورودی باشد. هر متد می‌تواند دارای خروجی نیز باشد که در این صورت لازم است نوع آن مشخص گردد. علاوه بر این، متدها، همانند صفات دارای قابلیت رویت هستند. قوانین و تعاریف انواع قابلیت رویت در صفات برای متدها نیز صادق است.

مثال: کلاس حساب (Account) شامل چهار متد زیر است:

- باز کردن (Open)
- برداشت وجه (Withdraw Funds)
- کسر از موجودی (Deduct Funds)
- تأیید موجودی (Verify Funds)

برای مدل‌سازی یک کلاس لازم است اجزای آن در یک قالب نمادین قرار گیرند. نماد یک کلاس از سه بخش مستطیل شکل تشکیل می‌شود. هر یک از این بخش‌ها به یکی از اجزای سه گانه کلاس اختصاص می‌یابد. همانطور که در شکل زیر مشاهده می‌نمایید، نام کلاس در اولین بخش مستطیلی شکل نماد کلاس قرار می‌گیرد. Stereotype نیز، در صورت نیاز در همین بخش و در بالای نام کلاس نشان داده می‌شود. مستطیل بخش میانی نماد کلاس، جهت درج و تعریف صفات کلاس، مورد استفده قرار می‌گیرد. ترتیب قرار گرفتن صفات مهم نیست. در بخش تحتانی نماد کلاس، متدهای کلاس تعریف می‌شوند. ترتیب قرار گرفتن متدها نیز مهم نیست. در فرآیند تعریف یک کلاس، ذکر بخش نام، ضروری است، اما دو بخش دیگر در صورت لزوم و وجود

اطلاعات مربوطه ذکر می‌شوند.

Account
- AccountNumber : long - PIN : int - Balance : long
+Open() + WithdrawFunds() + DeductFunds() + VerifyFunds()

روابط نمودار کلاس

همانطور که پیش از این نیز بیان شد، یکی از اجزای اصلی نمودار کلاس، روابط موجود بین کلاس‌ها است. روابط بین کلاس‌ها، مانند روابط بین انسان‌هاست. به عنوان مثال، اگر من و شما بخواهیم با یکدیگر کار کنیم لازم است با یکدیگر ارتباط برقرار نماییم. این ارتباط ممکن است از طرق مختلفی نظیر تلفن و پست الکترونیک انجام پذیرد که هرکدام در شرایط خاصی کاربرد دارد. بنابراین، در هر ارتباط، لازم است نوع و شرایط ارتباط به درستی مشخص گردد. از این رو UML، نمادهای خاصی را برای مشخص کردن نوع و نحوه این روابط ارائه می‌دهد. در این بخش با انواع روابط مابین کلاس‌ها، تعاریف و نمادهای آن آشنا می‌شوید.

مدل‌سازی انواع روابط بین کلاس‌های همکار در نمودار کلاس

۱- مدل‌سازی رابطه انجمنی (Association)

دو انسانی که همدیگر را می‌شناسند و امکان گفتگو و تبادل پیام با هم دارند، رابطه انجمنی با هم دارند. در رابطه انجمنی یک شیء بطور ساده درباره شیء دیگر می‌داند به همان طریقی که یک فرد ممکن است فرد دیگری را بشناسد. یک برنامه کامپیوتری شیءگرا از اجتماع تعدادی کلاس ایجاد شده است، کلاس‌های همکار بدون رابطه جزء و کل و هم‌هدف در یک بخش مشترک از برنامه، کلاس‌های همکار با رابطه جزء و کل و هم‌هدف در یک بخش مشترک از برنامه (جلوتر شرح داده می شود) و کلاس‌های غیرهمکار در دو بخش مختلف از برنامه، رابطه‌ای که میان کلاس‌های همکار بدون رابطه جزء و کل وجود دارد، رابطه انجمنی است.

کلاس‌های همکار بدون رابطه جزء و کل، جهت انجام وظایف خود از طریق مکانیزم پیام به گفتگو با یکدیگر می‌پردازند. در یک بیان ساده، هرگاه میان دو شیء بدون رابطه جزء و کل گفتگو باشد، کلاس‌های این دو شیء هم باهم همکار هستند و هم رابطه انجمنی میان آنها برقرار است.

هرگاه مابین دو کلاس را**بطه انجمنی** مطرح باشد، یعنی تبادل پیام مابین برخی متدهای دو

کلاس صورت گیرد، آنگاه در این شرایط **رابطه آگاهی** خواهیم داشت. مانند ارسال پیام از شیء بازیکن به شیء توپ پس از انجام متد شوت زدن مربوط به شیء بازیکن، برای تغییر مختصات توپ توسط صدا زدن متد تغییر مختصات توپ مربوط به شیء توپ. یعنی شیء بازیکن، باید در هنگام شوت‌زدن، مختصات توپ را توسط صدازدن متد تغییر مختصات توپ، تغییر دهد. به عنوان مثالی دیگر، کلاس Account به کلاس ATM Screen توسط یک خط ممتد وصل شده است زیرا هر دو مستقیماً باهم **رابطه انجمنی و تبادل پیام** دارند. Card Reader به Cash Dispenser وصل نشده است زیرا این دو با هم ارتباطی ندارند. اشیاء کلاس‌های همکار بدون رابطه جزء و کل از طریق صدازدن متدهای عمومی یکدیگر اقدام به گفتگو و تبادل پیام می‌کنند و این یعنی شناخت و آگاهی از یکدیگر.

توجه: رابطه انجمنی به عنوان رابطه «has knowledge of» نیز شناخته می‌شود.

هر رابطه انجمنی از سه قسمت اصلی تشکیل می‌شود که عبارتند از:

۱) کلاس‌های شرکت‌کننده در رابطه

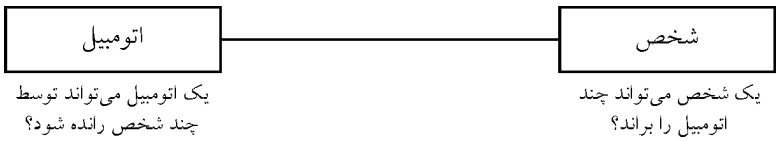
۲) خط ممتد رابطه که بین دو کلاس ترسیم می‌شود.

۳) نام رابطه که در وسط خط رابطه نوشته می‌شود و بیان‌کننده هدف رابطه است.

در مورد روابط انجمنی، بین کلاس‌ها، سوالات مهمی مطرح می‌شود. برای مثال، در مورد رابطه انجمنی شخص و اتومبیل، سوالات زیر مطرح است:

«یک شخص می‌تواند چند اتومبیل را براند؟»، «پاسخ: *..۱»

«یک اتومبیل می‌تواند توسط چند شخص رانده شود؟»، «پاسخ: *..۱»



UML برای پاسخگویی به چنین سوالاتی، مشخصه‌ای با نام **تعدد** (Multiplicity) را برای هر یک از کلاس‌های طرفین رابطه ارائه نموده است. تعدد اصطلاحی است که تعداد اشیای شرکت‌کننده در رابطه انجمنی را مشخص می‌کند. اعدادی که در پاسخ به این دو سوال داده می‌شود، مبین تعدد طرفین است. شیوه‌های مختلفی جهت تعیین تعدد وجود دارد، اما متداول‌ترین شیوه، بیان محدوده تعداد اشیای شرکت‌کننده در هر طرف رابطه است که به صورت «حداقل .. حداکثر» نشان داده می‌شود. در تعیین تعدد نمی‌توان از اعداد اعشاری استفاده نمود، بلکه فقط از اعداد صحیح مثبت استفاده می‌شود. در مواقعی که حداکثر تعداد اشیای رابطه مشخص نباشد و یا اینکه هیچ حد بالایی برای آن وجود نداشته باشد، از نماد «*» برای این منظور استفاده می‌شود. به عبارت دیگر، نماد «*» بیانگر عدم وجود یک حد بالا و پایین مشخص است و لذا در این حالت، تعداد صفر یا بیشتر شیء می‌توانند در رابطه مشارکت نمایند. در صورتی که حد بالا و پایین یکی باشد، می‌توان به اختصار فقط یکی از حدود را نمایش داد. برای مثال حد «۱..۱» را می‌توان به اختصار به صورت «۱» نمایش داد.

با توجه به مطالبی که ارائه شد، می‌توان انواع تعدد را در چهار دسته زیر طبقه‌بندی نمود:

۱- «یک به یک»

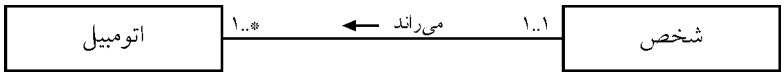
۲- «یک به چند»

۳- «چند به یک»

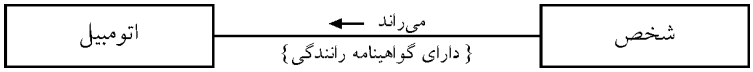
۴- «چند به چند»

توجه: رابطه «یک به چند» معادل رابطه «چند به یک» است، با این تفاوت که جهت رابطه در آنها عکس یکدیگر است.

مثال: تعدد رابطه انجمنی بین شخص و اتومبیل به صورت زیر است:

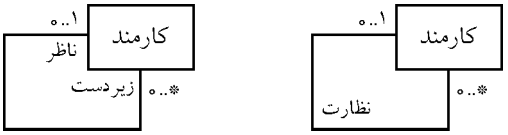


یک رابطه انجمنی ساده می‌تواند دارای شروط و محدودیت‌های خاص خود باشد تا صحت عملکرد رابطه تضمین گردد. برای روشن شدن این مطلب، در شکل زیر، رابطه شخص با اتومبیل نشان داده شده است. سوالی که اینجا مطرح است این است که آیا هر شخصی مجاز به راندن اتومبیل است؟ طبق قوانین راهنمایی و رانندگی فقط افراد دارای گواهینامه می‌توانند رانندگی کنند. بنابراین، محدودیتی باید در رابطه بین شخص و اتومبیل اعمال گردد. برای نمایش محدودیت‌ها نماد «{}» مورد استفاده قرار می‌گیرد. در شکل زیر محدودیت «{دارای گواهینامه رانندگی}» در نظر گرفته شده است. لازم به ذکر است که این محدودیت باید در طرف شخص قرار گیرد تا قبل از آنکه ارتباطی برقرار شود، محدودیت‌ها بررسی شده و نسبت به برقراری ارتباط تصمیم گیری شود.

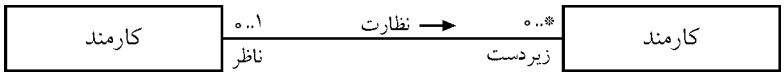


مدل‌سازی رابطه خودانجمنی (Self-Association)

در UML، یک رابطه‌ی خودانجمنی، رابطه‌ای است که یک کلاس با خودش دارد. لذا اگر اشیای یک کلاس با یکدیگر در ارتباط باشند، ارتباط مذکور از نوع خودانجمنی است. به عبارت دیگر، اگر ابتدا و انتهای یک رابطه انجمنی به یک کلاس اشاره داشته باشد، در این صورت رابطه‌ی مذکور از نوع خودانجمنی خواهد بود. در شکل زیر، دو مثال برای رابطه بازتابی ارائه شده است. توجه داشته باشید که در صورت مشخص کردن نقش دو طرف رابطه، ذکر نام رابطه، ضروری نیست.



برای سادگی در نحوه خواندن رابطه خودانجمنی توصیه می‌کنیم، رابطه را به شکل خطی ایجاد کنید، سپس رابطه را بخوانید:



شکل فوق گویای رابطه خودانجمنی (Self Association) است. به این معنی که یک کارمند هیچ زیردستی ندارد (چون مدیر نیست) یا چندین زیردست دارد (چون مدیر است) همچنین یک کارمند هیچ ناظری ندارد (چون مدیرکل است) یا یک ناظر دارد (چون مدیر کل نیست).

به کمک رابطه انجمنی ساده که در این قسمت به معرفی آن پرداختیم، امکان مدل‌سازی مشخصات خاص برخی روابط وجود ندارد. برای‌مثال رابطه بین «بازیکن» و «تیم» با رابطه بین «شخص» و «اتومبیل» تفاوت‌هایی دارد، در رابطه بین «بازیکن» و «تیم» مابین بازیکن و تیم رابطه جزء و کل مطرح است، به عبارت دیگر مابین کلاس‌های همکار یعنی بازیکن و تیم رابطه جزء و کل برقرار است. یعنی بازیکن بخشی یا جزئی از تیم محسوب می‌شود. در حالی در رابطه بین «شخص» و «اتومبیل» مابین شخص و اتومبیل رابطه جزء و کل مطرح نیست، به عبارت دیگر مابین کلاس‌های همکار یعنی شخص و اتومبیل رابطه جزء و کل برقرار نیست. یعنی شخص بخشی از اتومبیل و یا اتومبیل بخشی از شخص نیست، یعنی همانطور که گفتیم مابین شخص و اتومبیل رابطه انجمنی ساده برقرار است. بنابر مطلب بیان شده امکان نمایش رابطه جزء و کل مانند رابطه بین «بازیکن» و «تیم» توسط رابطه انجمنی ساده وجود ندارد.

برای پوشش چنین مشخصه‌ای از یک رابطه انجمنی، UML دو نوع رابطه به نام‌های تجمع و ترکیب ارائه نموده است. رابطه‌ی تجمع، نوع خاصی از رابطه انجمنی است که علاوه بر دارا بودن تمام ویژگی‌های رابطه انجمنی، یک سری ویژگی‌های مخصوص به خود نیز دارد. رابطه ترکیب نیز، نوع خاصی از رابطه تجمع است، لذا علاوه بر دارا بودن تمام ویژگی‌های رابطه تجمع، یک سری ویژگی‌های خاص خود را نیز دارد. در ادامه به معرفی رابطه انجمنی پیشرفته یا رابطه بخشی می‌پردازیم.

۲- مدل‌سازی رابطه انجمنی پیشرفته یا رابطه بخشی

همانطور که پیش‌تر از نیز گفتیم، هرگاه رابطه‌ای مابین یک واحد کل و تعدادی واحد جزء مطرح باشد، یعنی یک کلاس کل از همکاری تعدادی کلاس جزء تشکیل شده باشد، آنگاه در این شرایط رابطه انجمنی پیشرفته یا رابطه بخشی خواهیم داشت. رابطه انجمنی پیشرفته یا رابطه بخشی به دو رابطه تجمع و ترکیب تقسیم می‌گردد. در ادامه به تشریح دو رابطه مذکور می‌پردازیم.

الف) مدل‌سازی رابطه تجمع (Aggregation)

همانطور که پیش‌تر از این نیز گفتیم، رابطه تجمع رابطه‌ای مابین یک واحد کل و تعدادی واحد جزء است، یعنی یک کلاس کل از همکاری تعدادی کلاس جزء تشکیل شده باشد. برای مثال یک شیء کل «تیم» را در نظر بگیرید، که خود از چندین شیء جزء دیگر مانند اشیاء بازیکنان تشکیل شده است. به عنوان مثالی دیگر، یک شیء کل «اتومبیل» را در نظر بگیرید، که خود از

چندین شیء جزء دیگر مانند شیء فرمان، موتور، چرخ و غیره تشکیل شده است. به چنین روابطی تجمع گفته می‌شود. در رابطه تجمع، به طرف جزء، عضو (Member) و به طرف کل (Whole)، تجمع گفته می‌شود.

در رابطه **انجمنی‌ساده**، مابین طرفین همکار رابطه، رابطه جزء و کل مطرح نیست، در حالی که در رابطه **تجمع**، مابین طرفین همکار رابطه، رابطه جزء و کل مطرح است و از **تجمع** و سازماندهی اجزاء، یک موجودیت کامل‌تر ساخته می‌شود. برای مثال، برای ساخته شدن یک شیء کل «تیم» لازم است اشیاء جزء مختلفی با یکدیگر **مجتمع** شوند. یا برای مثالی دیگر، برای ساخته شدن یک شیء کل «اتومبیل» لازم است اشیاء جزء مختلفی با یکدیگر **مجتمع** شوند.

توجه: رابطه تجمع به عنوان رابطه «is part of» نیز شناخته می‌شود.

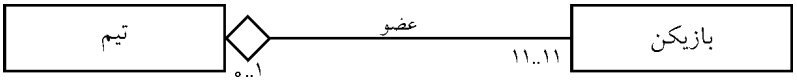
مراحل لازم جهت مدل‌سازی یک رابطه تجمع عبارتند از:

۱- ترسیم خط ممتد از سمت جزء به سمت کل

۲- ترسیم لوزی توخالی در انتهایی از خط رابطه که کلاس کل قرار دارد

۳- تعیین تعدد، نقش و محدودیت‌های طرفین رابط در صورت نیاز

شکل زیر نمونه‌ای از یک رابطه تجمع میان شیء کل «تیم» و اشیاء جزء «بازیکن» را نشان می‌دهد.



توجه: یک بازیکن، ممکن است عضو یک تیم باشد و یا نباشد، بدین معنی که یک بازیکن می‌تواند مستقل از یک تیم وجود داشته باشد، بنابراین از نماد ۱ .. ۰ استفاده شده است.

در رابطه تجمع، حیات شیء جزء به حیات شیء کل وابسته نیست. به بیان دیگر حیات شیء جزء و شیء کل به هم تقدم و تاخر دارند. یعنی ممکن است شیء جزء موجود باشد، بدون اینکه شیء کل ایجاد گردد و موجود باشد. در رابطه تجمع اگر شیء کل از بین برود، اشیاء جزء، همچنان به حیات خود ادامه می‌دهند. زیرا از ابتدای کار، اشیا جزء، توسط برنامه کامپیوتری ایجاد شده‌اند و فقط با پایان برنامه کامپیوتری اشیاء جزء از بین می‌روند و نه به هنگام از بین رفتن شیء کل. در بازی کامپوتری فوتبال، اشیا جزء یا همه بازیکن‌های تیم ملی توسط برنامه کامپیوتری ایجاد می‌شوند، اما شما بر حسب سلیقه ۱۱ نفر از آن‌ها را انتخاب می‌کنید و تیم خود یا شیء کل را می‌سازید. اگر هم تیم را منحل کنید، اشیاء بازیکن که توسط برنامه کامپیوتری از همان ابتدای اجرای برنامه ایجاد شده‌اند، همچنان هستند، و کافی است در یک دسته‌بندی دیگر تیم جدیدی را ایجاد نمایید. یا مانند بازی کامپیوتری اتومبیلرانی، اشیا جزء یا همه قطعات اتومبیل مانند انواع موتور و انوع فرمان توسط برنامه کامپیوتری ایجاد می‌شوند، اما شما بر حسب سلیقه قطعات مختلف ساخت اتومبیل را انتخاب می‌کنید و اتومبیل خود یا شیء کل را می‌سازید. اگر هم اتومبیل را از بین ببرید، اشیاء و قطعات اتومبیل همچون موتور و فرمان که توسط برنامه کامپیوتری از همان ابتدای اجرای برنامه ایجاد شده‌اند، همچنان هستند، و کافی است در یک دسته‌بندی دیگر

اتومبیل جدیدی را ایجاد نمایید.

ب) مدل‌سازی رابطه ترکیب (Composition)

همانطور که پیش‌تر از این نیز گفتیم، رابطه ترکیب رابطه‌ای مابین یک واحد کل و تعدادی واحد جزء است، یعنی یک کلاس کل از همکاری تعدادی کلاس جزء تشکیل شده باشد به نحوی که شیء کل و جزء با هم ایجاد شوند و با هم از بین بروند. برای مثال یک شیء کل «بازیکن» را در نظر بگیرید، که خود از چندین شیء جزء دیگر مانند سر، دست، پا و بدنه تشکیل شده است. به نحوی که شیء کل و جزء با هم ایجاد شوند و با هم از بین بروند. به عنوان مثالی دیگر، یک شیء کل «موتور اتومبیل» را در نظر بگیرید، که خود از چندین شیء جزء دیگر تشکیل شده است. به نحوی که شیء کل و جزء با هم ایجاد شوند و با هم از بین بروند. به چنین روابطی ترکیب گفته می‌شود. در رابطه ترکیب، به طرف جزء، عضو (Member) و به طرف کل (Whole)، ترکیب گفته می‌شود.

در رابطه **انجمنی‌ساده**، مابین طرفین همکار رابطه، رابطه جزء و کل مطرح نیست، در حالی که در رابطه **ترکیب**، مابین طرفین همکار رابطه، رابطه جزء و کل مطرح است و از **ترکیب** و سازماندهی اجزاء، یک موجودیت کامل‌تر ساخته می‌شود. برای مثال، برای ساخته شدن یک شیء کل «بازیکن» لازم است اشیاء جزء مختلفی با یکدیگر **ترکیب** شوند. یا برای مثالی دیگر، برای ساخته شدن یک شیء کل «موتور اتومبیل» لازم است اشیاء جزء مختلفی با یکدیگر **ترکیب** شوند.

توجه: رابطه تجمع به عنوان رابطه «is part of» نیز شناخته می‌شود.

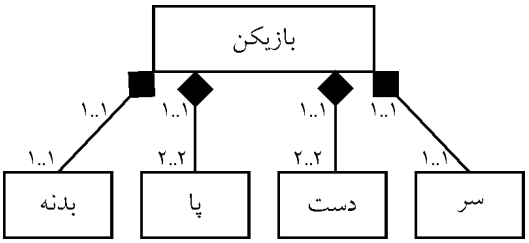
مراحل لازم جهت مدل‌سازی یک رابطه ترکیب عبارتند از:

۱- ترسیم خط ممتد از سمت جزء به سمت کل

۲- ترسیم لوزی توپر در انتهایی از خط رابطه که کلاس کل قرار دارد

۳- تعیین تعدد، نقش و محدودیت‌های طرفین رابط در صورت نیاز

شکل زیر نمونه‌ای از یک رابطه ترکیب میان شیء کل «بازیکن» و اشیاء جزء «سر، دست، پا و بدنه» را نشان می‌دهد.



توجه: حیات اجزای «سر، دست، پا و بدنه» کاملا به بازیکن، وابسته است، بدین معنی که اجزای «سر، دست، پا و بدنه» هرگز نمی‌توانند مستقل از یک بازیکن وجود داشته باشد، بنابراین از نماد ۱..۱ در سمت بازیکن استفاده شده است.

در رابطه ترکیب، حیات شیء جزء به حیات شیء کل وابسته است. به بیان دیگر حیات شیء

جزء و شیء کل به هم تقدم و تاخر ندارند. یعنی امکان ندارد شیء جزء موجود باشد، بدون اینکه شیء کل ایجاد گردد و موجود باشد. در رابطه ترکیب اگر شیء کل از بین برود، اشیاء جزء نیز به طور همزمان با شیء کل از بین می‌روند. زیرا از ابتدای کار، شیء کل، به طور همزمان از ایجاد و کنار هم قرار دادن اشیاء جزء توسط برنامه کامپیوتری ایجاد شده است و با پایان یافتن فعالیت شیء کل، شیء کل و جزء باهم و همزمان از بین می‌روند. در بازی کامپوتری فوتبال، شیء کل بازیکن از اشیاء جزء سر، دست، پا و بدنه تشکیل شده است. و پس از پایان فعالیت، شیء کل و جزء به طور همزمان باهم از بین می‌روند، در یک بیان ساده اشیاء کل و جزء در هنگام نیاز، باهم ایجاد می‌گردند و در صورت عدم نیاز، اشیاء کل و جزء با هم از بین می‌روند. در رابطه ترکیب، اشیاء جزء در شیء کل معنا و مفهوم دارند و به طور همزمان برای ساخت شیء کل ایجاد می‌گردند. اشیاء جزء در ترکیب به تنهایی معنا و مفهومی ندارند و برای خدمت به شیء کل ایجاد می‌گردند، و پس از، از بین رفتن شیء کل، اشیاء جزء هم به خدمت خود همزمان با پایان حیات شیء کل، پایان می‌دهند. اشیاء جزء در رابطه ترکیب فقط برای خدمت به شیء کل ایجاد می‌گردند و پس از پایان یافتن شیء کل، معنا و وظیفه دیگری ندارند و حیاتشان به پایان می‌رسد. برای مثال توسط برنامه کامپیوتری فوتبال، اشیاء جزء سر، دست، پا و بدنه به طور همزمان با ساخت بازیکن ایجاد می‌گردند و با پایان یافتن فعالیت بازیکن، اشیاء جزء سر، دست، پا و بدنه نیز به پایان فعالیت خود می‌رسند، در یک بیان ساده، در ابتدای اجرای بازی کامپیوتری فوتبال، شیء کل بازیکن از اشیاء جزء سر، دست، پا و بدنه به طور همزمان ایجاد می‌شود و تا پایان اجرای برنامه کامپیوتری هست که این ترکیب است، البته تک تک بازیکنان به نوبت به همین ترتیب توسط برنامه کامپیوتری ایجاد می‌شوند، اما حالا همه بازیکنان تیم ملی هستند، حال بر حسب سلیقه ۱۱ نفر از آن‌ها را انتخاب می‌کنید و تیم خود یا شیء کل را می‌سازید، که این تجمع است. که اگر تیم هم منحل شود، اشیاء بازیکن که توسط برنامه کامپیوتری ایجاد شده‌اند، همچنان هستند، و کافی است در یک دسته‌بندی دیگر تیم جدیدی ایجاد شود. برای مثالی دیگر توسط برنامه کامپیوتری اتومبیلرانی اشیاء جزء مربوطه به طور همزمان برای ساخت قطعات موتور، فرمان، چرخ و غیره ایجاد می‌گردند و با پایان یافتن فعالیت قطعات موتور، فرمان، چرخ و غیره، اشیاء جزء مربوطه نیز به پایان فعالیت خود می‌رسند. در یک بیان ساده، در ابتدای اجرای بازی کامپیوتری اتومبیلرانی، شیء کل موتور، فرمان، چرخ و غیره از اشیاء جزء مربوطه به طور همزمان ایجاد می‌شوند و تا پایان اجرای برنامه اتومبیلرانی هستند که این ترکیب است، البته تک تک قطعات موتور، فرمان، چرخ و غیره به نوبت به همین ترتیب توسط برنامه کامپیوتری ایجاد می‌شوند، اما حالا همه قطعات ساخت اتومبیل همچون موتور، فرمان، چرخ و غیره هستند، حال بر حسب سلیقه قطعات را انتخاب می‌کنید و اتومبیل خود یا شیء کل را می‌سازید. که این تجمع است. که اگر اتومبیل هم از بین برود، اشیاء و قطعات اتومبیل همچون موتور، فرمان، چرخ و غیره که توسط برنامه کامپیوتری از همان ابتدای اجرای برنامه ایجاد شده‌اند، همچنان هستند، و کافی است در یک دسته‌بندی دیگر اتومبیل جدیدی ایجاد شود. برای مثالی دیگر، یک پنجره سیستم عامل ویندوز را در نظر بگیرید، یک پنجره از چندین شیء جزء تشکیل شده است، برای مثال اشیاء جزء دکمه کمینه، بیشینه، بستن

پنجره و منوها، هنگامی که یک شیء کل پنجره بنا به درخواست اجرا می‌گردد، همزمان با آن تمام دکمه‌ها و منوها ایجاد می‌شوند. و با پایان دادن به فعالیت پنجره، شیء کل پنجره و اشیاء جزء به همراه هم و به طور همزمان به فعالیت خود پایان می‌دهند. در وادی زندگی نیز انسان کل با اشیاء جزء خود به طور همزمان متولد می‌شود و سپس در انتهای آن به حیات مادی خود به همراه اشیاء جزء خود به طور همزمان پایان می‌دهد، که این **ترکیب** است. اما این انسان در طول حیات خود در گروه‌ها، تیم‌ها، و کلاس‌های مختلفی شرکت می‌کند، که این **تجمع** است.

۳- مدل‌سازی رابطه وابستگی (Dependency)

همانطور که پیش‌تر از این در مورد رابطه وابستگی گفتیم، هرگاه مابین دو کلاس رابطه وابستگی مطرح باشد، رابطه وابستگی خواهیم داشت. در واقع هرگاه تغییرات مقادیر صفات یک شیء، روی مقادیر صفات شیء دیگری تأثیر بگذارد، این دو شیء به هم وابسته هستند. در واقع این رابطه زمانی رخ می‌دهد که مقادیر صفاتی از یک کلاس به مقادیر صفاتی از یک کلاس دیگر وابسته باشد. برای مثال در برنامه کامپیوتری فوتبال، مختصات شیء سربازیکن، دست بازیکن و پای بازیکن، همواره باید به مختصات شیء بدنه بازیکن وابسته باشد (مگر خشونت در بازی زیاد باشد!). در واقع مختصات سربازیکن، دست بازیکن و پای بازیکن، همواره باید با تغییرات مختصات بدنه بازیکن تغییر کند. وگرنه در هنگام حرکت بازیکن، سر بازیکن، دست بازیکن و پای بازیکن از بدنه بازیکن جلو یا عقب می‌افتند!

توجه: رابطه وابستگی به عنوان رابطه «depend upon» نیز شناخته می‌شود.

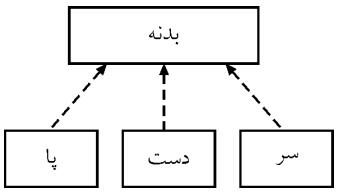
مراحل لازم جهت مدل‌سازی یک رابطه وابستگی عبارتند از:

۱- ترسیم یک خط جهت دار مقطع از سمت کلاس وابسته به سمت کلاس غیروابسته

۲- بیان نوع وابستگی بر روی خط جهت‌دار مقطع (Stereotype یا کلیشه)

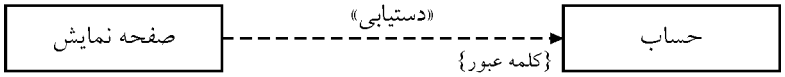
توجه: در UML کلیشه‌ها توسط نماد <<Stereotype>> مشخص می‌شوند.

شکل زیر نمونه‌ای از یک رابطه وابستگی میان «بدنه بازیکن» و «سر بازیکن، دست بازیکن، پای بازیکن»، را نشان می‌دهد.



توجه: رابطه سرویس گیرنده (client) و سرویس دهنده (server) میان دو کلاس، نوع دیگری از وابستگی میان دو کلاس است. در این گونه موارد کلاس سرویس گیرنده به نحوی به کلاس سرویس‌دهنده وابسته است و یک رابطه وابستگی برقرار می‌شود. در این نوع رابطه کلاس سرویس گیرنده برای انجام کارهای خود به داده‌های موجود در کلاس سرویس‌دهنده وابسته است. مانند وابستگی کلاس صفحه نمایش، به کلاس حساب، جهت مقدار دهی صفحه نمایش برای یک

مشتری خاص در سیستم ATM. برای نمایش محدودیت‌های دسترسی نماد «{}» مورد استفاده قرار می‌گیرد. در شکل زیر محدودیت «{کلمه عبور}» در نظر گرفته شده است.



مدل‌سازی رابطه وراثت بین کلاس‌ها در نمودار کلاس

به طورکلی هرگاه یک کلاس (ب)، **از نوع** یک کلاس (الف) باشد، گوییم بین دو کلاس مذکور رابطه **وراثت** برقرار است. در این صورت کلاس (ب) فرزند و کلاس (الف) پدر یا والد نامیده می‌شود. به عبارت دیگر همانطور که پیش از این نیز گفتیم، وراثت فرآیندی است که به وسیله آن یک کلاس (فرزند) می‌تواند صفات و متدهای کلاس دیگری (پدر) را کسب کند. به عبارت کلی‌تر یک کلاس فرزند ضمن به ارث بردن مجموعه‌ای از صفات و متدهای عمومی کلاس پدر، می‌تواند ویژگی‌های خاص و مختص خود را نیز به آنها اضافه کند. در رابطه وراثت هر تغییر در کلاس پدر بر کلاس فرزند نیز اثر می‌گذارد اما عکس این مطلب برقرار نیست. برای مثال، بین سیب و میوه، رابطه ارث‌بری برقرار است، سیب (فرزند) **یک نوع** میوه (پدر) است. بنابراین یک سیب دارای تمام ویژگی‌های یک میوه است. بنابراین از دیدگاه وراثت، سیب **نوعی** میوه است، زیرا تمام ویژگی‌های میوه را به ارث می‌برد و البته دارای یک سری ویژگی‌های مختص به خودش نیز هست. از دیدگاهی دیگر، میوه تعمیم یافته سیب است، زیرا ویژگی‌های عمومی سیب که در میوه‌های دیگر نیز مشترک است در میوه گردآوری شده است. ابرکلاس، همان کلاس پدر است که ویژگی‌ها و عملیات خود را در اختیار کلاس‌های دیگر (فرزندان) قرار می‌دهد. برای مثال، «سیب» یک ابرکلاس است، زیرا ویژگی‌های خود را در اختیار کلاس «سیب قرمز» قرار می‌دهد. زیرکلاس نیز همان کلاس فرزند است که برخی از صفات و عملیاتش متعلق به یک ابرکلاس است. کلاس «سیب قرمز» نمونه‌ای از یک زیرکلاس است.

دو رویکرد برای ایجاد سلسله مراتب وراثت وجود دارد:

۱- رویکرد «بالا به پایین» یا تخصیص (Specialization) که در آن، ابتدا کلاس‌های پدر، تعریف و سپس کلاس‌های فرزند ایجاد می‌شوند.

۲- رویکرد «پایین به بالا» یا تعمیم (Generalization) که در آن، از طریق تعمیم کلاس‌های فرزند به کلاس‌های پدر می‌رسیم.

توجه: رابطه وراثت به عنوان رابطه «is a» یا «kind of» یا «type of» نیز شناخته می‌شود. زیرا در این رابطه می‌گوییم: «A red apple is an apple» و یا «An apple is a fruit».

مراحل لازم جهت مدل‌سازی یک رابطه ارث‌بری عبارتند از:

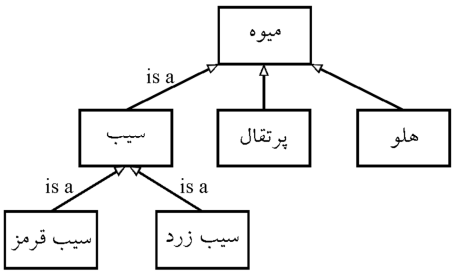
۱- ترسیم یک خط ممتد بین کلاس فرزند و کلاس پدر

۲- درج یک مثلث تو خالی در انتهایی از خط ممتد که کلاس پدر قرار دارد.

توجه: ذکر «is a» بر روی خط ممتد رابطه وراثت اختیاری است.

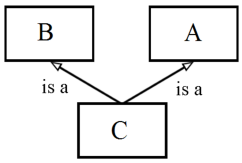
توجه: در تعریف رابطه وراثت، مشخصاتی نظیر تعدد و محدودیت وجود ندارد.

شکل مقابل نمونه‌ای از یک سلسله مراتب ارث‌بری میان «میوه»، «سیب» و «سیب قرمز»، را نشان می‌دهد.



توجه: همانطورکه در فصل قبل نیز گفتیم، هرگاه یک کلاس برخی از صفات و عملیات را از یک کلاس و برخی دیگر را از یک کلاس دیگر به ارث ببرد، در این حالت وراثت چندگانه رخ داده‌است.

شکل مقابل نمونه‌ای از مدل‌سازی **«ارث‌بری چندگانه»** (Multiple Inheritance) را نشان می‌دهد.



در شکل فوق کلاس C به صورت وراثت چندگانه از کلاس A و کلاس B ارث‌بری کرده است.

پس از مدل‌سازی ارتباطات ایستای میان کلاس‌های همکار نوبت به مدل‌سازی تعاملات پویای میان اشیاء همکار می‌رسد.

ی) مدل‌سازی تعاملات پویای میان اشیاء همکار

مدل‌سازی تعاملات پویای میان اشیاء همکار یا مدل‌سازی رفتاری، رفتار سیستم را در زمان اجرا نمایش می‌دهد. به بیان دیگر در این دیدگاه رفتارهای پویای سیستم مدل می‌شود.

مدل‌سازی رفتاری سیستم با استفاده از نمودارهای UML قابل انجام است:

الف) نمودار توالی (Sequence Diagram)

همانطورکه گفتیم نمودار کلاس ساختار ایستای داخل یک use case را نمایش می‌دهد. در یک سیستم در حال کار، به هر حال اشیاء با یکدیگر در تعامل هستند و این تعامل در طی زمان رخ می‌دهد. به بیان دیگر برای آنکه یک use case یا مورد کاربرد یا نیاز مرتع گردد، باید مجموعه‌ای از اشیاء با یکدیگر ارتباط برقرار کرده و پیام‌هایی را با یکدیگر رد و بدل نمایند. نمودار توالی نشان می‌دهد، برای مرتفع شدن یک use case یا مورد کاربرد یا نیاز، چه اشیایی باید چه پیام‌هایی را با چه ترتیبی ارسال کنند تا آن نیاز برآورده گردد. نمودار توالی، تعاملات پویا مابین اشیاء همکار را براساس زمان نشان می‌دهد.

توجه: Sequence Diagram یا نمودار توالی، Object Diagram یا نمودار شیء جهت مدل‌سازی تعاملات پویای میان اشیاء همکار داخل یک use case مورد استفاده قرار

می‌گیرد.

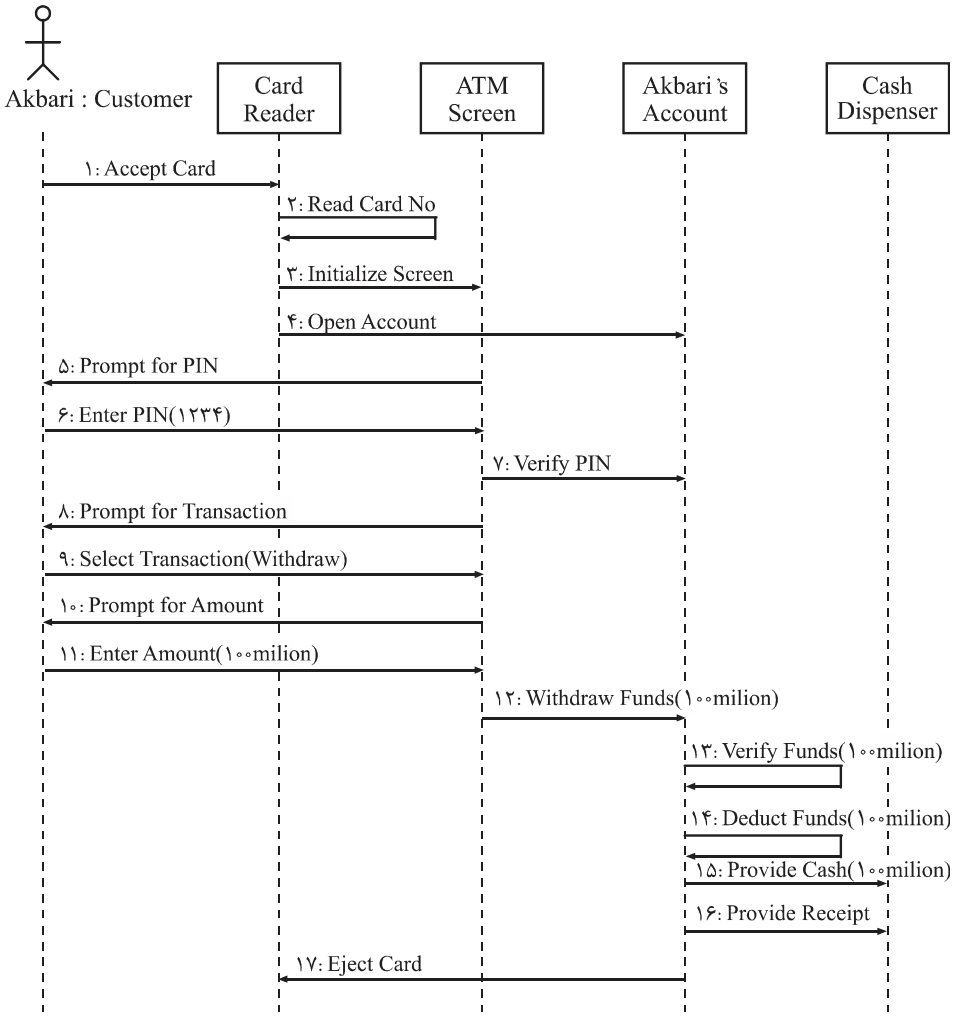
به طور کلی Sequence Diagram یا نمودارتوالی بر دو طبقه اصلی و فرعی می‌باشد:

نمودار توالی اصلی: نمایش روال تعاملات پویای میان اشیاء همکـار داخـل یـک use case از نقطه شروع تا رسیدن به یک نتیجه مطلوب (موفق)

توجه: برای رسم نمودار توالی اصلی از سناریوی اصلی استفاده می‌گردد.

مثال: نمایش روال تعاملات پویای میان اشیاء همکار، برای برداشت وجه موفق از یک حساب، مربوط به use case برداشت وجه در یک سیستم ATM.

در شکل زیر طرح معمولی برداشت صد میلیون ریال پول بدون هیچ مسئله‌ای مانند وارد کردن رمز عبور (PIN) اشتباه نشان داده شده است:



نمودار توالی فوق جریان پردازش را در use case برداشت وجه (Withdraw Money) نشان می‌دهد.

همچنین اشیایی که سیستم نیاز دارد تا use case برداشت وجه را به نتیجه برساند در بالاترین نقطه نمودار نشان داده شده است. هر فلش یک پیغام ارسالی بین «بازیگر با شیء» یا «شیء با شیء» را نمایش می‌دهد. تا عملیات موردنیاز را به انجام برساند.

با توجه به نمودار، روال کار در use case برداشت وجه، بدین ترتیب شروع می‌شود که مشتری کارتش را وارد کارت‌خوان می‌کند، سپس کارت‌خوان شماره کارت را می‌خواند، شیء حساب آقای اکبری را باز می‌کند و صفحه نمایش ATM را مقداردهی می‌نماید. صفحه نمایش از آقای اکبری می‌خواهد که PIN را وارد نماید. او ۱۲۳۴ را وارد می‌کند. PIN وارد شده تأیید می‌شود. صفحه نمایش انتخاب‌هایش را برای آقای اکبری آماده می‌کند و او برداشت صد میلیون ریال را انتخاب می‌کند. حساب آقای اکبری تأیید می‌کند که موجودی، حداقل شامل صد میلیون ریال است. سپس وجه از حساب کسر می‌شود و به صندوق اطلاع می‌دهد که صد میلیون ریال را آماده کند. همچنین یک رسید آماده شود. سرانجام به کارت‌خوان اطلاع داده می‌شود تا کارت را پس دهد. بنابراین این نمودار توالی، تمام جریان پردازشی use case مربوط به برداشت وجه موفق مربوط به توالی اصلی را با یک مثال نشان داد.

توجه: پس از اعتبارسنجی مشتری در مراحل ۱ تا ۷ و ارائه دریافت فرمان بعدی از سوی سیستم در مرحله ۸، در مرحله ۹، یعنی select transaction، از بین تراکنش‌های مختلف سیستم، مانند «برداشت وجه»، «انتقال وجه»، «تغییر رمز»، «نمایش موجودی» و غیره، مشتری تراکنش «برداشت وجه» را انتخاب کرده است. البته واضح است که مشتری در حال کار با مورد کاربرد، برداشت وجه است.

توجه: هر یک از چهار شیء فوق، به عنوان نماینده‌های یک ستون، در بالای شکل ظاهر می‌شوند. در زیر هر شیء خط چین‌های عمودی نشان داده شده است که به خط زندگی (Lifeline) موسوم است. محور عمودی بیانگر گذر زمان است. به بیان دیگر هر خط چین عمودی، خط زندگی یک شیء را نشان می‌دهد که تمام تعامل‌های آن را با اشیای دیگر از لحظه ایجاد تا زمان نابودی، مدل‌سازی می‌نماید. پیام‌هایی که در بین اشیاء گذر می‌کنند به شکل پیکان‌های افقی ظاهر می‌شوند. نام هر پیکان، متد و پارامترهای سرویس درخواستی را نشان می‌دهد. همچنین ترتیب پیام‌ها، با حرکت عمودی در امتداد محور اشیاء و رو به پایین مشخص می‌شود و به همین دلیل نیازی به شماره گذاری پیام‌ها نیست.

نمودار توالی فرعی: نمایش روال تعاملات پویای میان اشیاء همکار داخل یک use case از نقطه شروع تا رسیدن به یک نتیجه نامطلوب (ناموفق).

توجه: برای رسم نمودار توالی فرعی از سناریوهای فرعی استفاده می‌گردد.

مثال: نمایش روال تعاملات پویای میان اشیاء همکار، برای برداشت وجه ناموفق از یک حساب، مربوط به use case برداشت وجه در یک سیستم ATM. مواردی همچون تلاش برای برداشت پول از حساب بدون موجودی، تلاش برای برداشت پول با PIN اشتباه و غیره.

توجه: یک use case یک نمودار توالی اصلی و چندین نمودار توالی فرعی دارد.

ب) نمودار همکاری (Collaboration Diagram)

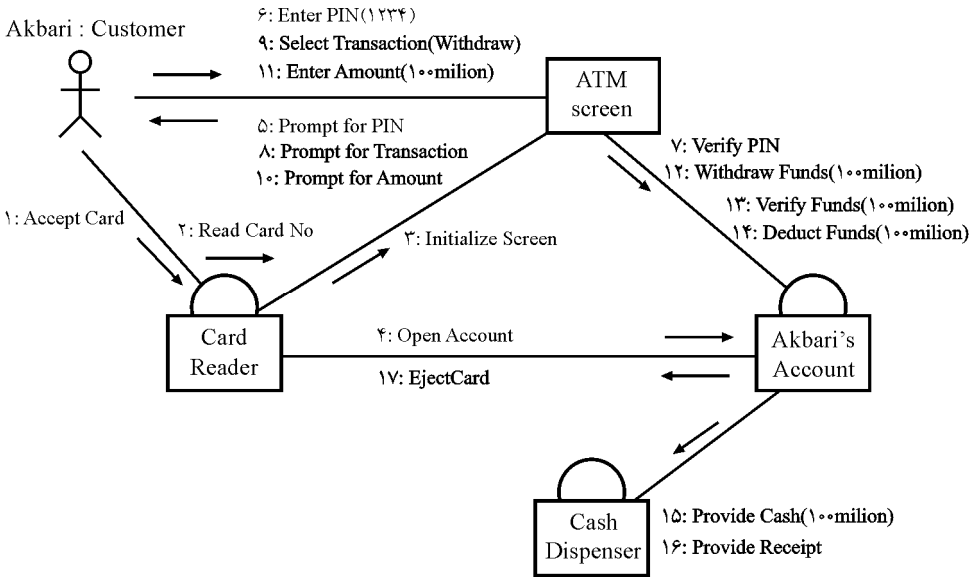
توجه: Collaboration Diagram یا نمودار همکاری در نسخه‌های امروزی UML، به نمودار **Communication Diagram** یا **نمودار ارتباط**، تغییر نام یافته است.

توجه: Collaboration Diagram یا نمودار همکاری، برای یک کاربرد خاص، جهت مدل‌سازی تعاملات پویای میان اشیاء همکار داخل یک use case مورد استفاده قرار می‌گیرد.

این نمودار نیز، مانند نمودار توالی، تعاملات بین اشیاء یک مورد کاربرد را نشان می‌دهد. در این نمودار نمی‌توان ترتیب زمانی را نشان داد. بنابراین، ترتیب ارسال پیام‌ها با شماره‌گذاری مشخص می‌شود. نمودار همکاری دقیقاً همان اطلاعات نمودار توالی را نشان می‌دهد. اگرچه، نمودار همکاری اطلاعات را به روش متفاوت و با یک هدف متفاوت نشان می‌دهد.

نمودار همکاری مثال قبل (نمودار توالی) به صورت زیر نشان داده شده است:

در نمودار همکاری زیر، مانند قبل، شیءها به شکل مستطیل و بازیگرها به شکل آدمک نشان داده شده است:



در حالی که در نمودار توالی، شیءها و ارتباطات بازیگرها به ترتیب زمان توضیح داده شده‌اند، نمودار همکاری شیءها و فعل و انفعالات بازیگرها را بدون توجه به زمان نشان می‌دهد.

مثلاً در این نمودار می‌بینید که کارت‌خوان به حساب آقای اکبری اطلاع می‌دهد تا باز شود و

حساب آقای اکبری به کارت‌خوان اطلاع می‌دهد تا کارت را پس دهد. همچنین شیءهایی که مستقیماً با دیگری ارتباط برقرار می‌کنند. با خطوطی که بین آن‌ها کشیده شده، نشان داده شده‌اند. اگر صفحه نمایش ATM و کارت‌خوان مستقیماً با یکدیگر رابطه داشته باشند، باید یک خط بین آنها کشیده شده باشد. نبودن این خط به این معنی است که هیچ ارتباط مستقیمی بین این دو شیء وجود ندارد.

بنابراین نمودارهای همکاری همان اطلاعات نمودارهای توالی را نشان می‌دهد اما افراد به دلایل متفاوتی به نمودارهای همکاری مراجعه می‌کنند. مهندسین تضمین کیفیت و معماران سیستم به این نمودارها نگاه می‌کنند تا توزیع شدن پردازش‌های بین شهرها را ببینند. فرض کنید که نمودار همکاری به شکل یک ستاره که در آن چند شیء که با یک شیء مرکزی ارتباط دارند، باشد. یک معمار سیستم ممکن است نتیجه بگیرد که سیستم خیلی به شیء مرکزی وابسته است و شیءها را دوباره طراحی نماید تا نیروی پردازش کردن را به طور یکنواخت توزیع کند. دیدن این نوع محاورات در یک نمودار توالی بسیار مشکل است.

توجه: همانند نمودار کلاس، نمودار همکاری نیز رابطه انجمنی میان اشیاء را نشان می‌دهد. به بیان دیگر در نمودار همکاری نیز باید مابین تمام کلاس‌هایی که در نمودار کلاس با یکدیگر رابطه انجمنی داشته‌اند، یک خط ارتباط رسم شود. پیام‌هایی که در بین اشیاء گذر می‌کنند به شکل پیکان‌هایی ظاهر می‌شوند. نام هر پیکان، متد و پارامترهای سرویس درخواستی را نشان می‌دهد. برای مثال، شیء حساب آقای اکبری یا Akbari Account به شیء ATM Screen توسط یک خط ممتد وصل شده است زیرا هر دو مستقیماً با دیگری **رابطه انجمنی و تبادل پیام** دارند. Card Reader به Cash Dispenser وصل نشده است زیرا این دو با هم ارتباطی ندارند. اشیاء کلاس‌های همکار از طریق صدا زدن متدهای عمومی یکدیگر اقدام به گفتگو و تبادل پیام می‌کنند. همچنین ترتیب پیام‌ها، به طور ذاتی و صریح مشخص نمی‌شود و به همین دلیل نیاز به شماره‌گذاری پیام‌ها می‌باشد.

ج) نمودار حالت (State Transition Diagram)

توجه: State Transition Diagram یا نمودار انتقال حالت (وضعیت)، به عنوان یک نمودار مکمل sequence Diagram، برای مشخص‌کردن وضوح بیشتر برای شناخت حالت‌های مختلف یک شیء مورد استفاده قرار می‌گیرد. به بیان دیگر نمودار انتقال حالت، چرخه حیات یک شیء را در حالت های مختلف (از زمانی که شیء ایجاد می‌شود تا زمانی که شیء از بین می‌رود) نمایش می‌دهد. در واقع این نمودار تمامی حالت‌های مختلفی را که یک شیء در طول حیات خود می‌تواند داشته باشد و همچنین نحوه انتقال بین حالت‌ها و وقایع باعث‌شونده این انتقال را نشان می‌دهد. به بیان دیگر نمودارهای حالت، راهی را آماده می‌کنند تا حالت‌های مختلف یک شیء را مدل کنند. در حالی که نمودارهای کلاس یک تصویر ثابت از کلاس‌ها و وابستگی آنها را نشان می‌دهد، نمودارهای حالت استفاده می‌شوند تا بیشتر، **رفتارهای پویای** یک سیستم را نمایش دهند.

توجه: برای تشخیص اینکه کلاسی دارای حالت‌های مختلفی است یا خیر، کافی است

صفت‌های آن کلاس مورد بررسی قرار گیرد، اینکه یک شیء از یک کلاس چگونه با مقدارهای متفاوت در برخی از صفات خود می‌تواند در حالت‌های مختلفی قرار گیرد. اگر این‌چنین صفتی برای یک کلاس موجود باشد، این نشانه خوبی برای تشخیص ایجاد نمودار حالت برای کلاس مورد نظر است.

برای مثال یک کلاس حساب بانکی می‌تواند به چندین حالت متفاوت وجود داشته باشد. می‌تواند «حساب باز دارای حداقل موجودی»، «حساب بسته» و «حساب باز بدون حداقل موجودی» باشد. یک حساب ممکن است در هر یک از این حالت‌ها، به طور متفاوتی رفتار کند. از نمودارهای حالت برای نشان دادن این اطلاعات استفاده می‌شود.

فرض کنید که برای انجام یک عملیات بانکی، اعم از ایجاد حساب، برداشت وجه و یا واریز وجه به بانک مراجعه می‌کنید. در هر کدام از این عملیات، بین «کارمند بانک»، «مشتری» **و** «حساب مشتری» اتفاقاتی می‌افتد که سناریوی هر کدام توسط نمودار توالی به خوبی قابل توصیف است. اما آیا می‌دانید که پس از انجام این عملیات، حساب بانکی مشتری در چه وضعیتی قرار می‌گیرد؟ آیا حساب بانکی مشتری همیشه در حالت ««حساب باز دارای حداقل موجودی» باقی می‌ماند؟ و یا این عملیات بانکی، بر روی حالت و وضعیت حساب تاثیر می‌گذارد؟ آیا نمودارهای توالی می‌توانند حالت‌های مختلف یک حساب بانکی را نشان داده و عوامل تغییردهنده آن را نیز مشخص نمایند؟

اجرای بعضی عملیات بر روی اشیاء ممکن است حالت و وضعیت آن‌ها را تغییر دهد. به عنوان مثال، شیء «حساب بانکی مشتری»، بر اثر اجرای عمل «برداشت وجه»، ممکن است به حالت «حساب باز بدون حداقل موجودی» وارد شود و یا حالت آن تغییر نکرده و همچنان در حالت «حساب باز دارای حداقل موجودی» باقی بماند.

همانطور که گفتیم حالت‌های یک شیء بر اساس مقادیر برخی از صفات شیء مشخص می‌شود. در شکل زیر، شیء حساب بانکی نشان داده شده است. در این شیء وقتی که مقدار صفت «موجودی»، به کمتر از ۱۰۰ هزار ریال برسد، شیء حساب بانکی وارد حالت «حساب باز بدون حداقل موجودی»، می‌شود. همچنین وقتی که مقدار موجودی به کمتر از ۱۰۰ هزار ریال برسد و در مدت ۳۰ روز به این حساب مراجعه نشود، بدین معنی که صفت «تاریخ آخرین عملیات»، به ۳۰ روز قبل اشاره کند، در این صورت حساب بانکی، وارد حالت «حساب بسته»، می‌شود و باید از سیستم حذف گردد.

حساب بانکی
محمد = نام صاحب حساب
۲۹۵۰۰۰ = موجودی
۹۱/۱/۷ = تاریخ آخرین عملیات

بنابراین، با توجه به اهمیت حالت‌های مختلف یک شیء و تاثیر آن‌ها بر عملکرد و رفتار سیستم، شناسایی حالت‌های مختف یک شیء به همراه عوامل تغییردهنده آن‌ها بسیار ضروری

است. نمودار حالت به مدل‌سازی حالت‌های مختلف یک شیء در طول حیات آن، یعنی از زمان ایجاد تا زمان نابودی آن، می‌پردازد. علاوه بر آن، در نمودارحالت، عوامل تغییردهنده حالت‌های یک شیء نیز مدل می‌شوند. همچنین، در هنگام تغییر حالت یک شیء ممکن است لازم باشد فعالیت‌هایی نیز انجام شود. این فعالیت‌ها نیز توسط نمودار حالت مدل‌سازی می‌شوند. برای مثال، به محض اینکه حسابی به حالت «حساب باز بدون حداقل موجودی»، وارد شود، موضوع باید به اطلاع دارنده حساب برسد.

همانطور که بیان شد، نمودار حالت، دوران حیات یک شیء را، در قالب اتفاقات و رخدادهایی که موجب تغییر حالت آن می‌شود، مدل‌سازی می‌کند. حالت یک شی مبین مقدار فعلی برخی صفات است، بدین معنی که مقدار فعلی برخی صفات یک شیء، حالت فعلی آن را مشخص می‌کند و تغییر مقادیر برخی صفات توسط اجرای عملیات مختلف، می‌تواند موجب تغییر حالت شیء گردد. زمانی که حالت یک شیء تغییر می‌نماید، نحوه پاسخگویی آن نیز تغییر می‌کند. برای مثال، وقتی که یک حساب بانکی در حالت «حساب باز دارای حداقل موجودی» است، مبلغ چک ممکن است پرداخت شود. در صورتی که چک، در حالت «حساب بسته»، برگشت داده می‌شود. نمودار توالی مبنای مناسبی برای ساخت نمودار حالت است. نمودار حالت، کل دوره حیات یک شیء را، از زمان ایجاد تا زمان نابودی آنرا، مورد بررسی قرار می‌دهد. در حالی که نمودار توالی، فقط برهه‌ای از زندگی شیء را، در طول یک سناریو نشان می‌دهد. بنابراین، برای به دست آوردن مجموعه حالت‌های مختلف یک شیء و مدل‌سازی نمودار حالت مربوطه، لازم است کلیه نمودارهای توالی که این شیء در آنها استفاده شده است، مورد بررسی قرار گیرند.

در ادامه به معرفی برخی از اصطلاحات مورد استفاده در نمودار حالت می‌پردازیم:

حالت (state): مشخص‌کننده وضعیت یک شیء است که بر اساس مقادیر فعلی برخی صفات آن تعیین می‌شود.

رخداد (event): به آنچه که موجب تغییر حالت یک شیء می‌گردد، رخداد گفته می‌شود. برای مثال، اگر عمل «برداشت وجه» به میزان زیاد انجام شود، رخداد «مانده حساب کمتر از ۱۰۰ هزار ریال»، واقع می‌شود و موجب تغییر حالت حساب از «حساب باز دارای حداقل موجودی» به «حساب باز بدون حداقل موجودی» خواهد شد. توجه داشته باشید که عمل «برداشت وجه»، همیشه موجب تغییر حالت حساب نمی‌شود.

اقدام ورودی (entry action): کارهایی که برای ورود به یک حالت انجام می‌شود. یک اقدام ورودی درون نماد حالت ظاهر می‌شود و یک کلمه entry و یک نماد : (colon) بعد از آن قرار می‌گیرد.

اقدام خروجی (exit action): کارهایی که برای خروج از یک حالت انجام می‌شود. یک اقدام خروجی درون نماد حالت ظاهر می‌شود و یک کلمه exit و یک نماد : (colon) بعد از آن قرار می‌گیرد.

فعالیت (activity): به عملیاتی که در یک حالت، اجرا می‌شوند، اما موجب تغییر حالت شیء

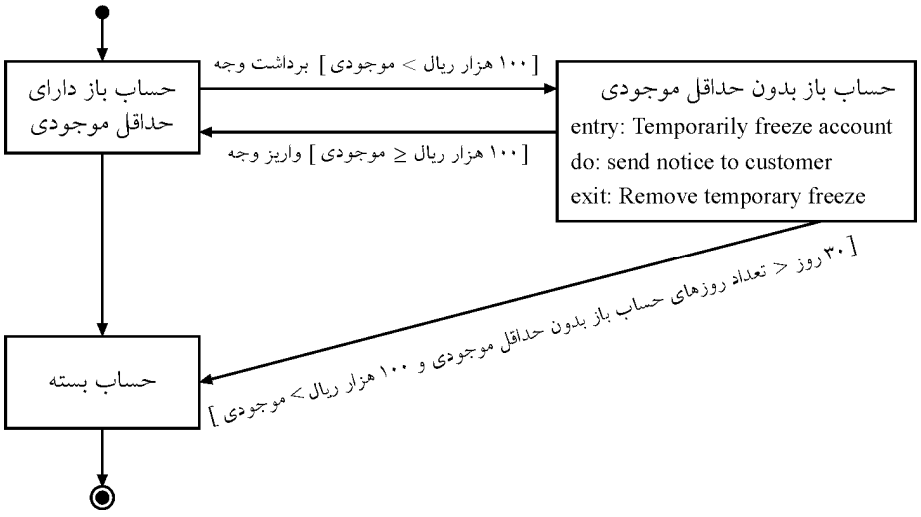
نمی‌گردند، فعالیت گفته می‌شود. مانند عمل «برداشت وجه» با حفظ حداقل موجودی که در حالت «حساب باز دارای موجودی حداقل» شیء حساب اجرا می‌شود، اما حالت آن را تغییر نمی‌دهد.

فعالیت داخلی: فعالیت داخلی، در شرایطی خاص، درون خود حالت برای اطلاع رسانی انجام می‌گردد. برای مثال وقتی که از یک حساب باز دارای حداقل موجودی، بیش از موجودی برداشت می‌شود، یک اخطار به مشتری فرستاده می‌شود. به عبارت دیگر وقتی شیء حساب بانکی به حالت «حساب باز بدون حداقل موجودی» وارد می‌شود، بهتر است این موضوع به اطلاع مشتری برسد. یک فعالیت داخلی درون نماد حالت ظاهر می‌شود و یک کلمه DO و یک نماد : (colon) بعد از آن قرار می‌گیرد.

پیکان انتقال: به پیکانی که بین دو حالت یک شیء کشیده می‌شود تا تغییر حالت آن را نشان دهد، پیکان انتقال گفته می‌شود.

توجه: هدف نمودار حالت، مدل‌سازی حالت‌ها، رخدادها و مشخص کردن ارتباط بین آنها است.

مثال: شکل زیر نمودار حالت را برای یک حساب بانکی نشان می‌دهد.



نمودار تغییر حالت متعلق به کلاس حساب (Account)

در این نمودار می‌توانیم حالت‌های مختلف یک حساب را ببینیم. همچنین می‌توانیم ببینیم که چگونه یک حساب از یک حالت به حالت دیگر منتقل می‌شود. برای مثال وقتی یک حساب در حالت «حساب باز دارای حداقل موجودی» است و مشتری درخواست بستن حساب را صادر می‌کند، حساب به حالت «حساب بسته» منتقل می‌شود. همانطور که گفتیم درخواستی از مشتری که منجر به تغییر حالت می‌شود «رخداد» نامیده می‌شود و رخداد چیزی است که موجب می‌شود یک انتقال از حالتی به حالت دیگر صورت گیرد. برای مثال اگر حساب در حالت «حساب باز

دارای حداقل موجودی» باشد و مشتری عمل «برداشت وجه» از حساب را با عدم حفظ حداقل موجودی انجام دهد، حساب از حالت «حساب باز دارای حداقل موجودی» به حالت حالت «حساب باز بدون حداقل موجودی» تغییر حالت می‌دهد.

در شکل فوق entry: Temporarily freeze account «اقدام ورودی» به معنی حساب به طور موقت مسدود شده، exit: Remove Temporary freeze «اقدام خروجی» به معنی حذف انسداد و do: Send notice to customer «فعالیت داخلی» است.

یک شرط که در براکت محصور شده است «شرط حفاظتی» (Guard Condition) نامیده می‌شود و وقوع یک انتقال (اینکه بتواند یا نتواند اتفاق بیفتد) را کنترل می‌کند.

در حالت ویژه، حالت شروع (Start State) و حالت پایان (Stop State) وجود دارد. «حالت شروع» یک شیء که مشخص‌کننده حالت اولیه و شروع زندگی آن است، باید به طور خاصی مدل‌سازی شود. برای این منظور، علاوه بر نماد حالت، باید یک دایره توپر به همراه یک پیکان که به حالت شروع اشاره دارد، ترسیم شود. توجه داشته باشید که حالت شروع، در برگیرنده همه اجزای مطرح شده است، یعنی نقطه توپر، پیکان و نماد حالت. بنابراین نقطه توپر و پیکان نیز جزء حالت شروع محسوب می‌شوند. نام رخدادی که باعث تغییر یک شیء از یک حالت به حالت دیگر می‌شود، بر روی پیکان انتقال بین دو حالت مذکور درج می‌گردد. جهت پیکان انتقال نیز، جهت تغییر حالت شیء را مشخص می‌کند. همانطور که هر شیء زمانی چرخه زندگی خود را از حالت شروع آغاز می‌کند، زمانی نیز به انتهای زندگی خود می‌رسد که به آن «حالت پایانی» گفته می‌شود، به طوری که پس از ورود به این حالت، به هیچ حالت دیگری نمی‌تواند وارد شود. نام حالت پایانی به وسیله یک دایره توپر مضاعف نمایش داده شده است و نشان می‌دهد که شیء درست قبل از اینکه از بین برود، در چه حالتی می‌باشد، شایان ذکر است که یک شیء فقط یک حالت شروع دارد، اما می‌تواند چندین حالت پایان داشته باشد. نمودارهای حالت برای هر کلاس ایجاد نمی‌شوند. آنها فقط برای کلاس‌های دارای حالات مختلف استفاده می‌شوند. اگر یک شیء از یک کلاس می‌تواند در چند حالت وجود داشته باشد و در هر حالت، متفاوت رفتار نماید، ممکن است بخواهد یک نمودار حالت برای آن ایجاد کنید. بسیاری از پروژه‌ها اصلاً به این نمودار نیازی ندارند. اگر آنها ایجاد شده‌اند، برنامه‌نویسان از آنها در زمان تولید کلاس‌ها استفاده می‌کنند، نمودار حالت خروجی کد ندارد بلکه به عنوان یک راهنما جهت تولید کد مورد استفاده قرار می‌گیرد. نمودارهای حالت فقط برای مستندسازی ایجاد شده‌اند.

توجه: همانطور که پیش از این نیز بیان شد، آنچه که باعث تغییر حالت یک شیء می‌گردد، تغییر مقادیر برخی صفات آن است و آنچه که باعث تغییر مقادیر صفات می‌شود، اجرای یکی از متدهای شیء صاحب آن صفات است. بنابراین، با توجه به اینکه اولا اجرای متدهای یک شیء می‌تواند عاملی برای تغییر حالت آن باشد، ثانیا نمودار حالت، توصیفگر حالات مختلف یک شیء و علل و عوامل تغییر حالت آن است و ثالثا نمودارهای توالی، نمایانگر ترتیب و تقدم اجرای متدهای مختلف اشیای سیستم هستند، در نتیجه می‌توان از نمودارهای توالی به نمودارهای حالت رسید.

توجه: اطلاعات به دست آمده از نمودار حالت در مدل تحلیل مربوط به کلاس‌های دارای حالات مختلف، در مدل طراحی، بخش طراحی مولفه، جهت تکمیل نمودن الگوریتم‌های مربوط به متدهای کلاس‌های دارای حالات مختلف مورد استفاده قرار می‌گیرد. اینکه متدهای یک کلاس دارای حالات مختلف در سطح طراحی مولفه در مواجه با حالت‌های مختلف یک شیء چگونه رفتار کنند از نمودار حالت استنباط می‌گردد. طراحی مولفه جلوتر در بخش مدل طراحی شرح داده می شود.

توجه: همه نمودارهای فوق (توالی، همکاری و حالت)، مدل‌سازی رفتاری یا تعاملی محسوب می‌گردند.

توجه: از آنجا که وقوع رخداد در نمودارهای توالی و همکاری (ارتباط) نقشی ندارد، به نمودارهای رفتاری توالی و همکاری، مدل‌سازی رفتاری ایستا و به نمودار حالت، مدل‌سازی رفتاری پویا نیز گفته می‌شود.

مدل طراحی: OOD (Object – Oriented Design)

پس از مدل تحلیل، نوبت به مدل طراحی می‌رسد. مدل طراحی به روش شی‌گراء شامل چهار بخش **طراحی داده**، **طراحی معماری**، **طراحی مؤلفه** و **طراحی واسط** می‌باشد.

طراحی داده

طراحی داده، شامل طراحی ساختمان داده‌ها و طراحی پرس‌وجوها می‌باشد. هنگامی که قابلیت‌های پایگاه داده، با قابلیت‌های زبان برنامه‌نویسی شی‌گراء، ترکیب شوند، نتیجه، پایگاه داده شی‌گراء یا Object – Oriented DBMS خواهد شد. پایگاه داده شی‌گراء، به برنامه‌های شی‌گراء اجازه می‌دهند که اطلاعات خود را به همان صورت اشیاء در بانک اطلاعاتی ذخیره، بازیابی و بروزرسانی کنند، بخاطر سازگاری که بین پایگاه داده شی‌گراء و زبان برنامه‌نویسی شی‌گراء وجود دارد، برنامه‌نویس می‌تواند هر دو ابزار را در یک محیط مجتمع داشته باشد.

توجه: امروزه در تولید نرم‌افزارها، ترکیبی از سادگی پایگاه داده رابطه‌ای و مفاهیم پایگاه داده شی‌گراء، مورد استفاده قرار می‌گیرد.

طراحی معماری

طراحی معماری، نمودار کلاس و نمودار توالی مرتبط با هر یک از موارد کاربرد را از مدل تحلیل، به عنوان ورودی دریافت کرده و توسط سبک شیءگرا (مبتنی بر ارسال پیام مابین اشیاء)، طراحی معماری را انجام می‌دهد.

در معنای عام، معماری به معنی نحوه ارتباط بخش‌های مختلف یک سازه است. در حیطه مهندسی نرم‌افزار نیز معماری به معنی نحوه ارتباط بخش‌های مختلف سازه‌ای به نام برنامه کامپیوتری است. طراحی معماری یا معماری نرم‌افزار، ساختار کلی نرم‌افزار و شیوه‌های یکپارچگی یک سیستم را بیان می‌کند. به عبارت دیگر، ساختار سلسله مراتبی **مولفه‌های برنامه (کلاس‌ها یا**

پیمانه‌ها)، شیوه تعامل مولفه‌ها با یکدیگر و **ساختمان داده‌های** مورد نیاز مولفه‌ها را نشان می‌دهد. معماری نرم‌افزار یک مدل قابل درک از چگونگی سازمان‌دهی سیستم است. در واقع نشان‌گر ساختمان داده‌ها و مؤلفه‌های برنامه‌ای است که برای ساختن یک سیستم کامپیوتری لازم است. به طور دقیق‌تر معماری نرم‌افزار شامل دو سطح از طراحی می‌باشد یعنی طراحی داده و طراحی معماری. در واقع این ساختار مانند یک نقشه ساختمان، مبنای ساخت نرم‌افزار قرار می‌گیرد.

توجه: در طراحی معماری، اسکلت، ساختار و چیدمان کلی مؤلفه‌های برنامه به این معنی که چه مولفه‌ای (کلاسی) با چه مولفه‌ای (کلاسی) دیگر در ارتباط است، بدون ذکر جزئیات مربوط به شرح متدهای کلاس‌های همکار داخل یک مؤلفه فرعی (یک بخش از نرم‌افزار). مانند اسکلت یک ساختمان که گویای جایگاه مؤلفه‌های ساختمان است اما هنوز آجر چینی نشده است. (اسکلت یک ساختمان بدون آجر چینی).

توجه: واحد مؤلفه (پیمانه) در شی‌گرایی، کلاس است که از کنار هم قرار گرفتن کلاس‌های همکار داخل هر use case یا مورد کاربرد یا نیاز، یک مؤلفه فرعی (مؤلفه بخشی) ایجاد می‌گردد، که از کنار هم قرار گرفتن مؤلفه‌های فرعی برنامه، مؤلفه اصلی (مؤلفه کلی یا برنامه اصلی) ایجاد می‌گردد.

توجه: در این مرحله هر use case یا مورد کاربرد یا نیاز با اطلاعات مربوط به نمودارهای کلاس و توالی به یک مؤلفه فرعی ولی بدون ذکر جزئیات تبدیل می‌گردد. همچنین در این مرحله نحوه چیدمان مولفه‌ها و ساختار کلی برنامه، کلاس‌های همکار، اشیاء همکار و تعریف ساختار پیام‌ها مابین فرستنده و گیرنده پیام‌ها اما بدون ذکر جزئیات مشخص می‌شود.

توجه: به طراحی معماری، طراحی کلی نیز گفته می‌شود.

طراحی مؤلفه

طراحی مؤلفه، طراحی معماری از همان فعالیت مدل طراحی را به عنوان ورودی دریافت کرده و طراحی مؤلفه را توسط ابزارهایی همچون شبه کد یا Activity Diagram یا Swimlane Diagram ایجاد می‌کند.

طراحی مولفه، فعالیت تبدیل طراحی معماری به نرم‌افزار است. در این مرحله، سطح انتزاع طراحی معماری به سطح انتزاع نرم‌افزار کاربردی نزدیک می‌گردد. طراحی در سطح مؤلفه‌ها، نرم‌افزار را در سطحی از انتزاع تصویر می‌کند که به کد نزدیک است. طراحی مولفه، به عنوان نقشه راهی دقیق، و نزدیک به زبان پیاده‌سازی، در فعالیت پیاده‌سازی نرم‌افزار، منجر به صرفه جویی در زمان و هزینه‌های تولید می‌گردد. در طراحی مولفه، مهندس نرم‌افزار باید ساختمان داده‌ها، واسط‌ها، ساختار پیام‌ها و متدها را با جزییات کافی به نمایش در آورد تا راهنمای تولید کد منبع زبان برنامه‌نویسی باشد.

توجه: در طراحی مؤلفه، اسکلت، ساختار و چیدمان کلی مؤلفه‌های برنامه به این معنی که چه مولفه‌ای (کلاسی) با چه مولفه‌ای (کلاسی) دیگر در ارتباط است، با ذکر جزئیات مربوط به شرح متدهای کلاس‌های همکار داخل یک مؤلفه فرعی (یک بخش از نرم‌افزار). مانند اسکلت یک

ساختمان که گویای جایگاه مؤلفه‌های ساختمان است و آجرچینی هم شده است. (اسکلت یک ساختمان به همراه آجرچینی).

توجه: در این مرحله هر use case یا مورد کاربرد یا نیاز با اطلاعات مربوط به نمودارهای کلاس و توالی به یک مؤلفه فرعی اما با ذکر جزئیات تبدیل می‌گردد. همچنین در این مرحله نحوه چیدمان مولفه‌ها و ساختار کلی برنامه، کلاس‌های همکار، اشیاء همکار و تعریف ساختار پیام‌ها مابین فرستنده و گیرنده پیام‌ها نیز با ذکر جزئیات مشخص مشخص می‌شود.

توجه: همانطور که گفتیم، در طراحی مؤلفه، باید جزئیات الگوریتمی متدهای کلاس تشریح شود. برای این منظور Activity Diagram یا نمودار فعالیت و Swimlane Diagram یا نمودار خط شنا مورد استفاده قرار می‌گیرد. نمودار فعالیت و نمودار خط‌شنا ساختاری مشابه فلوچارت دارد. نمودار فعالیت و نمودار خط‌شنا نه تنها شرح‌حال متدهای کلاس را تصویر می‌کند، بلکه مطابق آنچه پیش از این در بخش مدل تحلیل شی‌گراء یا OOA گفتیم، جهت مدل‌سازی سناریوهای اصلی و فرعی داخل یک use case نیز مورد استفاده قرار می‌گیرد. علاوه بر نمودار فعالیت و نمودار خط‌شنا، توسط شبه کد (PDL) نیز می‌توان جزئیات مربوط به شرح متدهای کلاس را نمایش داد.

توجه: اطلاعات به دست آمده از نمودار حالت در مدل تحلیل مربوط به کلاس‌های دارای حالات مختلف، در مدل طراحی، بخش طراحی مولفه، جهت تکمیل نمودن الگوریتم‌های مربوط به متدهای کلاس‌های دارای حالات مختلف مورد استفاده قرار می‌گیرد. اینکه متدهای یک کلاس دارای حالات مختلف در سطح طراحی مولفه در مواجه با حالت‌های مختلف یک شیء چگونه رفتار کنند از نمودار حالت استنباط می‌گردد.

توجه: از کنار هم قرار گرفتن کلاس‌های همکار درون یک use case به عنوان واحد مولفه، یک مولفه بزرگتر با عنوان مولفه فرعی (مولفه بخشی) ایجاد می‌گردد و از اجتماع مؤلفه‌های فرعی (مؤلفه بخشی)، مؤلفه اصلی (مؤلفه کلی یا برنامه اصلی) خلق می‌گردد. در سیستم ATM، اجتماع مولفه‌های فرعی «برداشت وجه»، «واریز وجه»، «انتقال وجه»، «تغیر کلمه عبور»، «پرداخت» و «نمایش موجودی»، منجر به ساخت مؤلفه اصلی (مؤلفه کلی یا برنامه اصلی) یا سیستم ATM، می‌گردد.

توجه: به طراحی مولفه، طراحی جزئی، طراحی تفصیلی و طراحی رویه‌ای نیز گفته می‌شود.

طراحی واسط

طراحی واسط یا همان واسط کاربر، براساس ورودی‌ها و خروجی‌های مورد نیاز کاربران نهایی به شکل نقشی بر روی کاغذ یا طرحی بر روی کامپیوتر ایجاد می‌گردد. مانند نحوه چیدمان منوها و فرم‌ها.

۴- فعالیت ساخت (پیاده‌سازی و تست)

پس از فعالیت مدل‌سازی (تحلیل و طراحی) نوبت به فعالیت ساخت (پیاده‌سازی و تست) می‌رسد.

پیاده سازی: OOP (Object – Oriented Programming)

پس از مدل طراحی نوبت به پیاده‌سازی می‌رسد. این کار توسط زبان‌های شیءگراء، مانند C++ و SQL Server انجام می‌گردد. در این مرحله، ساختار کلاس‌ها، ساختار و نحوه ارسال پیام‌ها مابین اشیاء، ساختمان داده‌ها و پرس‌وجوها برای ایجاد مؤلفه‌های فرعی (مؤلفه بخشی) و به تبع ایجاد مؤلفه اصلی (مؤلفه کلی یا برنامه اصلی) پیاده‌سازی می‌گردند.

توجه: مؤلفه یک قطعه‌ی آماده، کاربردی و پیاده‌سازی شده است که دارای واسط لازم جهت اتصال با سایر قطعات و استقرار در بخشی از یک سیستم عملیاتی می‌باشد. از آنجایی که در یک مولفه، فقط بخشی از کدها یا داده‌های نرم‌افزار قرار می‌گیرد، لذا هر مولفه ممکن است نیازمند ارتباط با مولفه‌های دیگر باشد تا بتواند از کدها یا داده‌های موجود در آنها استفاده نماید. برای انجام این ارتباط، در مولفه‌ها یک واسط در نظر گرفته می‌شود تا کدها یا داده‌های یک مولفه، از طریق واسط آن در اختیار مولفه‌های دیگر قرار گیرد، به این معنی که ارتباط بین مولفه‌ها، فقط از طریق واسط‌های آنها انجام می‌گیرد. در برنامه‌نویسی شیءگرا، متدهای عمومی کلاس‌ها، نقش واسط را ایفا می‌کنند. به عبارت دیگر کلاس‌های همکار، از طریق متدهای عمومی یکدیگر اقدام به گفتگو و تبادل پیام می‌کنند.

در دیدگاه شیءگراء به مؤلفه پیمانه نیز گفته می‌شود. در حیطه مهندسی نرم‌افزار شیءگرا، **واحد مؤلفه** یک قطعه‌ی عملیاتی مبتنی بر کلاس است که بر دو نوع می‌باشد:

۱) کلاس کاربردی: مانند کلاس دانشجو که برنامه‌نویس آن را می‌نویسد و به دلیل داشتن شرایط قابل حمل به شکل ذاتی، می‌تواند به عنوان یک قطعه‌ی آماده و قابل استفاده‌ی مجدد در پروژه‌های بعدی مورد استفاده مجدد قرار گیرد.

۲) کلاس سیستمی: مانند کلاس جعبه‌ی متن که کامپایلر تعاریف آن را فراهم می‌کند و می‌تواند به عنوان یک قطعه‌ی آماده و قابل استفاده‌ی مجدد، در پروژه‌ها مورد استفاده مجدد قرار گیرد.

برای مدل‌سازی این مرحله نمودار مولفه مورد استفاده قرار می‌گیرد.

نمودار مؤلفه (Component Diagram)

این نمودار به نحوه پیاده‌سازی فیزیکی نرم‌افزار می‌پردازد. در واقع، نمودار مولفه شامل کدهای نرم‌افزار است که در آن هر قطعه کد در قالب یک مولفه، دسته‌بندی می‌شود. کدها نیز چیزی جز کلاس‌های سیستم، در زمانی که جزئیات پیاده‌سازی به آن‌ها اضافه شده باشد، نیست. برای تولید یا توسعه یک سیستم نرم‌افزاری، ابتدا لازم است که نیازمندی‌های مشتری توسط تیم توسعه در فعالیت ارتباطات شناسایی گردد و سپس نرم‌افزار پیشنهادی، در قالب یک طرح منطقی در اختیار مشتری قرار گیرد. این طرح منطقی، با استفاده از نمودارهایی که تاکنون آموخته‌اید، مدل می‌شود. اما آیا UML به نحوه پیاده‌سازی نرم‌افزار و کیفیت سخت‌افزارهای مورد نیاز نیز توجهی دارد؟ پاسخ، قطعا مثبت است، UML علاوه بر ارائه یک طرح منطقی از نرم‌افزار، به جنبه‌های پیاده‌سازی و سخت‌افزاری آن نیز می‌پردازد. نمودار مولفه چگونگی پیاده‌سازی نرم‌افزار را مورد بررسی قرار

می‌دهد. نمودار استقرار نیز، معماری سخت‌افزارهای مورد نیاز را مدل می‌نماید. ترکیب این دو نمودار، نحوه اجرای نرم‌افزار را روی سخت‌افزار مشخص می‌کند. نمودار مولفه در ادامه و نمودار استقرار در بخش بعدی مورد بررسی قرار می‌گیرند.

توجه: Component Diagram یا نمودار مؤلفه جهت مدل‌سازی ساختار کلی پیاده‌سازی برنامه و ترتیب کامپایل مؤلفه‌های فرعی، برای ایجاد مؤلفه‌اصلی (مؤلفه کلی یا برنامه اصلی) مورد استفاده قرار می‌گیرد. به بیان دیگر نمودار مؤلفه یک دید فیزیکی از مدل سیستم به همراه مؤلفه‌های فرعی نرم‌افزار و روابط بین آنها را نشان می‌دهد.

توجه: نمودار مؤلفه زمانی که تولید کد تمام شده است ایجاد می‌گردد. بنابراین در اینجا منظور از مؤلفه فرعی یک مولفه فیزیکی کد است. به بیان دیگر مؤلفه فرعی در این مرحله، کلاس‌های همکار پیاده‌سازی شده داخل یک use case یا مورد کاربرد یا نیاز پیاده‌سازی شده است. دقت کنید که قبل از استفاده از نمودار مولفه هریک از کلاس‌های همکار موجود در نمودار کلاس مربوط به هر use case یا مورد کاربرد یا نیاز باید به یک مؤلفه فرعی حاوی کد منبع تبدیل شود.

توجه: در این مرحله هر use case یا مورد کاربرد یا نیاز، مطابق روالی که از فعالیت‌های ارتباط تا ساخت (پیاده‌سازی) طی می‌کند، سرانجام توسط کلاس‌های همکار پیاده‌سازی شده خود به یک مؤلفه فرعی (مؤلفه‌بخشی) پیاده‌سازی شده و کامپایل شده تبدیل می‌گردد، که از اجتماع این مؤلفه‌های فرعی (مؤلفه‌بخشی)، مؤلفه اصلی (مؤلفه کلی یا برنامه اصلی) خلق می‌گردد. و محصول نهایی آماده می‌گردد.

توجه: به یک مولفه فرعی که از تعدادی کلاس همکار پیاده‌سازی شده، ایجاد شده است، **مولفه محقق شده یا وابستگی محقق شده** (realization dependency) نیز گفته می‌شود.

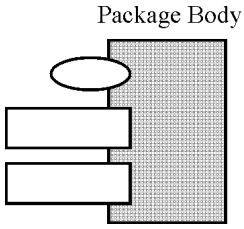
توجه: به یک مولفه فرعی، زیرسیستم (subsystem) نیز گفته می‌شود.

توجه: Component Diagram یا نمودار مؤلفه را می‌توان در زمره نمودارهای دید ایستا به شمار آورد. زیرا نمودارهای کلاس و مولفه از یک جنس هستند و شاید تنها سطح انتزاع آنها با یکدیگر متفاوت باشد.

در نمودار مؤلفه، دو نوع مؤلفه وجود دارد:

۱- مؤلفه قابل اجرا یا Package Body

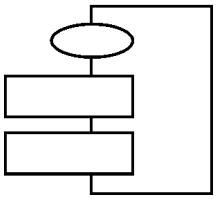
حاوی کد عملیات (Operation) کلاس می‌باشد. در زبان C++، Package Bodyها فایل‌های CPP، هستند. این نوع مؤلفه با نماد مقابل در UML نشان داده می‌شود:



۲- مؤلفه کتابخانه‌های کد یا Package Specification

حاوی تعاریف توابع سیستمی است. در زبان C++، Package Specificationها فایل‌های h.

هستند. این نوع مؤلفه با نماد مقابل در UML نشان داده می‌شود: Package Specification



به طور عمومی، بسته(Package) مجموعه‌ای از کلاس‌های پیاده‌سازی‌شده و کامپایل‌شده به عنوان واحد مولفه و یا تعدادی مولفه فرعی پیاده‌سازی‌شده و کامپایل‌شده است. که از اجتماع بسته‌ها یک سیستم نرم‌افزاری ایجاد می‌گردد.

توجه: به یک بسته(Package) ، زیرسیستم (subsystem) نیز گفته می‌شود.

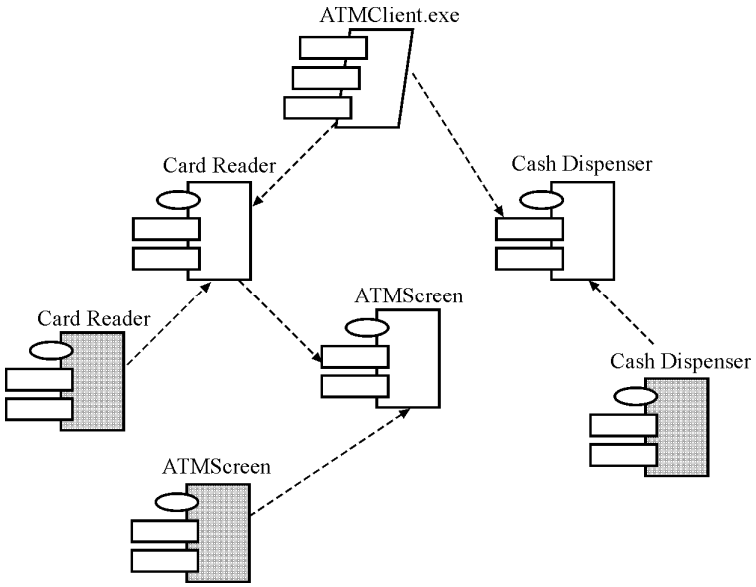
برای مثال سیستم ATM شامل دو بسته است:

۱- بسته ATMServer برای بخش سرویس‌دهنده

۲- بسته ATMClient برای بخش سرویس‌گیرنده

الف) بسته ATMClient

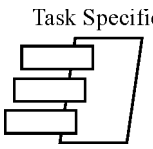
شکل زیر نمودار مؤلفه بسته سرویس‌گیرنده سیستم ATM یا ATMClient را نشان می‌دهد:



نمودار مولفه برای سرویس‌گیرنده ATMClient

در صورتی که زبان پیاده‌سازی C++ باشد، هر کلاس مؤلفه قابل اجرا (CPP.) و کتابخانه‌های کد (h.) خود را دارد. بنابراین هر کلاس در نمودار مؤلفه به یک کلاس پیاده‌سازی شده نگاشت می‌شود. برای مثال کلاس ATMScreen به یک کلاس پیاده‌سازی شده به نام ATMScreen

نگاشت می‌شود.



توجه: یک برنامه قابل اجرا (برنامه اصلی یا برنامه کلی) که از مولفه‌های فرعی (مولفه‌های بخشی) تشکیل شده است در نمودار مؤلفه UML توسط نماد زیر نمایش داده می‌شود.

توجه: یک فایل اجرایی معمولاً به عنوان یک Task Specification با یک پسوند EXE. نمایش داده می‌شود.

توجه: تنها نوع رابطه‌ای که می‌تواند بین مؤلفه‌ها در نمودار مولفه وجود داشته باشد، یک رابطه وابستگی (Dependency) است و به این معنی است که یک مؤلفه باید قبل از کدام مولفه دیگر کامپایل شود.

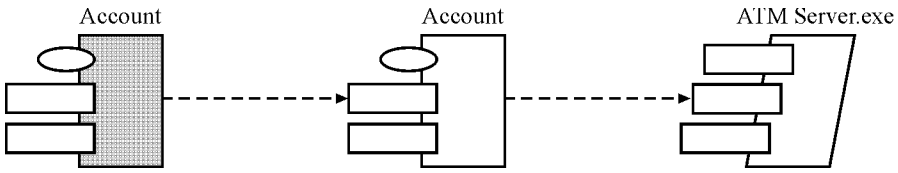
توجه: در نمودار مولفه، رابطه وابستگی بین مولفه‌ها توسط خطوط خط‌چین نشان‌داده می‌شود.

مثال: کلاس Card Reader به کلاس ATM Screen وابسته است. یعنی کلاس ATM Screen باید موجود باشد تا کلاس Card Reader کامپایل شود.

توجه: فایل اجرایی ATMClient.exe اولین باری که همه کلاس‌ها کامپایل شوند، می‌تواند ایجاد گردد.

ب) بسته ATMServer

شکل زیر نمودار مؤلفه بسته سرویس‌دهنده سیستم ATM یا ATMServer را نشان می‌دهد:



نمودار مولفه برای سرویس‌دهنده ATMServer

توجه: فایل اجرایی ATMServer.exe اولین باری که کلاس Account کامپایل شود، می‌تواند ایجاد گردد.

همان‌طور که در این مثال نشان داده شد، یک سیستم بسته به تعداد بسته‌ها می‌تواند چندین نمودار مؤلفه داشته باشد.

تست: OOT (Object – Oriented Testing)

پس از پیاده‌سازی نوبت به تست می‌رسد، در این مرحله کلیه موارد پیاده‌سازی شده از نظر خطاهای نحوی و خطاهای معنایی براساس لیست نیازمندی‌های مشتری (چک لیست) که در فعالیت ارتباطات تهیه شده بود، مورد وارسی قرار می‌گیرد تامشخص شود نرم‌افزار، براساس ورودی‌های موردنظر مشتری، خروجی‌های مورد انتظار مشتری را برآورده می‌سازد یا خیر.

توجه: این فعالیت در گام اول توسط برنامه‌نویسان و در صورت لزوم در گام بعدی توسط یک گروه تست مستقل (ITG) انجام می‌گردد.

توجه: ITG سرواژه عبارت Independent Testing Group و به معنی گروه تست مستقل است.

فعالیت استقرار

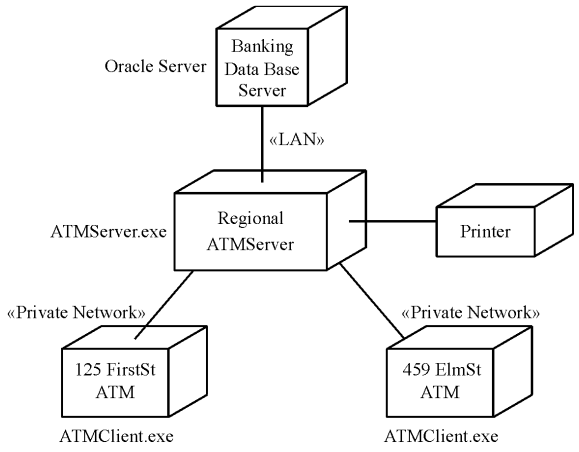
پس از تست، نوبت به فعالیت استقرار می‌رسد. هدف از فعالیت استقرار در گام اول، ارائه یک دید کلی از سخت‌افزارهای مورد نیاز سیستم است که مؤلفه‌های نرم‌افزاری باید بر روی آن‌ها قرار گیرند. و در گام دوم، نرم‌افزار به مشتری تحویل داده می‌شود و مشتری با بررسی محصول دریافتی، باز خوردهای به دست آمده براساس همین ارزیابی‌ها را به تیم نرم‌افزاری ارائه می‌دهد. این بازخوردها می‌توانند مبنایی برای ارتقاء و یا تصحیح نسخه‌ی بعدی نرم‌افزار باشد.

توجه: Deployment Diagram یا نمودار استقرار، جهت مدل‌سازی گره‌های سخت افزاری برای نصب نرم‌افزار مربوطه در فعالیت استقرار مورد استفاده قرار می‌گیرد.

نمودار استقرار (Deployment Diagram)

این نمودار معماری فیزیکی برای نصب یک سیستم مبتنی بر کامپیوتر را نشان می‌دهد. این نمودار می‌تواند کامپیوتر، دستگاه‌ها، اتصالات آنها با یکدیگر و نرم‌افزاری که روی هر دستگاه قرار می‌گیرد را نشان دهد. در اغلب نرم‌افزارهای امروزی کل ساختار یک نرم‌افزار بر روی یک گره سخت‌افزاری به شکل محلی قرار نمی‌گیرد. بلکه بخش‌های مختلف یک نرم‌افزار، بر روی گره‌های مختلف سخت‌افزاری به شکل غیر محلی توزیع می‌شوند. نمودار استقرار نشان می‌دهد که بخش‌های مختلف یک نرم‌افزار چگونه در گره‌های مختلف سخت‌افزاری توزیع می‌شوند.

سیستم ATM از سه بخش قابل اجرا (سرویس دهنده، سرویس گیرنده و پایگاه داده‌ها) بر روی گره‌های سخت‌افزاری مجزا تشکیل شده است. شکل زیر، نمودار استقرار سیستم ATM را نشان می‌دهد:



نمودار استقرار برای سیستم ATM

توجه: در بین گره‌ها، خطوطی رسم شده است، که نحوه ارتباط انجمنی مابین گره‌ها را بیان می‌کند، برای مثال «Regional ATMServer»، توسط شبکه محلی (LAN) به «Banking database server» متصل شده است. یک Stereotype برای توضیح ماهیت هر اتصال استفاده می‌شود.

توجه: نمودار استقرار به خوبی می‌تواند ساختار شبکه یک سیستم را نشان دهد. بنابراین، نمودار شبکه می‌تواند از روی نمودار استقرار استخراج گردد.

سیستم ATM یک سبک معماری سه طبقه دارد:

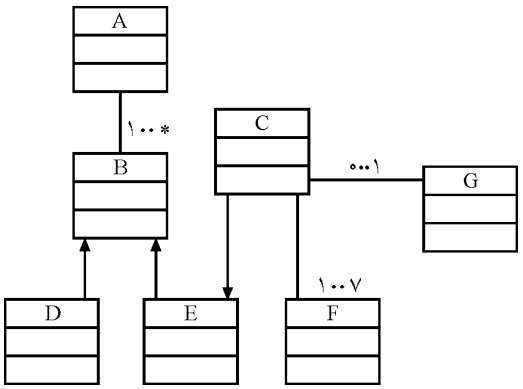
- سرویس‌دهنده
- سرویس‌گیرنده
- پایگاه داده‌ها

تست‌های فصل هفتم

۱- سیستم «مدیریت خانه» وجود وضعیت‌های نامطلوب مانند ورود غیرمجاز، آتش‌سوزی، ترکیدگی لوله آب و غیره را تشخیص می‌دهد و با انجام اقدامات لازم، خانه را در برابر این وضعیت‌ها محافظت می‌کند. این محصول، با استفاده از سنسورهای مناسب وضعیت‌های نامطلوب را کشف می‌کند و توسط صاحب‌خانه قابل برنامه‌ریزی است و در صورت کشف وضعیت نامطلوب به طور خودکار و از طریق خط تلفن، وضعیت را به اطلاع مرکز کنترل می‌رساند. در مدل مورد استفاده (use-case) این محصول، کدام‌یک از گزینه‌های زیر را به عنوان بازیگر (actor) در نظر گرفته نمی‌شود؟ (مهندسی IT – دولتی ۸۳)

۱) سنسور ۲) مرکز کنترل ۳) خط تلفن ۴) صاحب‌خانه

۲- با توجه به نمودار کلاس زیر گزینه‌ی صحیح را انتخاب کنید. (مهندسی IT – دولتی ۸۳)



۱) F بخشی از C می‌باشد. E نوعی از B می‌باشد. یک شیء از نوع G با چند شیء از نوع C رابطه دارد.

۲) C بخشی از E می‌باشد. D نوعی از B می‌باشد. یک شیء از نوع C با یک تا هفت شیء از نوع F رابطه دارد.

۳) E بخشی از C می‌باشد. E نوعی از B می‌باشد. یک شیء از نوع C با یک تا هفت شیء از نوع F رابطه دارد.

۴) C بخشی از E می‌باشد. B نوعی از D می‌باشد. یک شیء از نوع C با یک تا هفت شیء از نوع F رابطه دارد.

۳- کدام‌یک از نمودارهای UML در تجزیه و تحیل شیءگرا به شکل گرافیکی کارکردهای سیستم و ارتباط متقابل بین سیستم و موجودیت‌های خارج از سیستم را نشان می‌دهد؟ (مهندسی IT – دولتی ۸۴)

۱) use case diagram ۲) sequence diagram

۳) activity diagram ۴) collaboration diagram

۴- استخراج نیازمندی‌ها در شیءگرایی بر روی کدام‌یک از رویکردهای زیر استوار است؟ (مهندسی IT – دولتی ۸۴)

۱) مبتنی بر سرویس ۲) مبتنی بر هدف ۳) مبتنی بر فرآیند ۴) مبتنی بر وظیفه

۵- در مبحث شیءگرایی، زمانی که یک کلاس از تلفیق تعدادی کلاس دیگر تشکیل شود، دلالت بر چه نوع رابطه‌ای خواهد داشت؟ (مهندسی IT – آزاد ۸۵)

۱) رابطه وراثت (inheritance Relationship)

۲) رابطه انجمنی (Association Relationship)

۳) رابطه تجمعی (Aggregation Relationship)

۴) هیچ‌کدام

۶- در UML کدام‌یک از نمودارهای زیر، تعاملات درون مورد کاربردی (use-case) را نمایش می‌دهد؟ (مهندسی IT – آزاد ۸۶)

۱) نمودار کلاس (Class Diagram)

۲) نمودار تغییر حالت (State Transition Diagram)

۳) نمودار ترتیبی (Sequence Diagram)

۴) نمودار مؤلفه (Component Diagram)

۷- نمودارهای تعاملی (Interaction)، روابط میان کدام یک از گزینه‌های زیر را نمایش می‌دهد؟ (مهندسی IT – آزاد ۸۶)

۱) موارد کاربری (use cases) ۲) اشیاء (objects)

۳) کلاس‌ها (classes) ۴) بسته‌ها (packages)

۸- برای استخراج مشخصات رفتارهای ایستای (Static) موجودیت‌ها (Classes) کدام‌یک از ابزارهای زیر مناسب است؟ (مهندسی IT – دولتی ۸۷)

۱) Sequence Diagram, Use Case Model

۲) Collaboration Diagram, Use Case Model

۳) Sequence Diagram, Collaboration Diagram

۴) Use Case Model, State Transition Diagram

۹- یک انتقال حالت در نمودار حالت (State Chart Diagram) توسط کدام‌یک از عوامل زیر فعال می‌گردد؟ (مهندسی IT – آزاد ۸۷)

۱) عامل (actor) ۲) همکار (collaborator)

۳) رخداد (event) ۴) مؤلفه (component)

۱۰- کدام گزینه اولویت تعیین کلاس‌های سیستم و ترسیم نمودارهای ترتیبی (Sequence Diagrams) را به درستی بیان می‌نماید؟ (مهندسی IT – آزاد ۸۷)

۱) از آنجا که در نمودار ترتیبی به نمونه‌های کلاس نیاز داریم، لذا ابتدا می‌بایست کلیه کلاس‌های سیستم را مشخص نموده و سپس اقدام به ترسیم نمودار(های) ترتیبی نمود.
۲) همواره ابتدا نمودار(های) ترتیبی را رسم نموده و سپس اقدام به تعیین کلاس‌های سیستم می‌نماییم.
۳) نمودار ترتیبی و مشخص نمودن کلاس‌های سیستم دو فرآیند کاملاً متفاوت بوده و تقدم و تأخر در خصوص آنها مطرح نمی‌باشد.
۴) در ابتدا کلاس‌های اولیه را مشخص می‌نماییم و سپس نمودار(های) ترتیبی را رسم نموده و کلاس‌های سیستم را به روز می‌نماییم. به عبارت دیگر کلاس‌های سیستم به صورت تدریجی کامل می‌شود.

۱۱- کدام‌یک از نمودارهای UML ارایه‌کننده جریان کار در سطح سیستم، زیر سیستم، موارد کاربرد و کلاس‌ها می‌باشند؟ **(مهندسی IT – دولتی ۸۸)**

۱) Use Case Diagram ۲) Activity Diagram
۳) Class Diagram ۴) Collaboration Diagram

۱۲- تکمیل نمودن متدهای سیستم از وظایف کدام‌یک از نمودارهای زیر می‌باشد؟
(مهندسی IT – آزاد ۸۸)

۱) نمودار ترتیبی (Sequence Diagram) ۲) نمودار حالت (Statechart Diagram)
۳) نمودار مؤلفه (Component Diagram) ۴) نمودار استقرار (Deployment Diagram)

۱۳- در نمودار مؤلفه (Component Diagram)، رابطه وابستگی (Dependency) به چه منظور در میان مؤلفه‌ها قرار می‌گیرد؟ **(مهندسی IT – آزاد ۸۸)**

۱) مشخص نمودن ترتیب ایجاد مؤلفه‌ها ۲) مشخص نمودن ترتیب کامپایل
۳) نگاشت کلاس‌ها به مؤلفه‌ها ۴) نگاشت مؤلفه‌ها به بسته‌ها

۱۴- کدام عبارت صحیح است؟ **(مهندسی IT – دولتی ۸۹)**

۱) در تحلیل نیازها، افراز (Partitioning) مسأله منجر به تشریح دقیق‌تر داده‌ها، توابع و رفتار خواهد شد.
۲) Actorها در Use-Caseها افرادی هستند که کاربران نرم‌افزار خواهند بود.
۳) برای آنکه نمونه‌سازی نرم‌افزار عملی مؤثر باشد، نیاز به ابزاری برای توسعه‌ی سریع نمونه‌ها داریم تا طبق زمان‌بندی پیش برویم.
۴) وقتی که مستندات تعریف نیازهای نرم‌افزار به وسیله‌ی مشتری و توسعه‌دهنده، تأیید شد این اسناد تبدیل به مستنداتی غیرقابل تغییر خواهند شد.

۱۵- کدام عبارت نادرست است؟ **(مهندسی IT – دولتی ۸۹)**

۱) برای بازنگری یک مدل CRC کامل، بازنگری‌کننده، نیاز به توجه به تمام نمایش‌های موردکاربرد (Use Cases) دارد.

۲) مقادیری که به صفات یک شیء اختصاص می‌یابند، آن شیء را یکتا می‌نمایند.
۳) وراثت شامل مکانیزمی است که تغییرات در کلاس‌های سطح پایین به سرعت منجر به تعمیم روی تمام ابرکلاس‌ها می‌گردد.
۴) در حالتی که یک کلاس برخی از صفات و عملیات یک کلاس و برخی دیگر را از کلاس دیگر به ارث می‌برد، در این حالت وراثت چندگانه مطرح می‌گردد.

۱۶- کدام عبارت جزء ارتباطات عمومی بین کلاس‌های مشارکت‌کننده نیست و تحلیل‌گر نمی‌تواند براساس آن، کلاس‌های مشارکت را تشخیص دهد؟ (مهندسی IT – دولتی ۸۹)

۱) بخشی است از (...is-part-of)
۲) واقع شدن قبل از (...comes-before)
۳) آگاه است از (... has-knowledge-of)
۴) وابسته است به (...depends-upon)

۱۷- ارتباط میان اشیاء در شیءگرایی به چه صورت انجام می‌پذیرد؟ (مهندسی IT – آزاد ۸۹)

۱) شمول (Include)
۲) بسط (Extend)
۳) وابستگی (Dependency)
۴) تبادل پیام (Message Passing)

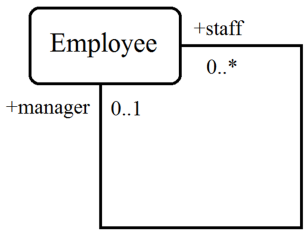
۱۸- کدام‌یک از موارد زیر برای گنجاندن در طراحی جزیی یک نرم‌افزار مناسب نیست؟ (مهندسی IT – دولتی ۹۰)

۱) حداکثر زمانی که استفاده‌کننده بایستی منتظر پاسخ سیستم بماند.
۲) نحوه ذخیره اطلاعات مربوط به عملیات انجام شده در هر روز
۳) اتفاقاتی که باید در صورت قطع اتصال از شبکه کامپیوتری در سیستم بیفتد.
۴) هیچ‌کدام

۱۹- در مدل‌سازی سیستم‌ها در جریان تحلیل و طراحی، کدام‌یک از نمودارهای زیر نوعاً نسخه‌ی فیزیکی ندارد و همواره منطقی باقی می‌ماند؟ (مهندسی IT – دولتی ۹۰)

۱) نمودار جریان داده (DFD)
۲) نمودار تعامل (Interaction Diagram)
۳) نمودار مورد کاربرد (Use Case Diagram)
۴) نمودار کلاس (Class Diagram)

۲۰- کلاس Employee (عضو اداره) با نقش‌های Staff (کارمندان) و manager (مدیر) را در نظر بگیرید. شکل زیر چه رابطه‌ای را بیان می‌نماید؟ (مهندسی IT – آزاد ۹۰)



۱) Self Aggregation
۲) Message to Self
۳) Self Association
۴) تنها با ملاحظه شکل نمی‌توان نوع رابطه را بیان نمود.

۲۱- کدام‌یک از موارد زیر، کاربرد اصلی روش CRC (Class-Responsibility-Collaborator) است؟ (مهندسی IT – دولتی ۹۲)

۱) تشخیص کلاس‌ها ۲) تشخیص روابط توارث بین کلاس‌ها
۳) تشخیص روابط کل – جزء بین کلاس‌ها ۴) تشخیص ترتیب تبادل پیغام‌ها بین کلاس‌ها

۲۲- کدام‌یک از نمودارهای UML زیر، نوعاً شیءگرا (یعنی مبتنی بر مفاهیم خاص شیءگرایی) نیست؟ (مهندسی IT – دولتی ۹۲)

۱) نمودار مورد کاربرد (Use Case) ۲) نمودار ترتیب (Sequence)
۳) نمودار رده (Class) ۴) نمودار ارتباط (Communication)

۲۳- کدام‌یک از گروه Diagramهای زیر برای OOA صحیح نیست؟ (مهندسی IT – دولتی ۹۳)

۱) Class Diagram, Business Use Case
۲) Sequence Diagram, Use Case Diagram, CRC
۳) Flow-Oriented Diagram, Deployment Diagram
۴) Activity Diagram, Swimlane Diagram, Object Diagram

۲۴- بکارگیری مدل ساختاری (structural model) برای نمایش طراحی معماری دارای کدام مشخصات زیر است؟ (مهندسی IT – دولتی ۹۴)

۱) یک مدل ساختاری عناصر تکراری را در کاربردهای مشابه نمایش می‌دهد.
۲) یک مدل ساختاری نمایش ساختار واحدها و مولفه‌های برنامه است.
۳) یک مدل ساختاری رفتار سیستم را در قبال رخدادها نشان می‌دهد.
۴) مورد ۱ و ۲ صحیح است.

۲۵- کدام نمودار UML، برای مدل‌سازی تعامل اشیاء به کار می‌رود؟ (مهندسی IT – دولتی ۹۶)

۱) حالت (State Diagram) ۲) مولفه (Component Diagram)
۳) فعالیت (Activity Diagram) ۴) توالی (Sequence Diagram)

۲۶- کدام نمودار UML، اساسا برای مدل‌سازی ساختار(معماری) محصول نرم‌افزاری به کار می‌رود؟ (مهندسی IT – دولتی ۹۷)

۱) نمودار فعالیت (Activity Diagram) ۲) نمودار مولفه (Component Diagram)
۳) نمودار توالی (Sequence Diagram) ۴) نمودار مورد کاربرد (Use-Case Diagram)

۲۷- کدام نمودار UML برای توصیف بصری موارد کاربرد (Use cases) مورد استفاده قرار می‌گیرد؟ (مهندسی IT – دولتی ۹۸)

۱) نمودار فعالیت ۲) ماشین حالت ۳) نمودار کلاس ۴) نمودار مولفه

۲۸- کدام نمودار UML می‌تواند برای مدل‌سازی منطق داخلی عملیات یک کلاس به کار برده شود؟ (مهندسی IT – دولتی ۹۹)

۱) نمودار شیء (Object Diagram) ۲) نمودار بسته (Package Diagram)
۳) نمودار فعالیت (Activity Diagram) ۴) نمودار مولفه (Component Diagram)

۲۹- کدام‌یک از نمودارهای UML زیر، محیط رایانش (Computing Environment) را نیز شامل می‌شود؟ (مهندسی IT – دولتی ۱۴۰۰)

۱) نمودار مستقرسازی (Deployment Diagram)

۲) نمودار شیء (Object Diagram)

۳) نمودار بسته (Package Diagram)

۴) نمودار فعالیت (Activity Diagram)

۳۰- کدام‌یک از موارد زیر در توصیف تفصیلی یک مورد کاربرد (Use Case) آورده می‌شود؟ (مهندسی IT – دولتی ۱۴۰۰)

۱) پیش‌شرایط و پس‌شرایط مورد کاربرد

۲) کلاس‌ها و اشیاء محقق‌کننده مورد کاربرد

۳) کد پیاده‌سازی شده مورد کاربرد

۴) موارد آزمون طراحی شده برای مورد کاربرد

پاسخ تست‌های فصل هفتم

۱- گزینه (۳) صحیح است.

Use Case Diagram یا نمودار مورد کاربرد، یا نمودار نیاز جهت مدل‌سازی لیست نیازمندی‌های مشتری مورد استفاده قرار می‌گیرد.

Use Caseها و Actorها محدوده سیستم در حال ساخت را مشخص می‌کنند. Use Case شامل تمام آن چیزهایی است که درون سیستم قرار دارد و Actor شامل تمام آن چیزهایی است که خارج از سیستم قرار دارد. هر فرد یا هر چیزی که با سیستم تعامل دارد، Actor یا بازیگر نامیده می‌شود. Use Caseها هر چیز موجود در داخل و محدوده سیستم را توصیف می کنند، در حالی که Actorها هر چیز موجود در خارج از محدوده سیستم را توصیف می‌کنند.

بازیگران، افراد و یا گاهی نرم‌افزار و یا سخت‌افزارهایی هستند که از سیستم استفاده می‌کنند و یا اطلاعاتی را برای سیستم فراهم می‌کنند. با اینکه همه بازیگران، عناصر خارجی سیستم هستند، اما با این حال هر عنصر خارج سیستم، بازیگر نیست. بازیگران استفاده کنندگان نهایی و یا تولیدکنندگان ابتدایی اطلاعات هستند، بنابراین عناصر خارجی سیستم که تنها وظیفه انتقال اطلاعات را دارند و نقش **رسانه‌انتقال** را ایفا می‌کنند، نمی‌توانند به عنوان بازیگر در نظر گرفته شوند.

برای مثال اگر یک سنسور از طریق رسانه بی‌سیم، اطلاعاتی را به سیستم مرکزی ارسال می‌کند، رسانه بی‌سیم نمی‌تواند بازیگر این سیستم باشد، بلکه سنسور بازیگر آن خواهد بود. بنابراین هر بازیگر اگر استفاده‌کننده باشد نقطه انتها و اگر تولیدکننده اطلاعات باشد نقطه ابتدای آن ارتباط است. بنابراین گزینه سوم نادرست است. زیرا خط تلفن، رسانه‌انتقال است. اما سایر گزینه‌ها به عنوان تولیدکننده اطلاعات یا مصرف‌کننده اطلاعات می‌توانند بازیگر در نظرگرفته شوند.

۲- گزینه (۲) صحیح است.

دو انسانی که همدیگر را می‌شناسند و امکان گفتگو و تبادل پیام با هم دارند، رابطه انجمنی با هم دارند. در رابطه انجمنی(Association Relationship) یک شیء بطور ساده درباره شیء دیگر می‌داند به همان طریقی که یک فرد ممکن است فرد دیگری را بشناسد. یک برنامه کامپیوتری شیءگرا از اجتماع تعدادی کلاس ایجاد شده است، کلاس‌های همکار بدون رابطه جزء و کل و هم‌هدف در یک بخش مشترک از برنامه، کلاس‌های همکار با رابطه جزء و کل و هم‌هدف در یک بخش مشترک از برنامه و کلاس‌های غیرهمکار در دو بخش مختلف از برنامه، رابطه‌ای که میان کلاس‌های همکار بدون رابطه جزء و کل وجود دارد، رابطه انجمنی است.

کلاس‌های همکار بدون رابطه جزء و کل، جهت انجام وظایف خود از طریق مکانیزم پیام به گفتگو با یکدیگر می‌پردازند. در یک بیان ساده، هرگاه میان دو شیء بدون رابطه جزء و کل گفتگو باشد، کلاس‌های این دو شیء هم باهم همکار هستند و هم رابطه انجمنی میان آنها برقرار است.

رابطه انجمنی به عنوان رابطه «has knowledge of» نیز شناخته می‌شود.

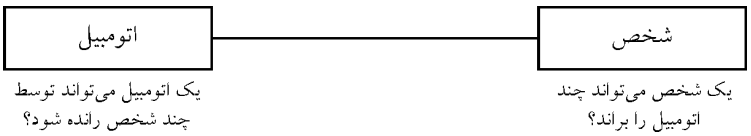
هر رابطه انجمنی از سه قسمت اصلی تشکیل می‌شود که عبارتند از:

۱) کلاس‌های شرکت‌کننده در رابطه
۲) خط ممتد رابطه که بین دو کلاس ترسیم می‌شود.
۳) نام رابطه که در وسط خط رابطه نوشته می‌شود و بیان‌کننده هدف رابطه است.

در مورد روابط انجمنی، بین کلاس‌ها، سوالات مهمی مطرح می‌شود. برای مثال، در مورد رابطه انجمنی شخص و اتومبیل، سوالات زیر مطرح است:

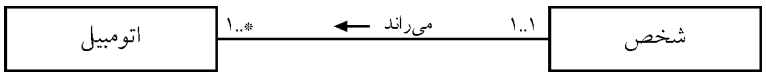
«یک شخص می‌تواند چند اتومبیل را براند؟»، «پاسخ: *..۱»

«یک اتومبیل می‌تواند توسط چند شخص رانده شود؟»، «پاسخ: *..۱»



UML برای پاسخگویی به چنین سوالاتی، مشخصه‌ای با نام **تعدد** (Multiplicity) را برای هر یک از کلاس‌های طرفین رابطه ارائه نموده است. تعدد اصطلاحی است که تعداد اشیای شرکت‌کننده در رابطه انجمنی را مشخص می‌کند. اعدادی که در پاسخ به این دو سوال داده می‌شود، مبین تعدد طرفین است. شیوه‌های مختلفی جهت تعیین تعدد وجود دارد، اما متداول‌ترین شیوه، بیان محدوده تعداد اشیای شرکت‌کننده در هر طرف رابطه است که به صورت «حداقل .. حداکثر» نشان داده می‌شود.

مثال: تعدد رابطه انجمنی بین شخص و اتومبیل به صورت زیر است:



مطابق نمودار مطرح شده در صورت سوال، رابطه کلاس A و B ، همچنین رابطه کلاس C و G و همچنین رابطه کلاس C و F از نوع رابطه انجمنی است.

همچنین یک شیء از نوع C حداقل با یک و حداکثر با هفت شیء ازنوع F رابطه دارد.

به طورکلی هرگاه یک کلاس (ب)، **از نوع** یک کلاس (الف) باشد، گوییم بین دو کلاس مذکور رابطه **وراثت** برقرار است. در این صورت کلاس (ب) فرزند و کلاس (الف) پدر یا والد نامیده می‌شود. به عبارت دیگر همانطور که پیش از این نیز گفتیم، وراثت فرآیندی است که به وسیله آن یک کلاس (فرزند) می‌تواند صفات و متدهای کلاس دیگری (پدر) را کسب کند. به عبارت کلی‌تر یک کلاس فرزند ضمن به ارث بردن مجموعه‌ای از صفات و متدهای عمومی کلاس پدر، می‌تواند ویژگی‌های خاص و مختص خود را نیز به آنها اضافه کند. در رابطه وراثت هر تغییر در کلاس پدر بر کلاس فرزند نیز اثر می‌گذارد اما عکس این مطلب برقرار نیست.

رابطه وراثت به عنوان رابطه «is a» یا «kind of» یا «type of» نیز شناخته می‌شود.

مراحل لازم جهت مدل‌سازی یک رابطه ارث بری عبارتند از:

۱- ترسیم یک خط ممتد بین کلاس فرزند و کلاس پدر

۲- درج یک مثلث تو خالی در انتهایی از خط ممتد که کلاس پدر قرار دارد.
ذکر «is a» بر روی خط ممتد رابطه وراثت اختیاری است.
در تعریف رابطه وراثت، مشخصاتی نظیر تعدد و محدودیت وجود ندارد.

رابطه کلاس D و B از نوع رابطه ارث‌بری است، بدین نحو که کلاس D فرزند کلاس B است. بنابراین کلاس D نوعی از کلاس B است.

رابطه کلاس E و B از نوع رابطه ارث‌بری است، بدین نحو که کلاس E فرزند کلاس B است. بنابراین کلاس E نوعی از کلاس B است.

رابطه کلاس C و E از نوع رابطه ارث‌بری است، بدین نحو که کلاس C فرزند کلاس E است. بنابراین کلاس C نوعی از کلاس E است.

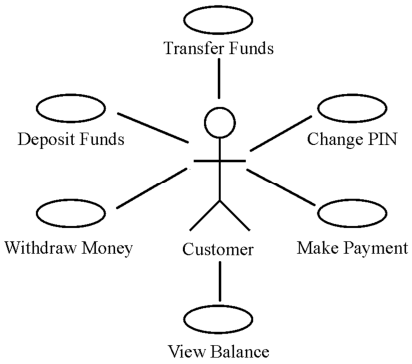
بنابر مطالب فوق واضح است که گزینه دوم درست است و گزینه‌های اول، سوم و چهارم نادرست هستند. سازمان سنجش آموزش کشور نیز گزینه دوم را به عنوان پاسخ درست اعلام کرده بود.

البته بهتر بود طراح محترم در گزینه دوم با توجه به استفاده از نماد ارث‌بری مابین کلاس C و E در نمودار مطرح شده به جای عبارت «C بخشی از کلاس E است» عبارت «C نوعی از کلاس E است» را به کار می‌برد، زیرا رابطه بخشی یا انجمنی پیشرفته رابطه‌ای مابین یک واحد کل و تعدادی واحد جزء است، یعنی یک کلاس کل از همکاری تعدادی کلاس جزء تشکیل شده باشد. که خود بر دو نوع تجمع با نماد لوزی توخالی و ترکیب با نماد لوزی توپر است. البته خود نماد ارث‌بری نشان‌داده شده در شکل صورت سوال نادرست است. زیرا در ارث‌بری نماد یک مثلث تو خالی در انتهایی از خط ممتد که کلاس پدر قرار دارد، درج می‌گردد. البته شاید هم طراح محترم و یا تایپیست محترم در نشانه‌گذاری و انتخاب نماد مناسب دچار خطا شده‌اند. یعنی اگر در رابطه بین C و E از نماد لوزی توخالی یا توپر استفاده می‌شد، آنگاه گزینه دوم بدون اصلاح و به همین شکل مطرح شده درست می‌بود. تستی که تا ابد راز نهفته در آن کشف نخواهد شد!

۳- گزینه (۱) صحیح است.

Use Case Diagram یا نمودار مورد کاربرد، یا نمودار نیاز جهت مدل‌سازی لیست نیازمندی‌های مشتری مورد استفاده قرار می‌گیرد.

مثال: مدل‌سازی لیست نیازمندی‌های مشتری برای سیستم ATM.



در نمودار فوق، لیست نیازمندی‌های مشتری، مدل‌سازی شده است.

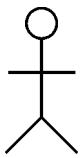
به هر یک از نیازهای فوق یک use case یا مورد کاربرد گفته می‌شود و به اجتماع این use caseها، Use Case Diagram یا نمودار مورد کاربرد گفته می‌شود.

Use Caseها و Actorها محدوده سیستم در حال ساخت را مشخص می‌کنند. Use Case شامل تمام آن چیزهایی است که درون سیستم قرار دارد و Actor شامل تمام آن چیزهایی است که خارج از سیستم قرار دارد. هر فرد یا هر چیزی که با سیستم تعامل دارد، Actor یا بازیگر نامیده می‌شود. Use Caseها هر چیز موجود در داخل و محدوده سیستم را توصیف می‌کنند، در حالی که Actorها هر چیز موجود در خارج از محدوده سیستم را توصیف می‌کنند.

بازیگران، افراد و یا گاهی نرم‌افزار و یا سخت‌افزارهایی هستند که از سیستم استفاده می‌کنند و یا اطلاعاتی را برای سیستم فراهم می‌کنند. با اینکه همه بازیگران، عناصر خارجی سیستم هستند، اما با این حال هر عنصر خارج سیستم، بازیگر نیست. بازیگران استفاده کنندگان نهایی و یا تولیدکنندگان ابتدایی اطلاعات هستند، بنابراین عناصر خارجی سیستم که تنها وظیفه انتقال اطلاعات را دارند و نقش **رسانه انتقال** را ایفا می‌کنند، نمی‌توانند به عنوان بازیگر در نظر گرفته شوند.

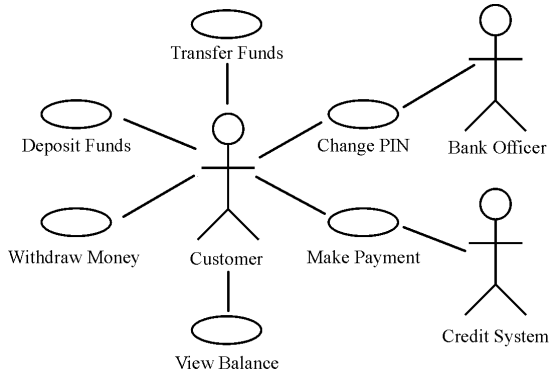
برای مثال اگر یک سنسور از طریق رسانه بی‌سیم، اطلاعاتی را به سیستم مرکزی ارسال می‌کند، رسانه بی‌سیم نمی‌تواند بازیگر این سیستم باشد، بلکه سنسور بازیگر آن خواهد بود. بنابراین هر بازیگر اگر استفاده‌کننده باشد نقطه انتها و اگر تولیدکننده اطلاعات باشد نقطه ابتدای آن ارتباط است.

در UML، بازیگران با آدمک‌هایی به شکل مقابل نشان داده می‌شوند:



Actor

نمونه‌ای از استفاده نمودار Use Case در شکل زیر نشان داده شده است: (سیستم ATM)



Sequence Diagram یا نمودار توالی و Collaboration Diagram یا نمودار همکاری جهت مدل‌سازی تعاملات پویای میان اشیاء همکار داخل یک use case مورد استفاده قرار می‌گیرند. بنابراین گزینه دوم و چهارم نادرست هستند.

Activity Diagram یا نمودار فعالیت و Swimlane Diagram یا نمودار خط شنا، جهت مدل‌سازی سناریوی اصلی و فرعی داخل یک use case مورد استفاده قرار می‌گیرد. به بیان دیگر نمودار فعالیت یا نمودار خط شنا، جهت مدل‌سازی روال انجام کارها داخل یک use case مورد استفاده قرار می‌گیرد. همچنین، در مدل طراحی در بخش طراحی مؤلفه، باید جزئیات الگوریتمی متدهای کلاس تشریح شود. برای این منظور Activity Diagram یا نمودار فعالیت و Swimlane Diagram یا نمودار خط‌شنا مورد استفاده قرار می‌گیرد. بنابراین گزینه سوم نیز نادرست است.

۴- گزینه (۱) صحیح است.

استخراج نیازمندی‌های مشتری در شیءگرایی، بر اساس موارد کاربرد (use case) است، که سرویس‌های سیستم را از دید عوامل خارج سیستم منعکس می کنند.

۵- گزینه (۳) صحیح است.

دو انسانی که همدیگر را می‌شناسند و امکان گفتگو و تبادل پیام با هم دارند، رابطه انجمنی با هم دارند. در رابطه انجمنی یک شیء بطور ساده درباره شیء دیگر می‌داند به همان طریقی که یک فرد ممکن است فرد دیگری را بشناسد. یک برنامه کامپیوتری شیءگرا از اجتماع تعدادی کلاس ایجاد شده است، کلاس‌های همکار بدون رابطه جزء و کل **(بدون تلفیق تعدادی کلاس)** و هم‌هدف در یک بخش مشترک از برنامه، کلاس‌های همکار با رابطه جزء و کل **(با تلفیق تعدادی کلاس)** و هم‌هدف در یک بخش مشترک از برنامه و کلاس‌های غیرهمکار در دو بخش مختلف از برنامه، رابطه‌ای که میان کلاس‌های همکار بدون رابطه جزء و کل وجود دارد، رابطه انجمنی است.

کلاس‌های همکار بدون رابطه جزء و کل، جهت انجام وظایف خود از طریق مکانیزم پیام به گفتگو با یکدیگر می‌پردازند. در یک بیان ساده، هرگاه میان دو شیء بدون رابطه جزء و کل **(بدون تلفیق تعدادی کلاس)** گفتگو باشد، کلاس‌های این دو شیء هم باهم همکار هستند و هم رابطه انجمنی میان آنها برقرار است. بنابراین گزینه دوم نادرست است.

رابطه انجمنی پیشرفته به دو رابطه تجمع و ترکیب تقسیم می‌گردد.

رابطه تجمع (Aggregation Relationship) رابطه‌ای مابین یک واحد کل و تعدادی واحد جزء است، یعنی یک کلاس کل از همکاری تعدادی کلاس جزء تشکیل شده باشد. **(با تلفیق تعدادی کلاس)**

در رابطه تجمع، حیات شیء جزء به حیات شیء کل وابسته نیست. به بیان دیگر حیات شیء جزء و شیء کل به هم تقدم و تاخر دارند. یعنی ممکن است شیء جزء موجود باشد، بدون اینکه شیء کل ایجاد گردد و موجود باشد. در رابطه تجمع اگر شیء کل از بین برود، اشیاء جزء، همچنان به حیات خود ادامه می‌دهند.

در رابطه **انجمنی‌ساده**، مابین طرفین همکار رابطه، رابطه جزء و کل مطرح نیست، در حالی که در رابطه **تجمع**، مابین طرفین همکار رابطه، رابطه جزء و کل مطرح است و از **تجمع** و سازماندهی اجزاء، یک موجودیت کامل‌تر ساخته می‌شود.

رابطه ترکیب (Composition Relationship) رابطه‌ای مابین یک واحد کل و تعدادی واحد جزء است، یعنی یک کلاس کل از همکاری تعدادی کلاس جزء تشکیل شده باشد. **(با تلفیق تعدادی کلاس)** به نحوی که شیء کل و جزء با هم ایجاد شوند و با هم از بین بروند.

در رابطه ترکیب، حیات شیء جزء به حیات شیء کل وابسته است. به بیان دیگر حیات شیء جزء و شیء کل به هم تقدم و تاخر ندارند. یعنی امکان ندارد شیء جزء موجود باشد، بدون اینکه شیء کل ایجاد گردد و موجود باشد. در رابطه ترکیب اگر شیء کل از بین برود، اشیاء جزء نیز به طور همزمان با شیء کل از بین می‌روند. زیرا از ابتدای کار، شیء کل، به طور همزمان از ایجاد و کنار هم قرار دادن اشیاء جزء توسط برنامه کامپیوتری ایجاد شده است و با پایان یافتن فعالیت شیء کل، شیء کل و جزء باهم و همزمان از بین می‌روند.

در رابطه **انجمنی‌ساده**، مابین طرفین همکار رابطه، رابطه جزء و کل مطرح نیست، در حالی که در رابطه **ترکیب**، مابین طرفین همکار رابطه، رابطه جزء و کل مطرح است و از **ترکیب** و سازماندهی اجزاء، یک موجودیت کامل‌تر ساخته می‌شود.

از آنجا که در گزینه‌ها رابطه ترکیب مطرح نشده است، آنرا کنار می‌گذاریم. بنابراین گزینه سوم درست است.

به طورکلی هرگاه یک کلاس (ب)، **از نوع** یک کلاس (الف) باشد، گوییم بین دو کلاس مذکور رابطه **وراثت** برقرار است. در این صورت کلاس (ب) فرزند و کلاس (الف) پدر یا والد نامیده می‌شود. به عبارت دیگر، وراثت فرآیندی است که به وسیله آن یک کلاس (فرزند) می‌تواند صفات و متدهای کلاس دیگری (پدر) را کسب کند. به عبارت کلی‌تر یک کلاس فرزند ضمن به ارث بردن مجموعه‌ای از صفات و متدهای عمومی کلاس پدر، می‌تواند ویژگی‌های خاص و مختص خود را نیز به آنها اضافه کند. بنابراین گزینه اول نادرست است.

۶- گزینه (۳) صحیح است.

مدل‌سازی تعاملات پویای میان اشیاء همکار یا مدل‌سازی رفتاری، رفتار سیستم را در زمان اجرا نمایش می‌دهد. به بیان دیگر در این دیدگاه رفتارهای پویای سیستم مدل می‌شود.

مدل‌سازی رفتاری سیستم با استفاده از نمودارهای UML قابل انجام است:

الف) نمودار توالی (Sequence Diagram)

نمودار کلاس ساختار ایستای داخل یک use case را نمایش می‌دهد. در یک سیستم در حال کار، به هر حال اشیاء با یکدیگر در تعامل هستند و این تعامل در طی زمان رخ می‌دهد. به بیان دیگر برای آنکه یک use case یا مورد کاربرد یا نیاز مرتفع گردد، باید مجموعه‌ای از اشیاء با یکدیگر ارتباط برقرار کرده و پیام‌هایی را با یکدیگر رد و بدل نمایند. نمودار توالی نشان می‌دهد، برای مرتفع شدن یک use case یا مورد کاربرد یا نیاز، چه اشیایی باید چه پیام‌هایی را با چه

ترتیبی ارسال کنند تا آن نیاز برآورده گردد. نمودار توالی، تعاملات پویا مابین اشیاء را براساس زمان نشان می‌دهد.

Sequence Diagram یا نمودار توالی، Object Diagram یا نمودار شیء جهت مدل‌سازی تعاملات پویای میان اشیاء همکار داخل یک use case مورد استفاده قرار می‌گیرد.

ب) نمودار همکاری (Collaboration Diagram)

Collaboration Diagram یا نمودار همکاری در نسخه‌های امروزی UML، به نمودار **Communication Diagram** یا **نمودار ارتباط**، تغییر نام یافته است.

Collaboration Diagram یا نمودار همکاری، برای یک کاربرد خاص، جهت مدل‌سازی تعاملات پویای میان اشیاء همکار داخل یک use case مورد استفاده قرار می‌گیرد.

این نمودار نیز، مانند نمودار توالی، تعاملات بین اشیاء یک مورد کاربرد را نشان می‌دهد.

ج) نمودار حالت (State Transition Diagram)

State Transition Diagram یا نمودار انتقال حالت (وضعیت)، به عنوان یک نمودار مکمل sequence Diagram، برای مشخص‌کردن وضوح بیشتر برای شناخت حالت‌های مختلف یک شیء مورد استفاده قرار می‌گیرد. بنابراین گزینه دوم نادرست است.

همه نمودارهای فوق(توالی، همکاری و حالت)، مدل‌سازی رفتاری محسوب می‌گردند.

بنابر مطالب فوق Sequence Diagram یا نمودار توالی و Collaboration Diagram یا نمودار همکاری جهت مدل‌سازی تعاملات پویای میان اشیاء همکار داخل یک use case مورد استفاده قرار می‌گیرند. از آنجاکه Collaboration Diagram یا نمودار همکاری در گزینه‌ها وجود ندارد، پس آنرا کنار می‌گذاریم، بنابراین گزینه سوم درست است.

Class Diagram یا نمودار کلاس جهت مدل‌سازی ارتباطات ایستای میان کلاس‌های همکار داخل یک use case مورد استفاده قرار می‌گیرد. بنابراین گزینه اول نادرست است.

Component Diagram یا نمودار مؤلفه جهت مدل‌سازی ساختار کلی پیاده‌سازی برنامه و ترتیب کامپایل مؤلفه‌های فرعی، برای ایجاد مؤلفه‌اصلی (مؤلفه کلی یا برنامه اصلی) مورد استفاده قرار می‌گیرد. به بیان دیگر نمودار مؤلفه یک دید فیزیکی از مدل سیستم به همراه مؤلفه‌های فرعی نرم‌افزار و روابط بین آنها را نشان می‌دهد. بنابراین گزینه چهارم نادرست است.

۷- گزینه (۲) صحیح است.

مدل‌سازی تعاملات پویای میان **اشیاء همکار** یا مدل‌سازی رفتاری، رفتار سیستم را در زمان اجرا نمایش می‌دهد. به بیان دیگر در این دیدگاه رفتارهای پویای سیستم مدل می‌شود.

مدل‌سازی رفتاری سیستم با استفاده از نمودارهای UML قابل انجام است:

الف) نمودار توالی (Sequence Diagram)

نمودار کلاس ساختار ایستای داخل یک use case را نمایش می‌دهد. در یک سیستم در حال کار، به هر حال اشیاء با یکدیگر در تعامل هستند و این تعامل در طی زمان رخ می‌دهد. به بیان

دیگر برای آنکه یک use case یا مورد کاربرد یا نیاز مرتفع گردد، باید مجموعه‌ای از اشیاء با یکدیگر ارتباط برقرار کرده و پیام‌هایی را با یکدیگر رد و بدل نمایند. نمودار توالی نشان می‌دهد، برای مرتفع شدن یک use case یا مورد کاربرد یا نیاز، چه اشیایی باید چه پیام‌هایی را با چه ترتیبی ارسال کنند تا آن نیاز برآورده گردد. نمودار توالی، تعاملات پویا مابین اشیاء همکار را براساس زمان نشان می‌دهد.

Sequence Diagram یا نمودار توالی، Object Diagram یا نمودار شیء جهت مدل‌سازی تعاملات پویای میان اشیاء همکار داخل یک use case مورد استفاده قرار می‌گیرد.

ب) نمودار همکاری (Collaboration Diagram)

Collaboration Diagram یا نمودار همکاری در نسخه‌های امروزی UML، به نمودار **Communication Diagram** یا **نمودار ارتباط**، تغییر نام یافته است.

Collaboration Diagram یا نمودار همکاری، برای یک کاربرد خاص، جهت مدل‌سازی تعاملات پویای میان اشیاء همکار داخل یک use case مورد استفاده قرار می‌گیرد.

این نمودار نیز، مانند نمودار توالی، تعاملات بین اشیاء یک مورد کاربرد را نشان می‌دهد.

ج) نمودار حالت (State Transition Diagram)

State Transition Diagram یا نمودار انتقال حالت (وضعیت)، به عنوان یک نمودار مکمل sequence Diagram، برای مشخص‌کردن وضوح بیشتر برای شناخت حالت‌های مختلف یک شیء مورد استفاده قرار می‌گیرد.

همه نمودارهای فوق (توالی، همکاری و حالت)، مدل‌سازی رفتاری محسوب می‌گردند.

بنابر مطالب فوق Sequence Diagram یا نمودار توالی و Collaboration Diagram یا نمودار همکاری جهت مدل‌سازی تعاملات پویای میان اشیاء همکار داخل یک use case مورد استفاده قرار می‌گیرند. بنابراین گزینه دوم درست است.

Use Case Diagram یا نمودار مورد کاربرد، یا نمودار نیاز جهت مدل‌سازی لیست نیازمندی‌های مشتری مورد استفاده قرار می‌گیرد. بنابراین گزینه اول نادرست است.

Class Diagram یا نمودار کلاس جهت مدل‌سازی ارتباطات ایستای میان کلاس‌های همکار داخل یک use case مورد استفاده قرار می‌گیرد. بنابراین گزینه سوم نیز نادرست است.

بسته (Package) مجموعه‌ای از کلاس‌های پیاده‌سازی‌شده و کامپایل‌شده به عنوان واحد مولفه و یا تعدادی مولفه فرعی پیاده‌سازی‌شده و کامپایل‌شده است. که از اجتماع بسته‌ها یک سیستم نرم‌افزاری ایجاد می‌گردد. بنابراین گزینه چهارم نیز نادرست است.

۸- گزینه (۳) صحیح است.

مدل‌سازی تعاملات پویای میان **اشیاء همکار** یا مدل‌سازی رفتاری، رفتار سیستم را در زمان اجرا نمایش می‌دهد. به بیان دیگر در این دیدگاه رفتارهای پویای سیستم مدل می‌شود.

مدل‌سازی رفتاری سیستم با استفاده از نمودارهای UML قابل انجام است:

الف) نمودار توالی (Sequence Diagram)

نمودار کلاس ساختار ایستای داخل یک use case را نمایش می‌دهد. در یک سیستم در حال کار، به هر حال اشیاء با یکدیگر در تعامل هستند و این تعامل در طی زمان رخ می‌دهد. به بیان دیگر برای آنکه یک use case یا مورد کاربرد یا نیاز مرتفع گردد، باید مجموعه‌ای از اشیاء با یکدیگر ارتباط برقرار کرده و پیام‌هایی را با یکدیگر رد و بدل نمایند. نمودار توالی نشان می‌دهد، برای مرتفع شدن یک use case یا مورد کاربرد یا نیاز، چه اشیایی باید چه پیام‌هایی را با چه ترتیبی ارسال کنند تا آن نیاز برآورده گردد. نمودار توالی، تعاملات پویا مابین اشیاء همکار را براساس زمان نشان می‌دهد.

Sequence Diagram یا نمودار توالی، Object Diagram یا نمودار شیء جهت مدل‌سازی تعاملات پویای میان اشیاء همکار داخل یک use case مورد استفاده قرار می‌گیرد.

ب) نمودار همکاری (Collaboration Diagram)

Collaboration Diagram یا نمودار همکاری در نسخه‌های امروزی UML، به نمودار **Communication Diagram** یا **نمودار ارتباط**، تغییر نام یافته است.

Collaboration Diagram یا نمودار همکاری، برای یک کاربرد خاص، جهت مدل‌سازی تعاملات پویای میان اشیاء همکار داخل یک use case مورد استفاده قرار می‌گیرد.

این نمودار نیز، مانند نمودار توالی، تعاملات بین اشیاء یک مورد کاربرد را نشان می‌دهد.

ج) نمودار حالت (State Transition Diagram)

State Transition Diagram یا نمودار انتقال حالت (وضعیت)، به عنوان یک نمودار مکمل Sequence Diagram، برای مشخص‌کردن وضوح بیشتر برای شناخت حالت‌های مختلف یک شیء مورد استفاده قرار می‌گیرد.

همه نمودارهای فوق (توالی، همکاری و حالت)، مدل‌سازی رفتاری محسوب می‌گردند.

بنابر مطالب فوق Sequence Diagram یا نمودار توالی و Collaboration Diagram یا نمودار همکاری جهت مدل‌سازی تعاملات پویای میان اشیاء همکار داخل یک use case مورد استفاده قرار می‌گیرند.

از آنجا که وقوع رخداد در نمودارهای توالی و همکاری (ارتباط) نقشی ندارد، به نمودارهای رفتاری توالی و همکاری، مدل‌سازی رفتاری ایستا و به نمودار حالت، مدل‌سازی رفتاری پویا نیز گفته می‌شود. بنابراین گزینه سوم درست است.

Use Case Diagram یا نمودار مورد کاربرد، یا نمودار نیاز جهت مدل‌سازی لیست نیازمندی‌های مشتری مورد استفاده قرار می‌گیرد. بنابراین گزینه‌های اول، دوم و چهارم نادرست هستند.

۹- گزینه (۳) صحیح است.

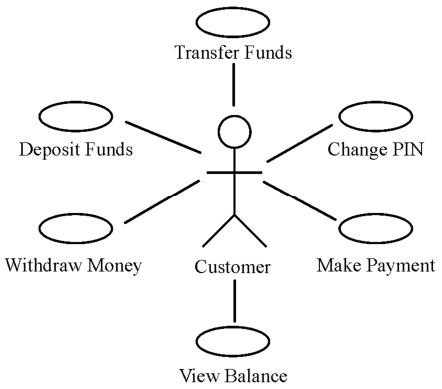
State Transition Diagram یا نمودار انتقال حالت (وضعیت)، به عنوان یک نمودار مکمل Sequence Diagram، برای مشخص‌کردن وضوح بیشتر برای شناخت حالت‌های مختلف یک

شیء مورد استفاده قرار می‌گیرد. به بیان دیگر نمودار انتقال حالت، چرخه حیات یک شیء را در حالت‌های مختلف (از زمانی که شیء ایجاد می‌شود تا زمانی که شیء از بین می‌رود) نمایش می‌دهد. در واقع این نمودار تمامی حالت‌های مختلفی را که یک شیء در طول حیات خود می‌تواند داشته باشد و همچنین نحوه انتقال بین حالت‌ها و وقایع باعث‌شونده این انتقال را نشان می‌دهد. به بیان دیگر نمودارهای حالت، راهی را آماده می‌کنند تا حالت‌های مختلف یک شیء را مدل کنند. نمودارهای حالت استفاده می‌شوند تا بیشتر، **رفتارهای پویای** یک سیستم را نمایش دهند.

به آنچه که موجب تغییر حالت یک شیء می‌گردد، رخداد(event) گفته می‌شود. بنابراین گزینه سوم درست است.

Use Case Diagram یا نمودار مورد کاربرد، یا نمودار نیاز جهت مدل‌سازی لیست نیازمندی‌های مشتری مورد استفاده قرار می‌گیرد.

مثال: مدل‌سازی لیست نیازمندی‌های مشتری برای سیستم ATM.



در نمودار فوق، لیست نیازمندی‌های مشتری، مدل‌سازی شده است.

به هر یک از نیازهای فوق یک use case یا مورد کاربرد گفته می‌شود و به اجتماع این use caseها، Use Case Diagram یا نمودار مورد کاربرد گفته می‌شود.

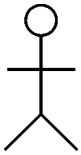
Use Caseها و Actorها محدوده سیستم در حال ساخت را مشخص می‌کنند. Use Case شامل تمام آن چیزهایی است که درون سیستم قرار دارد و Actor شامل تمام آن چیزهایی است که خارج از سیستم قرار دارد. هر فرد یا هر چیزی که با سیستم تعامل دارد، Actor یا بازیگر نامیده می‌شود. Use Caseها هر چیز موجود در داخل و محدوده سیستم را توصیف می کنند، در حالی که Actorها هر چیز موجود در خارج از محدوده سیستم را توصیف می‌کنند.

بازیگران، افراد و یا گاهی نرم‌افزار و یا سخت‌افزارهایی هستند که از سیستم استفاده می‌کنند و یا اطلاعاتی را برای سیستم فراهم می‌کنند. با اینکه همه بازیگران، عناصر خارجی سیستم هستند، اما با این حال هر عنصر خارج سیستم، بازیگر نیست. بازیگران استفاده کنندگان نهایی و یا تولیدکنندگان ابتدایی اطلاعات هستند، بنابراین عناصر خارجی سیستم که تنها وظیفه انتقال

اطلاعات را دارند و نقش **رسانه انتقال** را ایفا می‌کنند، نمی‌توانند به عنوان بازیگر در نظر گرفته شوند.

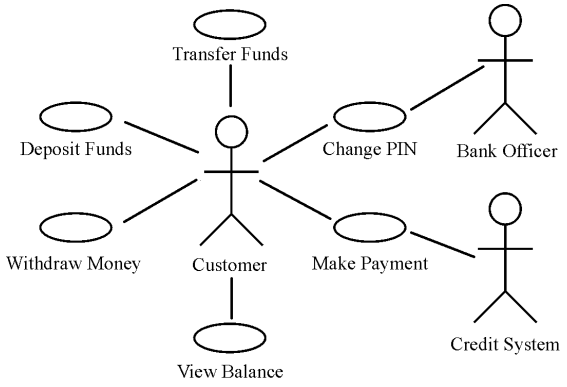
برای مثال اگر یک سنسور از طریق رسانه بی‌سیم، اطلاعاتی را به سیستم مرکزی ارسال می‌کند، رسانه بی‌سیم نمی‌تواند بازیگر این سیستم باشد، بلکه سنسور بازیگر آن خواهد بود. بنابراین هر بازیگر اگر استفاده‌کننده باشد نقطه انتها و اگر تولیدکننده اطلاعات باشد نقطه ابتدای آن ارتباط است.

در UML، بازیگران با آدمک‌هایی به شکل مقابل نشان داده می‌شوند:



Actor

نمونه‌ای از استفاده نمودار Use Case در شکل زیر نشان داده شده است: (سیستم ATM)



بنابر مطالب فوق گزینه اول نادرست است.

کلاس‌ها یا به تنهایی از عهده انجام مسئولیت‌های خود بر می‌آیند و یا از طریق همکاری با کلاس‌های همکار(collaborator) خود از عهده انجام مسئولیت‌های خود بر می‌آیند. بنابراین اگر کلاسی همکارانی دارد، باید در بخش مربوط به همکاران نوشته شود. بنابراین گزینه دوم نادرست است.

مؤلفه یک قطعه‌ی آماده، کاربردی و پیاده‌سازی شده است که دارای واسط لازم جهت اتصال با سایر قطعات و استقرار در بخشی از یک سیستم عملیاتی می‌باشد. از آنجایی که در یک مولفه، فقط بخشی از کدها یا داده‌های نرم‌افزار قرار می‌گیرد، لذا هر مولفه ممکن است نیازمند ارتباط با مولفه‌های دیگر باشد تا بتواند از کدها یا داده‌های موجود در آنها استفاده نماید. برای انجام این ارتباط، در مولفه‌ها یک واسط در نظر گرفته می‌شود تا کدها یا داده‌های یک مولفه، از طریق واسط آن در اختیار مولفه‌های دیگر قرار گیرد، به این معنی که ارتباط بین مولفه‌ها، فقط از طریق

واسط‌های آنها انجام می‌گیرد. در برنامه‌نویسی شیءگرا، متدهای عمومی کلاس‌ها، نقش واسط را ایفا می‌کنند. به عبارت دیگر کلاس‌های همکار، از طریق متدهای عمومی یکدیگر اقدام به گفتگو و تبادل پیام می‌کنند.

در دیدگاه شیءگرا به مؤلفه پیمانه نیز گفته می‌شود. در حیطه مهندسی نرم‌افزار شیءگرا، **واحد مؤلفه** یک قطعه‌ی عملیاتی مبتنی بر کلاس است که بر دو نوع می‌باشد:

۱) کلاس کاربردی: مانند کلاس دانشجو که برنامه‌نویس آن را می‌نویسد و به دلیل داشتن شرایط قابل حمل به شکل ذاتی، می‌تواند به عنوان یک قطعه‌ی آماده و قابل استفاده‌ی مجدد در پروژه‌های بعدی مورد استفاده مجدد قرار گیرد.

۲) کلاس سیستمی: مانند کلاس جعبه‌ی متن که کامپایلر تعاریف آن را فراهم می‌کند و می‌تواند به عنوان یک قطعه‌ی آماده و قابل استفاده‌ی مجدد، در پروژه‌ها مورد استفاده مجدد قرار گیرد. بنابراین گزینه چهارم نادرست است.

۱۰- گزینه (۴) صحیح است.

Sequence Diagram یا نمودار توالی جهت مدل‌سازی تعاملات پویای میان اشیاء همکار داخل یک use case مورد استفاده قرار می‌گیرد. بنابراین برای رسم نمودار توالی ابتدا باید کلاس‌های سیستم شناسایی شوند.

Class Diagram یا نمودار کلاس جهت مدل‌سازی ارتباطات ایستای میان کلاس‌های همکار داخل یک case use مورد استفاده قرار می‌گیرد.

پس از شناسایی موارد کاربرد و سناریونویسی برای هر یک از موارد کاربرد، زمان تعریف کلاس‌ها، صفات، متدها و ارتباطات میان کلاس‌های همکار برای هر یک از موارد کاربرد می‌رسد. برای شناسایی کلاس‌ها، صفات، متدها و ارتباطات میان کلاس‌ها برای هر یک از موارد کاربرد، از تکنیکی موسوم به CRC که سرواژه‌ی عبارت Class – Responsibility Collaborator و به معنی «مدل همکاری مسئولیت‌های کلاس‌ها» می‌باشد، استفاده می‌شود. مدل‌سازی CRC روشی ساده جهت تعیین و سازماندهی کلاس‌های داخل هر مورد کاربرد است. در این روش به هر کلاس یک کارت CRC اختصاص داده می‌شود که شامل سه بخش کلی زیر است:

نام کلاس

مسئولیت‌های کلاس (Responsibilities)

- صفات کلاس
- متدهای کلاس

همکاران کلاس (Collaborators)

کلاس‌ها یا به تنهایی از عهده انجام مسئولیت‌های خود بر می‌آیند و یا از طریق همکاری با کلاس‌های همکار خود از عهده انجام مسئولیت‌های خود بر می‌آیند. بنابراین اگر کلاسی همکارانی دارد، باید در بخش مربوط به همکاران نوشته شود.

برای کشف کلاس‌های همکار داخل هر use case، از سناریوی اصلی (نوشتاری یا نمودار) هر use case استفاده می‌گردد. برای این منظور، **اسامی** موجود داخل هر use case مورد جستجو قرار می‌گیرند. از آنجا که نیازها یا موارد کاربرد مشتری به تدریج و در طی تکرار مشخص می‌شوند و همچنین از آنجا که کلاس‌های همکار داخل هر مورد کاربرد یا نیاز هستند، بنابراین با تکامل و کامل شدن نیازها یا موارد کاربرد، به تبع کلاس‌های همکار هر مورد کاربرد نیز کامل می‌شود. بنابراین تشخیص تمام کلاس‌های برنامه، در همان ابتدای کار تقریبا غیر ممکن است، در واقع در طول پروژه و با گذشت زمان تحلیل‌گر متوجه نیازها و به تبع کلاس‌های جدیدی می‌شود که در ابتدای کار نیاز به آن‌ها چندان محسوس نبوده است. بنابراین می‌توان گفت روند تشخیص نیازها یا موارد کاربرد و به تبع کلاس‌های همکار به عنوان مرتفع‌کننده نیازها یا موارد کاربرد، یک فرآیند تکرارشونده است و در طول فرآیند تولید نرم‌افزار کامل و کامل‌تر می‌شوند. بنابراین گزینه چهارم درست است و گزینه‌های اول، دوم و سوم نادرست هستند.

۱۱- گزینه (۲) صحیح است.

Use Case Diagram یا نمودار مورد کاربرد، یا نمودار نیاز جهت مدل‌سازی لیست نیازمندی‌های مشتری مورد استفاده قرار می‌گیرد. بنابراین گزینه اول نادرست است.

Activity Diagram یا نمودار فعالیت و Swimlane Diagram یا نمودار خط شنا، جهت مدل سازی سناریوی اصلی و فرعی داخل یک use case **(نیاز یا مورد کاربرد یا زیرسیستم)** مورد استفاده قرار می‌گیرد. به بیان دیگر نمودار فعالیت یا نمودار خط شنا، جهت مدل‌سازی روال انجام کارها داخل یک use case مورد استفاده قرار می‌گیرد. همچنین، در مدل طراحی در بخش طراحی مؤلفه، باید جزئیات الگوریتمی **متدهای کلاس** تشریح شود. برای این منظور Activity Diagram یا نمودار فعالیت و Swimlane Diagram یا نمودار خط‌شنا مورد استفاده قرار می‌گیرد. بنابراین گزینه دوم درست است.

Class Diagram یا نمودار کلاس جهت مدل‌سازی ارتباطات ایستای میان کلاس‌های همکار داخل یک use case مورد استفاده قرار می‌گیرد. بنابراین گزینه سوم نادرست است.

Sequence Diagram یا نمودار توالی و Collaboration Diagram یا نمودار همکاری جهت مدل‌سازی تعاملات پویای میان اشیاء همکار داخل یک use case مورد استفاده قرار می‌گیرند. بنابراین گزینه چهارم نادرست است.

۱۲- گزینه (۲) صحیح است.

Sequence Diagram یا نمودار توالی و Collaboration Diagram یا نمودار همکاری جهت مدل‌سازی تعاملات پویای میان اشیاء همکار داخل یک use case مورد استفاده قرار می‌گیرند. بنابراین گزینه اول نادرست است.

اطلاعات به دست آمده از نمودار حالت در مدل تحلیل مربوط به کلاس‌های دارای حالات مختلف، در مدل طراحی، بخش طراحی مولفه، جهت تکمیل نمودن الگوریتم‌های مربوط به متدهای کلاس‌های دارای حالات مختلف مورد استفاده قرار می‌گیرد. اینکه متدهای یک کلاس

دارای حالات مختلف در سطح طراحی مولفه در مواجه با حالت‌های مختلف یک شیء چگونه رفتار کنند از نمودار حالت استنباط می‌گردد. بنابراین گزینه دوم درست است.

Component Diagram یا نمودار مؤلفه جهت مدل‌سازی ساختار کلی پیاده‌سازی برنامه و ترتیب کامپایل مؤلفه‌های فرعی، برای ایجاد مؤلفه‌اصلی (مؤلفه کلی یا برنامه اصلی) مورد استفاده قرار می‌گیرد. به بیان دیگر نمودار مؤلفه یک دید فیزیکی از مدل سیستم به همراه مؤلفه‌های فرعی نرم‌افزار و روابط بین آنها را نشان می‌دهد. بنابراین گزینه سوم نادرست است.

Deployment Diagram یا نمودار استقرار، جهت مدل‌سازی گره‌های سخت افزاری برای نصب نرم‌افزار مربوطه در فعالیت استقرار مورد استفاده قرار می‌گیرد. بنابراین گزینه چهارم نادرست است.

۱۳- گزینه (۲) صحیح است.

ترتیب ایجاد مولفه‌ها براساس رویکرد تکرار و تکامل نرم‌افزار توسط برنامه‌نویسان مشخص می‌شود. بنابراین گزینه اول نادرست است.

Component Diagram یا نمودار مؤلفه به نحوه پیاده‌سازی فیزیکی نرم‌افزار می‌پردازد. در واقع، نمودار مولفه شامل کدهای نرم‌افزار است که در آن هر قطعه کد در قالب یک مولفه، دسته‌بندی می‌شود. کدها نیز چیزی جز کلاس‌های سیستم، در زمانی که جزئیات پیاده‌سازی به آن‌ها اضافه شده باشد، نیست.

Component Diagram یا نمودار مؤلفه جهت مدل‌سازی ساختار کلی پیاده‌سازی برنامه و ترتیب کامپایل مؤلفه‌های فرعی، برای ایجاد مؤلفه‌اصلی (مؤلفه کلی یا برنامه اصلی) مورد استفاده قرار می‌گیرد. به بیان دیگر نمودار مؤلفه یک دید فیزیکی از مدل سیستم به همراه مؤلفه‌های فرعی نرم‌افزار و روابط بین آنها را نشان می‌دهد.

نمودار مؤلفه زمانی که تولید کد تمام شده است ایجاد می‌گردد. بنابراین در اینجا منظور از مؤلفه فرعی یک مولفه فیزیکی کد است. به بیان دیگر مؤلفه فرعی در این مرحله، کلاس‌های همکار پیاده‌سازی شده داخل یک use case یا مورد کاربرد یا نیاز پیاده‌سازی شده است. دقت کنید که قبل از استفاده از نمودار مولفه هریک از کلاس‌های همکار موجود در نمودار کلاس مربوط به هر use case یا مورد کاریرد یا نیاز باید به یک مؤلفه فرعی حاوی کد منبع تبدیل شود.

در این مرحله هر use case یا مورد کاربرد یا نیاز، مطابق روالی که از فعالیت‌های ارتباط تا ساخت (پیاده‌سازی) طی می‌کند، سرانجام توسط کلاس‌های همکار پیاده‌سازی شده خود به یک مؤلفه فرعی (مؤلفه‌بخشی) پیاده‌سازی شده و کامپایل شده تبدیل می‌گردد، که از اجتماع این مؤلفه‌های فرعی (مؤلفه‌بخشی)، مؤلفه اصلی (مؤلفه کلی یا برنامه اصلی) خلق می‌گردد. و محصول نهایی آماده می‌گردد.

تنها نوع رابطه‌ای که می‌تواند بین مؤلفه‌ها در نمودار مولفه وجود داشته باشد، یک رابطه وابستگی (Dependency) است و به این معنی است که یک مؤلفه باید قبل از کدام مولفه دیگر کامپایل شود. بنابراین گزینه دوم درست است.

نگاشت کلاس‌ها به مولفه‌ها قبل از استفاده از نمودار مولفه توسط برنامه‌نویسان در فعالیت پیاده‌سازی انجام می‌گردد. بنابراین گزینه سوم نیز نادرست است.

بسته (Package) مجموعه‌ای از کلاس‌های پیاده‌سازی‌شده و کامپایل‌شده به عنوان واحد مولفه و یا تعدادی مولفه فرعی پیاده‌سازی‌شده و کامپایل‌شده است. که از اجتماع بسته‌ها یک سیستم نرم‌افزاری ایجاد می‌گردد. نگاشت مولفه به بسته توسط کامپایل کردن مولفه‌ها انجام می‌گردد، در حالی که رابطه وابستگی در نمودار مولفه ترتیب کامپایل مولفه‌ها را نشان می‌دهد، بنابراین گزینه چهارم نیز نادرست است.

۱۴- گزینه (۱) صحیح است.

مدل تحلیل، مدل‌سازی عالم خارج به زبانی شبیه انسان است، اما مدل طراحی مدل‌سازی عالم داخل ماشین به زبانی شبیه زبان ماشین است. بنابراین برای اینکه ساخت برنامه کامپیوتری که به زبان ماشین است، امکان‌پذیر باشد، باید مدل تحلیل به مدل طراحی نگاشت شود تا مدل طراحی بتواند به عنوان نقشه راهی شبیه به زبان ماشین، راهگشا باشد.

مدل‌های تحلیل، کلی‌تر و انتزاعی‌تر هستند، و برای مدل‌سازی عالم خارج مورد استفاده قرار می‌گیرند، بدون آنکه به جزئیات نحوه پیاده‌سازی بپردازند، در واقع مدل تحلیل به کلی گویی به زبان انسان و حذف جزئیات نحوه پیاده‌سازی می‌پردازد. اما مدل‌های طراحی، جزئی‌تر و غیرانتزاعی‌تر هستند، و برای مدل‌سازی عالم داخل ماشین مورد استفاده قرار می‌گیرند، و به بیان جزئیات نحوه پیاده‌سازی می‌پردازند، در واقع مدل طراحی به جزئی‌گویی به زبان شبیه ماشین و درج جزئیات نحوه پیاده‌سازی می‌پردازد.

با نگاهی سطح به سطح، به حل مساله، سطوح مختلفی از انتزاع را خواهیم داشت. در بالاترین سطح انتزاع، راه حل به صورت کلی به زبان محیط مساله بیان می‌گردد. در سطوح پایین‌تر، راه حل به جزئیات پیاده‌سازی نزدیک‌تر می‌شود و در پایین‌ترین سطح انتزاع راه حل به صورتی بیان می‌شود تا مستقیما قابل پیاده‌سازی باشد.

تجرید یا انتزاع، روش و نگرشی در ارائه و توضیح است که در آن فقط اطلاعات مهمی که برای حل یک مساله لازم است، گردآوری و نگهداری می‌شود و از بقیه اطلاعات صرف نظر می‌گردد.

درک یکجای مساله به یکباره، اغلب غیر ممکن است. بنابراین با استفاده از مفهوم انتزاع، ابتدا بر کلیت مساله تمرکز می‌شود، اما برای حل کامل مساله جزئیات آن نیز باید در نهایت بررسی شوند! فعالیت پالایش با تعیین ریز به ریز عملیات، جزئیات مساله را آشکار می‌کند. این فعالیت روندی بالا به پایین و سلسله مراتبی دارد، در واقع در سطوح پایینی پالایش، جزئیات واضح‌تر می‌شوند. هرچه به سطوح پایین‌تر سلسله مراتب نزدیک‌تر می‌شوید، جزئیات بیشتری از مساله آشکار می‌شود، تا جایی که در پایین‌ترین سطح پالایش، دستورات زبان برنامه‌نویسی ارائه می‌گردند.

انتزاع و پالایش دو مفهوم مکمل هستند. انتزاع مهندسان نرم‌افزار را قادر می‌سازد تا داده‌ها،

عملکردها و رفتارها را باحذف جزئیات تعریف کنند. در حالی که پالایش بر محتوای داخلی داده‌ها، عملکردها و رفتارها تمرکز داشته و جزئیات آنها را تشریح می‌کند. این تشریح به صورت لایه به لایه و در یک ساختار سلسله مراتبی و بالا به پایین است که با پیشروی در عمق لایه‌ها، جزئیات بیشتری آشکار می‌شود. به پالایش، افراز یا بخش‌بندی(partitioning) نیز گفته می‌شود.

در متدولوژی ساخت‌یافته، در اولین مرحله مدل‌سازی (مدل تحلیل)، سیستم به دو وجه «داده» و «عملکرد» تفکیک می‌شود. سپس طی روندی سلسله مراتبی و مطابق با روش بالا به پایین، هر یک از این وجوه خود به مؤلفه‌های فرعی تجزیه می‌شوند. این روند تا به جایی ادامه می‌یابد که جزئیات توابع برنامه جهت پیاده‌سازی مشخص شوند.

همچنین در متدولوژی شیءگراء، در اولین مرحله‌ی مدل‌سازی (مدل تحلیل) سیستم در قالب کلاس‌ها (شامل داده (صفت) و عملکرد (متد)) نشان داده می‌شود. سپس طی روندی سلسله مراتبی و مطابق با روش بالا به پایین، کلاس‌ها با جزئیات بیشتری مشخص می‌شوند. این روند تا به جایی ادامه می‌یابد که جزئیات کلاس‌های برنامه جهت پیاده‌سازی مشخص شوند.

بنابر مطالب فوق واضح است که گزینه اول درست است.

Use Caseها و Actorها محدوده سیستم در حال ساخت را مشخص می‌کنند. Use Case شامل تمام آن چیزهایی است که درون سیستم قرار دارد و Actor شامل تمام آن چیزهایی است که خارج از سیستم قرار دارد. هر فرد یا هر چیزی که با سیستم تعامل دارد، Actor یا بازیگر نامیده می‌شود. Use Caseها هر چیز موجود در داخل و محدوده سیستم را توصیف می کنند، در حالی که Actorها هر چیز موجود در خارج از محدوده سیستم را توصیف می‌کنند.

بازیگران، افراد و یا گاهی نرم‌افزار و یا سخت‌افزارهایی هستند که از سیستم استفاده می‌کنند و یا اطلاعاتی را برای سیستم فراهم می‌کنند. لذا Actorها فقط افراد و کابران نرم‌افزار نیستند. بلکه می‌توانند نرم‌افزار و یا سخت‌افزارهایی باشند که از سیستم استفاده می‌کنند و یا اطلاعاتی را برای سیستم فراهم می‌کنند. بنابراین گزینه دوم نادرست است.

نمونه‌سازی می‌تواند توسط ابزارهای کامپیوتری و غیرکامپیوتری انجام گردد، اما صرف استفاده از ابزارها برای توسعه سریع نمونه‌ها، منجر به این نمی‌شود که پروژه طبق زمان‌بندی پیش‌رود. بلکه مدیریت زمان‌بندی پروژه و مدیریت ریسک‌های پروژه نیز در زمان تحویل پروژه مطابق زمان‌بندی مشخص شده موثر است. بنابراین گزینه سوم نیز نادرست است.

هنگامی مستندات مربوط به تعریف نیازمندی‌های مشتری، به مستنداتی غیرقابل تغییر بدل می گردد که ماهیت پروژه غیرتکاملی باشد و همه نیازها در همان ابتدای کار مشخص، ثابت و بدون تغییر باشد. در این شرایط مدل‌های فرآیند تولید نرم‌افزار غیرتکاملی برای روند تولید نرم‌افزار مورد استفاده قرار می‌گیرند.

اما هنگامی که ماهیت پروژه تکاملی باشد و همه نیازها در همان ابتدای کار مشخص، ثابت و بدون تغییر نباشد. آنگاه در این شرایط مدل‌های فرآیند تولید نرم‌افزار تکاملی برای روند تولید نرم‌افزار مورد استفاده قرار می‌گیرند. که به تبع منجر به این می‌شود که مستندات مربوط به تعریف نیازمندی‌های مشتری، به مستنداتی قابل تغییر در طول روند حرکت پروژه بدل گردد. لذا در چنین

شرایطی مهم‌ترین کار در فعالیت ارتباطات، مدیریت شناسایی نیازمندی‌ها و پیگیری تغییرات نیازمندی‌های مشتری است. بنابراین گزینه چهارم نیز نادرست است.

۱۵- گزینه (۳) صحیح است.

پس از شناسایی موارد کاربرد و سناریونویسی برای هر یک از موارد کاربرد، زمان تعریف کلاس‌ها، صفات، متدها و ارتباطات میان کلاس‌های همکار برای هر یک از موارد کاربرد می‌رسد. برای شناسایی کلاس‌ها، صفات، متدها و ارتباطات میان کلاس‌ها برای هر یک از موارد کاربرد، از تکنیکی موسوم به CRC که سرواژه‌ی عبارت Class – Responsibility Collaborator و به معنی «مدل همکاری مسئولیت‌های کلاس‌ها» می‌باشد، استفاده می‌شود. مدل‌سازی CRC روشی ساده جهت تعیین و سازماندهی کلاس‌های داخل هر مورد کاربرد است. در این روش به هر کلاس یک کارت CRC اختصاص داده می‌شود که شامل سه بخش کلی زیر است:

نام کلاس

مسئولیت‌های کلاس (Responsibilities)

- صفات کلاس
- متدهای کلاس

همکاران کلاس (Collaborators)

کلاس‌ها یا به تنهایی از عهده انجام مسئولیت‌های خود بر می‌آیند و یا از طریق همکاری با کلاس‌های همکار خود از عهده انجام مسئولیت‌های خود بر می‌آیند. بنابراین اگر کلاسی همکارانی دارد، باید در بخش مربوط به همکاران نوشته شود.

برای کشف کلاس‌های همکار داخل هر use case، از سناریوی اصلی (نوشتاری یا نموداری) هر use case استفاده می‌گردد. برای این منظور، **اسامی** موجود داخل هر use case مورد جستجو قرار می‌گیرند. از آنجا که نیازها یا موارد کاربرد مشتری به تدریج و در طی تکرار مشخص می‌شوند و همچنین از آنجا که کلاس‌های همکار داخل هر مورد کاربرد یا نیاز هستند، بنابراین با تکامل و کامل شدن نیازها یا موارد کاربرد، به تبع کلاس‌های همکار هر مورد کاربرد نیز کامل می‌شود. بنابراین تشخیص تمام کلاس‌های برنامه، در همان ابتدای کار تقریبا غیر ممکن است، در واقع در طول پروژه و با گذشت زمان تحلیل‌گر متوجه نیازها و به تبع کلاس‌های جدیدی می‌شود که در ابتدای کار نیاز به آن‌ها چندان محسوس نبوده است. بنابراین می‌توان گفت روند تشخیص نیازها یا موارد کاربرد و به تبع کلاس‌های همکار به عنوان مرتفع‌کننده نیازها یا موارد کاربرد، یک فرآیند تکرارشونده است و در طول فرآیند تولید نرم‌افزار کامل و کامل‌تر می‌شوند.

پس از اتمام مدل‌سازی CRC، کارت‌های CRC با سناریوی اصلی هر use case یا مورد کاربرد تطابق داده می‌شود. تا مشخص شود که آیا تمام کلاس‌های مربوط به use case درست شناسایی شده‌اند یا خیر. که در صورت نیاز، می‌بایست تغییرات لازم بر روی کارت‌های CRC اعمال گردد.

منظور از عبارت «تمام نمایش‌های مورد کاربرد (use case)» در گزینه اول، شیوه‌های نمایش

شرح حال هر use case است، این شرح حال use case همان سناریوهای اصلی و فرعی داخل use case است که به دو شیوه نوشتاری (متنی) و گرافیکی (نموداری) قابل نمایش است. به بیان روال حرکت قدم به قدم کارها داخل یک use case از نقطه شروع تا رسیدن به یک نتیجه مطلوب (موفق) سناریوی اصلی گفته می‌شود. و به بیان روال حرکت قدم به قدم کارها داخل یک use case از نقطه شروع تا رسیدن به یک نتیجه نامطلوب (ناموفق) سناریوی فرعی گفته می‌شود. بنابراین گزینه اول درست است.

کلاس مکانیزمی است که توسط آن، مفهوم کپسوله‌سازی پیاده‌سازی می‌شود.

بنابراین مطابق تعریف بسته‌بندی، کلاس نیز صفات و متدهای مرتبط به هم را در یک بسته، بسته‌بندی می‌کند. درون یک کلاس، صفات و متدها یا هردو می‌توانند به صورت اختصاصی (private) و یا عمومی (public) باشند

وجه تمایز کلاس‌ها در تفاوت در صفات آن‌ها است. مانند تفاوت در صفات کلاس استاد و کلاس دانشجو.

به نمونه‌ای از یک کلاس، شیء گفته می‌شود. مانند دانشجو اکبری از کلاس دانشجو.

معرفی یک کلاس یک انتزاع یا تجرید **منطقی** (logical abstraction) است که یک نوع داده جدید را معرفی می‌کند و نشان می‌دهد که یک شی از این نوع، چه شکل و شمایلی دارد. اما معرفی یک شیء از یک کلاس، یک موجودیت **فیزیکی** (physical entity) از نوع یک کلاس را ایجاد می‌کند، بدین معنی که با معرفی یک شیء، فضای حافظه‌ای به شیء اختصاص داده می‌شود ولی در معرفی یک کلاس فضای حافظه‌ای تخصیص نمی‌یابد.

وجه تمایز اشیاء در مقادیر صفات اشیاء است. مانند تفاوت در مقادیر دانشجو اکبری و دانشجو احمدی. بنابراین گزینه دوم نیز درست است.

به طورکلی هرگاه یک کلاس (ب)، **از نوع** یک کلاس (الف) باشد، گوییم بین دو کلاس مذکور رابطه **وراثت** برقرار است. در این صورت کلاس (ب) فرزند و کلاس (الف) پدر یا والد نامیده می‌شود. به عبارت دیگر همانطور که پیش از این نیز گفتیم، وراثت فرآیندی است که به وسیله آن یک کلاس (فرزند) می‌تواند صفات و متدهای کلاس دیگری (پدر) را کسب کند. به عبارت کلی‌تر یک کلاس فرزند ضمن به ارث بردن مجموعه‌ای از صفات و متدهای عمومی کلاس پدر، می‌تواند ویژگی‌های خاص و مختص خود را نیز به آنها اضافه کند. **در رابطه وراثت هر تغییر در کلاس پدر بر کلاس فرزند نیز اثر می‌گذارد اما عکس این مطلب برقرار نیست.** ابرکلاس، همان کلاس پدر است که ویژگی‌ها و عملیات خود را در اختیار کلاس‌های دیگر (فرزندان) قرار می‌دهد. زیر کلاس نیز همان کلاس فرزند است که برخی از صفات و عملیاتش متعلق به یک ابرکلاس است.

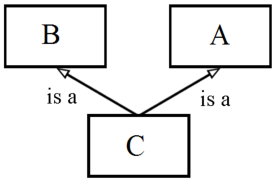
دو رویکرد برای ایجاد سلسله مراتب وراثت وجود دارد:

۱- رویکرد «بالا به پایین» یا تخصیص (Specialization) که در آن، ابتدا کلاس‌های پدر، تعریف و سپس کلاس‌های فرزند ایجاد می‌شوند.

۲- رویکرد «پایین به بالا» یا تعمیم (Generalization) که در آن، از طریق تعمیم کلاس‌های فرزند به کلاس‌های پدر می‌رسیم.

بنابر مطالب فوق، واضح است که گزینه سوم نادرست است.

هرگاه یک کلاس برخی از صفات و عملیات را از یک کلاس و برخی دیگر را از یک کلاس دیگر به ارث ببرد، در این حالت وراثت چندگانه رخ داده‌است. شکل زیر نمونه‌ای از مدل‌سازی «**ارث‌بری چندگانه**» (Multiple Inheritance) را نشان می‌دهد.



در شکل فوق کلاس C به صورت وراثت چندگانه از کلاس A و کلاس B ارث‌بری کرده است. بنابراین گزینه چهارم نیز درست است.

۱۶- گزینه (۲) صحیح است.

پس از شناسایی موارد کاربرد و سناریونویسی برای هر یک از موارد کاربرد، زمان تعریف کلاس‌ها، صفات، متدها و ارتباطات میان کلاس‌های همکار برای هر یک از موارد کاربرد می‌رسد. برای شناسایی کلاس‌ها، صفات، متدها و ارتباطات میان کلاس‌ها برای هر یک از موارد کاربرد، از تکنیکی موسوم به CRC که سرواژه‌ی عبارت Class – Responsibility Collaborator و به معنی «مدل همکاری مسئولیت‌های کلاس‌ها» می‌باشد، استفاده می‌شود. مدل‌سازی CRC روشی ساده جهت تعیین و سازماندهی کلاس‌های داخل هر مورد کاربرد است. در این روش به هر کلاس یک کارت CRC اختصاص داده می‌شود که شامل سه بخش کلی زیر است:

نام کلاس

مسئولیت‌های کلاس (Responsibilities)

- صفات کلاس
- متدهای کلاس

همکاران کلاس (Collaborators)

کلاس‌ها یا به تنهایی از عهده انجام مسئولیت‌های خود بر می‌آیند و یا از طریق همکاری با کلاس‌های همکار خود از عهده انجام مسئولیت‌های خود بر می‌آیند. بنابراین اگر کلاسی همکارانی دارد، باید در بخش مربوط به همکاران نوشته شود.

برای کشف کلاس‌های همکار داخل هر use case، از سناریوی اصلی (نوشتاری یا نموداری) هر use case استفاده می‌گردد. برای این منظور، **اسامی** موجود داخل هر use case مورد جستجو قرار می‌گیرند. از آنجا که نیازها یا موارد کاربرد مشتری به تدریج و در طی تکرار مشخص می‌شوند

و همچنین از آنجا که کلاس‌های همکار داخل هر مورد کاربرد یا نیاز هستند، بنابراین با تکامل و کامل شدن نیازها یا موارد کاربرد، به تبع کلاس‌های همکار هر مورد کاربرد نیز کامل می‌شود. بنابراین تشخیص تمام کلاس‌های برنامه، در همان ابتدای کار تقریبا غیر ممکن است، در واقع در طول پروژه و با گذشت زمان تحلیل‌گر متوجه نیازها و به تبع کلاس‌های جدیدی می‌شود که در ابتدای کار نیاز به آن‌ها چندان محسوس نبوده است. بنابراین می‌توان گفت روند تشخیص نیازها یا موارد کاربرد و به تبع کلاس‌های همکار به عنوان مرتفع‌کننده نیازها یا موارد کاربرد، یک فرآیند تکرارشونده است و در طول فرآیند تولید نرم‌افزار کامل و کامل‌تر می‌شوند.

سه نوع رابطه مختلف بین کلاس‌های همکار در مدل CRC وجود دارد:

رابطه آگاه است از (has knowledge of)

دو انسانی که همدیگر را می‌شناسند و امکان گفتگو و تبادل پیام با هم دارند، رابطه انجمنی با هم دارند. در رابطه انجمنی یک شیء بطور ساده درباره شیء دیگر می‌داند به همان طریقی که یک فرد ممکن است فرد دیگری را بشناسد. یک برنامه کامپیوتری شیءگرا از اجتماع تعدادی کلاس ایجاد شده است، کلاس‌های همکار **بدون** رابطه جزء و کل و هم‌هدف در یک بخش مشترک از برنامه، کلاس‌های همکار **با** رابطه جزء و کل و هم‌هدف در یک بخش مشترک از برنامه و کلاس‌های غیرهمکار در دو بخش مختلف از برنامه.

رابطه‌ای که میان کلاس‌های همکار بدون رابطه جزء و کل جهت گفتگوی میان اشیاء آنها وجود دارد، رابطه انجمنی است.

کلاس‌های همکار بدون رابطه جزء و کل، جهت انجام وظایف خود از طریق مکانیزم پیام به گفتگو با یکدیگر می‌پردازند. در یک بیان ساده، هرگاه میان دو شیء بدون رابطه جزء و کل گفتگو باشد، کلاس‌های این دو شیء هم باهم همکار هستند و هم رابطه انجمنی میان آنها برقرار است.

هرگاه مابین دو کلاس را**بطه انجمنی** مطرح باشد، یعنی تبادل پیام مابین برخی متدهای اشیای دو کلاس صورت گیرد، آنگاه در این شرایط **رابطه آگاهی** خواهیم داشت. مانند ارسال پیام از شیء بازیکن به شیء توپ پس از انجام متد شوت زدن مربوط به شیء بازیکن، برای تغییر مختصات توپ توسط صدا زدن متد تغییر مختصات توپ مربوط به شیء توپ. یعنی شیء بازیکن، باید در هنگام شوت‌زدن، مختصات توپ را توسط صدازدن متد تغییر مختصات توپ، تغییر دهد.

رابطه بخشی است از (is part of)

هرگاه رابطه‌ای مابین یک واحد کل و تعدادی واحد جزء مطرح باشد، یعنی یک کلاس کل از همکاری تعدادی کلاس جزء تشکیل شده باشد، آنگاه در این شرایط رابطه بخشی خواهیم داشت. این رابطه خود بر دو نوع **تجمع** و **ترکیب** می‌باشد.

رابطه وابسته است به (depends upon)

هر گاه مابین دو کلاس رابطه وابستگی مطرح باشد، رابطه وابستگی خواهیم داشت. در واقع

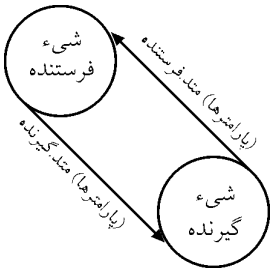
هرگاه تغییرات مقادیر صفات یک شیء، روی مقادیر صفات شیء دیگری تأثیر بگذارد، این دو شیء به هم وابسته هستند. در واقع این رابطه زمانی رخ می‌دهد که مقادیر صفاتی از یک کلاس به مقادیر صفاتی از یک کلاس دیگر وابسته باشد. برای مثال در برنامه کامپیوتری فوتبال، مختصات شیء سر بازیکن، دست بازیکن و پای بازیکن، همواره باید به مختصات شیء بدنه بازیکن وابسته باشد (مگر خشونت در بازی زیاد باشد!). در واقع مختصات سر بازیکن، دست بازیکن و پای بازیکن، همواره باید با تغییرات مختصات بدنه بازیکن تغییر کند. وگرنه در هنگام حرکت بازیکن، سر بازیکن، دست بازیکن و پای بازیکن از بدنه بازیکن جلو یا عقب می‌افتند!

به کلاس‌های همکار، **کلاس‌های مشارکت‌کننده** نیز گفته می‌شود.

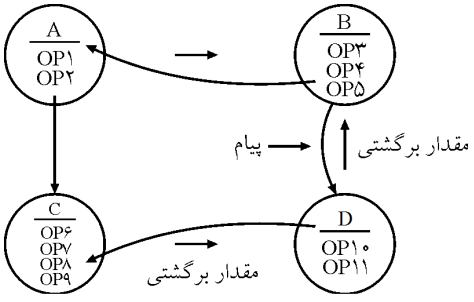
۱۷- گزینه (۴) صحیح است.

راه ایجاد ارتباط بین اشیاء، ارسال پیام است، در واقع پیام مکانیزمی است که اشیاء به وسیله آن با هم ارتباط برقرار می‌کنند. یک پیام باعث می‌شود رفتاری در شیء گیرنده انجام شود. بنابراین ارسال یک پیام از یک شیء مبدا به شیء مقصد به معنی صدا زدن یکی از متدهای شیء مقصد است. یک پیام شامل شیء مقصد (شیء گیرنده). متد (متدی که در شیء مقصد پیام را دریافت می‌کند) و پارامترهای مربوطه می‌باشد.

تعامل میان اشیاء در شکل زیر نشان داده شده است، یک عمل در داخل شیء فرستنده پیامی به شکل زیر تولید می‌کند: (پارامترها) متد شیء مقصد. نام شیء مقصد که در آن، مقصد، شیء گیرنده‌ای را تعریف می‌کند که توسط پیام صدا زده می‌شود.



مثال: چهار شیء A، B، C و D با مبادله پیام با یکدیگر ارتباط برقرار می‌کنند.



مبادله پیام‌های بین اشیاء

اگر شیء B بخواهد متد OP10 از شیء D را اجرا کند، پیامی به شکل زیر به D ارسال می‌کند:

D.OP10(data)

شیء D نیز به عنوان بخشی از اجرای OP10 ممکن است، پیامی به شکل زیر به C بفرستد:

C.OP8(data)

C عمل OP8 را می‌یابد، آن را اجرا می‌کند و سپس یک مقدار بازگشتی مناسب به D ارسال می‌کند. عمل OP10 کامل می‌شود و مقداری را به B بازمی‌گرداند.

توجه: برای آن که از یک شیء درخواست شود تا یکی از متدهای خود را انجام دهد، باید پیامی به آن داده شود تا معلوم شود چه باید بکند. شیء گیرنده ابتدا با انجام متد درخواست شده و سپس با بازگرداندن کنترل به شیء مبدا، این پیام را پاسخ می‌دهد.

شمول یا رابطه شامل بودن (Include) و بسط یا رابطه توسعه‌دادن (Extend) مربوط به انواع روابط بین Use Caseها می‌باشد. رابطه‌ی وابستگی (Dependency) مربوط به روابط میان کلاس‌ها یا ترتیب کامپایل میان مولفه‌های برنامه می‌باشد.

۱۸- گزینه (۴) صحیح است.

قبل از پاسخ به سوال مطرح شده، به سوال زیر پاسخ دهید:

سوال:کدام‌یک از موارد زیر برای نوشتن مناسب نیست؟

۱) مداد ۲) خودکار ۳) اتود ۴) هیچ‌کدام

پاسخ - گزینه (۴) صحیح است. زیرا همه موارد فوق برای نوشتن **مناسب** هستند. مداد مناسب است. خودکار مناسب است و اتود هم مناسب است.

به طراحی مولفه، طراحی جزئی، طراحی تفصیلی و طراحی رویه‌ای نیز گفته می‌شود.

طراحی مولفه، فعالیت تبدیل طراحی معماری به نرم‌افزار است. در این مرحله، سطح انتزاع طراحی معماری به سطح انتزاع نرم‌افزار کاربردی نزدیک می‌گردد. طراحی در سطح مؤلفه‌ها، نرم افزار را در سطحی از انتزاع تصویر می‌کند که به کد نزدیک است. طراحی مولفه، به عنوان نقشه راهی دقیق، و نزدیک به زبان پیاده‌سازی، در فعالیت پیاده‌سازی نرم‌افزار، منجر به صرفه جویی در زمان و هزینه‌های تولید می‌گردد. در طراحی مولفه، مهندس نرم‌افزار باید ساختمان داده‌ها، واسطها و الگوریتم‌ها را با جزییات کافی به نمایش در آورد تا راهنمای تولید کد منبع زبان برنامه‌نویسی باشد.

طراحی مؤلفه ساخت‌یافته، طراحی معماری از همان فعالیت مدل طراحی را به عنوان ورودی دریافت کرده و طراحی مؤلفه را توسط ابزارهایی همچون شبه کد یا فلوچارت ایجاد می‌کند.

در طراحی مؤلفه ساخت‌یافته، اسکلت، ساختار و چیدمان کلی مولفه‌های (توابع) برنامه به این معنی که چه مولفه‌ای (تابعی) چه مولفه‌ای (تابعی) دیگر را صدا می‌زند، با ذکر جزئیات داخلی مولفه‌ها (توابع) مشخص می‌گردد (ساختار درختی برنامه با ذکر جزئیات مولفه‌ها (توابع)). مانند اسکلت یک ساختمان که گویای جایگاه مؤلفه‌های ساختمان است و آجرچینی هم شده است. (اسکلت یک ساختمان به همراه آجرچینی).

طراحی مؤلفه شیءگرا، طراحی معماری از همان فعالیت مدل طراحی را به عنوان ورودی دریافت کرده و طراحی مؤلفه را توسط ابزارهایی همچون شبه کد یا Activity Diagram یا Swimlane Diagram ایجاد می‌کند.

در طراحی مؤلفه شیءگرا، اسکلت، ساختار و چیدمان کلی مؤلفه‌های برنامه به این معنی که چه مولفه‌ای (کلاسی) با چه مولفه‌ای (کلاسی) دیگر در ارتباط است، با ذکر جزئیات مربوط به شرح متدهای کلاس‌های همکار داخل یک مؤلفه فرعی (یک بخش از نرم‌افزار). مانند اسکلت یک ساختمان که گویای جایگاه مؤلفه‌های ساختمان است و آجرچینی هم شده است. (اسکلت یک ساختمان به همراه آجرچینی).

مطابق آنچه گفتیم، در طراحی مولفه می‌بایست جزئیات مولفه‌ها توسط ابزارهای مربوطه نوشته شود.

از آنجاکه در عبارت مطرح شده در گزینه اول «حداکثر زمانی که استفاده‌کننده بایستی منتظر پاسخ سیستم بماند.» به جزئیات انجام کار پرداخته شده است. بنابراین برای گنجاندن در طراحی جزئی یا طراحی مولفه مناسب است.

از آنجاکه در عبارت مطرح شده در گزینه دوم «نحوه ذخیره اطلاعات مربوط به عملیات انجام شده در هر روز» به جزئیات انجام کار پرداخته شده است. بنابراین برای گنجاندن در طراحی جزئی یا طراحی مولفه مناسب است.

از آنجاکه در عبارت مطرح شده در گزینه سوم «اتفاقاتی که باید در صورت قطع اتصال از شبکه کامپیوتری در سیستم بیفتد.» به جزئیات انجام کار پرداخته شده است. بنابراین برای گنجاندن در طراحی جزئی یا طراحی مولفه مناسب است.

بنابراین گزینه چهارم یعنی «هیچکدام» انتخاب می‌گردد، زیرا گزینه‌های اول، دوم و سوم مناسب طراحی مولفه هستند و نه نامناسب.

۱۹- گزینه (۳) صحیح است.

مدل تحلیل به روش ساخت‌یافته از سه بخش مدل‌سازی داده‌ای، مدل‌سازی عملکردی و مدل‌سازی رفتاری تشکیل شده است، مدل‌سازی داده‌ای شامل تحلیل موجودیت‌ها و تحلیل پرس و جوها می‌باشد. تحلیل موجودیت‌ها توسط ابزار مدل ER و تحلیل پرس و جوها توسط ابزار حساب رابطه‌ای مدل می‌شوند، مدل‌سازی عملکردی توسط ابزار DFD و مدل‌سازی رفتاری توسط ابزار STD مدل می‌شود. نمودار جریان‌داده پس از نگاشت به طراحی معماری و گذر به طراحی مولفه در مدل طراحی سرانجام در فعالیت پیاده‌سازی به نسخه فیزیکی (کد) تبدیل می‌گردد. در واقع پیاده‌سازی عملکرد در فعالیت پیاده‌سازی، طراحی مؤلفه از مدل طراحی را به عنوان ورودی دریافت کرده و توسط یک زبان برنامه‌نویسی پیاده‌سازی عملکرد را انجام می‌دهد. بنابراین گزینه اول پاسخ سؤال نیست.

Use Case Diagram یا نمودار مورد کاربرد، Requirement Diagram یا نمودار نیاز جهت مدل‌سازی لیست نیازمندی‌های مشتری مورد استفاده قرار می‌گیرد.

Use Case Diagram یا نمودار مورد کاربرد، هیچگاه به طور مستقیم، پیاده‌سازی نمی‌شود، بلکه مبنایی برای استخراج دیگر نمودارها قرار می‌گیرد، بنابراین ساختار نمودار کاربرد به پیاده‌سازی نرم‌افزار یا نسخه فیزیکی تبدیل نمی‌گردد.

Use Caseها و Actorها محدوده سیستم در حال ساخت را مشخص می‌کنند. Use Case شامل تمام آن چیزهایی است که درون سیستم قرار دارد و Actor شامل تمام آن چیزهایی است که خارج از سیستم قرار دارد. هر فرد یا هر چیزی که با سیستم تعامل دارد، Actor یا بازیگر نامیده می‌شود. Use Caseها هر چیز موجود در داخل و محدوده سیستم را توصیف می‌کنند، در حالی که Actorها هر چیز موجود در خارج از محدوده سیستم را توصیف می‌کنند. بنابراین گزینه سوم پاسخ سؤال است.

پس از شناسایی کلاس‌ها و کلاس‌های همکار برای هر یک از موارد کاربرد در مدل CRC، زمان مدل‌سازی نموداری و ساختاری توسط نمودار کلاس می‌رسد.

Class Diagram یا نمودار کلاس جهت مدل‌سازی ارتباطات ایستای میان کلاس‌های همکار داخل یک case use مورد استفاده قرار می‌گیرد.

در فعالیت مدل تحلیل، جزئیات مربوط به کلاس‌ها، شامل جزئیات دقیق صفات و متدها، مشخص نمی‌گردد، بیان این جزئیات تا بخش طراحی مولفه از مدل طراحی به تأخیر می‌افتد.

نمودار کلاس به عنوان منبع اصلی تولید کد محسوب می‌گردد. هر کلاس دارای حداقل یک مسئولیت خواهد بود که در سلسله مراحل پالایش کلاس، این مسئولیت‌ها به مجموعه‌ای از صفات و متدها بدل می‌گردند. به نحوی که صفات و متدها بتوانند از عهده مسئولیت‌های کلاس برآیند.

نمودار کلاس پس از عبور از طراحی معماری و گذر به طراحی مولفه در مدل طراحی سرانجام در فعالیت پیاده‌سازی به نسخه فیزیکی (کد) تبدیل می‌گردد. در واقع فعالیت پیاده‌سازی، طراحی مؤلفه از مدل طراحی را به عنوان ورودی دریافت کرده و توسط یک زبان برنامه‌نویسی شی‌گراء پیاده‌سازی عملکرد را انجام می‌دهد. در این مرحله، ساختار کلاس‌ها، ساختار و نحوه ارسال پیام‌ها مابین اشیاء، ساختمان داده‌ها و پرس‌وجوها برای ایجاد مؤلفه‌های فرعی (مؤلفه بخشی) و به تبع ایجاد مؤلفه اصلی (مؤلفه کلی یا برنامه اصلی) پیاده‌سازی می‌گردند. بنابراین گزینه چهارم پاسخ سؤال نیست.

پس از مدل‌سازی ارتباطات ایستای میان کلاس‌های همکار نوبت به مدل‌سازی تعاملات پویای میان اشیاء همکار می‌رسد.

مدل‌سازی تعاملات پویای میان اشیاء همکار یا مدل‌سازی رفتاری، رفتار سیستم را در زمان اجرا نمایش می‌دهد. به بیان دیگر در این دیدگاه رفتارهای پویای سیستم مدل می‌شود.

مدل‌سازی رفتاری سیستم با استفاده از نمودارهای UML قابل انجام است:

الف) نمودار توالی (Sequence Diagram)

Sequence Diagram یا نمودار توالی، Object Diagram یا نمودار شیء جهت مدل‌سازی

تعاملات پویای میان اشیاء همکار داخل یک use case مورد استفاده قرار می‌گیرد.

ب) نمودار همکاری (Collaboration Diagram)

Collaboration Diagram یا نمودار همکاری در نسخه‌های امروزی UML، به نمودار **Communication Diagram** یا **نمودار ارتباط**، تغییر نام یافته است.

Collaboration Diagram یا نمودار همکاری، برای یک کاربرد خاص، جهت مدل‌سازی تعاملات پویای میان اشیاء همکار داخل یک use case مورد استفاده قرار می‌گیرد.

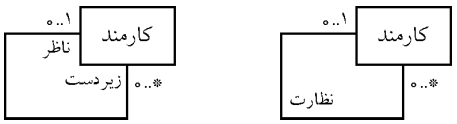
ج) نمودار حالت (State Transition Diagram)

State Transition Diagram یا نمودار انتقال حالت (وضعیت)، به عنوان یک نمودار مکمل Sequence Diagram، برای مشخص‌کردن وضوح بیشتر برای شناخت حالت‌های مختلف یک شیء مورد استفاده قرار می‌گیرد.

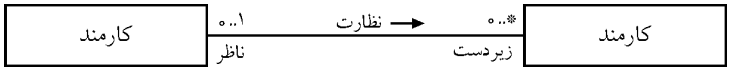
همه نمودارهای فوق (توالی، همکاری و حالت)، مدل‌سازی رفتاری یا تعاملی محسوب می‌گردند و در فعالیت پیاده‌سازی به نسخه فیزیکی بدل می‌گردند. بنابراین گزینه دوم پاسخ سؤال نیست.

۲۰- گزینه (۳) صحیح است.

در UML، یک رابطه‌ی خودانجمنی، رابطه‌ای است که یک کلاس با خودش دارد. لذا اگر اشیای یک کلاس با یکدیگر در ارتباط باشند، ارتباط مذکور از نوع خودانجمنی است. به عبارت دیگر، اگر ابتدا و انتهای یک رابطه انجمنی به یک کلاس اشاره داشته باشد، در این صورت رابطه‌ی مذکور از نوع خودانجمنی خواهد بود. در شکل زیر، دو مثال برای رابطه بازتابی ارائه شده است. توجه داشته باشید که در صورت مشخص کردن نقش دو طرف رابطه، ذکر نام رابطه، ضروری نیست.



برای سادگی در نحوه خواندن رابطه خودانجمنی توصیه می‌کنیم، رابطه را به شکل خطی ایجاد کنید، سپس رابطه را بخوانید:



شکل مطرح شده در صورت سوال گویای رابطه خودانجمنی (Self Association) می‌باشد. به این معنی که یک کارمند هیچ زیردستی ندارد (چون مدیر نیست) یا چندین زیردست دارد (چون مدیر است) همچنین یک کارمند هیچ ناظری ندارد (چون مدیرکل است) یا یک ناظر دارد (چون مدیر کل نیست). بنابراین گزینه سوم درست است.

هرگاه رابطه‌ای مابین یک واحد کل و تعدادی واحد جزء مطرح باشد، یعنی یک کلاس کل از

همکاری تعدادی کلاس جزء تشکیل شده باشد، آنگاه در این شرایط رابطه انجمنی پیشرفته یا رابطه بخشی خواهیم داشت. رابطه انجمنی پیشرفته یا رابطه بخشی به دو رابطه تجمع (Aggregation) و ترکیب (Composition) تقسیم می‌گردد. در رابطه انجمنی پیشرفته یا رابطه بخشی رابطه‌ای به شکل Self Aggregation وجود ندارد. بنابراین گزینه اول نادرست است.

راه ایجاد تعاملات پویا مابین اشیاء همکار، ارسال پیام (Message) است، در واقع پیام مکانیزمی است که اشیاء به وسیله آن با هم ارتباط برقرار می‌کنند. یک پیام باعث می‌شود رفتاری در شیء گیرنده انجام شود. بنابراین ارسال یک پیام از یک شیء مبدا به شیء مقصد به معنی صدا زدن یکی از متدهای شیء مقصد است. یک پیام شامل شیء مقصد (شیء گیرنده). متد (متدی که در شیء مقصد پیام را دریافت می‌کند) و پارامترهای مربوطه می‌باشد.

مدل‌سازی تعاملات پویای میان اشیاء همکار یا مدل‌سازی رفتاری، رفتار سیستم را در زمان اجرا نمایش می‌دهد. به بیان دیگر در این دیدگاه رفتارهای پویای سیستم مدل می‌شود.

مدل‌سازی رفتاری سیستم با استفاده از نمودارهای UML قابل انجام است:

الف) نمودار توالی (Sequence Diagram)

Sequence Diagram یا نمودار توالی، Object Diagram یا نمودار شیء جهت مدل‌سازی تعاملات پویای میان اشیاء همکار داخل یک use case مورد استفاده قرار می‌گیرد.

ب) نمودار همکاری (Collaboration Diagram)

Collaboration Diagram یا نمودار همکاری در نسخه‌های امروزی UML، به نمودار **Communication Diagram** یا **نمودار ارتباط**، تغییر نام یافته است.

Collaboration Diagram یا نمودار همکاری، برای یک کاربرد خاص، جهت مدل‌سازی تعاملات پویای میان اشیاء همکار داخل یک use case مورد استفاده قرار می‌گیرد.

ج) نمودار حالت (State Transition Diagram)

State Transition Diagram یا نمودار انتقال حالت (وضعیت)، به عنوان یک نمودار مکمل Sequence Diagram، برای مشخص‌کردن وضوح بیشتر برای شناخت حالت‌های مختلف یک شیء مورد استفاده قرار می‌گیرد.

در صورتی که یک شیء به خود پیامی ارسال کند به آن message to self گفته می‌شود مانند خواندن شماره کارت توسط دستگاه کارت‌خوان در یک سیستم ATM.

۲۱- گزینه (۱) صحیح است.

پس از شناسایی موارد کاربرد و سناریونویسی برای هر یک از موارد کاربرد، زمان تعریف کلاس‌ها، صفات، متدها و ارتباطات میان کلاس‌های همکار برای هر یک از موارد کاربرد می‌رسد. برای شناسایی کلاس‌ها، صفات، متدها و ارتباطات میان کلاس‌ها برای هر یک از موارد کاربرد، از تکنیکی موسوم به CRC که سرواژه‌ی عبارت Class – Responsibility Collaborator و به معنی «مدل همکاری مسئولیت‌های کلاس‌ها» می‌باشد، استفاده می‌شود. مدل‌سازی CRC روشی ساده

جهت تعیین و سازماندهی کلاس‌های داخل هر مورد کاربرد است. در این روش به هر کلاس یک کارت CRC اختصاص داده می‌شود که شامل سه بخش کلی زیر است:

نام کلاس

مسئولیت‌های کلاس(Responsibilities)

- صفات کلاس
- متدهای کلاس

همکاران کلاس(Collaborators)

کلاس‌ها یا به تنهایی از عهده انجام مسئولیت‌های خود بر می‌آیند و یا از طریق همکاری با کلاس‌های همکار خود از عهده انجام مسئولیت‌های خود بر می‌آیند. بنابراین اگر کلاسی همکارانی دارد، باید در بخش مربوط به همکاران نوشته شود.

برای کشف کلاس‌های همکار داخل هر use case، از سناریوی اصلی (نوشتاری یا نموداری) هر use case استفاده می‌گردد. برای این منظور، **اسامی** موجود داخل هر use case مورد جستجو قرار می‌گیرند. بنابر آنچه گفتیم، گزینه اول درست است.

تشخیص روابط کل – جزء و توارث بین کلاس‌ها توسط نمودار کلاس انجام می‌گردد. بنابراین گزینه‌های دوم و سوم نادرست هستند.

کلاس‌ها با یکدیگر تعاملات پویا ندارند و با یکدیگر تبادل پیام نمی‌کنند در واقع کلاس‌ها همچون یک قاب عکس ثابت، فقط با یکدیگر ارتباطات ایستا دارند برای‌مثال فقط مشخص است که استاد و دانشجو با یکدیگر ارتباط دارند. اما اشیاء با یکدیگر تعاملات پویا در گذر زمان دارند و با یکدیگر اقدام به تبادل پیام می‌کنند، که این تبادلات پیام توسط نمودار توالی یا همکاری نشان داده می‌شود. مانند پیام‌هایی که میان مشتری و سیستم رد و بدل می‌شود تا مشتری مبلغی از حساب خود توسط یک دستگاه خودپرداز، برداشت کند. این تعاملات پویا توسط نمودار توالی و همکاری نشان داده می‌شود.

مدل‌سازی تعاملات پویای میان اشیاء همکار یا مدل‌سازی رفتاری، رفتار سیستم را در زمان اجرا نمایش می‌دهد. به بیان دیگر در این دیدگاه رفتارهای پویای سیستم مدل می‌شود.

مدل‌سازی رفتاری سیستم با استفاده از نمودارهای UML قابل انجام است:

الف) نمودار توالی (Sequence Diagram)

Sequence Diagram یا نمودار توالی، Object Diagram یا نمودار شیء جهت مدل‌سازی تعاملات پویای میان اشیاء همکار داخل یک use case مورد استفاده قرار می‌گیرد.

ب) نمودار همکاری (Collaboration Diagram)

Collaboration Diagram یا نمودار همکاری در نسخه‌های امروزی UML، به نمودار **Communication Diagram** یا **نمودار ارتباط**، تغییر نام یافته است.

Collaboration Diagram یا نمودار همکاری، برای یک کاربرد خاص، جهت مدل‌سازی

تعاملات پویای میان اشیاء همکار داخل یک use case مورد استفاده قرار می‌گیرد.

ج) نمودار حالت (State Transition Diagram)

State Transition Diagram یا نمودار انتقال حالت (وضعیت)، به عنوان یک نمودار مکمل sequence Diagram، برای مشخص‌کردن وضوح بیشتر برای شناخت حالت‌های مختلف یک شیء مورد استفاده قرار می‌گیرد.

بنابرآنچه گفتیم، گزینه چهارم نیز نادرست است.

۲۲- گزینه (۱) صحیح است.

در فعالیت ارتباطات (مهندسی نیازمندی‌ها) لیست نیازمندی‌های مشتری از طریق ارتباط با مشتری و ابزارهایی همچون گفتگو، مشاهده مصاحبه، پرسش‌نامه، بازدید، نمونه‌سازی دورانداختنی و نمونه‌سازی تکاملی، جمع‌آوری و آماده می‌گردد.

مهم‌ترین کار، در فعالیت ارتباطات، مدیریت شناسایی نیازمندی‌ها و پیگری تغییرات نیازمندی‌های مشتری است.

لیست نیازمندی‌های مشتری در این مرحله به صورت متن نوشته می‌شود.

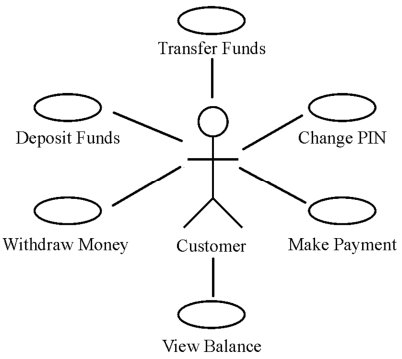
مثال: نمایش لیست نیازمندی‌های مشتری برای سیستم ATM.

۱- واریز وجه
۲- انتقال وجه
۳- برداشت وجه
۴- تغییر رمز
۵- پرداخت
۶- نمایش موجودی

پس از جمع‌آوری نیازمندی‌ها در فعالیت ارتباطات، نوبت به مدل تحلیل (مدل‌سازی نیازمندی‌ها) می‌رسد. مدل‌سازی که فعالیتی فنی به شمار می‌رود، نیازمندی‌ها را باید به گونه‌ای مدل نماید که برای سازنده و مشتری قابل فهم باشد.

Use Case Diagram یا نمودار مورد کاربرد، Requirement Diagram یا نمودار نیاز جهت مدل‌سازی لیست نیازمندی‌های مشتری مورد استفاده قرار می‌گیرد.

مثال: مدل‌سازی لیست نیازمندی‌های مشتری برای سیستم ATM.



در نمودار فوق، لیست نیازمندی‌های مشتری، مدل‌سازی شده است.

به هر یک از نیازهای فوق یک use case یا مورد کاربر گفته می‌شود و به اجتماع این use caseها، Use Case Diagram یا نمودار مورد کاربرد گفته می‌شود.

از آنجا که Use Case Diagram یا نمودار مورد کاربرد یا نمودار نیاز مستقل از مفاهیم شیءگرایی (کلاس، وراثت و چندریختی) است، بنابراین می‌توان مدل‌سازی لیست نیازمندی‌های مشتری در مرحله **مدل تحلیل متدولوژی ساخت‌یافته** (مهندسی‌نرم‌افزار ساخت‌یافته) را توسط این نمودار انجام داد. بنابراین گزینه اول درست است.

سایر گزینه‌ها یعنی نمودارهای کلاس، ترتیبی (توالی) و ارتباط، مختص متدولوژی شیءگرا هستند.

Collaboration Diagram یا نمودار همکاری در نسخه‌های امروزی UML، به نمودار **Communication Diagram** یا **نمودار ارتباط**، تغییر نام یافته است.

۲۳- گزینه (۳) صحیح است.

مدل تحلیل به روش ساخت‌یافته از سه بخش مدل‌سازی داده‌ای، مدل‌سازی عملکردی و مدل‌سازی رفتاری تشکیل شده است، مدل‌سازی داده‌ای شامل تحلیل موجودیت‌ها و تحلیل پرس و جوها می‌باشد. تحلیل موجودیت‌ها توسط ابزار مدل ER و تحلیل پرس و جوها توسط ابزار حساب رابطه‌ای مدل می‌شوند، مدل‌سازی عملکردی توسط ابزار DFD و مدل‌سازی رفتاری توسط ابزار STD مدل می‌شود.

DFD یا Data Flow Diagram یا نمودار جریان داده جهت مدلسازی عملکردی مورد استفاده قرار می‌گیرد.

مدل‌سازی عملکردی یا مدل‌سازی جریان‌گرا (Flow-Oriented Models)، در زبان UML وجود نداشته و تحلیل شیءگرا (OOA) براساس آن انجام نمی‌شود. در عوض این مدل یکی از مهم‌ترین مدل‌های مدل تحلیل ساخت‌یافته می‌باشد.

هریک از نمودارهای UML در فعالیت‌های چارچوبی فرآیند تولید نرم‌افزار بر اساس متدولوژی شیءگرا مدل‌سازی خاص خود را بر عهده دارند.

۱- فعالیت ارتباط (مهندسی نیازمندی‌ها)

در این مرحله لیست نیازمندی‌های مشتری از طریق ارتباط با مشتری و ابزارهایی همچون گفتگو، مشاهده مصاحبه، پرسش‌نامه، بازدید، نمونه‌سازی دوراندا‌ختنی و نمونه‌سازی تکاملی، جمع‌آوری و آماده می‌گردد.

لیست نیازمندی‌های مشتری در این مرحله به صورت متن نوشته می‌شود.

۲- فعالیت برنامه‌ریزی

برنامه‌ریزی یعنی هنر حرکت از مبدأ موجود به مقصد مطلوب برای رسیدن به نتیجه‌ای مطلوب بر اساس خواسته‌های مورد نیاز در یک زمان مشخص.

۳- فعالیت مدل‌سازی (تحلیل و طراحی)

مدل تحلیل: OOA (Object-Oriented Analysis)

پس از جمع‌آوری نیازمندی‌ها در فعالیت ارتباطات، نوبت به مدل تحلیل (مدل‌سازی نیازمندی‌ها) می‌رسد.

مدل تحلیل، شامل مراحل زیر می‌باشد:

الف) مدل‌سازی محیط عملیاتی یک کسب و کار

Business use case یا مورد کاربرد کسب و کار، جهت مدل‌سازی محیط عملیاتی یک کسب و کار مورد استفاده قرار می‌گیرد.

ب) مدل‌سازی لیست نیازمندی‌های مشتری

Use Case Diagram یا نمودار مورد کاربرد، Requirement Diagram یا نمودار نیاز جهت مدل‌سازی لیست نیازمندی‌های مشتری مورد استفاده قرار می‌گیرد.

ج) سناریونویسی

در این مرحله برای هر use case (مورد کاربرد یا نیاز) سناریو یا شرح حال نوشته می‌شود.

سناریو بر دو طبقه اصلی و فرعی می‌باشد.

نحوه نمایش سناریوی اصلی و فرعی به دو روش زیر می‌باشد:

نوشتاری (متنی): در این روش سناریوی اصلی و فرعی به صورت متنی نوشته می‌شود.

گرافیکی (نموداری): در این روش سناریوی اصلی و فرعی به صورت نمودار، مدل‌سازی می‌شود.

Activity Diagram یا نمودار فعالیت و Swimlane Diagram یا نمودار خط شنا، جهت مدل‌سازی سناریوی اصلی و فرعی داخل یک use case (نیاز یا مورد کاربرد یا زیرسیستم) مورد استفاده قرار می‌گیرد.

د) کشف کلاس‌های همکار داخل هر use case یا مورد کاربرد

پس از شناسایی موارد کاربرد و سناریونویسی برای هر یک از موارد کاربرد، زمان تعریف کلاس‌ها، صفات، متدها و ارتباطات میان کلاس‌های همکار برای هر یک از موارد کاربرد می‌رسد. برای شناسایی کلاس‌ها، صفات، متدها و ارتباطات میان کلاس‌ها برای هر یک از موارد کاربرد، از تکنیکی موسوم به CRC که سرواژه‌ی عبارت Class – Responsibility Collaborator و به معنی «مدل همکاری مسئولیت‌های کلاس‌ها» می‌باشد، استفاده می‌شود.

هـ) مدل‌سازی ارتباطات ایستای میان کلاس‌های همکار

پس از شناسایی کلاس‌ها و کلاس‌های همکار برای هر یک از موارد کاربرد در مدل CRC، زمان مدل‌سازی نموداری و ساختاری توسط نمودار کلاس می‌رسد. Class Diagram یا نمودار کلاس جهت مدل‌سازی ارتباطات ایستای میان کلاس‌های همکار داخل یک use case مورد

استفاده قرار می‌گیرد.

در فعالیت مدل تحلیل، جزئیات مربوط به کلاس‌ها، شامل جزئیات دقیق صفات و متدها، مشخص نمی‌گردد، بیان این جزئیات تا بخش طراحی مولفه از مدل طراحی به تأخیر می‌افتد.

ی) مدل‌سازی تعاملات پویای میان اشیاء همکار

مدل‌سازی تعاملات پویای میان اشیاء همکار یا مدل‌سازی رفتاری، رفتار سیستم را در زمان اجرا نمایش می‌دهد. به بیان دیگر در این دیدگاه رفتارهای پویای سیستم مدل می‌شود.

مدل‌سازی رفتاری سیستم با استفاده از نمودارهای UML قابل انجام است:

الف) نمودار توالی (Sequence Diagram)

Sequence Diagram یا نمودار توالی، Object Diagram یا نمودار شیء جهت مدل‌سازی تعاملات پویای میان اشیاء همکار داخل یک use case مورد استفاده قرار می‌گیرد.

ب) نمودار همکاری (Collaboration Diagram)

Collaboration Diagram یا نمودار همکاری در نسخه‌های امروزی UML، به نمودار **Communication Diagram** یا **نمودار ارتباط**، تغییر نام یافته است.

Collaboration Diagram یا نمودار همکاری، برای یک کاربرد خاص، جهت مدل‌سازی تعاملات پویای میان اشیاء همکار داخل یک use case مورد استفاده قرار می‌گیرد.

ج) نمودار حالت (State Transition Diagram)

State Transition Diagram یا نمودار انتقال حالت (وضعیت)، به عنوان یک نمودار مکمل sequence Diagram، برای مشخص‌کردن وضوح بیشتر برای شناخت حالت‌های مختلف یک شیء مورد استفاده قرار می‌گیرد.

مدل طراحی: OOD (Object – Oriented Design)

پس از مدل تحلیل، نوبت به مدل طراحی می‌رسد. مدل طراحی به روش شی‌گراء شامل چهار بخش **طراحی داده**، **طراحی معماری**، **طراحی مؤلفه** و **طراحی واسط** می‌باشد.

طراحی داده

طراحی داده، شامل طراحی ساختمان داده‌ها و طراحی پرس‌وجوها می‌باشد.

طراحی معماری

طراحی معماری یا معماری نرم‌افزار، ساختار کلی نرم‌افزار و شیوه‌های یکپارچگی یک سیستم را بیان می‌کند.

طراحی مؤلفه

طراحی مؤلفه، طراحی معماری از همان فعالیت مدل طراحی را به عنوان ورودی دریافت کرده و طراحی مؤلفه را توسط ابزارهایی همچون شبه کد یا Activity Diagram یا Swimlane Diagram ایجاد می‌کند.

در این مرحله هر use case یا مورد کاربرد یا نیاز با اطلاعات مربوط به نمودارهای کلاس و توالی به یک مؤلفه فرعی اما با ذکر جزئیات تبدیل می‌گردد.

طراحی واسط

طراحی واسط یا همان واسط کاربر، براساس ورودی‌ها و خروجی‌های مورد نیاز کاربران نهایی به شکل نقشی بر روی کاغذ یا طرحی بر روی کامپیوتر ایجاد می‌گردد. مانند نحوه چیدمان منوها و فرم‌ها.

۴- فعالیت ساخت (پیاده‌سازی و تست)

پس از فعالیت مدل‌سازی (تحلیل و طراحی) نوبت به فعالیت ساخت (پیاده‌سازی و تست) می‌رسد.

پیاده سازی: OOP (Object – Oriented Programming)

پس از مدل طراحی نوبت به پیاده‌سازی می‌رسد. این کار توسط زبان‌های شی‌گراء، مانند C++ و SQL Server انجام می‌گردد.

Component Diagram یا نمودار مؤلفه جهت مدل‌سازی ساختار کلی پیاده‌سازی برنامه و ترتیب کامپایل مؤلفه‌های فرعی، برای ایجاد مؤلفه‌اصلی (مؤلفه کلی یا برنامه اصلی) مورد استفاده قرار می‌گیرد.

تست: OOT (Object – Oriented Testing)

پس از پیاده‌سازی نوبت به تست می‌رسد، در این مرحله کلیه موارد پیاده‌سازی شده از نظر خطاهای نحوی و خطاهای معنایی براساس لیست نیازمندی‌های مشتری (چک لیست) که در فعالیت ارتباطات تهیه شده بود، مورد وارسی قرار می‌گیرد.

فعالیت استقرار

پس از تست، نوبت به فعالیت استقرار می‌رسد. هدف از فعالیت استقرار در گام اول، ارائه یک دید کلی از سخت‌افزارهای مورد نیاز سیستم است که مؤلفه‌های نرم‌افزاری باید بر روی آن‌ها قرار گیرند. و در گام دوم، نرم‌افزار به مشتری تحویل داده می‌شود و مشتری با بررسی محصول دریافتی، باز خوردهای به دست آمده براساس همین ارزیابی‌ها را به تیم نرم‌افزاری ارائه می‌دهد.

Deployment Diagram یا نمودار استقرار، جهت مدل‌سازی گره‌های سخت افزاری برای نصب نرم‌افزار مربوطه در فعالیت استقرار مورد استفاده قرار می‌گیرد.

در ادامه به بررسی گزینه‌ها می‌پردازیم:

Business Use Case مربوط به مدل تحلیل شیءگرا است.

Class Diagram مربوط به مدل تحلیل و طراحی شیءگرا است.

CRC مربوط به مدل تحلیل شیءگرا است.

Use Case Diagram مربوط به مدل تحلیل شیءگرا است.

Sequence Diagram مربوط به مدل تحلیل و طراحی شیءگرا است.

Flow-Oriented Diagram مربوط به مدل تحلیل ساخت‌یافته است.
Deployment Diagram مربوط به فعالیت استقرار شیءگرا است. بنابراین گزینه سوم درست است.
Object Diagram مربوط به مدل تحلیل و طراحی شیءگرا است.
Activity Diagram, Swimlane Diagram مربوط به مدل تحلیل و طراحی شیءگرا است.

۲۴- گزینه (۲) صحیح است.

مدل طراحی به روش ساخت‌یافته شامل چهار بخش **طراحی داده**، **طراحی معماری**، **طراحی مؤلفه** و **طراحی واسط** می‌باشد.

طراحی معماری، تحلیل عملکرد (DFD) از مدل تحلیل را به عنوان ورودی دریافت کرده و توسط سبک ساخت‌یافته(فراخوانی و بازگشت)، طراحی معماری را انجام می‌دهد.

طراحی معماری یا معماری نرم‌افزار، ساختار کلی نرم‌افزار و شیوه‌های یکپارچگی یک سیستم را بیان می‌کند. به عبارت دیگر، ساختار سلسله مراتبی **مولفه‌های برنامه (توابع یا پیمانه‌ها)**، شیوه تعامل مولفه‌ها با یکدیگر و **ساختمان داده‌های** مورد نیاز مولفه‌ها را نشان می‌دهد. معماری نرم‌افزار یک مدل قابل درک از چگونگی سازمان‌دهی سیستم است. در واقع نشان‌گر ساختمان داده‌ها و مؤلفه‌های برنامه‌ای است که برای ساختن یک سیستم کامپیوتری لازم است. به طور دقیق‌تر معماری نرم‌افزار شامل دو سطح از طراحی می‌باشد یعنی طراحی داده و طراحی معماری. در واقع این ساختار مانند یک نقشه ساختمان، مبنای ساخت نرم‌افزار قرار می‌گیرد.

در طراحی معماری، اسکلت، ساختار و چیدمان کلی مولفه‌های (توابع) برنامه به این معنی که چه مولفه‌ای (تابعی) چه مولفه‌ای (تابعی) دیگر را صدا می‌زند، بدون ذکر جزئیات داخلی مولفه‌ها (توابع) مشخص می‌گردد (ساختار درختی برنامه بدون ذکر جزئیات مولفه‌ها (توابع)). مانند اسکلت یک ساختمان که گویای جایگاه مؤلفه‌های ساختمان است اما هنوز آجرچینی نشده است. (اسکلت یک ساختمان بدون آجرچینی).

در ساخت‌یافتگی یک مدل ساختاری ساختار واحدها و مولفه‌های برنامه را نشان می‌دهد و نه رفتار سیستم در قبال رخدادها را. در ساخت‌یافتگی رفتار سیستم در قبال رخدادها در مدل تحلیل به کمک نمودار انتقال حالت یا STD نشان داده می‌شود.

در ساخت‌یافتگی، STD برای مدل‌سازی رفتاری سیستم در قبال رخدادهای عالم خارج نرم‌افزار مورد استفاده قرار می‌گیرد.

مدل‌سازی رفتاری توسط STD، در سیستم‌های بی‌درنگ مورد استفاده قرار می‌گیرد.

هر نرم‌افزاری که توسط سنسور و حسگر، عالم خارج را شنود و مورد ارزیابی قرار می‌دهد، تا در موقع لزوم و در زمان واقعی، حقیقی و تا دیر نشده عکس‌العمل نشان دهد، یک نرم‌افزار بی‌درنگ است.

مدل طراحی به روش شیءگرا شامل چهار بخش **طراحی داده**، **طراحی معماری**، **طراحی مؤلفه** و **طراحی واسط** می‌باشد.

طراحی معماری، نمودار کلاس و نمودار توالی مرتبط با هر یک از موارد کاربرد را از مدل تحلیل، به عنوان ورودی دریافت کرده و توسط سبک شیءگرا (مبتنی بر ارسال پیام مابین اشیاء)، طراحی معماری را انجام می‌دهد.

طراحی معماری یا معماری نرم‌افزار، ساختار کلی نرم‌افزار و شیوه‌های یکپارچگی یک سیستم را بیان می‌کند. به عبارت دیگر، ساختار سلسله مراتبی **مولفه‌های برنامه (کلاس‌ها یا پیمانه‌ها)**، شیوه تعامل مولفه‌ها با یکدیگر و **ساختمان داده‌های** مورد نیاز مولفه‌ها را نشان می‌دهد. معماری نرم‌افزار یک مدل قابل درک از چگونگی سازمان‌دهی سیستم است. در واقع نشان‌گر ساختمان داده‌ها و مؤلفه‌های برنامه‌ای است که برای ساختن یک سیستم کامپیوتری لازم است. به طور دقیق‌تر معماری نرم‌افزار شامل دو سطح از طراحی می‌باشد یعنی طراحی داده و طراحی معماری. در واقع این ساختار مانند یک نقشه ساختمان، مبنای ساخت نرم‌افزار قرار می‌گیرد.

در طراحی معماری، اسکلت، ساختار و چیدمان کلی مؤلفه‌های برنامه به این معنی که چه مولفه‌ای (کلاسی) با چه مولفه‌ای (کلاسی) دیگر در ارتباط است، بدون ذکر جزئیات مربوط به شرح متدهای کلاس‌های همکار داخل یک مؤلفه فرعی (یک بخش از نرم‌افزار). مانند اسکلت یک ساختمان که گویای جایگاه مؤلفه‌های ساختمان است اما هنوز آجر چینی نشده است. (اسکلت یک ساختمان بدون آجر چینی).

واحد مؤلفه (پیمانه) در شیءگرایی، کلاس است که از کنار هم قرار گرفتن کلاس‌های همکار داخل هر use case یا مورد کاربرد یا نیاز، یک مؤلفه فرعی (مؤلفه بخشی) ایجاد می‌گردد، که از کنار هم قرار گرفتن مؤلفه‌های فرعی برنامه، مؤلفه اصلی (مؤلفه کلی یا برنامه اصلی) ایجاد می‌گردد.

در این مرحله هر use case یا مورد کاربرد یا نیاز با اطلاعات مربوط به نمودارهای کلاس و توالی به یک مؤلفه فرعی ولی بدون ذکر جزئیات تبدیل می‌گردد. همچنین در این مرحله نحوه چیدمان مولفه‌ها و ساختار کلی برنامه، کلاس‌های همکار، اشیاء همکار و تعریف ساختار پیام‌ها مابین فرستنده و گیرنده پیام‌ها اما بدون ذکر جزئیات مشخص می‌شود.

در شیءگرایی یک مدل ساختاری ساختار واحدها و مولفه‌های برنامه را نشان می‌دهد و نه رفتار سیستم در قبال رخدادها را. در شیءگرایی رفتار سیستم در قبال رخدادها در مدل تحلیل به کمک نمودار حالت نشان داده می‌شود.

اطلاعات به دست آمده از نمودار حالت در مدل تحلیل مربوط به کلاس‌های دارای حالات مختلف، در مدل طراحی، بخش طراحی مولفه، جهت تکمیل نمودن الگوریتم‌های مربوط به متدهای کلاس‌های دارای حالات مختلف مورد استفاده قرار می‌گیرد. اینکه متدهای یک کلاس دارای حالات مختلف در سطح طراحی مولفه در مواجه با حالت‌های مختلف یک شیء چگونه رفتار کنند از نمودار حالت استنباط می‌گردد.

بنابرآنچه گفتیم، واضح است که گزینه‌های اول، سوم و چهارم نادرست و گزینه دوم درست است.

۲۵- گزینه (۴) صحیح است.

صورت سوال به این شکل است:

کدام نمودار UML، برای مدل‌سازی تعامل اشیاء به کار می‌رود؟

۱) حالت (State Diagram)

گزینه اول پاسخ سوال نیست، زیرا State Transition Diagram یا نمودار انتقال حالت (وضعیت)، به عنوان یک نمودار مکمل sequence Diagram، برای مشخص‌کردن وضوح بیشتر برای شناخت حالت‌های مختلف یک شیء مورد استفاده قرار می‌گیرد. به بیان دیگر نمودار انتقال حالت، چرخه حیات یک شیء را در حالت های مختلف (از زمانی که شیء ایجاد می‌شود تا زمانی که شیء از بین می‌رود) نمایش می‌دهد. در واقع این نمودار تمامی حالت‌های مختلفی را که یک شیء در طول حیات خود می‌تواند داشته باشد و همچنین نحوه انتقال بین حالت‌ها و وقایع باعث‌شونده این انتقال را نشان می‌دهد. به بیان دیگر نمودارهای حالت، راهی را آماده می‌کنند تا حالت‌های مختلف یک شیء را مدل کنند.

۲) مولفه (Component Diagram)

گزینه دوم پاسخ سوال نیست، زیرا Component Diagram یا نمودار مؤلفه جهت مدل‌سازی ساختار کلی پیاده‌سازی برنامه و ترتیب کامپایل مؤلفه‌های فرعی، برای ایجاد مؤلفه‌اصلی (مؤلفه کلی یا برنامه اصلی) مورد استفاده قرار می‌گیرد. به بیان دیگر نمودار مؤلفه یک دید فیزیکی از مدل سیستم به همراه مؤلفه‌های فرعی نرم‌افزار و روابط بین آنها را نشان می‌دهد.

۳) فعالیت (Activity Diagram)

گزینه سوم پاسخ سوال نیست، زیرا Activity Diagram یا نمودار فعالیت و Swimlane Diagram یا نمودار خط شنا، جهت مدل‌سازی سناریوی اصلی و فرعی داخل یک use case (نیاز یا مورد کاربرد یا زیرسیستم) مورد استفاده قرار می‌گیرد. به بیان دیگر نمودار فعالیت یا نمودار خط شنا، جهت مدل‌سازی روال انجام کارها داخل یک use case مورد استفاده قرار می گیرد. همچنین، در طراحی مؤلفه، باید جزئیات الگوریتمی متدهای کلاس تشریح شود. برای این منظور Activity Diagram یا نمودار فعالیت و Swimlane Diagram یا نمودار خط‌شنا مورد استفاده قرار می‌گیرد. نمودار فعالیت و نمودار خط‌شنا ساختاری مشابه فلوچارت دارد.

۴) توالی (Sequence Diagram)

گزینه چهارم پاسخ سوال است. زیرا Sequence Diagram یا نمودار توالی، Object Diagram یا نمودار شیء جهت مدل‌سازی تعاملات پویای میان اشیاء همکار داخل یک use case مورد استفاده قرار می‌گیرد.

۲۶- گزینه (۲) صحیح است.

صورت سوال به این شکل است:

کدام نمودار UML، برای اساسا برای مدل‌سازی ساختار(معماری) محصول نرم‌افزاری به

کار می‌رود؟

۱) نمودار فعالیت (Activity Diagram)

گزینه اول پاسخ سوال نیست، زیرا Activity Diagram یا نمودار فعالیت و Swimlane Diagram یا نمودار خط شنا، جهت مدل‌سازی سناریوی اصلی و فرعی داخل یک use case (نیاز یا مورد کاربرد یا زیرسیستم) مورد استفاده قرار می‌گیرد. به بیان دیگر نمودار فعالیت یا نمودار خط شنا، جهت مدل‌سازی روال انجام کارها داخل یک use case مورد استفاده قرار می‌گیرد. همچنین، در طراحی مؤلفه، باید جزئیات الگوریتمی متدهای کلاس تشریح شود. برای این منظور Activity Diagram یا نمودار فعالیت و Swimlane Diagram یا نمودار خط‌شنا مورد استفاده قرار می‌گیرد. نمودار فعالیت و نمودار خط‌شنا ساختاری مشابه فلوچارت دارد.

۲) نمودار مولفه (Component Diagram)

گزینه دوم پاسخ سوال است، زیرا Component Diagram یا نمودار مؤلفه جهت مدل‌سازی ساختار کلی پیاده‌سازی برنامه به عبارت دیگر ساختار (معماری) محصول نرم‌افزاری و ترتیب کامپایل مؤلفه‌های فرعی، برای ایجاد مؤلفه‌اصلی (مؤلفه کلی یا برنامه اصلی) مورد استفاده قرار می گیرد. به بیان دیگر نمودار مؤلفه یک دید فیزیکی از مدل سیستم به همراه مؤلفه‌های فرعی نرم‌افزار و روابط بین آنها را نشان می‌دهد.

توجه: دقت کنید که در صورت سوال به ساختار (معماری) یک «**محصول نرم افزاری**» تاکید شده است، یعنی فعالیت پیاده‌سازی.

۳) نمودار توالی (Sequence Diagram)

گزینه سوم پاسخ سوال نیست. زیرا Sequence Diagram یا نمودار توالی، Object Diagram یا نمودار شیء جهت مدل‌سازی تعاملات پویای میان اشیاء همکار داخل یک use case مورد استفاده قرار می‌گیرد.

۴) نمودار مورد کاربرد (Use-Case Diagram)

گزینه چهارم پاسخ سوال نیست. زیرا Use Case Diagram یا نمودار مورد کاربرد، Requirement Diagram یا نمودار نیاز جهت مدل‌سازی لیست نیازمندی‌های مشتری مورد استفاده قرار می‌گیرد.

۲۷- گزینه (۱) صحیح است.

Activity Diagram یا نمودار فعالیت و Swimlane Diagram یا نمودار خط شنا، جهت مدل‌سازی سناریوی اصلی و فرعی داخل یک use case (نیاز یا مورد کاربرد یا زیرسیستم) یا «**توصیف بصری موارد کاربرد**» مورد استفاده قرار می‌گیرد. به بیان دیگر نمودار فعالیت یا نمودار خط شنا، جهت مدل‌سازی روال انجام کارها داخل یک use case مورد استفاده قرار می‌گیرد. توصیف بصری به معنی مدل‌سازی است.

۲۸- گزینه (۳) صحیح است.

در طراحی مؤلفه، باید جزئیات الگوریتمی متدهای کلاس (منطق داخلی عملیات یک کلاس) تشریح شود، منظور از عملیات یک کلاس در صورت سوال، همان متدهای یک کلاس است. برای این منظور Activity Diagram یا نمودار فعالیت و Swimlane Diagram یا نمودار خطشنا مورد استفاده قرار می‌گیرد. نمودار فعالیت و نمودار خطشنا ساختاری مشابه فلوچارت دارد. نمودار فعالیت و نمودار خطشنا نه تنها شرح‌حال متدهای کلاس را تصویر می‌کند، بلکه در بخش مدل تحلیل شی‌گراء یا OOA، جهت مدل‌سازی سناریوهای اصلی و فرعی داخل یک use case نیز مورد استفاده قرار می‌گیرد. علاوه بر نمودار فعالیت و نمودار خطشنا، توسط شبه کد (PDL) نیز می‌توان جزئیات مربوط به شرح متدهای کلاس را نمایش داد. دقت کنید که Sequence Diagram یا نمودار توالی، Object Diagram یا نمودار شیء جهت مدل‌سازی تعاملات پویای میان اشیاء همکار داخل یک use case مورد استفاده قرار می‌گیرد.

۲۹- گزینه (۱) صحیح است.

نمودار استقرار یا مستقرسازی (Deployment Diagram)، جهت مدل‌سازی گره‌های سخت‌افزاری یا محیط رایانش (Computing Environment) برای نصب نرم‌افزار مربوطه در فعالیت استقرار مورد استفاده قرار می‌گیرد.

۳۰- گزینه (۱) صحیح است.

Use Case Diagram یا نمودار مورد کاربرد، Requirement Diagram یا نمودار نیاز جهت مدل‌سازی لیست نیازمندی‌های مشتری مورد استفاده قرار می‌گیرد. به یک use case یا مورد کاربرد، نیاز (requirement) یا زیرسیستم (subsystem) نیز گفته می‌شود.

صورت گزینه‌ها به صورت زیر است:

۱) پیش‌شرایط و پس‌شرایط مورد کاربرد

گزینه اول پاسخ سوال نیست، زیرا در مرحله سناریونویسی **(توصیف تفصیلی)** برای هر use case (مورد کاربرد یا نیاز) سناریو یا شرح حال نوشته می‌شود.

سناریو بر دو طبقه اصلی و فرعی می‌باشد:

سناریوی اصلی: بیان روال حرکت قدم به قدم کارها داخل یک use case از نقطه شروع تا رسیدن به یک نتیجه مطلوب (موفق). (حرکت از خود موجود و رسیدن به خود مطلوب). «**یعنی پیش‌شرایط و پس‌شرایط مورد کاربرد در توصیف تفصیلی یک مورد کاربرد (Use Case) آورده می‌شود.**»

سناریوی فرعی: بیان روال حرکت قدم به قدم کارها داخل یک use case از نقطه شروع تا رسیدن به یک نتیجه نامطلوب (ناموفق). (حرکت از خود موجود و رسیدن به خود نامطلوب).

۲) کلاس‌ها و اشیاء محقق‌کننده مورد کاربرد

گزینه دوم پاسخ سوال نیست، زیرا کلاس‌ها و اشیاء محقق‌کننده مورد کاربرد پس از مرحله سناریونویسی **(توصیف تفصیلی) و توسط** تکنیکی موسوم به CRC کشف می‌شوند.

۳) کد پیاده‌سازی شده مورد کاربرد

گزینه سوم پاسخ سوال نیست، زیرا کد پیاده‌سازی شده مورد کاربرد پس از سناریونویسی (توصیف تفصیلی) و مراحل بعدی همچون مدل طراحی و در نهایت در مرحله پیاده‌سازی ایجاد می‌شود.

۴) موارد آزمون طراحی شده برای مورد کاربرد

گزینه چهارم پاسخ سوال است، زیرا موارد آزمون طراحی شده برای مورد کاربرد پس از سناریونویسی (توصیف تفصیلی) و مراحل بعدی و در نهایت در مرحله تست مورد استفاده قرار می‌گیرد. Use Case Diagram یا نمودار مورد کاربرد، می‌تواند به عنوان مدل لیست نیازمندی‌ها، به شکل چک لیست برای فعالیت تست مورد استفاده قرار بگیرد.

۸ تست و استقرار نرم‌افزار

مقدمه

پس از پیاده‌سازی نوبت به تست می‌رسد، در این مرحله کلیه‌ی موارد پیاده‌سازی شده از نظر خطاهای معنایی براساس لیست نیازمندی‌های مشتری (چک لیست) که در فعالیت ارتباطات تهیه و در مدل تحلیل مدل‌سازی شده است مورد تست و بررسی قرار می‌گیرد تا مشخص شود نرم‌افزار ایجاد شده براساس ورودی‌های مورد نظر مشتری، خروجی‌های مورد انتظار مشتری را برآورده می‌سازد یا خیر.

هر محصولی که به دست انسان ساخته می‌شود به دلیل ماهیت انسانی و جایزالخطا بودنش، برای آن که از صحت و درستی تقریبی آن اطمینان حاصل شود، باید با توجه به نیازمندی‌های مورد نظر مشتری مورد تست و ارزیابی قرار گیرد تا اشکالات آن شناسایی و برطرف شود تا سرانجام محصولی مطلوب و ایده‌آل برای مشتری فراهم گردد.

این قاعده برای تولید و توسعه نرم‌افزار که گاه با جان انسان در ارتباط است مستثنی نیست!

بنابراین پس از ساخت هر محصول نرم‌افزاری، همواره این احتمال وجود دارد که برخی از بخش‌های آن بر اساس نیازمندی‌های مشتری عمل نکند. انسان است و جایز الخطا بودنش. هنگامی‌که محصول نرم‌افزاری ایجاد می‌گردد، در واقع از فعالیت‌های تحلیل، طراحی و پیاده‌سازی عبور کرده‌است، بنابراین اگر عملکردی ناسازگار با نیازمندی‌های مشتری آشکار گردد، در همان فعالیت‌ها رخ داده‌است. حال اگر این خطاها و عملکردهای ناسازگار با نیازمندی‌های مشتری توسط تیم توسعه نرم‌افزار کشف (uncover) نگردد، آنگاه در آینده توسط مشتری کشف می‌گردد و این یعنی ایجاد یک محصول نرم‌افزاری که مورد رضایت‌مندی مشتری قرار نمی‌گیرد. بنابراین فعالیت تست نرم‌افزار (software testing) به عنوان یک فعالیت کنترل‌کننده سازگاری و تطابق محصول نرم‌افزاری ایجاد شده با لیست نیازمندی‌های مشتری، قبل از تحویل محصول نرم‌افزاری به مشتری محسوب می‌گردد. و بیانگر بازبینی نهایی مراحل تحلیل، طراحی و پیاده‌سازی است.

فعالیت تست که به بررسی برآورده‌سازی کلیه نیازمندی‌های وظیفه‌مندی و غیروظیفه‌مندی

مشتری می‌پردازد، بخشی از فعالیت صحت و اعتبارسنجی (verification and validation) یا V&V است که خود نیز به عنوان بخشی از فعالیت‌های تضمین کیفیت نرم‌افزار (software quality assurance) محسوب می‌گردد. فعالیت تست تلاش می‌کند تا اغلب خطاهای نرم‌افزار را قبل از تحویل به مشتری کشف کند. این کار با ساخت موارد تست (test case) به عنوان مجموعه ورودی‌های نرم‌افزار انجام می‌گردد. ورودی‌های مورد نظر مشتری، که وقتی نرم‌افزار آن‌ها را اجرا می‌کند، نرم‌افزار نتواند خروجی‌های مورد انتظار مشتری را برآورده‌سازد. که این ورودی‌ها ثابت می‌کنند نرم‌افزار داری خطاست که منجر به این می‌شود نرم‌افزار در ارائه خروجی مورد انتظار مشتری در مواجه با ورودی مورد نظر مشتری ناتوان گردد. بنابراین می‌توان گفت هدف تست، ساخت ورودی‌هایی است که خطاهای برنامه را آشکار سازد. نتیجه اینکه فعالیت تست در جهت تضمین کیفیت نرم‌افزار (software quality assurance) قدم برمی‌دارد.

توجه: دقت کنید که تست نرم‌افزار با هدف کشف خطاهای برنامه قبل از تحویل نرم‌فزار به مشتری انجام می‌گردد و نمی‌توان مدعی بود که نرم‌افزار پس از تحویل به مشتری به طور کامل عاری از خطا است. به بیان دیگر عمل تست، صحت و درستی کلی نرم‌افزار را بررسی نمی‌کند، بلکه صحت و درستی نرم‌افزار را بر اساس حالات تست انجام شده مشخص می‌کند. زیرا ممکن است حالت‌هایی موجود باشند که منجر به ایجاد خطا گردند در حالی که پس از تست نرم‌افزار همچنان از نگاه سازنده پنهان مانده‌اند.

توجه: عمل تست، مشابه صید ماهی از دریاچه است، تست‌کننده، همان ماهی‌گیر و دریاچه همان نرم‌افزار است و ماهی‌ها همان خطاهای برنامه. اگر ماهی‌گیر نتواند از دریاچه ماهی صید کند، نمی‌توان گفت که دریاچه ماهی ندارد! اگر تست‌کننده نتواند از نرم‌افزار خطا صید کند، نمی‌توان گفت که نرم‌افزار خطا ندارد!

توجه: معمولاً حدود ۴۰ درصد فعالیت‌های کل پروژه روی تست متمرکز می‌گردد.

توجه: برنامه‌ریزی برای تست باید مدت‌ها قبل از آنکه مرحله تست شروع شود، انجام گیرد.به عنوان مثال، لیست نیازمندی‌های مشتری، برای استفاده در فعالیت تست به عنوان چک‌لیست باید از مدت‌ها قبل آماده گردد.

توجه: تست تمام و کمال و اجرای تمامی ترکیبات مسیرهای نرم‌افزار در ضمن تست امکان‌پذیر نیست.

توجه: فرآیند تولید نرم‌افزار شامل مراحل تحلیل، طراحی و پیاده‌سازی عملی «سازنده» است، زیرا سعی بر ساخت نرم‌افزار دارد درحالی‌که تست نرم‌افزار عملی «مخرب» است، زیرا سعی دارد درستی آنچه که در مراحل تحلیل، طراحی و پیاده‌سازی ساخته شده‌است را نقض کند و خطاهای آن را آشکار سازد. از نظر روان‌شناختی، اگر سازنده نرم‌افزار، تست را انجام دهد، سعی می‌کند درستی عملکرد نرم‌افزار را نشان دهد تا اینکه خطاهای آن را آشکار سازد! بنابراین تست، مستلزم آن است که سازنده دید پیش‌داوری خود، مبنی بر درستی نرم‌افزار را رها کند و نرم‌افزار را آنگونه تست کند که باید تست گردد.

توجه: سازنده نرم‌افزار همیشه مسئول تست برنامه می‌باشد تا از صحت عملکرد آن اطمینان حاصل نماید. همچنین برای پروژه‌های بزرگ‌تر یک «**گروه تست مستقل**[1]» وظیفه تست مجدد را بر عهده می‌گیرد.

صحت و اعتبارسنجی

تست یک محصول نرم‌افزاری از دو دیدگاه باید مورد بررسی قرار گیرد:

۱- صحت (verificaion)

این دیدگاه به دیدگاه صحت معروف است که در اینجا روند ساخت یا درستی ساخت در سطح هر یک از مولفه‌ها مستقل از مولفه‌های دیگر برنامه مورد ارزیابی قرار می‌گیرد.

در صحت، **درستی عملکرد یا روند اجرای** یک بخش، یک جزء یا یک مولفه از نرم‌افزار مستقل از سایر مولفه‌های دیگر برنامه مورد ارزیابی قرار می‌گیرد.

در این دیدگاه پرسش زیر مطرح است:

«آیا محصول را درست ساخته‌ایم؟[2]»

توجه: اصل صحت (verificaion) توسط روش تست جعبه سفید و تکنیک‌های مرتبط با آن محقق می‌گردد.

توجه: روش تست جعبه سفید و تکنیک‌های مرتبط با آن جلوتر شرح داده خواهد شد.

۲- اعتبارسنجی (validation)

این دیدگاه به دیدگاه اعتبارسنجی معروف است که در اینجا روند کار یا درستی کار در سطح اجتماع مولفه‌هایی از برنامه یا کل مولفه‌های برنامه در کنار یکدیگر مورد ارزیابی قرار می‌گیرد. تا مشخص شود نرم‌افزاری که ایجاد شده است منطبق بر نیازمندی‌های مشتری است یا خیر. به عبارت دیگر مشخص شود که آیا نرم‌افزار ایجاد شده بر اساس ورودی‌های مورد نظر مشتری، خروجی‌های مورد انتظار مشتری را برآورده می‌سازد یا خیر.

در اعتبارسنجی، **درستی عملکرد یا روند اجرای** چندین مولفه در کنار یکدیگر و یا کل مولفه‌های نرم‌افزار ایجاد شده در کنار یکدیگر به همراه نقاط اتصال مابین آن‌ها مورد ارزیابی قرار می‌گیرد.

در این دیدگاه پرسش زیر مطرح است:

«آیا محصول درستی ساخته‌ایم؟[3]»

توجه: اصل اعتبارسنجی(validation) توسط روش تست جعبه سیاه و تکنیک‌های مرتبط با آن محقق می‌گردد.

---

[1] ITG : Independent Testing Group
[2] Are we building the product right?
[3] Are we building the right product?

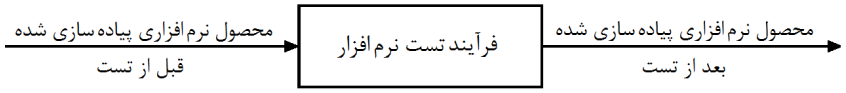
توجه: روش تست جعبه سیاه و تکنیک‌های مرتبط با آن جلوتر شرح داده خواهد شد.

تست نرم‌افزار

تست نرم‌افزار نظامی است یکپارچه، شامل فرآیندها، روش‌ها و تکنیک‌ها که منجر به ایجاد نرم‌افزاری با حداقل خطا می‌گردد. بگونه‌ای که نرم‌افزار بتواند براساس ورودی‌های مورد نظر مشتری، خروجی‌های مورد انتظار مشتری را برآورده سازد.

فرآیند تست نرم‌افزار

هر پروژه‌ی نرم‌افزاری، چه بزرگ و چه کوچک پس از پیاده‌سازی مراحلی را طی می‌نماید که در طی آن محصول نرم‌افزاری پیاده‌سازی شده به یک محصول نرم‌افزاری با حداقل خطا تبدیل می گردد. الگو و قالبی که چگونگی طی مراحل مختلف تست نرم‌افزار را تعریف می‌نماید، اصطلاحاً **فرآیند تست نرم‌افزار** نامیده می‌شود. شکل زیر فرآیند فعالیت تست نرم‌افزار و ورودی و خروجی آن را نشان می‌دهد:



همانطور که ملاحظه می‌نمایید، ورودی فرآیند تست نرم‌افزار، محصول نرم‌افزاری پیاده‌سازی شده قبل از تست و خروجی آن، یک محصول نرم‌افزاری پیاده‌سازی شده بعد از تست و به تبع با حداقل خطا است.

فرآیند تست نرم‌افزار در میان دیگر فعالیت‌های مهندسی نرم‌افزار، اغلب بیشترین کار را به خود اختصاص می‌دهد. اگر بدون برنامه‌ریزی اجرا شود، زمان هدر می‌رود، کار بیهوده صرف می‌شود و خطاها کشف نشده باقی می‌مانند. بنابراین منطقی به نظر می‌رسد که یک راهبرد سیستماتیک برای فرآیند تست نرم‌افزار وضع شود، زیرا اگر مهندس نرم‌افزار خطاها را نیابد، مشتری خواهد یافت!

به طور کلی فرآیند تست نرم‌افزار از سطح مؤلفه شروع شده و به سمت تست یکپارچه کل سیستم کامپیوتری ادامه می‌یابد.

روش‌های تست نرم افزار

روش‌های ارزیابی برای تست نرم‌افزار را «روش‌های تست نرم‌افزار» می‌گوییم و به دو شکل «**روش تست جعبه سفید**» و «**روش تست جعبه سیاه**» می‌باشد.

توجه: روش تست جعبه سفید و روش تست جعبه سیاه جلوتر شرح داده خواهد شد.

تکنیک‌های تست نرم‌افزار

تکنیک‌های ارزیابی برای تست نرم‌افزار را «تکنیک‌های تست نرم‌افزار» می‌گوییم و به دو شکل «**تکنیک‌های روش تست جعبه سفید**» و «**تکنیک‌های روش تست جعبه سیاه**» می‌باشد.

توجه: تکنیک‌های روش تست جعبه سفید و تکنیک‌های روش تست جعبه سیاه جلوتر شرح داده خواهد شد.

مراحل فرآیند تست نرم‌افزار

به طور کلی مراحل مرتبط با فرآیند تست نرم‌افزار صرف‌نظر از اندازه، پیچیدگی پروژه و زمینه‌ی کاربردی آن و مستقل از متدولوژی ساخت‌یافته و شیءگرا به چهار مرحله تست واحد، تست جامعیت (تست یکپارچگی یا تست تدریجی)، تست اعتبارسنجی و تست سیستم تقسیم می‌شود، به بیان دیگر مراحل در هر دو دسته‌ی متدولوژی ساخت‌یافته و شیءگرا همین‌ها خواهند بود، اما کارهایی که در هر فعالیت در متدولوژی ساخت‌یافته و شیءگرا انجام می‌شود شباهت‌ها و تفاوت‌هایی خواهد داشت.

در ادامه مراحل فرآیند تست نرم‌افزار بیان خواهد شد:

۱- تست واحد[1]

تست واحد، روند ساخت یا درستی ساخت را در سطح هر یک ازمولفه‌ها مستقل از مولفه‌های دیگر برنامه مورد تست و ارزیابی قرار می‌دهد.

به بیان دیگر تست واحد **صحت عملکرد یا روند اجرای** یک بخش، یک جزء یا یک مولفه از نرم‌افزار را مستقل از سایر مولفه‌های دیگر برنامه مورد تست و ارزیابی قرار می‌دهد.

توجه: تست واحد جهت محقق کردن اصل **صحت (verificaion)**، روش تست جعبه سفید و تکنیک‌های مرتبط با آنرا مورد استفاده قرار می‌دهد.

روش تست جعبه سفید[2]

روش تست جعبه سفید، روند ساخت یا درستی ساخت را در سطح هریک ازمولفه‌ها مستقل از مولفه‌های دیگر برنامه مورد تست و ارزیابی قرار می‌دهد.

به بیان دیگر روش تست جعبه سفید **صحت عملکرد** یا **روند اجرای یک بخش**، یک جزء یا یک مولفه از نرم‌افزار را مستقل از سایر مولفه‌های دیگر برنامه مورد تست و ارزیابی قرار می‌دهد.

توجه: به روش تست جعبه سفید، روش تست «جعبه شفاف یا جعبه شیشه‌ای[3]» نیز گفته می‌شود.

تکنیک‌های روش تست جعبه سفید

تکنیک‌های روش تست جعبه سفید شامل «**تکنیک تست مسیر پایه**»، «**تکنیک تست ساختار کنترلی شرط**»، «**تکنیک تست ساختار کنترلی حلقه**» و «**تکنیک تست جریان داده**» می‌باشد که در ادامه به بررسی آن‌ها می‌پردازیم.

۱- تکنیک تست مسیر پایه[4]

تکنیک تست مسیر پایه، طراحِ حالات تست را از میزان پیچیدگی منطقی مولفه‌ها آگاه می‌سازد

---

[1] Unit Testing
[2] White-Box Testing
[3] Glass Box
[4] Basic Path Testing

و کمک می‌کند تا مسیرهای پایه‌ای اجرایی، شناسایی شوند. عمل تست بر روی مجموعه مسیرهای پایه تضمین می‌کند که هر دستور یک مولفه حداقل یک بار در حین تست اجرا گردد. به این ترتیب خطاهای موجود در دستورات مولفه مورد نظر آشکار خواهد شد.

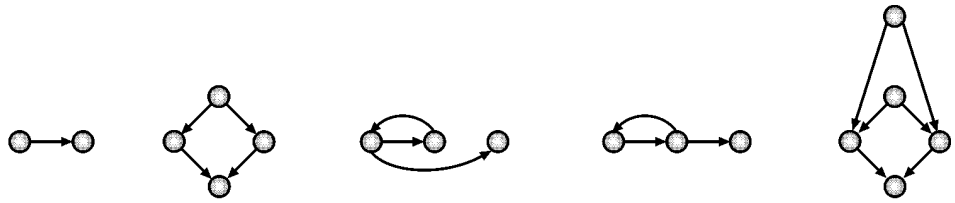
توجه: هر مسیر پایه شامل تعدادی از دستورات برنامه است.

توجه: نام مسیرهایی که توسط تکنیک تست مسیر پایه معرفی می‌شود، مسیرهای مستقل (independent path) است.

نشانه‌گذاری گراف جریان[1]

برای به‌دست آوردن مسیرهای پایه(مستقل) مولفه مورد نظر، ابتدا باید گرافی از مولفه مورد نظر را تولید نمود. این گراف باید جریان منطقی مولفه مورد نظر را با استفاده از یک سری از نشانه‌گذاری‌ها، نمایش دهد. شکل زیر انواع نمادهای گراف را نشان می‌دهد:

توجه: در این شکل‌ها هر دایره بیانگر قسمتی از کد برنامه است.

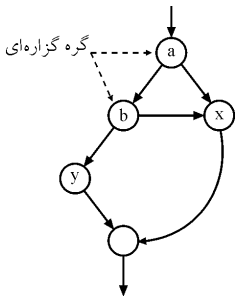


گراف جریان همراه شرط‌های مرکب

هنگام مواجه با شرط‌های مرکب در یک مولفه، تولید گراف جریان قدری پیچیده‌تر می‌شود. شرط مرکب زمانی رخ می‌دهد که یک یا چند عملگر بولی (AND، OR، NOR و غیره) در یک دستور شرطی وجود داشته باشند. در ادامه گراف جریان شرط‌های AND و OR نشان داده شده است:

گراف جریان شامل عملگر OR

```
If a OR b then
      procedure x
   else
      procedure y
   END IF
```



[1] Flow Graph Notation

توجه: دقت داشته باشید که برای هر یک از شرایط a و b در دستور If a OR b یک گره جداگانه ایجاد می‌شود. اگر مقدار متغیر a درست باشد، مستقل از اینکه مقدار متغیرb چه باشد کنترل برنامه به x منتقل می‌شود. همچنین اگر مقدار متغیر a نادرست باشد حال بسته به اینکه مقدار متغیر b چه باشد کنترل برنامه به x یا y منتقل می‌شود. اگر مقدار متغیر b درست باشد کنترل برنامه به x و اگر مقدار متغیر b نادرست باشد کنترل برنامه به y منتقل می‌گردد.

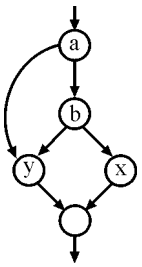
گره گزاره‌ای

هر گره که حاوی یک شرط باشد، گره گزاره‌ای خوانده می‌شود و با دو یا چند پیکان که از آن خارج می‌گردند، مشخص می‌شود.

مثال: در مثال فوق گره‌های a و b گره‌های گزاره‌ای محسوب می‌گردند.

گراف کنترل جریان شامل عملگر AND

```
If a AND b then
        procedure x
    else
        procedure y
END IF
```



توجه: دقت داشته باشید که برای هر یک از شرایط a و b در دستور If a AND b یک گره جداگانه ایجاد می‌شود. اگر مقدار متغیر a نادرست باشد مستقل از اینکه مقدار متغیرb چه باشد کنترل برنامه به y منتقل می‌شود. همچنین اگر مقدار متغیر a درست باشد حال بسته به اینکه مقدار متغیر b چه باشد کنترل برنامه به x یا y منتقل می‌شود. اگر مقدار متغیر b درست باشد کنترل برنامه به x و اگر مقدار متغیر b نادرست باشد کنترل برنامه به y منتقل می‌گردد.

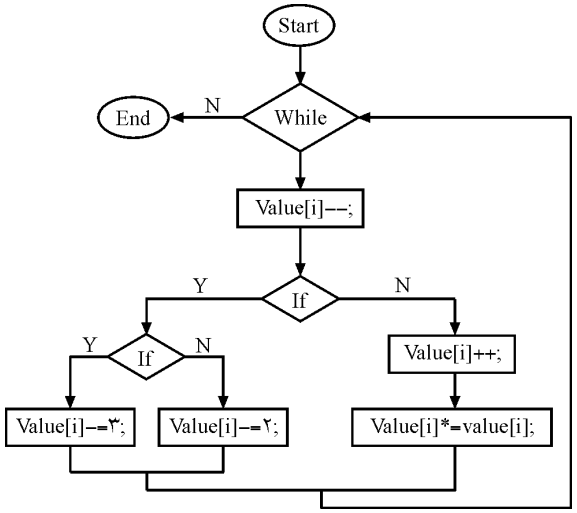
مثال: در مثال فوق گره‌های **a** و **b** گره‌های گزاره‌ای محسوب می‌گردند.

مثال: قطعه کد زیر مربوط به یک مولفه مورد نظر را در نظر بگیرید. می‌خواهیم روش تست جعبه سفید را برای آن انجام دهیم:

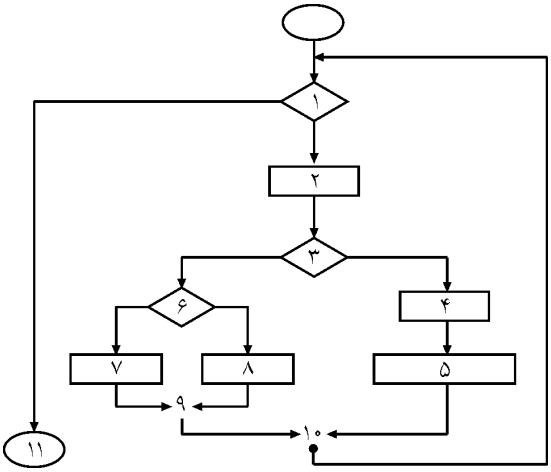
```
while (value [i]>=5)}
    value[i]--;
    if (value[i]>=10){
        if (value[i]>=15)
            value[i]-=3;
        else
            value[i]-=2;
        }
```

```
    else
        {
        value[i]++;
        value*=value[i];
        }
}
```

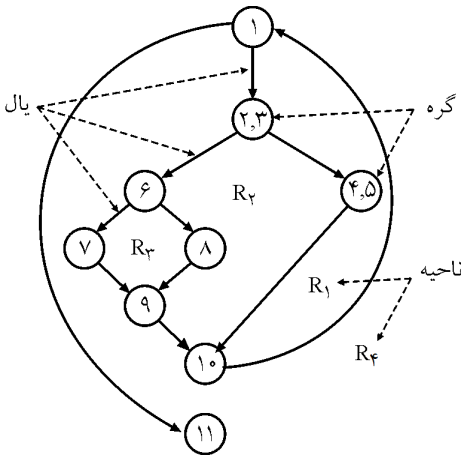
در ابتدا باید فلوچارت قطعه کد فوق را ترسیم نماییم:



حال فلوچارت حاصل را مطابق شکل زیر با اعداد نام‌گذاری می‌کنیم. به طوری که تمامی مستطیل‌ها و لوزی‌ها به همراه تمامی خروجی‌های آن‌ها شماره‌گذاری شوند:



از روی فلوچارت عددگذاری شده، می‌توان گرافی مطابق شکل مقابل ایجاد نمود. این نمودار به «گراف جریان» یا «گراف برنامه» موسوم است.



حال نوبت به تعیین مسیرهای پایه (مستقل) در گراف جریان می‌رسد. هر مسیر پایه، مسیر مستقلی است که از ابتدای گراف شروع شده، سپس یک ناحیه از گراف را دور زده و به گره پایانی گراف ختم می‌شود. از این رو ابتدا باید نواحی موجود در گراف جریان را به دست آورد.

تعداد ناحیه = تعداد یال − تعداد گره + ۲

$$V(G) = E - N + 2$$

پس در این گراف خواهیم داشت:

$$\text{تعداد ناحیه} \Rightarrow V(G) = ۱۱ - ۹ + ۲ = ۴$$

این چهار ناحیه توسط R_1 تا R_4 در گراف جریان مشخص شده‌اند.

حال باید به دنبال مسیرهای پایه‌ای (مستقل) باشیم که شرایط زیر را دارند:

۱- از نقطه آغازین گراف (گره شماره ۱) شروع شود.

۲- از یکی از چهار ناحیه عبور کند.

۳- به گره انتهایی (گره شماره ۱۱) ختم شود.

توجه: هر مسیر پایه (مستقل) جدید باید حتماً هم دارای گره‌های جدیدتر نسبت به مسیرهای پایه‌ای به دست آمده قبلی باشد و هم اینکه ناحیه جدیدی را معرفی کند. براین اساس مسیرهای پایه‌ای (مستقل) زیر به دست خواهند آمد:

مسیر شماره یک : ۱ → ۱۱

مسیر شماره دو : ۱ → ۲ → ۳ → ۴ → ۵ → ۱۰ → ۱ → ۱۱

مسیر شماره سه : ۱ → ۲ → ۳ → ۶ → ۸ → ۹ → ۱۰ → ۱ → ۱۱

مسیر شماره چهار: ۱ → ۲ → ۳ → ۶ → ۷ → ۹ → ۱۰ → ۱ → ۱۱

توجه: دقت کنید که هر مسیر جدید یک یال جدید معرفی می‌کند. مسیر زیر:

۱ → ۲ → ۳ → ۴ → ۵ → ۱۰ → ۱ → ۲ → ۳ → ۶ → ۸ → ۹ → ۱۰ → ۱ → ۱۱

به عنوان مسیر پایه (مستقل) در نظر گرفته نمی‌شود، زیرا صرفاً تلفیقی از مسیرهای مشخص شده از قبل است و از هیچ یال جدیدی عبور نمی‌کند.

حال اگر بتوان مراحل تست را طوری طراحی کرد که اجرای هر یک از این چهار مسیر، حتمی باشد آنگاه به طور قطع می‌توان ادعا کرد که هر دستور از مولفه مورد نظر حداقل یک بار اجرا خواهد شد و هر یک از دستورات شرطی در هر یک از حالات درست یا غلط اجرا می‌شود.

توجه: مجموعه پایه منحصر به فرد نیست. در حقیقت، چند مجموعه پایه متفاوت را می‌توان برای یک برنامه به دست آورد.

توجه: کلیه مسیرهای پایه (مستقل) به دست آمده «مجموعه پایه» نامیده می‌شود.

برای مثال مسیرهای ۱ تا ۴ در یک مجموعه پایه قرار می‌گیرند.

توجه: مساحت‌های محصور شده توسط یال‌ها و گره‌ها را «ناحیه» می‌نامند. هنگام شمارش نواحی، مساحت خارج از گراف نیز به عنوان یک ناحیه در نظر گرفته می‌شود (مانند ناحیه R4).

پیچیدگی سیکلوماتیک[1] یا چرخشی یا دورانی

پیچیدگی سیکلوماتیک یک معیار نرم‌افزاری است که میزان کمی از پیچیدگی منطقی یک مولفه نرم‌افزاری را ارایه می‌دهد. در تکنیک تست مسیر پایه، مقدار محاسبه شده برای پیچیدگی سیکلوماتیک، همان تعداد مسیرهای پایه (مستقل) یا اندازه «مجموعه پایه» یک مولفه می‌باشد.

سؤال: چگونه بدانیم که به دنبال چند مسیر باید باشیم؟

پاسخ را در محاسبه پیچیدگی سیکلوماتیک می‌توان جستجو کرد.

پیچیدگی سیکلوماتیک، ریشه در نظریه گراف‌ها دارد و یک معیار نرم‌افزاری سودمند را فراهم می‌آورد. پیچیدگی سیکلوماتیک به یکی از سه روش زیر محاسبه می‌شود:

۱- تعداد نواحی گراف جریان، متناظر با پیچیدگی سیکلوماتیک است.

۲- پیچیدگی سیکلوماتیک، V(G)، برای یک گراف جریان به صورت زیر تعریف می‌شود:

$$V(G) = E - N + ۲$$

که در آن E تعداد «یال‌ها» و N تعداد «گره‌های» موجود در گراف جریان است.

توجه: خطی که از یک گره به گره بعدی به مشکل مستقیم رسم شود، یک یال نامیده می‌شود.

توجه: به دنباله‌ای از یال‌های متوالی، یک مسیر گفته می‌شود.

۳- پیچیدگی سیکلوماتیک، V(G) برای یک گراف جریان به صورت زیر تعریف می‌شود:

$$V(G) = P + ۱$$

که در آن P تعداد «گره‌های گزاره‌ای» موجود در گراف جریان است.

حال یک بار دیگر به گراف جریان مثال قبل نگاه کنید، مشاهده می‌کنید که پیچیدگی

[1] Cyclomatic Complexity

سیکلوماتیک را می‌توانید با به کارگیری هر یک از الگوریتم‌های بالا محاسبه کنید:

۱- گراف جریان چهار ناحیه دارد. $(R_۴, R_۳, R_۲, R_۱)$

۲- $V(G) = ۱۱ - ۹ + ۲ = ۴$

۳- $V(G) = ۳ + ۱ = ۴$

بنابراین، پیچیدگی سیکلوماتیک گراف جریان مثال قبل برابر ۴ است.

توجه: مقدار $V(G)$ یک حد نهایی برای تعداد مسیرهای تشکیل‌دهنده، «مجموعه پایه» ارایه می‌کند و در نتیجه یک حد نهایی برای تعداد تست‌هایی ارایه می‌کند که باید طراحی و اجرا شوند تا تضمین شود که کلیه دستورات مولفه مورد نظر تحت پوشش قرار دارند.

توجه: تکنیک تست مسیر پایه، جهت کشف مسیرهای پایه (مستقل) در مولفه مورد نظر مورد استفاده قرار می‌گیرد.

تکنیک تست مسیر پایه، یکی از چند تکنیک مربوط به روش تست جعبه سفید است. گرچه تکنیک تست مسیرهای پایه، مشخص کننده مسیرهای پایه در مولفه مورد نظر است، اما به تنهایی کافی نیست. در ادامه، برخی دیگر از تکنیک‌های مکمل تکنیک تست مسیر پایه را مورد بحث قرار می‌دهیم، این تکنیک‌ها در کنار تکنیک تست مسیر پایه موجب افزایش کیفیت روش تست جعبه سفید می‌گردد.

۲- تکنیک تست ساختار کنترلی شرط

تکنیک تست ساختار کنترلی شرط، یکی از تکنیک‌های روش تست جعبه سفید است که انحصاراً بر اعتبار ساختار شرط تاکید دارد.

شرط‌ها می‌توانند به صورت‌های زیر بیان شوند:

الف) یک متغیر ساده، بولی یا عبارتی رابطه‌ای که عملگر NOT می‌تواند قبل از آن برای منفی کردنش مورد استفاده قرار گیرد.

ب) یک عبارت رابطه‌ای به صورت زیر:

$E_۱$ (عملگر رابطه‌ای) $E_۲$

که $E_۱$ و $E_۲$ عبارات محاسباتی و عملگر رابطه‌ای نیز یکی از عملگرهای $(\neq, =, \leq, \geq, <, >)$ می‌باشد.

ج) یک عبارت مرکب شامل شرط‌های ساده، عملگرهای بولی و پرانتز.

اگر شرطی نادرست باشد، در آن صورت، حداقل یک جزء از شرط نادرست است، از این رو، انواع خطاها در یک شرط شامل موارد زیر است:

الف) خطای عملگر بولی (نادرست، جاافتاده، اضافی)

ب) خطای متغیر بولی

ج) خطای پرانتز

د) خطای عملگر رابطه‌ای

هـ) خطای عبارات محاسباتی

توجه: تکنیک تست ساختار کنترلی شرط، جهت حصول اطمینان از نبود خطا در ساختار شرط‌ها در مولفه مورد نظر مورد استفاده قرار می‌گیرد.

۳- تکنیک تست ساختار کنترلی حلقه[1]

تکنیک تست ساختار کنترلی حلقه، یکی از تکنیک‌های روش تست جعبه سفید است که انحصاراً بر اعتبار ساختار حلقه تاکید دارد.

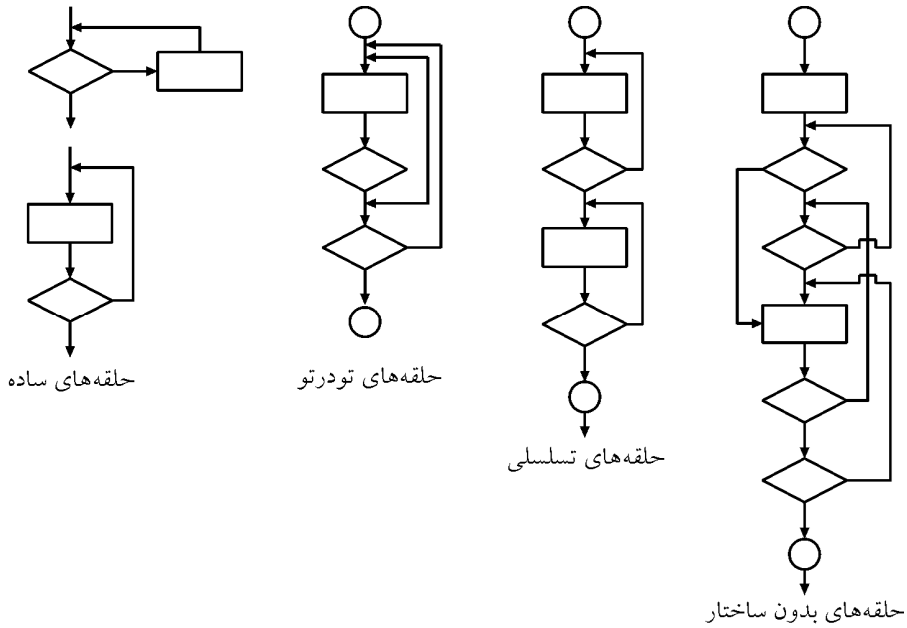
به طور کلی ساختار حلقه‌ها به چهار طبقه زیر تقسیم می‌گردد:

الف) حلقه‌های ساده

ب) حلقه‌های تودرتو یا متداخل

ج) حلقه‌های تسلسلی یا الحاقی

د) حلقه‌های بدون ساختار



الف) حلقه‌های ساده

برای یک حلقه ساده با حداکثر n بار تکرار، تست‌های زیر را می‌توان انجام داد:

۱- عدم اجرای حلقه برای کنترل قطعه کد درون حلقه.

۲- یک بار اجرای حلقه (ساختار حلقه یک بار اجرا شود.)

[1] Loop Testing

۳- دو بار اجرای حلقه (ساختار حلقه دو بار اجرا شود.)
۴- k بار اجرای حلقه (تا زمانی که شرط k < n برقرار باشد.)
۵- n-1، n و n+1 اجرای ساختار حلقه برای کنترل خروج موفق از حلقه.

ب) حلقه‌های تودرتو یا متداخل

۱- از داخلی‌ترین حلقه شروع کنید. سایر حلقه‌های بیرونی را در حداقل مقدار تکرار خود قرار دهید.
۲- تست حلقه ساده را برای داخلی‌ترین حلقه با مقادیر معمولی تکرار اجرا کنید.
۳- به سمت حلقه‌های بیرونی حرکت کنید و همین روند را برای حلقه بعدی تکرار کنید.
۴- این روند را تا تست بیرونی ترین حلقه ادامه دهید.

ج) حلقه‌های تسلسلی یا الحاقی

در صورتی که مستقل از یکدیگر باشند می‌توان از روش تست حلقه ساده استفاده کرد ولی اگر حلقه‌ها وابسته باشند، یعنی مقدار اولیه یک حلقه به حلقه دیگر وابسته باشد، باید از روش تست حلقه‌های تودرتو یا متداخل استفاده کرد.

د) حلقه‌های بدون ساختار

علت وجود حلقه‌های بدون ساختار استفاده از دستوراتی مثل دستور go to است. حلقه‌های بدون ساختار برای اجرای تست باید مجدداً طراحی شوند زیرا این نوع حلقه‌ها جزو اصول برنامه نویسی ساخت‌یافته نیستند.

توجه: تکنیک تست ساختار کنترلی حلقه، جهت حصول اطمینان از نبود خطا در ساختار حلقه‌ها در مولفه مورد نظر مورد استفاده قرار می‌گیرد.

۴-تکنیک تست جریان داده[1]

تکنیک تست جریان داده به کنترل خطای متغیرها در هنگام تعریف و استفاده از آن‌ها می‌پردازد. بنابراین مسیرهای این روش تست می‌بایست بر مبنای محل تعریف (definition) متغیر مورد نظر و محل استفاده (use) متغیر مورد نظر در مولفه مورد تست تعریف گردد. زیرا متغیرها در قسمتی از مولفه مورد نظر تعریف می‌شوند و در بخشی دیگر از همان مولفه مورد استفاده قرار می‌گیرند. برای‌مثال متغیر x در گره اول مولفه‌ای از برنامه به شکل int x=2 تعریف می‌شود و در گره سوم همان مولفه در یک عبارت شرطی به شکل x<y مورد استفاده قرار می‌گیرد. بنابراین برای ایجاد مسیرهای تست جریان داده، کافیست مسیرهایی انتخاب شود که به ازای هر گره تعریف یک متغیر، گره استفاده از متغیر مورد نظر را نیز در خود جای دهد. به بیان دیگر مسیر تست جریان داده، می‌بایست گره تعریف متغیر و استفاده از متغیر مورد نظر را در مسیر پیمایش خود ملاقات

[1] Data Flow Testing

کند. ایده‌ی اصلی تست جریان داده بر این پایه استوار است که خطاهای مربوط به متغیرها، یا در زمان تعریف یا در زمان استفاده از متغیرها رخ می‌دهد. و تست جریان داده هر دو حالت این خطاها را مورد تست قرار می‌دهد. از آنجا که تکنیک تست جریان داده از تعریف (definition) و استفاده (use) متغیر برای ساخت مسیر تست جریان داده بهره می‌برد، به آن تکنیک DU Testing نیز گفته می‌شود.

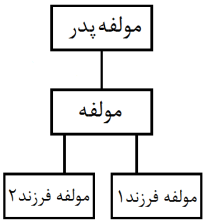
توجه: تکنیک تست جریان داده، جهت حصول اطمینان از نبود خطا در ساختار جریان داده‌ها، همچون کلیه متغیرهای مولفه، ساختمان داده‌های مولفه، ورودی‌ها و خروجی‌های مولفه، در مولفه مورد نظر مورد استفاده قرار می‌گیرد.

توجه: روش تست جعبه سفید و تکنیک‌های مرتبط با آن بر اساس لیست نیازمندی‌های وظیفه‌مندی و غیروظیفه‌مندی موجود در فعالیت ارتباطات، مدل تحلیل و مدل طراحی مولفه متناظر موجود در مدل طراحی بخش طراحی مولفه انجام می‌گردد.

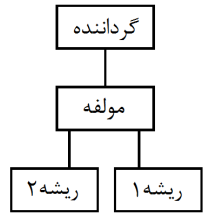
تا به اینجا روش تست جعبه سفید و تکنیک‌های مرتبط با آن یعنی تکنیک تست مسیر پایه، تکنیک تست ساختار کنترلی شرط، تکنیک تست ساختار کنترلی حلقه و تکنیک تست جریان داده را جهت تحقق اصل **صحت** (verificaion) در تست واحد یک مولفه مورد نظر مورد بررسی قرار دادیم. در ادامه به بررسی دقیق‌تر تست واحد می‌پردازیم.

از آنجاکه در تست واحد، یک مولفه از نرم‌افزار، مستقل از مولفه‌های دیگر مورد تست قرار می‌گیرد، بنابراین برای تست یک مولفه نیاز به ایجاد مولفه گرداننده (Driver) و مولفه ریشه (Stub) می‌باشد.

مثال: در یک معماری فراخوانی و بازگشت، مولفه پدر، پدر مولفه مورد نظر است، که مولفه مورد نظر را فراخوانی می‌کند. همچنین مولفه‌های فرزند۱ و فرزند۲ به عنوان مولفه‌های فرزند مولفه مورد نظر محسوب می‌گردند. شکل مقابل گویای مطلب است:



حال اگر قرار باشد، مولفه مورد نظر به تنهایی مورد تست قرار گیرد، باید جایگزین مناسبی برای مولفه پدر و مولفه‌های فرزند در نظر گرفته شود. شکل مقابل گویای مطلب است:



گرداننده

مولفه گرداننده، جای مولفه پدر قرار می‌گیرد که حالات تست را دریافت کرده و به «مولفه مورد تست» پاس می‌دهد. سپس نتایج خروجی توسط مولفه مورد تست به خروجی موردنظر منتقل می‌گردد.

ریشه

مولفه ریشه، جایگزین مؤلفه‌های فرزند می‌شود که توسط «مولفه مورد تست»، فراخوانی می گردد.

توجه: مولفه‌های گرداننده و ریشه نسبت به مولفه‌های اصلی پدر و فرزند، باید خلاصه‌تر باشند، زیرا فقط موظف به تامین نیازهای مولفه مورد تست هستند و نه تامین نیازهای خودشان.

توجه: تولید مولفه‌های گرداننده و ریشه به دلیل هزینه و زمان صرف شده و عدم تحویل با محصول نهایی سربار دارد. اگر این مولفه‌ها خلاصه باشد سربار کم خواهد بود. به عبارت دیگر از آنجا که مؤلفه‌های گرداننده و ریشه در محصول نهایی کاربرد ندارند، و فقط به منظور تست واحد ایجاد گردیده‌اند، هزینه و زمان صرف شده برای ساخت آن‌ها به عنوان سربار سیستم تلقی می‌شود، حال اگر پیچیدگی آنها بالا باشد، سربار سیستم بالاتر هم خواهد رفت.

توجه: بسیاری از مؤلفه‌ها (واحدها) را نمی‌توان به طور کامل با نرم‌افزارهای گرداننده و ریشه، تست واحد نمود. در چنین موارد، تست کامل تا مرحله تست جامعیت به تعویق می‌افتد.

مثال: تست واحد مانند تست قطعات (مولفه‌های) یک خودرو توسط یک خودروساز به شکل مستقل به کمک گرداننده و ریشه‌های مرتبط با آن است.

پس از تست واحد نوبت به تست جامعیت یا یکپارچگی می‌رسد، بنابراین در ادامه به بررسی تست جامعیت یا یکپارچگی می‌پردازیم.

۲- تست جامعیت یا یکپارچه‌سازی [1]

تست جامعیت یا یکپارچه‌سازی، در پایان تست واحد آغاز می‌شود، یعنی زمانی‌که همه مولفه‌ها توسط تست واحد یک به یک مورد تست و ارزیابی قرار گرفتند و خطاهای مولفه‌ها برطرف شدند.

تست جامعیت یا یکپارچه‌سازی، روند کار یا درستی کار را در سطح اجتماع مولفه‌هایی از برنامه یا کل مولفه‌های برنامه در کنار یکدیگر مورد تست و ارزیابی قرار می‌دهد. تا مشخص شود نرم‌افزاری که ایجاد شده است منطبق بر نیازمندی‌های مشتری است یا خیر. به عبارت دیگر مشخص شود که آیا نرم‌افزار ایجاد شده بر اساس ورودی‌های مورد نظر مشتری، خروجی‌های مورد انتظار مشتری را برآورده می‌سازد یا خیر.

به بیان دیگر تست جامعیت یا یکپارچه‌سازی، اعتبارسنجی عملکرد یا روند اجرای چندین مولفه در کنار یکدیگر و یا کل مولفه‌های نرم‌افزار ایجاد شده در کنار یکدیگر به همراه **نقاط اتصال** مابین آن‌ها را مورد تست و ارزیابی قرار می‌دهد.

در یک نرم‌افزار ممکن است هر یک از مؤلفه‌های برنامه به تنهایی درست عمل کنند اما ترکیب آنها با یکدیگر به درستی عمل نکند. زیرا ممکن است به دلیل عدم تنظیم درست نقاط اتصال و

[1] Integration Testing

ارتباطی مابین مولفه‌ها، این مولفه‌ها قادر نباشند در کنار یکدیگر، کارکرد مورد انتظار را نشان دهند. بنابراین تست جامعیت، جهت تست تدریجی یکپارچه‌شدن مولفه‌های برنامه در کنار یکدیگر مورد استفاده قرار می‌گیرد.

توجه: منظور از نقاط اتصال مابین مولفه‌های مرتبط با هم، همان کانال یا محل تبادل داده مابین مولفه‌های مرتبط با هم هست، مانند فراخونی با مقدار یا فراخوانی با ارجاع. اگر این نقاط اتصال مابین مولفه‌های مرتبط با هم درست تنظیم نشوند ممکن است داده‌ها در گذر از این نقاط اتصال از بین بروند مانند حالتی که یک متغیر دوبایتی به یک متغیر یک بایتی در روش فراخوانی با مقدار پاس داده می‌شود، در واقع در این حالت بخشی از داده‌ها از دست رفته است. این نقاط اتصال به **واسط مولفه** نیز موسوم است.

توجه: تست جامعیت جهت محقق کردن اصل **اعتبارسنجی (validation)**، روش تست جعبه سیاه و تکنیک‌های مرتبط با آنرا مورد استفاده قرار می‌دهد.

روش تست جعبه سیاه[1]

روش **تست جعبه سیاه**، روند کار یا درستی کار را در سطح اجتماع مولفه‌هایی از برنامه یا کل مولفه‌های برنامه در کنار یکدیگر مورد تست و ارزیابی قرار می‌دهد. تا مشخص شود نرم‌افزاری که ایجاد شده است منطبق بر نیازمندی‌های مشتری است یا خیر. به عبارت دیگر مشخص شود که آیا نرم‌افزار ایجاد شده بر اساس ورودی‌های مورد نظر مشتری، خروجی‌های مورد انتظار مشتری را برآورده می‌سازد یا خیر.

به بیان دیگر تست جعبه سیاه اعتبارسنجی **عملکرد یا روند اجرای** چندین مولفه در کنار یکدیگر و یا کل مولفه‌های نرم‌افزار ایجاد شده در کنار یکدیگر به همراه نقاط اتصال مابین آن‌ها را مورد تست و ارزیابی قرار می‌دهد.

توجه: به روش تست جعبه سیاه، روش تست «رفتاری» نیز گفته می‌شود.

تکنیک‌های روش تست جعبه سیاه

تکنیک‌های روش تست جعبه سیاه شامل «**تکنیک تست بخش‌بندی معادل یا افراز هم‌ارزی**»، «**تکنیک تست تحلیل مقادیر مرزی**»، «**تکنیک تست تشابه**»، «**تکنیک تست آرایه متعامد**»، «**تکنیک تست مبتنی بر گراف**» و «**تکنیک تست مبتنی بر مدل**» می‌باشد که در ادامه به بررسی آن‌ها می‌پردازیم.

تکنیک تست بخش‌بندی معادل یا افراز هم‌ارزی[2]

در این روش دامنه داده‌های ورودی برنامه به دسته‌های متفاوتی تقسیم می‌شود و موارد تست بر اساس این دسته‌ها انتخاب می‌شوند. هدف اصلی استفاده از این روش، کم کردن‌تعداد موارد

[1] Black-Box Testing
[2] Equivalence Partitioning

تست است. بدین صورت که از یک مجموعه موارد تست یکسان، تنها یک مورد تست اعمال شود.

به بیان بهتر در بخش‌بندی معادل دامنه ورودی برنامه به رده‌های مختلفی تقسیم می‌شود، سپس یک مورد از هر رده به عنوان نماینده مورد تست واقع می‌شود و نتیجه به دست آمده به کلیه موارد آن رده تعمیم داده می‌شود.

توجه: در رده‌های داده‌ای معتبر باید این مساله تست گردد که آیا نرم‌افزار مورد تست، بر اساس ورودی‌های مورد نظر و معتبر مشتری، خروجی‌های مورد انتظار مشتری را برآورده می‌سازد یا خیر. همچنین در رده‌های داده‌ای نامعتبر باید این مساله تست گردد که آیا نرم‌افزار مورد تست، بر اساس ورودی‌های نامعتبر مشتری، پیغام مناسب را مبنی بر ورودی‌های نامعتبر مشتری گزارش می‌کند یا خیر.

مثال: همان‌طور که می‌دانید، قدر مطلق اعداد منفی، برابر مثبت همان اعداد است و قدر مطلق اعداد مثبت با خود آن عدد برابر است. می‌خواهیم قدر مطلق یک عدد را محاسبه کنیم. این عمل در یک تابع انجام می‌شود.

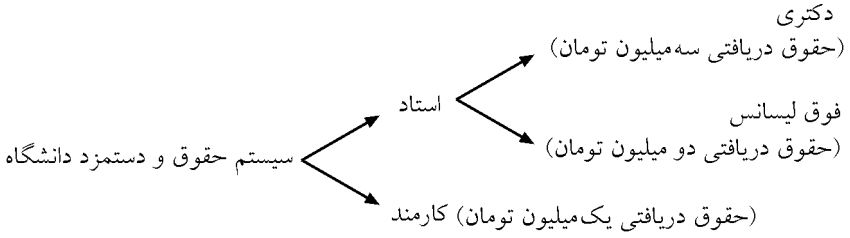
$f(x)=|a|$

فرض کنید بخواهیم تمامی اعداد موجود را در این تابع تست کنیم. خواهید دید که چه کار حجیمی ایجاد خواهد شد.

برای این کار کل مجموعه اعداد را به دو رده اعداد مثبت و منفی تقسیم می‌کنیم و از هر رده عددی را برمی‌گزینیم و تابع را با این دو عدد انتخابی تست کرده و نتیجه را برای همه اعداد رده مورد نظر تعمیم می‌دهیم.

مثال: سیستم حقوق و دستمزد یک دانشگاه را در نظر بگیرید:

اگر بخواهیم این سیستم را تعریف و مورد تست قرار دهیم کافی است تمامی افراد را به رده‌هایی مانند زیر تقسیم و حقوق هر یک از گروه‌ها را تعریف نماییم. در غیر این صورت می بایست برای هر فرد میزان حقوق تست می‌شد!



تکنیک تست تحلیل مقادیر مرزی[1]

معمولاً تعداد زیادی از خطاها در حالات مرزی دامنه ورودی رخ می‌دهند. بنابراین این روش

[1] Boundary Value Analysis

بر کنترل مقادیر مرزی در موارد تست تمرکز دارد. این روش مکمل روش بخش‌بندی معادل است و موارد تست را طوری انجام می‌دهد که شامل مقادیر مرزی هر دسته یا رده نیز باشد.

توجه: در رده‌های داده‌ای معتبر باید این مساله تست گردد که آیا نرم‌افزار مورد تست، بر اساس ورودی‌های مرزی مورد نظر و معتبر مشتری، خروجی‌های مورد انتظار مشتری را برآورده می‌سازد یا خیر. همچنین در رده‌های داده‌ای نامعتبر باید این مساله تست گردد که آیا نرم‌افزار مورد تست، بر اساس ورودی‌های برون مرزی نامعتبر مشتری، پیغام مناسب را مبنی بر ورودی‌های برون مرزی نامعتبر مشتری گزارش می‌کند یا خیر.

مثال: در ورود مقادیر نمرات دانشجویان در سیستم آموزش دانشگاه، مقادیر ۰ و ۲۰ مقادیر مرزی و مقادیر ۱- و ۲۱ مقادیر برون مرزی هستند.

مثال: در محاسبه قدر مطلق اعداد، عدد صفر، یک مقدار مرزی است.

مثال: در محاسبه حقوق استادی که دانشجوی دکتری است، مقدار حقوق وی یک مقدار مرزی بین فوق‌لیسانس و دکتری محسوب می‌شود.

تکنیک تست تشابه[1]

در برخی از مواقع قابلیت اطمینان نرم‌افزار بسیار مهم و حساس می‌باشد، مثل سیستم کنترل هواپیما و سیستم ترمز اتومبیل، در چنین کاربردهایی، غالباً از نرم‌افزارها و سخت‌افزارهای اضافی برای به حداقل رساندن امکان خطا استفاده می‌شود. جهت رسیدن به این هدف، تیم‌های مجزای مهندسی نرم‌افزار، نسخه‌های مستقلی از برنامه کاربردی را با خصوصیات مشابه تولید می‌کنند.

این نسخه‌های مستقل، مبنای تکنیک دیگری از تست جعبه سیاه می‌باشد که تست تشابه، مقایسه یا پشت به پشت نامیده می‌شود.

هنگامی که برنامه کاربردی در نسخه‌های مستقل و مشابه پیاده‌سازی شده باشد، موارد تستی که با استفاده از تکنیک‌های دیگر روش جعبه سیاه طراحی شدند (مثل بخش‌بندی معادل)، برای همه نسخه‌های نرم‌افزار به عنوان ورودی در نظر گرفته می‌شود.

اگر خروجی‌ها یکسان باشند، فرض می‌شود که همه پیاده‌سازی‌ها درست بوده‌اند. اگر به ازای ورودی خاصی، نسخه‌ای خروجی نامعتبری را تولید نماید آن‌گاه آن نسخه از دور رقابت خارج می‌گردد و این رویه تا انتخاب معتبرترین نسخه تکرار می‌گردد.

تکنیک تست آرایه متعامد[2]

بعضی از برنامه‌ها دامنه ورودی محدودی دارند، یعنی تعداد پارامترهای ورودی کمی داشته و علاوه بر آن مقداری که هر پارامتر می‌تواند دریافت کند محدود است. برای‌مثال ۲ پارامتر ورودی که هر یک ۲ مقدار مجزا می‌گیرند. در این حالت برای تست نرم‌افزار می‌توان تمام حالت‌های

[1] Comparison Testing
[2] Orthogonal Array Testing

ورودی برنامه را اعمال نمود و مرحله تست را به طور جامع انجام داد. پس ۴ = $۲^۲$ مورد تست متفاوت امکان پذیر است. در واقع هر چه تعداد پارامترها و محدوده مقادیر آنها بزرگتر باشد، انجام تست جامع مشکل‌تر خواهد بود. تکنیک تست آرایه متعامد، در مقایسه با تست جامع، با موارد تست کمتر، پوشش خوبی را جهت تست مناسب برنامه ارائه می‌کند.

تکنیک تست آرایه متعامد در یافتن خطاهای مرتبط با خطاهای ناحیه‌ای مفید می‌باشد. خطای ناحیه‌ای به گروهی از خطاها اطلاق می‌شود که به منطق نادرست در نقطه‌ای از یک برنامه مربوط می‌شوند.

مثال:

```
If a AND b then
      procedure x
   else
      procedure y
END IF
```

توجه: اگر مقدار متغیر a صفر باشد مستقل از اینکه مقدار متغیرb چه باشد کنترل برنامه به y منتقل می‌شود. همچنین اگر مقدار متغیر a یک باشد بسته به اینکه مقدار متغیر b چه باشد کنترل برنامه به x یا y منتقل می‌شود. اگر مقدار متغیر b صفر باشد کنترل برنامه به y و اگر مقدار متغیر b یک باشد کنترل برنامه به x منتقل می‌گردد.

موارد تست	a	b	ناحیه تست
1	0	0	Y
2	0	1	Y
3	1	0	Y
4	1	1	X

در تکنیک تست آرایه متعامد مثال فوق به جای استفاده از همه موارد تست ۱ تا ۴، فقط یکی از موارد تست ۱تا ۳ برای تست ناحیه y و مورد تست ۴ برای تست ناحیه x مورد استفاده قرار می‌گیرد. در واقع در تست برنامه فوق توسط تکنیک تست آرایه متعامد، ۲ مورد تست به جای ۴ مورد تست، مورد استفاده قرار گرفته‌است.

تکنیک تست مبتنی بر گراف[1]

تکنیک تست مبتنی بر گراف در تست واسط کاربر مورد استفاده قرار می گیرد. یکی از بخش های ضروری هر سیستم کامپیوتری، واسط کاربر آن سیستم است که جهت ارتباط سیستم با محیط اطرافش استفاده می‌شود. کیفیت این ارتباط تأثیر چشمگیری در کارایی سیستم و همینطور

[1] Graph Based Testing

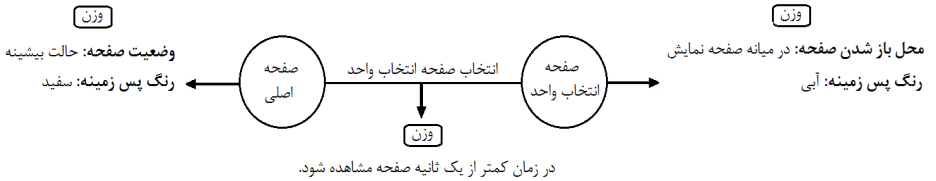
رضایت‌مندی مشتری دارد، چه بسا سیستم‌هایی که علی‌رغم کارکردهای مناسبشان، به دلیل ضعف در طراحی واسط کاربرشان، از بازار تجاری حذف شدند.

واسط کاربر چه برای یک دستگاه پخش موسیقی دیجیتال طراحی شده باشد و چه برای سیستم کنترل یک هواپیما، آنچه که اهمیت دارد، قابلیت استفاده است. اگر سازوکارهای واسط، از طراحی خوبی برخوردار باشد، کاربر با استفاده از ریتمی ملایم به تعامل با دستگاه می‌پردازد که به او امکان می‌دهد تا بدون هیچ‌گونه تلاش زیادی به اهداف خود دست پیدا کند. ولی اگر واسط کاربر، خوب طراحی نشده باشد، کاربر سردرگم می‌شود و نتیجه نهایی، چیزی جز ناراحتی و سرخوردگی کاربر نخواهد بود. واسط کاربر، پنجره‌ای فراروی نرم‌افزار است. واسط در بسیاری موارد، ادراک کاربر از کیفیت سیستم را شکل می‌دهد. اگر این پنجره غبار گرفته شود، موج‌دار شود یا شکسته شود، کاربر ممکن است، سیستمی پرقدرت را پس بزند. نتیجه اینکه تست واسط کاربری که برازنده انسان باشد، از اهمیت فراوانی برخوردار است.

امروزه واسط‌های گرافیکی کاربر به علت استفاده از اجزای قابل استفاده مجدد بسیار سریع‌تر و دقیق‌تر ایجاد می‌شوند. اما در عین حال پیچیدگی آنها نیز افزایش می‌یابد. این مسئله باعث بروز مشکلات بیشتر در طراحی و اجرای موارد تست مربوط به آنها می‌گردد. از آنجا که بسیاری از رابط‌های گرافیکی امروزی دارای شکل و شمایل تقریبا مشابه هستند، می‌توان به تست‌های استاندارد دست یافت. به دلیل تعداد زیاد حالت‌های ممکن در عملیات رابط‌های گرافیکی کاربر، تست باید با استفاده از ابزارهای خودکار انجام شود. طی چند سال اخیر، انواع ابزارهای تست رابط‌های گرافیکی کاربر به بازار ارائه شده است.

در تکنیک تست مبتنی بر گراف عناصر واسط کاربر به عنوان اشیاء در نظر گرفته می‌شوند. سپس هر شیء به عنوان یک گره در گراف تست در نظر گرفته می‌شود. همچنین اگر روابطی مابین اشیاء باشد، توسط یک یال ارتباط آنها نشان داده می‌شود. در گراف تست، وزن گره‌ها و یال‌ها نشان‌دهنده خصوصیات آنها است.

مثال: شیء «**صفحه اصلی برنامه**» و شیء «**صفحه انتخاب واحد**» را در سیستم انتخاب واحد اینترنتی یک دانشگاه در نظر بگیرید، زمانی که در صفحه اصلی برنامه، کلید انتخاب واحد توسط کاربر انتخاب می‌گردد، کنترل برنامه به صفحه انتخاب واحد منتقل می‌گردد. که گراف تست مبتنی بر گراف آن به صورت زیر است:



در زمان کمتر از یک ثانیه صفحه مشاهده شود.

بنابراین تست‌کننده با دنبال‌کردن گراف مذکور و تطابق وزن گره‌ها و یال‌ها با خروجی‌های برنامه، متوجه درستی یا نادرستی کارکرد برنامه می شود. برای‌مثال مدت زمان صرف شده برای

انتقال کنترل از صفحه اصلی برنامه به صفحه انتخاب واحد کمتر از یک ثانیه در نظر گرفته شده است، بنابراین اگر این زمان از زمان در نظر گرفته شده بیشتر باشد، باید نسبت به رفع آن اقدام‌های لازم صورت گیرد.

تکنیک تست مبتنی بر مدل[1]

Use Case Diagram یا نمودار مورد کاربرد، Requirement Diagram یا نمودار نیاز جهت مدل‌سازی لیست نیازمندی‌های مشتری در فعالیت ارتباطات مورد استفاده قرار می‌گیرد.

Use Case Diagram یا نمودار مورد کاربرد، می‌تواند به عنوان مدل لیست نیازمندی‌ها، به شکل چک لیست برای فعالیت تست مورد استفاده قرار بگیرد. بنابراین تکنیک تست مبتنی بر مدل از اطلاعات موجود در مدل لیست نیازمندی‌ها به عنوان مبنایی برای تولید موارد تست، بهره می‌برد. جهت تست دقیق‌تر کارکرد سناریوهای موجود در Use Caseها، نمودارهای رفتاری همچون نمودار توالی و حالت نیز در تکنیک تست مبتنی بر مدل مورد استفاده قرار می‌گیرند.

انواع تست جامعیت یا یکپارچگی

به طور کلی جهت تست جامعیت دو شیوه تست «**جامعیت غیرتدریجی**» و «**تست جامعیت تدریجی**» وجود دارد که در ادامه به بررسی آنها می‌پردازیم.

تست جامعیت غیر تدریجی یا غیرافزایشی

در تست جامعیت غیرتدریجی، که به انفجار بزرگ[2] نیز موسوم است. ابتدا کل ساختار برنامه یکپارچه می‌شود و سپس کل ساختار برنامه مورد تست قرار می‌گیرد.آنچه واضح است این است که نتیجه تست جامعیت غیرتدریجی یک لیست طولانی از خطاها خواهد بود. که مشخص هم نخواهد بود خطاها مربوط به کدام بخش برنامه است. و اگر هم مشخص باشد، تصحیح آن بسیار دشوار خواهد بود.

تست جامعیت تدریجی یا افزایشی

تست جامعیت تدریجی در نقطه مقابل تست جامعیت غیرتدریجی قرار دارد. در این رویکرد ساختار برنامه به شکل تدریجی و قدم به قدم به سمت یکپارچگی حرکت می‌کند و همگام با این حرکت و تکامل بخش کامل شده از برنامه نیز به شکل تدریجی و قدم به قدم مورد تست قرار می‌گیرد. به بیان دیگر در تست جامعیت تدریجی، کار با کوچک‌ترین مؤلفه شروع شده و پس از خطایابی، مولفه دیگری به مجموعه جاری اضافه می‌شود. این کار تا زمانی که کل نرم‌افزار تحت پوشش خطایابی قرار داده شود، ادامه می‌یابد.

نتیجه تست جامعیت تدریجی لیستی از خطاها خواهد بود که قدم به قدم آشکار می‌شود. و مشخص هم خواهد بود که خطاها مربوط به کدام بخش از برنامه است. که به تبع تصحیح خطا

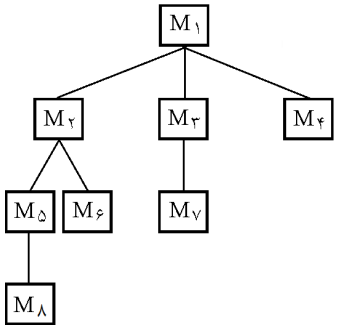
[1] Model Based Testing
[2] Big Bang

آسان‌تر و ساده‌تر نیز خواهد بود. جهت این پیمایش و کنار هم قرار دادن مولفه‌ها دو رویکرد تست جامعیت تدریجی بالا به پایین و تست جامعیت تدریجی پایین به بالا وجود دارد، که در ادامه به بررسی آنها می‌پردازیم.

تست جامعیت تدریجی بالا به پایین

تست جامعیت تدریجی بالا به پایین روشی افزایشی برای تست ساختار برنامه است. در این روش عمل تست از بالاترین پیمانه یعنی مولفه اصلی برنامه(main program) شروع شده و طی سلسله مراتب بالا به پایین مولفه‌ها به مجموعه تست اضافه می‌گردند. به بیان دیگر در تست جامعیت تدریجی بالا به پایین، ابتدا مولفه اصلی برنامه مورد تست قرار می‌گیرد و در ساختار معماری قرار می‌گیرد. سپس مولفه‌های فرزند مولفه اصلی برنامه مورد تست قرار می‌گیرند و این کار در یک روند بالا به پایین در مورد تمام مولفه‌های برنامه انجام می‌گردد تا تست جامعیت برنامه کامل گردد. مولفه‌های زیرین مولفه اصلی برنامه به دو شیوه **اول عمق (Depth-First)** و **اول سطح (Breadth-First)** می‌توانند پس از پیمایش در ساختار تست قرار داده شوند.

شکل زیر ساختار سلسله مراتبی مؤلفه‌های یک برنامه را نشان می‌دهد:



توجه: برای پیمایش اول عمق، همانند پیمایش درخت‌ها در درس ساختمان داده‌ها، می‌توان یکی از سه روش NLR، LNR و LRN را انتخاب نمود.

بنابراین ترتیب ترکیب مولفه‌ها به شیوه اول عمق به صورت زیر است: (روش NLR)

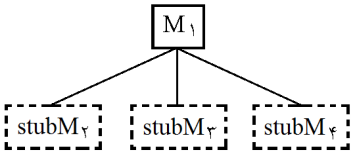
$$M_۱ - M_۲ - M_۵ - M_۸ - M_۶ - M_۳ - M_۷ - M_۴$$

همچنین ترکیب مولفه‌ها به شیوه اول سطح به صورت زیر است:

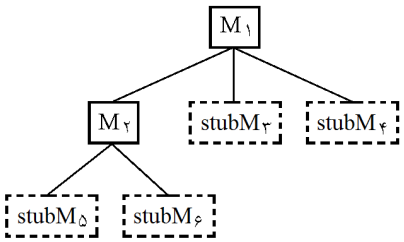
$$M_۱ - M_۲ - M_۳ - M_۴ - M_۵ - M_۶ - M_۷ - M_۸$$

فرایند تست جامعیت تدریجی بالا به پایین طی مراحل زیر انجام می‌شود:

۱- مولفه اصلی برنامه به عنوان گرداننده تست به کار می‌رود و مولفه‌های ریشه، به جای تمام مؤلفه‌هایی که مستقیماً در سطح بعدی مولفه اصلی برنامه قرار دارند، جایگزین می‌شوند. (در مثال $M_۲$، $M_۳$ و $M_۴$)



۲- بسته به روش پیمایش انتخاب شده (اول عمق یا اول سطح)، مولفه‌های ریشه، به نوبت، جای خود را به مؤلفه‌های واقعی می‌دهند، این کار در یک روند بالا به پایین در مورد تمام مولفه‌های برنامه انجام می‌گردد تا تست جامعیت برنامه کامل گردد.



توجه: در تست جامعیت تدریجی بالا به پایین پس از درج هر مولفه در ساختار تست برنامه، بخش مورد نظر یا کل ساختار برنامه باید مجددا مورد تست و ارزیابی قرار گیرد، این عمل مذکور به **تست بازگشت** یا **رگرسیون** موسوم است که جلوتر بیشتر در مورد آن صحبت خواهیم کرد.

توجه: در تست جامعیت تدریجی بالا به پایین از بین مولفه‌های سربار ریشه و گرداننده، فقط مولفه ریشه مورد استفاده قرار می‌گیرد.

اساسی‌ترین مشکل این روش هنگامی رخ می‌دهد که تست سطوح بالاتر نیازمند سطوح پایین‌تر باشد، برای حل آن می‌توان به جای سطوح پایین‌تر از مولفه‌های ریشه استفاده نمود. اما این راه‌حل سبب می‌شود تا مولفه‌های سطوح بالاتر را نتوان به طور کامل تست کرد، زیرا آنها با فرض این که مولفه‌های سطوح پایین‌تر، مولفه‌های ریشه هستند تست شده‌اند و تست کامل آنها زمانی رخ می‌دهد که هیچ مولفه ریشه‌ای در درخت کنترل باقی نماند. برای حل این مشکل انتخاب‌های زیر وجود دارند:

۱- به تأخیر انداختن بسیاری از تست‌ها تا اینکه مولفه‌های ریشه جای خود را به مولفه‌های واقعی بدهند.

۲- توسعه تعدادی مولفه‌های ریشه که با اجرای عملیاتی محدود، مولفه واقعی را شبیه‌سازی می‌کنند.

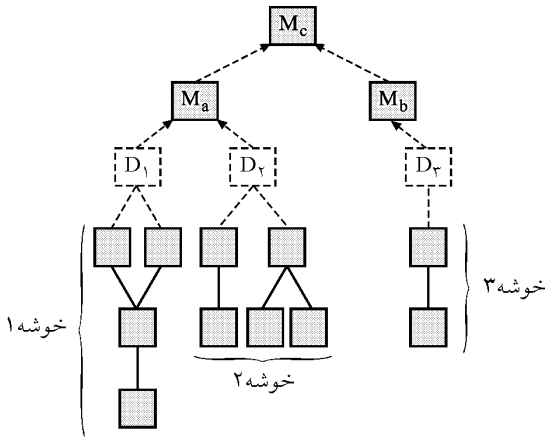
۳- استفاده از تست جامعیت تدریجی پایین به بالا که در ادامه به آن می‌پردازیم.

تست جامعیت تدریجی پایین به بالا

در جامعیت تدریجی پایین به بالا، تست مولفه‌ها با یکی از مولفه‌هایی که در پایین‌ترین سطح از سلسله مراتب ساختار برنامه قرار دارد آغاز و به سمت سایر مولفه‌ها در بخش بالای برنامه هدایت می‌شود. سپس مولفه‌های بررسی شده در کنار هم جمع می‌شوند تا یک خوشه را تشکیل

دهند. از آنجا که در این روش حرکت از پایین به بالا است لذا به مولفه‌های ریشه نیازی وجود ندارد. اما برای تست خوشه‌ها به مولفه‌های گرداننده نیاز می‌باشد، تا به طور موقت کار مولفه‌های سطوح بالاتر را انجام دهند. پس از تست خوشه‌ها مولفه‌های گرداننده با مؤلفه‌های واقعی جایگزین می‌شوند.

شکل زیر، تست جامعیت تدریجی پایین به بالا را نشان می‌دهد:



هر یک از خوشه‌ها با استفاده از یک مولفه گرداننده (که به صورت بلوک نقطه‌چین نشان داده شده است)، مورد تست قرار می‌گیرد. پس از تست خوشه‌های ۱ و ۲ به کمک گرداننده‌های $D_۱$ و $D_۲$، این گرداننده‌ها جای خود را به یک مولفه واقعی برای ارتباط خوشه‌های ۱ و ۲ با مؤلفه‌های واقعی M_a می‌دهند. به طور مشابه، گرداننده $D_۳$ پس از تست خوشه ۳ جای خود را به مؤلفه واقعی M_b می‌دهد. در انتها M_a و M_b با مؤلفه M_c مجتمع می‌شوند.

در این روش تعداد مولفه‌های گرداننده زیاد است. برای کاهش تعداد آنها توصیه می‌گردد دو سطح بالای ساختار برنامه با روش بالا به پایین یکپارچه شوند و برای مابقی ساختار از یکپارچه‌سازی پایین به بالا استفاده گردد. این امر باعث کاهش تعداد مولفه‌های گرداننده و آسان‌تر شدن یکپارچه‌سازی خوشه‌ها می‌شود.

توجه: این عمل اخیر موسوم به «**تست ساندویچ**» نیز می‌باشد.

توجه: در تست جامعیت تدریجی پایین به بالا از بین مولفه‌های سربار ریشه و گرداننده، فقط مولفه گرداننده مورد استفاده قرار می‌گیرد.

تست بازگشت یا رگرسیون[1]

هر بار که یک مولفه جدید به مجموعه تست جامعیت تدریجی بالا به پایین اضافه می‌گردد، ممکن است نرم‌افزار به دلایل مختلفی همچون ایجاد مسیرهای جدید جریان داده، فراخوانی مولفه-

[1] Regression Testing

های جدید، و یا اعمال ورودی و خروجی دستخوش تغییر و تحول شود. این تغییرات ممکن است بر آنچه که قبلاً تست شده است تأثیر بگذارد و باعث بروز خطا شود. بنابراین تست مجدد لازم و ضروری است، این تست مجدد به **تست بازگشت** یا **رگرسیون** موسوم است. در این نوع تست، تعدادی از تست‌هایی که قبلاً انجام شده است مجدداً اجرا می‌شوند تا اطمینان حاصل شود که تغییرات باعث وقوع خطا شده است یا خیر. این تست ممکن است به صورت دستی و خودکار در حالات زیر انجام شود:

۱- تست تمام نرم‌افزار
۲- تست مؤلفه‌هایی که تحت تأثیرات جانبی قرار گرفته‌اند.
۳- تست مؤلفه‌های تغییریافته

تست دود[1]

تست دود یک روش تست جامعیت است که به طور متداول برای تست بسته‌های نرم‌افزاری مبتنی بر مولفه (قابل استفاده مجدد) مورد استفاده قرار می‌گیرد که زمان تحویل نیز در آن از اهمیت بالایی برخوردار است.

این روش امکان تولید نرم‌افزار و تست‌های فراوان را به طور هم‌زمان در یک محدوده زمانی کم فراهم می‌آورد. این تست طی مراحل زیر انجام می‌شود:

۱- مؤلفه‌های نرم‌افزاری که به کد ترجمه شده‌اند، در قالب یک «**بنا**[2]» یکپارچه می‌شوند. یک «**بنا**» شامل تمام فایل‌های داده، کتابخانه‌ها و مولفه‌های قابل استفاده مجدد است که برای پیاده‌سازی یک یا چند عملکرد محصول موردنیاز است.
۲- یک سری از تست‌ها طراحی می‌شوند تا خطاهایی را آشکار نمایند که باعث می‌شوند یک «**بنا**» به طور منظم عمل خود را انجام ندهد، هدف، یافتن خطاهای بازدارنده‌ای است که بالاترین احتمال به تأخیر انداختن پروژه را دارند.
۳- این «**بنا**» با بناهای دیگر یکپارچه می‌شود و محصول کامل (به شکل فعلی) به صورت روزانه با این روش تست می‌گردد. یکپارچه‌سازی می‌تواند توسط «**تست جامعیت تدریجی بالا به پایین**» یا «**تست جامعیت تدریجی پایین به بالا**» انجام گردد.

توجه: از آنجاکه یکپارچگی بناها به صورت تدریجی صورت می‌گیرد و تست‌های دود به طور روزانه اجرا می‌شوند، ناسازگاری‌ها و خطاها خیلی زود کشف می‌شوند، لذا احتمال وارد آمدن آسیب‌های جدی به زمان‌بندی پروژه، کاهش می‌یابد.

توجه: از آنجاکه یکپارچگی بناها به صورت تدریجی صورت می‌گیرد، احتمال یافتن نقایص طراحی در سطح طراحی معماری و طراحی مولفه بیشتر می‌شود. اگر این نقایص، زودهنگام

[1] Smoke Testing
[2] Build

برطرف شوند، محصول باکیفیت‌تری تولید خواهد شد.

توجه: از آنجاکه یکپارچگی بناها به صورت تدریجی صورت می‌گیرد، با درج هر بنا کشف خطاهای عملیاتی به صورت تدریجی و آسان‌تر صورت می‌گیرد.

توجه: از آنجاکه یکپارچگی بناها به صورت تدریجی صورت می‌گیرد، با گذشت هر روز، مقدار بیشتری از نرم‌افزار یکپارچه می‌شود. این موضوع، روحیه تیم را بهبود می‌بخشد و شاخص خوبی از پیشرفت کار را به مدیران می‌دهد.

مثال: تست جامعیت یا یکپارچگی مانند تست قطعات (مولفه‌های) یک خودرو توسط یک خودروساز به شکل تدریجی در کنار یکدیگر به کمک گرداننده و ریشه‌های مرتبط با آن است.

پس از تست جامعیت یا یکپارچگی نوبت به تست اعتبارسنجی می‌رسد، بنابراین در ادامه به بررسی تست اعتبارسنجی می‌پردازیم.

۳- تست اعتبارسنجی[1]

تست اعتبارسنجی، در پایان تست جامعیت یا یکپارچگی آغاز می‌شود، یعنی زمانی‌که اجتماع کل مولفه‌های برنامه در کنار یکدیگر قرار گرفتند و نرم‌افزار به طور کامل مونتاژ شد و خطاهای واسط و نقاط اتصال مابین مولفه‌ها کشف و برطرف شدند.

تست اعتبارسنجی، روند کار یا درستی کار را در سطح اجتماع کل مولفه‌های برنامه در کنار یکدیگر در شرایط **سهل و آسان** مورد تست و ارزیابی قرار می‌دهد. تا مشخص شود نرم‌افزاری که ایجاد شده است منطبق بر نیازمندی‌های مشتری است یا خیر. به عبارت دیگر مشخص شود که آیا نرم‌افزار ایجاد شده بر اساس ورودی‌های مورد نظر مشتری **در** شرایط **سهل** و **آسان**، خروجی‌های مورد انتظار مشتری را برآورده می‌سازد یا خیر.

به بیان دیگر تست اعتبارسنجی، اعتبارسنجی عملکرد یا روند اجرای کل مولفه‌های نرم‌افزار ایجاد شده در کنار یکدیگر را در شرایط **سهل و آسان** مورد تست و ارزیابی قرار می‌دهد.

مثال: تست اعتبارسنجی مانند تست قطعات (مولفه‌های) یک خودرو توسط یک خودروساز به شکل کامل در کنار یکدیگر در شرایط سهل و آسان داخل محیط کارخانه خودروساز بدون لحاظ کردن شرایط سخت و دشوار همچون شرایط بارانی و کویری است.

توجه: تست اعتبارسنجی جهت محقق کردن اصل اعتبارسنجی (validation)، روش تست جعبه سیاه و تکنیک‌های مرتبط با آنرا مورد استفاده قرار می‌دهد.

توجه: در این مرحله کلیه‌ی موارد پیاده‌سازی شده، براساس لیست نیازمندی‌های وظیفه‌مندی و غیروظیفه‌مندی مشتری (چک لیست) که در فعالیت ارتباطات تهیه و در مدل تحلیل و طراحی مدل‌سازی شده است مورد وارسی قرار می‌گیرد تا مشخص شود نرم‌افزار ایجاد شده براساس ورودی‌های مورد نظر مشتری، خروجی‌های مورد انتظار مشتری را برآورده می‌سازد یا خیر. در

[1] Validation Testing

انتهای این تست یک لیست از نیازمندی‌های مشتری که برآورده نشده است تهیه می‌گردد تا رفع گردند. هنگامی‌که یک **نرم‌افزار سفارشی** (Custom Software) تنها برای یک مشتری توسعه می‌یابد، «**آزمون پذیرش**[1]» اجرا می‌شود تا مشتری را قادر به اعتبارسنجی کلیه‌ی خواسته‌ها کند. آزمون پذیرش، که به جای مهندس نرم‌افزار، توسط مشتری انجام می‌شود، می‌تواند از یک آزمون غیررسمی تا یک سری آزمون‌های برنامه‌ریزی شده‌ی سیستماتیک را شامل شود. درواقع، آزمون پذیرش را می‌توان در عرض یک هفته یا یک ماه انجام داد و لذا کشف خطاهایی که ممکن است سیستم را با گذشت زمان تنزل دهد، امکان‌پذیر می‌شود. اگر نرم‌افزار به عنوان محصولی توسعه می‌یابد که قرار است مشتریان زیادی از آن استفاده کنند، انجام «آزمون پذیرش» توسط یکایک آنها امکان‌پذیر نیست. اکثر سازندگان محصولات نرم‌افزاری از فرآیندی موسوم به تست آلفا (**α**) و بتا (**β**)، برای کشف خطاهایی استفاده می‌کنند که به نظر می‌رسد فقط کاربر نهایی قادر به یافتن آنها می‌باشد.

تست آلفا (α)

تست آلفا در مکان سازنده نرم‌افزار و توسط مشتری انجام می‌شود. بدین صورت که مشتری در یک محیط کنترل‌شده توسط سازنده، کنار سازنده می‌نشیند و نرم‌افزار را تست می‌کند. سازنده نیز همزمان خطاهای موجود را یادداشت می‌کند.

تست بتا (β)

تست بتا در مکان مشتری و توسط یک یا چند کاربر نهایی انجام می‌شود. برخلاف تست آلفا، سازنده معمولاً حضور ندارد و سیستم در محیط واقعی خود مستقر می‌باشد. بنابراین، تست بتا، یک کاربرد «زنده» از نرم‌افزار در محیطی است که سازنده قادر به کنترل آن نیست. مشتری کلیه مشکلات (واقعی یا تصوری) را که طی تست بتا یافته است، یادداشت می‌کند و آنها را در فاصله های زمانی منظم به سازنده گزارش می‌دهد. مهندسان نرم‌افزار، با توجه به مشکلات گزارش شده طی تست بتا، اصلاحات لازم را انجام می‌دهند و سپس محصول نرم‌افزاری نهایی را برای ارایه به مشتریان آماده می‌کنند.

شکل دیگری از تست بتا، که به «**آزمون پذیرش مشتری**[2]» موسوم است، گاهی اجرا می‌شود، یعنی زمانی‌که یک **نرم‌افزار سفارشی** تحت **قرارداد** به مشتری تحویل داده می‌شود. مشتری یک سری تست‌های مشخص انجام می‌دهد تا خطاها را قبل از پذیرفتن نرم‌افزار از سازنده آن کشف کند.

پس از تست اعتبارسنجی نوبت به تست سیستم می‌رسد، بنابراین در ادامه به بررسی تست سیستم می‌پردازیم.

---

[1] Acceptance Test
[2] Customer Acceptance Testing

۴- تست سیستم[1]

تست سیستم، در پایان تست اعتبارسنجی آغاز می‌شود، یعنی زمانی‌که اجتماع کل مولفه‌های برنامه در کنار یکدیگر قرار گرفتند و نرم‌افزار در شرایط **سهل و آسان** مورد تست و ارزیابی قرار گرفت.

تست سیستم، روند کار یا درستی کار را در سطح اجتماع کل مولفه‌های برنامه در کنار یکدیگر در شرایط **سخت** و **دشوار** مورد تست و ارزیابی قرار می‌دهد. تا مشخص شود نرم‌افزاری که ایجاد شده است منطبق بر نیازمندی‌های مشتری است یا خیر. به عبارت دیگر مشخص شود که آیا نرم‌افزار ایجاد شده بر اساس ورودی‌های مورد نظر مشتری **در** شرایط **سخت** و **دشوار**، خروجی‌های مورد انتظار مشتری را برآورده می‌سازد یا خیر.

به بیان دیگر تست سیستم، اعتبارسنجی عملکرد یا روند اجرای کل مولفه‌های نرم‌افزار ایجاد شده در کنار یکدیگر را در شرایط **سخت** و **دشوار** مورد تست و ارزیابی قرار می‌دهد.

مثال: تست اعتبارسنجی مانند تست قطعات (مولفه‌های) یک خودرو توسط یک خودروساز به شکل کامل در کنار یکدیگر در شرایط سخت و دشوار خارج از محیط کارخانه خودروساز با لحاظ کردن شرایط سخت و دشوار همچون شرایط بارانی و کویری است.

توجه: تست سیستم جهت محقق کردن اصل **اعتبارسنجی (validation)**، روش تست جعبه سیاه و تکنیک‌های مرتبط با آنرا مورد استفاده قرار می‌دهد.

انواع تست سیستم

به طور کلی جهت تست سیستم شش شیوه «**تست بازیابی**»، «**تست امنیت**»، «**تست فشار**»، «**تست حساسیت**»، «**تست کارایی**» و «**تست استقرار**» وجود دارد که در ادامه به بررسی آنها می‌پردازیم.

تست بازیابی[2]

بسیاری از سیستم‌های کامپیوتری باید بعد از بروز خرابی، پس از یک زمان از پیش تعیین شده پردازش خود را از سر گیرند. در برخی موارد، سیستم باید در برابر نقایص تحمل داشته باشد، یعنی نقایص پردازش نباید باعث شود عملکرد کلی سیستم متوقف شود.

تست بازیابی، نرم‌افزار را در شرایط سخت و دشوار و به روش‌های گوناگون وادار به شکست می‌کند و کنترل می‌کند که نرم‌افزار بتواند عملیات خود را به درستی ترمیم نماید. اگر ترمیم به صورت خودکار باشد، مقداردهی دوباره، بازیابی داده و شروع دوباره هر کدام، مورد ارزیابی قرار می‌گیرند و در صورتی که ترمیم نیازمند دخالت انسان باشد آنگاه میانگین زمان ترمیم (MTTR) ارزیابی می‌شود تا مشخص گردد که قابل قبول است یا خیر. نتیجه اینکه سیستم‌هایی که نیاز به

[1] System Testing

[2] Recovery Testing

تحمل‌پذیری خطا دارند، باید تحت تست بازیابی قرار گیرند. برای مثال فرض کنید در یک برنامه قرار است عمل تقسیم دو عدد انجام بگیرد، در تست بازیابی برای فراهم کردن شرایط سخت و دشوار برای این برنامه، می بایست مقدار مخرج را به صفر مقدار دهی نمود، اگر برنامه به‌طور خودکار و بدون دخالت انسان اقدام به صدور پیامی مبنی بر دریافت مقدار مناسب برای مخرج کسر به کاربر ارسال کند، آنگاه این بدین معنی خواهد بود که این برنامه در مدت زمان کوتاهی به طور خودکار اقدام به ترمیم خود نموده است. اما اگر جلوگیری از ورود مقدار صفر برای مخرج کسر در برنامه لحاظ نشده باشد، آنگاه ترمیم برنامه نیازمند دخالت انسان است که مدت زمان این ترمیم حائز اهمیت فراوان است. بنابراین تست بازیابی زمان و قدرت بازیابی یک سیستم کامپیوتری را مورد بررسی قرار می‌دهد.

توجه: MTTR سرواژه عبارت Mean Time To Repair و به معنی میانگین زمان ترمیم است.

تست امنیت[1]

هر سیستم کامپیوتری که اطلاعات حساس و مهمی را مدیریت و پردازش می‌کند یا عملیاتی را انجام می‌دهد که نفوذ در آن می‌تواند باعث متضررشدن مالکان آن گردد، هدف مناسبی برای نفوذکنندگان است. انگیزه این نفوذ می‌تواند به دلایل مختلفی همچون سرگرمی، لجاجت کارمندان سرخورده یک سازمان و منافع شخصی باشد. این تست سعی می‌کند مکانیزم امنیتی موجود در یک سیستم کامپیوتری را تست کند تا احیاناً جلوی نفوذهای ناخواسته گرفته شود.

در واقع در تست امنیت این مساله بررسی می‌شود که در هنگام وقوع حملات نفوذ به سیستم کامپیوتری، آیا سیستم کامپیوتری می‌تواند با این حملات مقابله کند و آنها را دفع کند و از دسترسی‌های غیر مجاز به سیستم کامپیوتری جلوگیری کند و یا نه پس می‌افتد. بنابراین در این شرایط تست‌کننده باید نقش یک فرد نفوذکننده را ایفا کند و از تمام توان و امکانات خود جهت نفوذ به سیستم کامپیوتری تحت تست خود استفاده کند. در این شرایط با کشف هر راه نفوذ توسط تست‌کننده باید اقدامات امنیتی لازم جهت مسدودسازی آن حفره امنیتی انجام گردد.

هر سیستم کامپیوتری، با صرف زمان، هزینه و منابع کافی قابل نفوذ است. هدف طراحان سیستم‌های کامپیوتری باید بدین گونه باشد که هزینه نفوذ به سیستم از سود حاصل از اطلاعاتی که یک نفوذکننده به دست می‌آورد بیشتر باشد. نتیجه اینکه سیستم‌هایی که نیاز به امنیت بالایی دارند، باید تحت تست امنیت در شرایط سخت و دشوار قرار گیرند.

تست فشار[2]

تست فشار مربوط به تست شرایط غیرمعمول از نظر میزان داده و حافظه نرم‌افزار است. در این نوع تست، سیستم طوری اجرا می‌گردد تا هر تقاضای غیرمعمول اعم از درخواست تعداد منابع نامحدود، حجم زیاد درخواست، تعداد وقفه‌های بیش از حد و یا درخواست حافظه زیاد به

[1] Security Testing
[2] Stress Testing

سیستم وارد شود و سپس کارایی سیستم و نرم‌افزار مشاهده گردد. اینکه نرم‌افزار حالش چطور است. به عبارت دیگر در تست فشار بررسی می‌شود که نرم‌افزار را قبل از شکست تا کجا می‌توان تحت فشار قرار داد. برای‌مثال، نرم‌افزار را می‌توان تحت حالاتی تست کرد که نرخ ورود داده افزایش یابد و یا حافظه زیادی درخواست شود. نتیجه اینکه سیستم‌هایی که نیاز به تحمل فشار بالایی دارند، باید تحت تست فشار در شرایط سخت و دشوار قرار گیرند. مانند تست سایت سازمان سنجش آموزش کشور در شرایط سخت و دشوار در لحظه ثبت نام یک آزمون به شکل واقعی و کاربران بسیار زیاد.

تست حساسیت[1]

شکل دیگری از تست فشار، تکنیکی موسوم به «**تست حساسیت**» است. در برخی شرایط که بیشتر در الگوریتم‌های ریاضی رخ می‌دهد، ممکن است دامنه بسیار کوچکی از داده‌های موجود در مرز داده‌های معتبر برای یک برنامه، باعث پردازش زیاد یا حتی نادرست یا تنزل زیاد در کارایی شود. تست حساسیت می‌کوشد تا در بازه‌های معتبر داده‌های ورودی، مقادیری را بیابد که باعث مختل شدن کارکرد نرم‌افزار می‌گردد. برای مثال فرض کنید در یک برنامه قرار است عمل تقسیم دو عدد انجام بگیرد، در تست حساسیت برای فراهم کردن شرایط سخت و دشوار برای این برنامه، می‌بایست مقدار مخرج را به صفر مقداردهی نمود، اگر برنامه به‌طور خودکار و بدون دخالت انسان اقدام به صدور پیامی مبنی بر دریافت مقدار مناسب برای مخرج کسر به کاربر ارسال کند، آنگاه این بدین معنی خواهد بود که این برنامه در مدت زمان کوتاهی به طور خودکار اقدام به ترمیم خود نموده است. اما اگر جلوگیری از ورود مقدار صفر برای مخرج کسر در برنامه لحاظ نشده باشد، آنگاه ترمیم برنامه نیازمند دخالت انسان است. بنابراین تست حساسیت، **حساسیت مقداری** یک سیستم کامپیوتری را در یک بازه مقداری معتبر مورد بررسی قرار می‌دهد.

تست کارایی[2]

این تست اغلب برای نرم‌افزارهای بی‌درنگ[3] و توکار[4] یا جاسازی شده یا تعبیه‌شده یا نهفته انجام می‌شود. زیرا نرم‌افزار بی‌درنگی که نیازمندی‌های وظیفه‌مندی (کارکردی) را مرتفع می‌کند ولی نیازمندی‌های غیروظیفه‌مندی (غیرکارکردی)، همچون کارایی زمان پاسخ کوتاه را مرتفع نمی‌کند مناسب برای محل نصب در سیستم‌های بی‌درنگ نخواهد بود. تست کارآیی برای این منظور طراحی می‌گردد تا بتوان کارایی زمان اجرای نرم‌افزار را در سیستم اندازه‌گیری نمود. این تست در تمامی مراحل فرآیند تست نرم‌افزار یعنی تست واحد، تست جامعیت و تست اعتبارسنجی می‌تواند مورد استفاده قرار گیرد، اما بررسی کارایی کل سیستم تا زمانی‌که کلیه بخش‌های مختلف نرم‌افزار به هم ملحق نشود امکان پذیر نخواهد بود. تست کارایی اغلب با تست

---

[1] Sensitivity Testing
[2] Performance Testing
[3] Real Time
[4] Embedded Software

فشار همراه می‌شود تا کارایی سیستم در شرایط سخت و دشوار نیز بررسی شود.

سیستم‌های بی‌درنگ به شدت به زمان وابسته هستند. بنابراین وابستگی زیاد سیستم‌های بی درنگ به متغیر زمان باعث دشوارتر شدن فرآیند تست آنها شده‌است. زیرا تست‌کننده نه تنها باید موارد روتین نظیر تست‌های جعبه سفید و سیاه را در نظر بگیرد، بلکه باید مدیریت رویدادها (پردازش وقفه‌ها)، مدیریت زمان‌بندی ورود داده‌ها و همگام‌سازی کارهای انجام شده روی داده‌ها را مدنظر داشته باشد. همچنین در بسیاری از موارد، داده‌هایی که به عنوان ورودی به سیستم در یک وضعیت داده می‌شود، ممکن است خروجی درستی را ایجاد کند اما اگر همان داده‌ها در وضعیت دیگری به سیستم داده شود، ممکن است نتیجه درستی حاصل نگردد و منجر به بروز خطا شود. علاوه بر این، ارتباط تنگاتنگ بین سیستم‌های بی‌درنگ و محیط‌های سخت‌افزاریشان ممکن است منجر به سخت‌ترشدن تست شود، زیرا در تست نرم‌افزار باید تاثیر نقص سخت‌افزار بر پردازش نرم‌افزار نیز در نظر گرفته شود.

به طور کلی برای تست سیستم‌های بی‌درنگ چهار مرحله زیر وجود دارد:

تست وظیفه[1]: نخستین گام در تست نرم‌افزار بی‌درنگ این است که هر یک از وظایف به طور مستقل تست شوند. یعنی برای هر وظیفه، تست‌های جعبه سیاه و جعبه سفیدی طراحی و اجرا شود. طی این تست‌ها هر وظیفه به طور مستقل اجرا می‌شود. تست وظایف باعث کشف خطاهای موجود در منطق و عملکرد نرم افزار می‌شود، ولی خطاهای **رفتاری** و **زمان‌بندی** را آشکار نمی‌کند.

تست رفتاری[2]: در این مرحله، کلیه رفتارهای شناسایی شده در مدل تحلیل، مبنای تست رفتاری یک سیستم بی‌درنگ قرار می‌گیرد و براساس همان رفتارهای شناسایی شده نرم‌افزار مورد تست و ارزیابی قرار می‌گیرد.

تست بین وظایف[3]: در این مرحله، به موقع انجام شدن و همگام بودن وظایف مرتبط باهم از نظر زمانی، مورد تست و ارزیابی قرار می‌گیرد.

تست سیستم[4]: در این مرحله، نرم‌افزار و سخت‌افزار مربوط به یک سیستم بی‌درنگ به شکل یکپارچه در شرایط سخت و دشوار تحت تست و بررسی قرارمی‌گیرد.

مثال: در فضاپیمای «راه‌یاب» ۳ کار مهم در نرم‌افزار آن تعبیه شده است که عبارتند از:

T_1: **به صورت دوره‌ای سلامت سیستم‌ها و نرم‌افزار فضاپیما را چک می‌کند.**

T_2: **داده‌های تصویری را پردازش می‌کند.**

T_3: **هر از گاهی بر روی وضعیت تجهیزات آزمایش می‌کند.**

---

[1] Task Testing
[2] Behavioral Testing
[3] Intertask Testing
[4] System Testing

اولویت سه کار به ترتیب T_1 و T_2 و T_3 هستند یعنی T_1 بالاترین اولویت و T_3 پایین‌ترین را دارند. هر کار که اولویت بالاتر داشته باشد و آماده باشد کار دیگر را قبضه (preempt) می‌کند. در هر بار اجرای T_1 یک تایمر به بالاترین مقدار خود مقداردهی می‌شود. اگر احیاناً زمان تایمر منقضی شود، فرض می‌شود که مشکلی در اجرای نرم‌افزار فضاپیما به وجود آمده است. در این حالت تمام پردازش‌ها متوقف می‌شوند و نرم‌افزار به طور کامل بار می‌شود و تمام سیستم‌ها آزمایش می‌شوند و همه چیز از نقطه شروع آغاز می‌شود. T_1 و T_3 در یک ساختار داده‌ای مشترک هستند و برای دسترسی به آن از سمافور باینری S استفاده می‌کنند. سناریوی زیر را در نظر بگیرید که به ترتیب پیش می‌رود.

۱- T_3 شروع به کار می‌کند.

۲- T_3 سمافور S را در اختیار می‌گیرد و وارد ناحیه بحرانی می‌شود.

۳- T_1 که دارای الویت بالاتری است T_3 را قبضه می‌کند و شروع به اجرا می‌کنند.

۴- T_1 اقدام به ورود به ناحیه بحرانی می‌کند ولی بلوک می‌شود. T_3 کار خود در ناحیه بحرانی را پی‌گیرد.

۵- T_2، T_3 را قبضه می‌کند و شروع به اجرا می‌کند.

۶- T_2 به دلیلی مستقل از T_1 و T_3، معلق می‌شود. T_3 دوباره ادامه می‌دهد.

۷- T_3 ناحیه بحرانی را ترک می‌کند و سمافور *S* آزاد می‌شود.

T_1، T_3 را قبضه می‌کند و سمافور را در اختیار می‌گیرد و وارد ناحیه بحرانی می‌شود.

۱) در این سناریو ممکن است مشکل زمانی به وجود آید و اگر اولویت T_2 را کمتر از T_3 قرار دهیم مشکل حل می‌شود.

۲) در این سناریو اولویت داشتن T_1 نسبت به T_2 و T_3 خود را نشان می‌دهد و سیستم به درستی کار می‌کند.

۳) اگر بین کارها سهم زمانی برقرار کنیم زمان پاسخ تضمین می‌شود و مشکلات احتمالی زمانی از بین می‌روند.

۴) این سیستم به درستی کار نمی‌کند و می‌تواند شکست بخورد و تایمر منقضی شود.

پاسخ: گزینه (۴) صحیح است.

با توجه به صورت سوال، با اجرای T_1 تایمر با یک مقدار بالا مقداردهی می‌شود و بعد از اتمام زمان آن، سیستم بطور کامل بار می‌گردد. در سناریوی ارائه شده در مرحله‌ی ۳، تایمر T_1 مقداردهی و شروع به شمارش معکوس می‌کند. در مرحله‌ی ۴، T_1 به دلیل حضور T_3 در ناحیه بحرانی مسدود می‌گردد. در مرحله ۵، T_2 به دلیل داشتن اولویت بالاتر نسبت به T_3، T_3 را قبضه می‌کند. بنابراین پردازنده در اختیار T_2 قرار می‌گیرد. در مرحله ۶، T_2 به دلیلی مستقل از T_1و T_3 معلق می‌شود، بدین معنی که پردازنده را واگذار می‌کند، حالا به هر دلیلی (مثل عملیات ورودی و خروجی و قرارگرفتن در وضعیت منتظر). در هر حال حاضر پردازنده آزاد است و فقط T_3 در صف آماده پردازنده قرار دارد، زیرا T_1 مسدود و در صف سمافور قرار دارد و T_2 هم معلق است. بنابراین پردازنده در اختیار T_3 قرار داده می‌شود. بنابراین T_3 می‌تواند ادامه کار خود را در ناحیه

بحرانی انجام و تمام کند و T_1 را از ابتدای صف سمافور بیدار کند تا در صف آماده پردازنده قرار گیرد. حال اگر دقیقا در همین لحظه هم تعلیق T_2 تمام شود و T_2 هم در صف آماده پردازنده قرار بگیرد، از آنجا که T_1 اولویت بیشتری نسبت به T_2 دارد، T_1 می‌تواند پردازنده را در اختیار بگیرد و وارد ناحیه بحرانی گردد و قبل از Timeout تمام گردد و T_2 هم پس از مدتی تمام می‌گردد. در واقع اگر همه‌ی شرایط فوق به موقع رخ دهد، سیستم به درستی کار می‌کند و کارهای T_1،T_2و T_3 تمام می‌شوند و سناریوی مطرح شده می‌تواند به طور دوره‌ای در این نرم افزار تعبیه شده تکرار گردد.

اما نکته در اینجاست که اگر تعلیق یاد شده در مرحله ۶، به موقع رخ ندهد و دیر اتفاق بیافتد، باعث می‌شود پردازنده در اختیار T_2 باقی بماند، بنابراین ادامه کار T_3 در ناحیه بحرانی به تعویق می‌افتد. بنابراین صدور مجوز ورود T_1 به ناحیه بحرانی به تعویق می‌افتد. زیرا صدور این مجوز مستلزم اتمام کار T_3 در ناحیه بحرانی است که در حال حاضر پردازنده را در اختیار ندارد و درگیر تاخیر ناشی از کار طولانی T_2 است، در این شرایط زمان در حال سپری شدن است، در حالی‌که از مدت‌ها قبل شمارنده تایمر T_1، شمارش معکوس خود را آغاز کرده است. زمان می‌گذرد ولی همچنان T_2 معلق نشده است. زمان به سرعت در حال سپری شدن است، T_2 آنقدر تعلل و تأخیر ایجاد می‌کند که تا سرانجام شمارنده T_1، Timeout کند. در نتیجه سیستم مجدداً راه اندازی می‌گردد. مطابق فرض مسأله منقضی شدن تایمر (Timeout)، نشانه وجود مشکل در اجرای نرم افزار فضاپیما است که باید تمام پردازش‌ها متوقف شوند و نرم افزار به طور کامل بار شود و تمام سیستم‌ها تست شوند و همه چیز از نقطه شروع آغاز گردد. بنابراین از آنجا که احتمال وقوع Timeout مطابق سناریوی فوق دراین سیستم وجود دارد، بنابراین ممکن است این سیستم به‌درستی کار نکند، بنابراین گزینه دوم نادرست و گزینه چهارم درست است.

گزینه اول نیز نادرست است، زیرا با افزایش اولویت T_3 نسبت به T_2، بازهم این سیستم ممکن است به درستی کار نکند، ممکن است کار T_3 در ناحیه بحرانی آنقدر طولانی گردد که باز هم تایمر منقضی گردد.

گزینه سوم نیز نادرست است، زیرا وجود سهم زمانی برای خروج سریع یک کار موجود در ناحیه بحرانی کاری نمی‌تواند بکند، یک کار حاضر در ناحیه بحرانی، خودش باید از ناحیه بحرانی خارج گردد. برش زمانی فقط سبب انتقال پردازنده به کارها می‌گردد، حال اگر مدت زمان فعالیت یک کار در ناحیه بحرانی زیاد باشد، به تبع تعداد سهم زمانی بیشتری را مصرف می‌کند، که باز هم منجر به ایجاد تأخیر می‌گردد.

توجه: مشکل اساسی در این سیستم این است که کاری روی شمارنده تایمر نقش دارد که خود در شرایط رقابتی ممکن است در مدت زمانی که تایمر، شمارش معکوس خود را آغاز کرده است، وارد ناحیه بحرانی نشود.

تست استقرار[1]

در بسیاری از موارد، نرم‌افزار باید در سکوهای مختلف و تحت سیستم عامل‌های متفاوت اجرا شود. تست استقرار که به تست پیکربندی نیز موسوم است، نرم‌افزار را در هریک از این محیط‌ها مورد تست و ارزیابی قرار می‌دهد. همچنین در تست استقرار روال نصب نرم‌افزار و کلیه مستندات راهنمای نرم‌افزار که مورد استفاده کاربران نهایی می‌باشد مورد تست و ارزیابی قرار می‌گیرد. برای مثال نسخه تحت وب یک فروشگاه اینترنتی کتاب را در نظر بگیرید. در تست استقرار این برنامه تحت وب، می‌بایست کارکرد این برنامه در مرورگرها و سیستم‌عامل‌های مختلف مورد تست و ارزیابی قرار گیرد، تا از صحت عملکرد برنامه اطمینان حاصل گردد.

تست برنامه‌های خاص

با پیچیده‌تر شدن نرم‌افزارهای کامپیوتری، نیاز به روش‌های تست ویژه نیز رشد یافته است. روش‌های تست جعبه سفید و تست جعبه سیاه در همه محیط‌ها، طراحی‌ها و کاربردها قابل اجرا هستند، ولی گاه روش‌ها و دستورالعمل‌هایی منحصربه‌فرد در مورد تست، ضرورت پیدا می‌کند. در این بخش به دستورالعمل‌هایی درباره تست برنامه‌های خاص می‌پردازیم.

تست معماری‌های سرویس‌دهنده و سرویس گیرنده

ماهیت توزیع شده محیط‌های شبکه‌ای، مسایل کارآیی مرتبط با پردازش تراکنش‌ها، وجود انواع سایت‌های سخت‌افزاری، ارتباطات شبکه‌ای، درخواست‌های چندگانه از پایگاه داده‌های متمرکز و غیرمتمرکز (توزیع شده) و غیره همگی دست به دست هم داده تا تست نرم‌افزارهای مبتنی بر شبکه دشوارتر از نرم‌افزارهای مستقل از شبکه گردد و منجر به افزایش چشمگیر زمان و هزینه برای تست این نوع از نرم‌افزارها گردد.

تست مستندات راهنمای کاربر

همانطور که پیش‌تر نیز گفتیم، نرم‌افزار محصولی است که به واسطه‌ی فرآیند تولید نرم‌افزار و تحت نظارت مهندسی نرم‌افزار، تحلیل، طراحی و پیاده‌سازی و تست می‌گردد تا در نهایت بر اساس ورودی‌های موردنظر مشتری، خروجی‌های مورد انتظار مشتری را برآورده سازد و شامل مؤلفه‌های زیر می‌باشد:

۱- ساختار داده‌ای: محل نگهداری داده‌های محیط عملیاتی به شکل متغیرها و جداول.

۲- عملکرد: دستورات یا کدهای قابل اجرا که باعث انجام وظایف مورد نظر می‌شود که به برنامه کاربردی موسوم است.

۳- مستندات: شامل توصیف مدل‌های تحلیل و طراحی نزد سازنده و راهنمای کاربر نزد کاربران نهایی و مشتری.

علاوه بر تست بخش داده‌ای و عملکردی یک محصول نرم‌افزاری، تست بخش مستندات

[1] Deployment Testing

راهنمای کاربر، به عنوان یکی از مولفه‌های یک محصول نرم‌افزاری لازم و ضروری است. در تست مستندات راهنمای کاربر، راحتی و سادگی قابل استفاده بودن مستندات به عنوان راهنمای استفاده از نرم افزار در شرایط مختلف روند اجرای برنامه، مورد تست و بررسی قرار می‌گیرد.

خطاهای موجود در مستندات راهنمای کاربر، می‌تواند به اندازه خطاهای موجود در داده ها یا عملکرد برنامه، ناخوشایند باشد. هیچ چیز در هنگام استفاده از مستندات راهنمای کاربر، ناراحت‌کننده‌تر از این نیست که دستورالعمل‌های موجود در دفترچه راهنما یا راهنمای آنلاین یک برنامه دنبال شود، اما نتایج مورد انتظار مشاهده نشود.

به طور کلی تست مستندات راهنمای کاربر در دو مرحله زیر انجام می‌گردد:

۱- تست نگارش مستندات راهنمای کاربر: در این مرحله، نحوه نگارش مستندات راهنمای کاربر از نظر ادبی بررسی می‌گردد.

۲- تست زنده مستندات راهنمای کاربر: در این مرحله، راهکارهای کمکی مستندات راهنمای کاربر به صورت زنده از نظر تطابق با نرم‌افزار بررسی می‌گردد.

اشکال زدایی[1]

عملیات تست، خطاهای موجود در برنامه را آشکار می‌سازد و این وظیفه واحد اشکال‌زدایی است تا آن را رفع کند. در حقیقت اشکال‌زدایی با اجرای یک مورد تست آغاز می‌گردد. یعنی اشکال‌زدایی تست نیست، اما همیشه در نتیجه تست اتفاق می‌افتد. اگرچه اشکال‌زدایی را می‌توان همانند فعالیت تست، به صورت سیستماتیک و منظم هدایت کرد اما این فرآیند تا حد زیادی هنر می‌باشد. زیرا ممکن است خطاهای مشاهده شده با منبع تولیدکننده آن به ظاهر هیچ ارتباطی نداشته باشد. علت این موضوع به دو مفهوم «**نشانه خطا**[2]» و «**علت خطا**[3]» و رابطه مابین آنها مرتبط است. نشانه خطا همان علائمی هست که تست‌کننده به هنگام تست آنها را کشف می‌کند و به تبع علت خطا هم دستوراتی از برنامه است که منجر به ایجاد خطا شده است. در اغلب موارد رابطه مابین نشانه خطا و علت خطا به شکل مستقیم قابل مشاهده نیست. که هنر اشکال‌زدایی کشف همین روابط ساده یا پیچیده مابین نشانه خطا و علت خطا است. برای‌مثال در برخی مواقع محل وقوع نشانه خطا با محل علت خطا متفاوت است، بدین معنی‌که محل وقوع نشانه خطا در یک مولفه و محل علت خطا در مولفه‌ای دیگر است که این مساله منجر به پیچیدگی رابطه مابین محل وقوع نشانه خطا و علت خطا می‌گردد. علت این پیچیدگی ممکن است به اتصال(Coupling) بالای مابین دو مولفه مرتبط باشد.

توجه: اشکال‌زدایی به دنبال اجرای تستی شروع می‌شود که منجر به شناسایی خطا گردیده است.

توجه: در هنگام تصحیح کردن یک خطا باید تمامی جوانب مرتبط با آن خطا را در نظر داشت

---

[1] Debugging
[2] Error Manifestation
[3] Error Cause

زیرا به راحتی تصحیح یک خطا می‌تواند باعث بروز خطا یا خطاهای جدیدی در قسمت‌های دیگر شود.

روش‌های اشکال زدایی

به طور کلی برای اشکال‌زدایی سه روش زیر وجود دارد:

۱- روش حذف علت[1]

در روش حذف خطا، لیستی از علت‌های احتمالی خطا تهیه می‌گردد. سپس تست‌کننده هر بار یکی از علت‌های احتمالی خطا را با نشانه خطا تحت نظر قرار می‌دهد و بررسی می‌کند که ببیند با حذف علت خطا، نشانه خطا نیز حذف می‌گردد یا خیر. که اغلب ادامه این روند منجر به کشف رابطه مابین نشانه خطا و علت خطا می‌گردد.

۲- روش پسگرد یا عقب‌گرد[2]

عقب‌گرد یک روش اشکال‌زدایی نسبتا متداول است که در اشکال‌زدایی برنامه‌های کوچک اغلب موفق است. نحوه کار بدین صورت است که با شروع از محلی که نشانه خطا کشف شده است، کد منبع، رو به عقب و به طور دستی مورد ردگیری قرار می‌گیرد تا محل علت خطا کشف شود. البته با افزایش تعداد خطوط کد، تعداد مسیرهای رو به عقب افزایش می یابد، که در این حالت این روش دیگر کارآمد نخواهد بود.

۳- روش نیروی مطلق[3]

این روش متداول‌ترین و ناکارآمدترین روش اشکال‌زدایی است و تنها زمانی مورد استفاده قرار می‌گیرد که روش‌های دیگر موفق به کشف محل علت خطا نشوند. نحوه کار بدین صورت است که تست‌کننده به کمک امکانات کامپایلر برنامه را خط به خط در جستجوی محل علت خطا ملاقات می‌کند، چاره‌ی دیگری هم نیست، چون روش‌های دیگر موفق به کشف محل علت خطا نشده‌اند که مجبور به بررسی خط به خط برنامه شده‌ایم. البته با این امید که بتوانیم سرانجام محل علت خطا را کشف کنیم. اما این کار مانند پیداکردن سوزن در انبار کاه است، که شاید هم محل علت خطا کشف نشود، شاید هم بشود، به هر حال آخرین راه حل همین روش است، اما امیدواریم موفق به کشف محل علت خطا شویم.

آخرین فعالیت، از فعالیت‌های چارچوبی، فعالیت استقرار است، که در ادامه به آن می‌پردازیم:

فعالیت استقرار

پس از فعالیت تست نوبت به فعالیت استقرار می‌رسد. در این مرحله، نرم‌افزار به مشتری تحویل داده می‌شود و مشتری با بررسی محصول دریافتی، بازخوردهای به دست آمده براساس همین ارزیابی‌ها را به تیم نرم‌افزاری ارائه می‌دهد. این بازخوردها می‌توانند مبنایی برای ارتقاء و یا

---

[1] Cause Elimination
[2] Backtracking
[3] Brtue Force

تصحیح نسخه‌ی بعدی نرم‌افزار باشد.

نگهداری نرم‌افزار

در سیستم‌های بزرگ نرم‌افزاری، نگهداری و پشتیبانی نرم‌افزار، یکی از وظایف عمده برنامه‌نویسان و شرکت‌های ارایه‌دهنده نرم‌افزار محسوب می‌شود. در برخی از موارد هزینه‌های نگهداری نرم‌افزار بیشتر از هزینه‌های تولید می‌باشد، اما می‌توان با اعمال اقداماتی از پیش تعیین شده نگهداری نرم‌افزار را تاحدودی آسان‌تر و کم‌هزینه‌تر کرد. در یک بیان ساده نگهداری به معنی تکرار مجدد فعالیت‌های چارچوبی از ارتباط تا استقرار جهت مرتفع کردن نیازها یا اشکالات احتمالی نرم‌افزار است.

امروزه بحث نگهداری نرم‌افزار یکی از خدمات ویژه‌ای است که سازندگان نرم‌افزار می‌توانند در جهت ارایه محصولات خود، دست به تبلیغ و رقابت زنند. آنها می‌توانند یکی از مهم‌ترین مزیت‌های محصول خود را، ارائه خدمات پس از فروش، نگهداری و پشتیبانی کامل و ارایه ضمانت‌های طولانی مدت معرفی کنند. به بیانی دیگر دستکاری یک محصول نرم‌افزاری پس از تحویل آن به منظور تصحیح خطاها، بهبود کارایی یا سایر صفات و تطبیق محصول برای یک محیط تغییریافته را نگهداری نرم‌افزار می‌گویند.

به طور کلی انواع نگهداری نرم‌افزار به چهار دسته زیر تقسیم می‌گردد:

۱- نگهداری تصحیحی[1]

نگهداری تصحیحی به معنی تصحیح یک نیاز برنامه‌نویسی‌شده توسط سازنده و به تبع برآورده شده اما نادرست است. مانند توانایی نرم‌افزار در برداشت از حساب بدون موجودی، که این مساله در نگهداری تصحیحی باید اصلاح گردد.

۲- نگهداری تطابقی[2]

نگهداری تطابقی به معنی تطابق یک نیاز برنامه‌نویسی‌شده توسط سازنده و به تبع برآورده شده و درست است که به علت تغییرات محیط برنامه همچون تغییرات در سخت‌افزار، سیستم عامل، پایگاه داده‌ها، پروتکل‌های شبکه‌ای و قوانین محیط عملیاتی، باید به پیروی از این تغیرات نحوه برآورده‌سازی آن توسط نرم‌افزار تغییر کند. مانند تغییر محاسبه ارزش افزوده یک فروشگاه کتاب از هفت درصد به هشت درصد که این مساله در نگهداری تطابقی باید با قوانین جدید انطباق داده شود. یا مانند تغییر تکنولوژی برنامه‌نویسی یک برنامه برای تطابق با یک سیستم عامل جدید.

۳- نگهداری تکمیلی[3]

نگهداری تکمیلی به معنی اضافه‌کردن یک نیاز برنامه‌نویسی‌نشده توسط سازنده اما بیان شده

[1] Corrective Maintenance
[2] Adaptive Maintenance
[3] Perfective Maintenance

توسط مشتری و یا به معنی اضافه‌کردن یک نیاز جدید مشتری و به تبع برنامه‌نویسی‌نشده توسط سازنده است که به تازگی توسط مشتری بیان شده‌است و باید توسط سازنده پیاده‌سازی گردد.

۴- نگهداری پیش‌گیرانه[1]

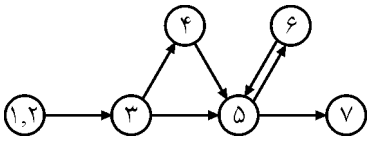
نرم‌افزار کامپیوتری در اثر تغییرات زیاد، کارایی خود را از دست می‌دهد، از این رو «مهندسی مجدد نرم‌افزار» یا نگهداری پیشگیرانه با تاکید بر اصول مهندسی‌نرم‌افزار در سرتاسر چرخه حیات نرم‌افزار مانع از ناکارآمدی نرم‌افزار می‌گردد. اغلب به علت طراحی نامناسب نرم‌افزار، و یا اعمال مکرر تغییرات در نرم‌افزار بدون رعایت اصول مهندسی نرم‌افزار، هزینه‌های نگهداری نرم‌افزار رشد فزاینده‌ای پیدا می‌کند. بنابراین برای کاهش هزینه‌های نگهداری آتی و سازگارکردن نرم‌افزار با تغییرات، مجموعه‌ای از اعمال پیشگیرانه صورت می‌گیرد که اغلب **مهندسی مجدد نرم‌افزار** (software reengineering) نامیده می‌شود. مهندسی مجدد نرم‌افزار فرآیندی زمانبر و پرهزینه است و اغلب چندین سال یکبار در نرم‌افزارها انجام می‌شود.

[1] Preventive Maintenance

تست‌های فصل هشتم

۱- برنامه زیر و گراف جریان (flow graph) متناظر با آن را در نظر بگیرید: (مهندسی IT – دولتی ۸۴)

```
1       Read x, y;
2       z = 10;
3       if (x<y) then
4           z = y-x;
5       while (z>10)
6           z = z-1;
7       end
```



با در نظر گرفتن مجموعه موردهای تست (test case) (x=1, y=6) و (x=25 , y=10) کدام گره‌ها در گراف جریان پوشش داده نمی‌شوند؟

۱) ۵ ۲) ۶ ۳) ۴ و ۶ ۴) ۵ و ۶

۲- تفاوت بین verification و validation چیست؟ (مهندسی IT – آزاد ۸۴)

۱) در Verification به کیفیت ساخت و در Validation به کیفیت محصول توجه می‌شود.

۲) در Verification به خوب بودن محصول و در Validation به خوب بودن کار توجه می‌شود.

۳) در Verification به نیازمندی‌های کاربر و در Validation به نیازمندی‌های نرم‌افزار توجه می‌شود.

۴) گزینه ۲ و ۳ صحیح است.

۳- Cyclomatic Complexity اندازه‌ای کمی از پیچیدگی منطقی نرم‌افزار است. با توجه به گراف جریان زیر، مقدار Cyclomatic Complexity کدام‌یک از گزینه‌های زیر می‌باشد؟

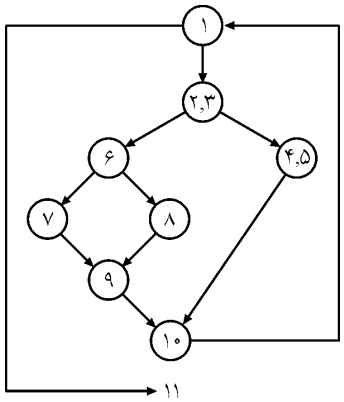
(مهندسی IT – دولتی ۸۵)

۱) ۸

۲) ۵

۳) ۴

۴) ۳



۴- آزمون مسیر پایه به کدام‌یک از آزمون‌های نرم‌افزار تعلق دارد؟ (مهندسی IT – آزاد ۸۵)

۱) آزمون جعبه سیاه ۲) آزمون مقایسه‌ای

۳) آزمون آرایه متعامد ۴) آزمون جعبه سفید

۵- تست حساسیت (Sensitivity Testing) به کدام‌یک از گونه‌های تست زیر متعلق است؟

(مهندسی IT – آزاد ۸۶)

۱) تست کارایی (Performance Testing) ۲) تست بازیابی (Recovery Testing)

۳) تست واحد (Unit Testing) ۴) تست فشار (Stress Testing)

۶- قانون ۴۰-۲۰-۴۰ نشان‌دهنده این است که کمترین توان تیم روی ..... است. (مهندسی IT – دولتی ۸۷)

۱) تست ۲) کد زدن ۳) تحلیل و طراحی ۴) تخمین و برنامه‌ریزی

۷- ترتیب فعالیت‌هایی که برای تست نرم‌افزار انجام می‌شود کدام‌یک از گزینه‌های زیر می‌باشد؟

(مهندسی IT – آزاد ۸۷)

۱) واحد، اعتبار، یکپارچگی، سیستم ۲) واحد، اعتبار، سیستم، یکپارچگی

۳) واحد، یکپارچگی، سیستم، اعتبار ۴) واحد، یکپارچگی، اعتبار، سیستم

۸- کدام‌یک از مراحل، به عنوان فعالیتی از مراحل دور حیات سیستم تلقی نمی‌شود؟ (مهندسی IT – دولتی ۸۸)

۱) تست (Test)

۲) مدل‌سازی سیستم (Modeling)

۳) تغییر روایت (Versioning)

۴) مهندسی نیازمندی (User Requirement Engineering)

۹- «مهندسی مجدد نرم‌افزار» در کدام‌یک از نگهداری‌های زیر مورد تأکید قرار می‌گیرد؟

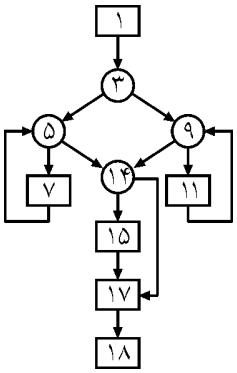
(مهندسی IT – آزاد ۸۹)

۱) نگهداری پیش‌گیرانه ۲) نگهداری بهبودی

۳) نگهداری تطابقی ۴) نگهداری تصحیحی

۱۰- پیچیدگی سایکلوماتیک گراف جریان روبرو، برابر با کدامیک از مقادیر زیر است؟

(مهندسی IT – دولتی ۹۱)



۱) ۶

۲) ۵

۳) ۷

۴) ۱۱

۱۱- تکنیک تحلیل مقادیر مرزی (Boundary value analysis) برای کدام‌یک از موارد زیر استفاده می‌شود؟ (مهندسی IT – دولتی ۹۱)

۱) تحلیل امکان‌پذیری ۲) تحلیل خواسته‌ها
۳) آزمون جعبه سیاه ۴) آزمون جعبه سفید

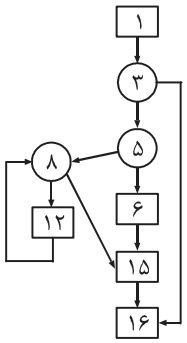
۱۲- کدام‌یک از روش‌های آزمون زیر، روشی معمول برای آزمون سیستم (System Testing) است؟ (مهندسی IT – دولتی ۹۲)

۱) آزمون حلقه‌های تکرار (Loop Testing) ۲) آزمون دود (Smoke Testing)
۳) آزمون بازیابی (Recovery Testing) ۴) آزمون جریان داده (Data Flow Testing)

۱۳- کدام جمله در مورد تست درست است؟ (مهندسی IT – دولتی ۹۳)

۱) در تست بالا به پایین بایستی زیر برنامه‌هایی را به عنوان Driver ساخت.
۲) در تست واحدها، سربار ساختن Driverها و یا stubها وجود دارد.
۳) در تست پایین به بالا، سربار ساختن زیربرنامه‌های مجازی به نام stub وجود دارد.
۴) هر سه مورد فوق

۱۴- Cyclomatic complexity برای گراف جریان برنامه زیر چقدر است؟ (مهندسی IT – دولتی ۹۴)



۱) ۳
۲) ۴
۳) ۹
۴) ۱۱

۱۵- کدام آزمون، روشی معمول برای اعتبارسنجی نرم‌افزار سفارشی (Custom Software) است؟ (مهندسی IT – دولتی ۹۶)

۱) آزمون واحد ۲) آزمون پذیرش ۳) آزمون بِتا ۴) آزمون رگرسیون

۱۶- کاربرد اصلی نمودار موجودیت – رابطه (ERD)، کدام است؟ (مهندسی IT – دولتی ۹۶)

۱) طراحی موارد آزمون در آزمون جریان داده‌ها
۲) طراحی معماری لایه‌ای
۳) طراحی پایگاه داده‌ها
۴) طراحی روابط مولفه‌ها

۱۷- کدام عبارت در مورد آزمون آلفا (Alpha Testing)، درست نیست؟ (مهندسی IT – دولتی ۹۷)

۱) توسط کاربران انجام می‌شود.

۲) در یک محیط کنترل شده انجام می‌شود.
۳) در غیاب ایجادکننده سیستم انجام می‌شود.
۴) برای اعتبارسنجی (Validation) انجام می‌شود.

۱۸- کدام‌یک از آزمون‌های زیر، نوعی آزمون سیستمی (System Testing) به شمار می‌آید؟

(مهندسی IT – دولتی ۱۴۰۰)

۱) آزمون امنیت ۲) آزمون آلفا ۳) آزمون بتا ۴) آزمون واحد

پاسخ تست‌های فصل هشتم

۱- گزینه (۲) صحیح است.

توسط مورد تست (x=1, y=6) گره ۶ ملاقات نمی‌شود. زیرا:

$x = ۱, y = ۶ \Rightarrow x < y \Rightarrow$ گره ۴ اجرا می‌شود و $z = ۵$ می‌شود

اما دستور while درست نبوده و گره ۶ اجرا نمی‌شود.

توسط مورد تست (x=25 , y=10) گره ۴ و ۶ ملاقات نمی‌شود. زیرا:

$x = ۲۵ , y = ۱۰ \quad x \not< y \Rightarrow$

while اجرا می‌شود اما شرط while درست نبوده و گره ۶ ملاقات نمی‌شود.

نتیجه اینکه گره ۶ در هیچ‌یک از موردهای تست مطرح‌شده ملاقات نمی شود.

توسط مورد تست (x=12, y=1) همه گره‌های برنامه ملاقات می شوند.

۲- گزینه (۱) صحیح است.

تست یک محصول نرم‌افزاری از دو دیدگاه باید مورد بررسی قرار گیرد:

۱- صحت (verificaion)

این دیدگاه به دیدگاه صحت معروف است که در اینجا روند ساخت یا درستی ساخت در سطح هر یک از مولفه‌ها مستقل از مولفه‌های دیگر برنامه مورد ارزیابی قرار می‌گیرد.

در صحت، **درستی عملکرد یا روند اجرای** یک بخش، یک جزء یا یک مولفه از نرم‌افزار مستقل از سایر مولفه‌های دیگر برنامه مورد ارزیابی قرار می‌گیرد.

در این دیدگاه پرسش زیر مطرح است:

«آیا محصول را درست ساخته‌ایم؟»[1]

۲- اعتبارسنجی (validation)

این دیدگاه به دیدگاه اعتبارسنجی معروف است که در اینجا روند کار یا درستی کار در سطح اجتماع مولفه‌هایی از برنامه یا کل مولفه‌های برنامه در کنار یکدیگر مورد ارزیابی قرار می‌گیرد. تا مشخص شود نرم‌افزاری که ایجاد شده است منطبق بر نیازمندی‌های مشتری است یا خیر. به عبارت دیگر مشخص شود که آیا نرم‌افزار ایجاد شده بر اساس ورودی‌های مورد نظر مشتری، خروجی‌های مورد انتظار مشتری را برآورده می‌سازد یا خیر.

در اعتبارسنجی، **درستی عملکرد یا روند اجرای** چندین مولفه در کنار یکدیگر و یا کل مولفه‌های نرم‌افزار ایجاد شده در کنار یکدیگر به همراه نقاط اتصال مابین آن‌ها مورد ارزیابی قرار می‌گیرد.

در این دیدگاه پرسش زیر مطرح است:

«آیا محصولِ درستی ساخته‌ایم؟»[2]

---

[1] Are we building the product right?
[2] Are we building the right product?

۳- گزینه (۳) صحیح است.

پیچیدگی سیکلوماتیک یک معیار نرم‌افزاری است که میزان کمی از پیچیدگی منطقی یک مولفه نرم‌افزاری را ارایه می‌دهد. در تکنیک تست مسیر پایه، مقدار محاسبه شده برای پیچیدگی سیکلوماتیک، همان تعداد مسیرهای پایه (مستقل) یا اندازه «مجموعه پایه» یک مولفه می‌باشد.

پیچیدگی سیکلوماتیک به یکی از سه روش زیر محاسبه می‌شود:

۱- تعداد نواحی گراف جریان، متناظر با پیچیدگی سیکلوماتیک است.

۲- پیچیدگی سیکلوماتیک، V(G)، برای یک گراف جریان به صورت زیر تعریف می‌شود:

$$V(G) = E - N + ۲$$

که در آن E تعداد «یال‌ها» و N تعداد «گره‌های» موجود در گراف جریان است.

۳- پیچیدگی سیکلوماتیک، V(G) برای یک گراف جریان به صورت زیر تعریف می‌شود:

$$V(G) = P + ۱$$

که در آن P تعداد «گره‌های گزاره‌ای» موجود در گراف جریان است.

پیچیدگی سیکلوماتیک را می‌توانید با به کارگیری هر یک از الگوریتم‌های بالا محاسبه کنید:

۱- گراف جریان چهار ناحیه دارد. $(R_۴, R_۳, R_۲, R_۱)$

۲- $V(G) = ۱۱ - ۹ + ۲ = ۴$

۳- $V(G) = ۳ + ۱ = ۴$

بنابراین، پیچیدگی سیکلوماتیک گراف مطرح شده در صورت سوال برابر ۴ است.

۴- گزینه (۴) صحیح است.

تکنیک‌های روش تست جعبه سفید شامل تکنیک تست مسیر پایه، تکنیک تست ساختار کنترلی شرط، تکنیک تست ساختار کنترلی حلقه و تکنیک تست جریان داده می‌باشد.

۵- گزینه (۴) صحیح است.

شکل دیگری از تست فشار، تکنیکی موسوم به «تست حساسیت» است. در برخی شرایط که بیشتر در الگوریتم‌های ریاضی رخ می‌دهد، ممکن است دامنه بسیار کوچکی از داده‌های موجود در مرز داده‌های معتبر برای یک برنامه، باعث پردازش زیاد یا حتی نادرست یا تنزل زیاد در کارایی شود. تست حساسیت می‌کوشد تا در بازه‌های معتبر داده‌های ورودی، مقادیری را بیابد که باعث مختل شدن کارکرد نرم‌افزار می‌گردد. برای مثال فرض کنید در یک برنامه قرار است عمل تقسیم دو عدد انجام بگیرد، در تست حساسیت برای فراهم کردن شرایط سخت و دشوار برای این برنامه، می بایست مقدار مخرج را به صفر مقدار دهی نمود، اگر برنامه به‌طور خودکار و بدون دخالت انسان اقدام به صدور پیامی مبنی بر دریافت مقدار مناسب برای مخرج کسر به کاربر ارسال کند، آنگاه این بدین معنی خواهد بود که این برنامه در مدت زمان کوتاهی به طور خودکار اقدام به ترمیم خود نموده است. اما اگر جلوگیری از ورود مقدار صفر برای مخرج کسر در برنامه لحاظ نشده باشد، آنگاه ترمیم برنامه نیازمند دخالت انسان است. بنابراین تست حساسیت، حساسیت مقداری یک سیستم کامپیوتری را در یک بازه مقداری معتبر مورد بررسی قرار می‌دهد.

۶- گزینه (۲) صحیح است.

طبق قانون ۴۰-۲۰-۴۰، ۴۰٪ تلاش فرآیند تولید نرم‌افزار بر روی تحلیل و طراحی بوده و ۲۰٪ بر روی پیاده‌سازی و ۴۰٪ بر روی تست استوار است.

۷- گزینه (۴) صحیح است.

به طور کلی مراحل مرتبط با فرآیند تست نرم‌افزار صرف‌نظر از اندازه، پیچیدگی پروژه و زمینه‌ی کاربردی آن و مستقل از متدولوژی ساخت‌یافته و شیءگرا به پنج مرحله تست واحد، تست جامعیت (تست یکپارچگی یا تست تدریجی)، تست اعتبارسنجی و تست سیستم تقسیم می‌شود.

۸- گزینه (۳) صحیح است.

به طور کلی فعالیت‌های مرتبط با فرآیند تولید نرم‌افزار صرف‌نظر از اندازه، پیچیدگی پروژه و زمینه‌ی کاربردی آن و مستقل از متدولوژی ساخت‌یافته و شیءگرا به پنج فعالیت ارتباطات (مهندسی نیازمندی‌ها)، برنامه‌ریزی، مدل‌سازی (تحلیل و طراحی)، ساخت (پیاده‌سازی و تست) و استقرار تقسیم می‌شود. تغییر روایت، یعنی ورژن بعدی، که ورژن بعدی چرخه حیات مربوط به خود را دارد، مانند ساختمانی که تخریب می‌شود و مجددا ساخته می‌شود، واضح است که ساختمان جدید پس از ساخت، در چرخه حیات جدید خود قرار می‌گیرد.

۹- گزینه (۱) صحیح است.

به طور کلی انواع نگهداری نرم‌افزار به چهار دسته زیر تقسیم می‌گردد:

۱- نگهداری تصحیحی[1]

نگهداری تصحیحی به معنی تصحیح یک نیاز برنامه‌نویسی‌شده توسط سازنده و به تبع برآورده شده اما نادرست است. مانند توانایی نرم‌افزار در برداشت از حساب بدون موجودی، که این مساله در نگهداری تصحیحی باید اصلاح گردد.

۲- نگهداری تطابقی[2]

نگهداری تطابقی به معنی تطابق یک نیاز برنامه‌نویسی‌شده توسط سازنده و به تبع برآورده شده و درست است که به علت تغییرات محیط برنامه همچون تغییرات در سخت‌افزار، سیستم عامل، پایگاه داده‌ها، پروتکل‌های شبکه‌ای و قوانین محیط عملیاتی، باید به پیروی از این تغیرات نحوه برآورده‌سازی آن توسط نرم‌افزار تغییر کند. مانند تغییر محاسبه ارزش افزوده یک فروشگاه کتاب از هفت درصد به هشت درصد که این مساله در نگهداری تطابقی باید با قوانین جدید انطباق داده شود. یا مانند تغییر تکنولوژی برنامه‌نویسی یک برنامه برای تطابق با یک سیستم عامل جدید.

[1] Corrective Maintenance
[2] Adaptive Maintenance

۳- نگهداری تکمیلی[1]

نگهداری تکمیلی به معنی اضافه‌کردن یک نیاز برنامه‌نویسی‌نشده توسط سازنده اما بیان شده توسط مشتری و یا به معنی اضافه‌کردن یک نیاز جدید مشتری و به تبع برنامه‌نویسی‌نشده توسط سازنده است که به تازگی توسط مشتری بیان شده‌است و باید توسط سازنده پیاده‌سازی گردد.

۴- نگهداری پیش‌گیرانه[2]

نرم‌افزار کامپیوتری در اثر تغییرات زیاد، کارایی خود را از دست می‌دهد، از این رو «مهندسی مجدد نرم‌افزار» یا نگهداری پیشگیرانه با تاکید بر اصول مهندسی‌نرم‌افزار در سرتاسر چرخه حیات نرم‌افزار مانع از ناکارآمدی نرم‌افزار می‌گردد.

۱۰- گزینه (۲) صحیح است.

پیچیدگی سیکلوماتیک یک معیار نرم‌افزاری است که میزان کمی از پیچیدگی منطقی یک مولفه نرم‌افزاری را ارایه می‌دهد. در تکنیک تست مسیر پایه، مقدار محاسبه شده برای پیچیدگی سیکلوماتیک، همان تعداد مسیرهای پایه (مستقل) یا اندازه «مجموعه پایه» یک مولفه می‌باشد.

پیچیدگی سیکلوماتیک به یکی از سه روش زیر محاسبه می‌شود:

۱- تعداد نواحی گراف جریان، متناظر با پیچیدگی سیکلوماتیک است.

۲- پیچیدگی سیکلوماتیک، V(G)، برای یک گراف جریان به صورت زیر تعریف می‌شود:

$$V(G) = E - N + ۲$$

که در آن E تعداد «یال‌ها» و N تعداد «گره‌های» موجود در گراف جریان است.

۳- پیچیدگی سیکلوماتیک، V(G) برای یک گراف جریان به صورت زیر تعریف می‌شود:

$$V(G) = P + ۱$$

که در آن P تعداد «گره‌های گزاره‌ای» موجود در گراف جریان است.

پیچیدگی سیکلوماتیک را می‌توانید با به کارگیری هر یک از الگوریتم‌های بالا محاسبه کنید:

۱- گراف جریان چهار ناحیه دارد. $(R_۵, R_۴, R_۳, R_۲, R_۱)$

۲- $V(G) = ۱۳ - ۱۰ + ۲ = ۵$

۳- $V(G) = ۴ + ۱ = ۵$

بنابراین، پیچیدگی سیکلوماتیک گراف مطرح شده در صورت سوال برابر ۵ است.

۱۱- گزینه (۳) صحیح است.

تکنیک‌های روش تست جعبه سیاه شامل تکنیک تست بخش‌بندی معادل یا افراز هم‌ارزی، تکنیک تست تحلیل مقادیر مرزی، تکنیک تست تشابه، تکنیک تست آرایه متعامد، تکنیک تست مبتنی بر گراف و تکنیک تست مبتنی بر مدل می‌باشد.

---

[1] Perfective Maintenance
[2] Preventive Maintenance

۱۲- گزینه (۳) صحیح است.

به طور کلی مراحل مرتبط با فرآیند تست نرم‌افزار صرف‌نظر از اندازه، پیچیدگی پروژه و زمینه‌ی کاربردی آن و مستقل از متدولوژی ساخت‌یافته و شیءگرا به پنج مرحله تست واحد، تست جامعیت (تست یکپارچگی یا تست تدریجی)، تست اعتبارسنجی و تست سیستم تقسیم می‌شود.

تست واحد جهت محقق کردن اصل صحت(verificaion)، روش تست جعبه سفید و تکنیک‌های مرتبط با آنرا مورد استفاده قرار می‌دهد.

تست جامعیت جهت محقق کردن اصل اعتبارسنجی(validation)، روش تست جعبه سیاه و تکنیک‌های مرتبط با آنرا مورد استفاده قرار می‌دهد.

تست اعتبارسنجی جهت محقق کردن اصل اعتبارسنجی(validation)، روش تست جعبه سیاه و تکنیک‌های مرتبط با آنرا مورد استفاده قرار می‌دهد.

تست سیستم جهت محقق کردن اصل اعتبارسنجی(validation)، روش تست جعبه سیاه و تکنیک‌های مرتبط با آنرا مورد استفاده قرار می‌دهد.

تکنیک‌های روش تست جعبه سفید شامل تکنیک تست مسیر پایه، تکنیک تست ساختار کنترلی شرط، تکنیک تست ساختار کنترلی حلقه و تکنیک تست جریان داده می‌باشد.

تکنیک‌های روش تست جعبه سیاه شامل تکنیک تست بخش‌بندی معادل یا افراز هم‌ارزی، تکنیک تست تحلیل مقادیر مرزی، تکنیک تست تشابه، تکنیک تست آرایه متعامد، تکنیک تست مبتنی بر گراف و تکنیک تست مبتنی بر مدل می‌باشد.

گزینه اول نادرست است. زیرا، تکنیک تست حلقه مربوط به روش تست جعبه سفید است که در مرحله‌ی تست واحد مورد استفاده قرار می‌گیرد.

گزینه دوم نادرست است. زیرا، تست دود مربوط به تست جامعیت یا یکپارچه‌سازی است.

گزینه سوم درست است. زیرا، تست بازیابی مربوط به تست سیستم است. به طور کلی در تست سیستم، نرم‌افزار در شرایط سخت و دشوار مورد تست و بررسی قرار می‌گیرد، مانند تست ترمز یک اتومبیل در شرایط کویری و سخت و دشوار مانند زمان اعلام اسامی قبول‌شدگان کنکور.

گزینه چهارم نادرست است. زیرا، تکنیک تست جریان داده مربوط به روش تست جعبه سفید است که در مرحله‌ی تست واحد مورد استفاده قرار می‌گیرد.

۱۳- گزینه (۲) صحیح است.

گزینه اول نادرست است. زیرا، در تست بالا به پایین بایستی زیربرنامه‌هایی را به عنوان Stub ساخت.

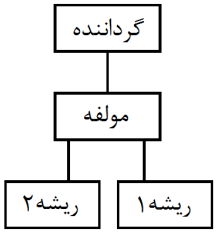
گزینه دوم درست است. زیرا، از آنجاکه در تست واحد، یک مولفه از نرم‌افزار، مستقل از مولفه‌های دیگر مورد تست قرار می‌گیرد، بنابراین برای تست یک مولفه نیاز به ایجاد مولفه گرداننده (Driver) و مولفه ریشه (Stub) می‌باشد.

تولید مولفه‌های گرداننده و ریشه به دلیل هزینه و زمان صرف شده و عدم تحویل با محصول

نهایی سربار دارد. اگر این مولفه‌ها خلاصه باشد سربار کم خواهد بود. به عبارت دیگر از آنجا که مؤلفه‌های گرداننده و ریشه در محصول نهایی کاربرد ندارند، و فقط به منظور تست واحد ایجاد گردیده‌اند، هزینه و زمان صرف شده برای ساخت آن‌ها به عنوان سربار سیستم تلقی می‌شود، حال اگر پیچیدگی آنها بالا باشد، سربار سیستم بالاتر هم خواهد رفت.

گزینه سوم نادرست است. زیرا، در تست پایین به بالا بایستی زیربرنامه‌هایی را به عنوان Driver ساخت.

به شکل زیر توجه کنید.



گرداننده (Driver)

مولفه گرداننده، جای مولفه پدر قرار می‌گیرد که حالات تست را دریافت کرده و به «مولفه مورد تست» پاس می‌دهد. سپس نتایج خروجی توسط مولفه به خروجی موردنظر منتقل می‌گردد.

ریشه (Stub)

مولفه ریشه، جایگزین مؤلفه‌های فرزند می‌شود که توسط «مولفه مورد تست»، فراخوانی می گردد.

۱۴- گزینه (۲) صحیح است.

پیچیدگی سیکلوماتیک یک معیار نرم‌افزاری است که میزان کمی از پیچیدگی منطقی یک مولفه نرم‌افزاری را ارایه می‌دهد. در تکنیک تست مسیر پایه، مقدار محاسبه شده برای پیچیدگی سیکلوماتیک، همان تعداد مسیرهای پایه (مستقل) یا اندازه «مجموعه پایه» یک مولفه می‌باشد.

پیچیدگی سیکلوماتیک به یکی از سه روش زیر محاسبه می‌شود:

۱- تعداد نواحی گراف جریان، متناظر با پیچیدگی سیکلوماتیک است.

۲- پیچیدگی سیکلوماتیک، V(G)، برای یک گراف جریان به صورت زیر تعریف می‌شود:

$$V(G) = E - N + ۲$$

که در آن E تعداد «یال‌ها» و N تعداد «گره‌های» موجود در گراف جریان است.

۳- پیچیدگی سیکلوماتیک، V(G) برای یک گراف جریان به صورت زیر تعریف می‌شود:

$$V(G) = P + ۱$$

که در آن P تعداد «گره‌های گزاره‌ای» موجود در گراف جریان است.

پیچیدگی سیکلوماتیک را می‌توانید با به کارگیری هر یک از الگوریتم‌های بالا محاسبه کنید:

۱- گراف جریان چهار ناحیه دارد. $(R_۴, R_۳, R_۲, R_۱)$

۲- $V(G) = ۱۰ - ۸ + ۲ = ۴$

۳- $V(G) = ۳ + ۱ = ۴$

بنابراین، پیچیدگی سیکلوماتیک گراف مطرح شده در صورت سوال برابر ۴ است.

۱۵- گزینه (۲) صحیح است.

صورت سوال به این شکل است:

کدام آزمون، روشی معمول برای اعتبارسنجی نرم‌افزار سفارشی (Custom Software) است؟

۱) آزمون واحد

گزینه اول پاسخ سوال نیست، زیرا به طور کلی مراحل مرتبط با فرآیند تست نرم‌افزار به چهار مرحله تست واحد، تست جامعیت(تست یکپارچگی یا تست تدریجی)، تست اعتبارسنجی و تست سیستم تقسیم می‌شود.

۲) آزمون پذیرش

گزینه دوم پاسخ سوال است، زیرا در آزمون اعتبارسنجی، هنگامی‌که یک **نرم‌افزار سفارشی** (Custom Software) تنها برای **یک** مشتری توسعه می‌یابد، «**آزمون پذیرش**» اجرا می‌شود تا مشتری را قادر به اعتبارسنجی کلیه‌ی خواسته‌ها کند.

۳) آزمون بتا

گزینه سوم پاسخ سوال نیست، زیرا اگر نرم‌افزار به عنوان محصولی توسعه می‌یابد که قرار است مشتریان **زیادی** از آن استفاده کنند، انجام «آزمون پذیرش» توسط یکایک آنها امکان‌پذیر نیست. اکثر سازندگان محصولات نرم‌افزاری از فرآیندی موسوم به تست آلفا (α) و بتا (β)، برای کشف خطاهایی استفاده می‌کنند که به نظر می‌رسد فقط کاربر نهایی قادر به یافتن آنها می‌باشد.

۴) آزمون رگرسیون

گزینه چهارم پاسخ سوال نیست. زیرا هر بار که یک مولفه جدید به مجموعه تست جامعیت تدریجی بالا به پایین اضافه می‌گردد، ممکن است نرم‌افزار به دلایل مختلفی همچون ایجاد مسیرهای جدید جریان داده، فراخوانی مولفه‌های جدید، و یا اعمال ورودی و خروجی دستخوش تغییر و تحول شود. این تغییرات ممکن است بر آنچه که قبلاً تست شده است تأثیر بگذارد و باعث بروز خطا شود. بنابراین تست مجدد لازم و ضروری است، این تست مجدد به تست بازگشت یا رگرسیون موسوم است. در این نوع تست، تعدادی از تست‌هایی که قبلاً انجام شده است مجدداً اجرا می‌شوند تا اطمینان حاصل شود که تغییرات باعث وقوع خطا شده است یا خیر.

۱۶- گزینه (۳) صحیح است.

صورت سوال به این شکل است:

کاربرد اصلی نمودار موجودیت – رابطه (ERD)، کدام است؟

۱) طراحی موارد آزمون در آزمون جریان داده‌ها

گزینه اول پاسخ سوال نیست، زیرا تکنیک آزمون جریان داده به کنترل خطای متغیرها در هنگام

تعریف و استفاده از آن‌ها می‌پردازد. بنابراین مسیرهای این روش تست می‌بایست بر مبنای محل تعریف (definition) متغیر مورد نظر و محل استفاده (use) متغیر مورد نظر در مولفه مورد تست تعریف گردد. زیرا متغیرها در قسمتی از مولفه مورد نظر تعریف می‌شوند و در بخشی دیگر از همان مولفه مورد استفاده قرار می‌گیرند. برای‌مثال متغیر x در گره اول مولفه‌ای از برنامه به شکل int x=2 تعریف می‌شود و در گره سوم همان مولفه در یک عبارت شرطی به شکل x<y مورد استفاده قرار می‌گیرد. بنابراین برای ایجاد مسیرهای تست جریان داده، کافیست مسیرهایی انتخاب شود که به ازای هر گره تعریف یک متغیر، گره استفاده از متغیر مورد نظر را نیز در خود جای دهد. به بیان دیگر مسیر تست جریان داده، می‌بایست گره تعریف متغیر و استفاده از متغیر مورد نظر را در مسیر پیمایش خود ملاقات کند. ایده‌ی اصلی تست جریان داده بر این پایه استوار است که خطاهای مربوط به متغیرها، یا در زمان تعریف یا در زمان استفاده از متغیرها رخ می‌دهد. و تست جریان داده هر دو حالت این خطاها را مورد تست قرار می‌دهد. از آنجا که تکنیک تست جریان داده از تعریف (definition) و استفاده (use) متغیر برای ساخت مسیر تست جریان داده بهره می‌برد، به آن تکنیک DU Testing نیز گفته می‌شود.

تکنیک تست جریان داده، جهت حصول اطمینان از نبود خطا در ساختار جریان داده‌ها، همچون کلیه متغیرهای مولفه، ساختمان داده‌های مولفه، ورودی‌ها و خروجی‌های مولفه، در مولفه مورد نظر مورد استفاده قرار می‌گیرد.

۲) طراحی معماری لایه‌ای

گزینه دوم پاسخ سوال نیست، زیرا مربوط به طراحی معماری است و نه طراحی داده.

۳) طراحی پایگاه داده‌ها

گزینه سوم پاسخ سوال است، زیرا پس از جمع‌آوری لیست نیازمندی‌های مشتری در فعالیت ارتباطات نوبت به مدل تحلیل (مدل‌سازی لیست نیازمندی‌های مشتری) می‌رسد. مدل‌سازی که فعالیتی فنی به شمار می‌رود نیازمندی‌ها را باید به گونه‌ای مدل نماید که برای سازنده و مشتری قابل فهم باشد. مدل تحلیل به روش ساخت‌یافته از سه بخش مدل‌سازی داده‌ای، مدل‌سازی عملکردی و مدل‌سازی رفتاری تشکیل شده است، مدل‌سازی داده‌ای شامل تحلیل موجودیت‌ها و تحلیل پرس و جوها می‌باشد. تحلیل موجودیت‌ها توسط ابزار مدل ER و تحلیل پرس و جوها توسط ابزار حساب رابطه‌ای مدل می‌شوند، مدل‌سازی عملکردی توسط ابزار DFD و مدل‌سازی رفتاری توسط ابزار STD مدل می‌شود.، ERD یا Entity Relationship Diagram یا نمودار جریان داده جهت مدل‌سازی داده‌ای مورد استفاده قرار می‌گیرد. پس از مدل تحلیل، نوبت به مدل طراحی می‌رسد، مدل طراحی به روش ساخت‌یافته شامل چهار بخش **طراحی‌داده**، **طراحی‌معماری**، **طراحی‌مؤلفه** و **طراحی‌واسط** می‌باشد. طراحی داده **(طراحی پایگاه داده‌ها)** بر دو بخش طراحی جدول و طراحی پرس و جو می‌باشد. طراحی جدول از بخش طراحی داده، تحلیل موجودیت (ERD) از مدل تحلیل را به عنوان ورودی دریافت کرده و توسط مدل رابطه‌ای، **طراحی جدول** را انجام می‌دهد. طراحی پرس و جو از بخش طراحی داده، تحلیل پرس و جو از مدل تحلیل را به

عنوان ورودی دریافت کرده و توسط جبر رابطه‌ای، طراحی پرس و جو را انجام می‌دهد.

۴) طراحی روابط مولفه‌ها

گزینه چهارم پاسخ سوال نیست. زیرا طراحی معماری، تحلیل عملکرد (DFD) از مدل تحلیل را به عنوان ورودی دریافت کرده و توسط یکی از سبک‌های معماری (مانند فراخوانی و بازگشت)، طراحی معماری را انجام می‌دهد.

طراحی معماری یا معماری نرم‌افزار، ساختار کلی نرم‌افزار و شیوه‌های یکپارچگی یک سیستم را بیان می‌کند. به عبارت دیگر، ساختار سلسله مراتبی **مولفه‌های برنامه (توابع یا پیمانه‌ها) یعنی روابط مابین مولفه‌ها**، شیوه تعامل مولفه‌ها با یکدیگر و **ساختمان داده‌های** مورد نیاز مولفه‌ها را نشان می‌دهد. معماری نرم‌افزار یک مدل قابل درک از چگونگی سازمان‌دهی سیستم است. در واقع نشان‌گر ساختمان داده‌ها و مؤلفه‌های برنامه‌ای است که برای ساختن یک سیستم کامپیوتری لازم است. به طور دقیق‌تر معماری نرم‌افزار شامل دو سطح از طراحی می‌باشد یعنی طراحی داده و طراحی معماری. در واقع این ساختار مانند یک نقشه ساختمان، مبنای ساخت نرم‌افزار قرار می‌گیرد.

در طراحی معماری، اسکلت، ساختار و چیدمان کلی مولفه‌های (توابع) برنامه به این معنی که چه مولفه‌ای (تابعی) چه مولفه‌ای (تابعی) دیگر را صدا می‌زند، بدون ذکر جزئیات داخلی مولفه‌ها (توابع) مشخص می‌گردد (ساختار درختی برنامه بدون ذکر جزئیات مولفه‌ها (توابع)). مانند اسکلت یک ساختمان که گویای جایگاه مؤلفه‌های ساختمان است اما هنوز آجرچینی نشده است. (اسکلت یک ساختمان بدون آجرچینی).

طراحی مؤلفه، طراحی معماری از همان فعالیت مدل طراحی را به عنوان ورودی دریافت کرده و طراحی مؤلفه را توسط ابزارهایی همچون شبه کد یا فلوچارت ایجاد می‌کند.

طراحی مولفه، فعالیت تبدیل طراحی معماری به نرم‌افزار است. در این مرحله، سطح انتزاع طراحی معماری به سطح انتزاع نرم‌افزار کاربردی نزدیک می‌گردد. طراحی در سطح مؤلفه‌ها، نرم‌افزار را در سطحی از انتزاع تصویر می‌کند که به کد نزدیک است. طراحی مولفه، به عنوان نقشه راهی دقیق، و نزدیک به زبان پیاده‌سازی، در فعالیت پیاده‌سازی نرم‌افزار، منجر به صرفه جویی در زمان و هزینه‌های تولید می‌گردد. در طراحی مولفه، مهندس نرم‌افزار باید ساختمان داده‌ها، واسط‌ها و الگوریتم‌ها را با جزییات کافی به نمایش در آورد تا راهنمای تولید کد منبع زبان برنامه‌نویسی باشد. طراحی واسط یا همان واسط کاربر، براساس ورودی‌ها و خروجی‌های مورد نیاز کاربران نهایی به شکل نقشی بر روی کاغذ یا طرحی بر روی کامپیوتر ایجاد می‌گردد. مانند نحوه چیدمان منوها و فرم‌ها.

۱۷- گزینه (۳) صحیح است.

صورت سوال به این شکل است:

کدام عبارت در مورد آزمون آلفا (Alpha Testing)، درست نیست؟

به طور کلی مراحل مرتبط با فرآیند تست نرم‌افزار صرف‌نظر از اندازه، پیچیدگی پروژه و

زمینه‌ی کاربردی آن و مستقل از متدولوژی ساخت‌یافته و شیءگرا به چهار مرحله تست واحد، تست جامعیت (تست یکپارچگی یا تست تدریجی)، تست اعتبارسنجی و تست سیستم تقسیم می‌شود.

تست اعتبارسنجی(Validation Testing)، در پایان تست جامعیت یا یکپارچگی آغاز می‌شود، یعنی زمانی‌که اجتماع کل مولفه‌های برنامه در کنار یکدیگر قرار گرفتند و نرم‌افزار به طور کامل مونتاژ شد و خطاهای واسط و نقاط اتصال مابین مولفه‌ها کشف و برطرف شدند.

تست اعتبارسنجی، روند کار یا درستی کار را در سطح اجتماع کل مولفه‌های برنامه در کنار یکدیگر در شرایط **سهل و آسان** مورد تست و ارزیابی قرار می‌دهد. تا مشخص شود نرم‌افزاری که ایجاد شده است منطبق بر نیازمندی‌های مشتری است یا خیر. به عبارت دیگر مشخص شود که آیا نرم‌افزار ایجاد شده بر اساس ورودی‌های مورد نظر مشتری **در** شرایط **سهل** و **آسان**، خروجی‌های مورد انتظار مشتری را برآورده می‌سازد یا خیر.

به بیان دیگر تست اعتبارسنجی، اعتبارسنجی عملکرد یا روند اجرای کل مولفه‌های نرم‌افزار ایجاد شده در کنار یکدیگر را در شرایط **سهل و آسان** مورد تست و ارزیابی قرار می‌دهد.

مثال: تست اعتبارسنجی مانند تست قطعات (مولفه‌های) یک خودرو توسط یک خودروساز به شکل کامل در کنار یکدیگر در شرایط سهل و آسان داخل محیط کارخانه خودروساز بدون لحاظ کردن شرایط سخت و دشوار همچون شرایط بارانی و کویری است.

توجه: تست اعتبارسنجی جهت محقق کردن اصل اعتبارسنجی (validation)، روش تست جعبه سیاه و تکنیک‌های مرتبط با آنرا مورد استفاده قرار می‌دهد.

توجه: در این مرحله کلیه‌ی موارد پیاده‌سازی شده، براساس لیست نیازمندی‌های وظیفه‌مندی و غیروظیفه‌مندی مشتری (چک لیست) که در فعالیت ارتباطات تهیه و در مدل تحلیل و طراحی مدل‌سازی شده است مورد وارسی قرار می‌گیرد تا مشخص شود نرم‌افزار ایجاد شده براساس ورودی‌های مورد نظر مشتری، خروجی‌های مورد انتظار مشتری را برآورده می‌سازد یا خیر. در انتهای این تست یک لیست از نیازمندی‌های مشتری که برآورده نشده است تهیه می‌گردد تا رفع گردند. هنگامی‌که یک **نرم‌افزار سفارشی** (Custom Software) تنها برای یک مشتری توسعه می‌یابد، «**آزمون پذیرش**[1]» اجرا می‌شود تا مشتری را قادر به اعتبارسنجی کلیه‌ی خواسته‌ها کند. آزمون پذیرش، که به جای مهندس نرم‌افزار، توسط مشتری انجام می‌شود، می‌تواند از یک آزمون غیررسمی تا یک سری آزمون‌های برنامه‌ریزی شده‌ی سیستماتیک را شامل شود. درواقع، آزمون پذیرش را می‌توان در عرض یک هفته یا یک ماه انجام داد و لذا کشف خطاهایی که ممکن است سیستم را با گذشت زمان تنزل دهد، امکان‌پذیر می‌شود. اگر نرم‌افزار به عنوان محصولی توسعه می‌یابد که قرار است مشتریان زیادی از آن استفاده کنند، انجام «آزمون پذیرش» توسط یکایک آنها امکان‌پذیر نیست. اکثر سازندگان محصولات نرم‌افزاری از فرآیندی موسوم به تست آلفا (**α**) و

[1] Acceptance Test

بتا (β)، برای کشف خطاهایی استفاده می‌کنند که به نظر می‌رسد فقط کاربر نهایی قادر به یافتن آنها می‌باشد.

تست آلفا (α)

تست آلفا در مکان سازنده نرم‌افزار و توسط مشتری انجام می‌شود. بدین صورت که مشتری در یک محیط کنترل‌شده توسط سازنده، کنار سازنده می‌نشیند و نرم‌افزار را تست می‌کند. سازنده نیز همزمان خطاهای موجود را یادداشت می‌کند.

تست بتا (β)

تست بتا در مکان مشتری و توسط یک یا چند کاربر نهایی انجام می‌شود. برخلاف تست آلفا، سازنده معمولاً حضور ندارد و سیستم در محیط واقعی خود مستقر می‌باشد. بنابراین، تست بتا، یک کاربرد «زنده» از نرم‌افزار در محیطی است که سازنده قادر به کنترل آن نیست. مشتری کلیه مشکلات (واقعی یا تصوری) را که طی تست بتا یافته است، یادداشت می‌کند و آنها را در فاصله‌های زمانی منظم به سازنده گزارش می‌دهد. مهندسان نرم‌افزار، با توجه به مشکلات گزارش شده طی تست بتا، اصلاحات لازم را انجام می‌دهند و سپس محصول نرم‌افزاری نهایی را برای ارایه به مشتریان آماده می‌کنند.

شکل دیگری از تست بتا، که به «**آزمون پذیرش مشتری**[1]» موسوم است، گاهی اجرا می‌شود، یعنی زمانی‌که یک **نرم‌افزار سفارشی** تحت **قرارداد** به مشتری تحویل داده می‌شود. مشتری یک سری تست‌های مشخص انجام می‌دهد تا خطاها را قبل از پذیرفتن نرم‌افزار از سازنده آن کشف کند.

۱) توسط کاربران انجام می‌شود.

گزینه اول پاسخ سوال نیست، زیرا آزمون آلفا (**Alpha Testing**)، در مکان سازنده نرم‌افزار و توسط مشتری انجام می‌شود.

۲) در یک محیط کنترل شده انجام می‌شود.

گزینه دوم پاسخ سوال نیست، زیرا آزمون آلفا (**Alpha Testing**)، در مکان سازنده نرم‌افزار و توسط مشتری انجام می‌شود. بدین صورت که **مشتری در یک محیط کنترل‌شده** توسط سازنده، کنار سازنده می‌نشیند و نرم‌افزار را تست می‌کند. سازنده نیز همزمان خطاهای موجود را یادداشت می‌کند.

۳) در غیاب ایجادکننده سیستم انجام می‌شود.

گزینه سوم پاسخ سوال است. زیرا آزمون آلفا (**Alpha Testing**)، در مکان سازنده نرم‌افزار و توسط مشتری انجام می‌شود. بدین صورت که مشتری در یک محیط کنترل‌شده توسط سازنده،

[1] Customer Acceptance Testing

کنار سازنده می‌نشیند و نرم‌افزار را تست می‌کند. سازنده نیز همزمان خطاهای موجود را یادداشت می‌کند.

۴) برای اعتبارسنجی (Validation) انجام می‌شود.

گزینه چهارم پاسخ سوال نیست. زیرا اکثر سازندگان محصولات نرم‌افزاری از فرآیندی موسوم به تست آلفا (**α**) و بتا (**β**)، برای کشف خطاهایی استفاده می‌کنند که به نظر می‌رسد فقط کاربر نهایی قادر به یافتن آنها می‌باشد. تست آلفا (**α**) و بتا (**β**) در مرحله تست **اعتبارسنجی (Validation)** انجام می‌شود.

۱۸- گزینه (۱) صحیح است.

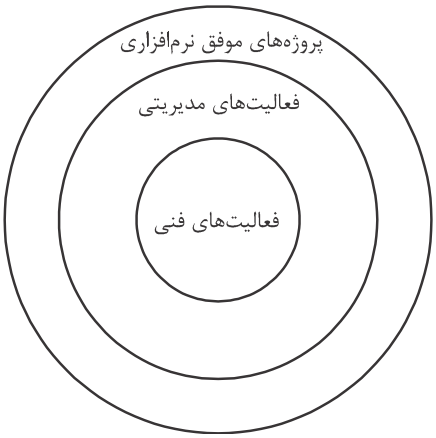
به طور کلی جهت تست سیستم شش شیوه «تست بازیابی»، «تست امنیت»، «تست فشار»، «تست حساسیت»، «تست کارایی» و «تست استقرار» وجود دارد.

---

۹ مدیریت پروژه‌های نرم‌افزاری

مقدمه

شرط لازم برای موفقیت پروژه‌های نرم‌افزاری، انجام مناسب فعالیت‌های فنی و شرط کافی برای موفقیت پروژه‌های نرم‌افزاری، انجام مناسب فعالیت‌های مدیریتی است. شکل زیر گویای مطلب است:



مدیریت پروژه شامل برنامه‌ریزی، نظارت و کنترل افراد، فرآیند و ریسک‌هایی می‌شود که به موازات تکامل نرم‌افزار از یک مفهوم مقدماتی به استقرار عملیاتی کامل رخ می‌دهد. هر شخصی در فرآیند تولید نرم‌افزار حوزه‌ی مدیریتی خاص خود را دارد. ولی دامنه فعالیت‌های مدیریتی یک شخص بسته به جایگاه مدیریتی وی تغییر می‌کند. مهندس نرم‌افزار، فعالیت‌های روزانه، برنامه‌ریزی، نظارت و وظایف فنی کنترلی خود را مدیریت می‌کند. مدیران پروژه، کار تیم نرم‌افزاری را برنامه‌ریزی، نظارت و سازماندهی می‌کنند و مدیران ارشد، ارتباط میان مدیران پروژه و مدیران تجاری را هماهنگ می‌کنند. ساخت نرم‌افزار کامپیوتری کاری پیچیده است، به ویژه اگر

شامل تعداد بسیاری از افراد شود که در مدتی نسبتا طولانی مشغول به کار باشند، از همین رو است که پروژه‌ها نیاز به مدیریت دارند. با شروع فعالیت‌های مدیریتی، یک برنامه‌ریزی پروژه ایجاد می‌شود. در این برنامه‌ریزی، فرآیند و وظایفی که قرار است اجرا شود، افرادی که کار را انجام می‌دهند و سازوکارهایی برای ارزیابی ریسک‌ها، کنترل تغییرات و ارزیابی کیفیت تعریف می‌شود.

مایلر پیج جونز در مقدمه‌ی کتاب خود درباب مدیریت پروژه‌های نرم‌افزاری، بیانیه‌ای دارد که انعکاس آن را در گفتار بسیاری از مشاوران مهندسی نرم‌افزار می‌توان یافت.

من از ده‌ها بنگاه تجاری، چه خوب و چه بد، دیدن کرده‌ام و کار ده‌ها مدیر داده‌پردازی (باز هم چه خوب و چه بد) را زیر نظر گرفته‌ام. و با حیرت، شاهد تقلای بیهوده‌ی این مدیران در پروژه‌های کابوس‌وار بوده‌ام که یا زیر فشار مهلت‌های غیرممکن خرد شده‌اند یا سیستم‌هایی را تحویل داده‌اند که کاربران را دچار جنون می‌کنند و در ادامه هم باید زمان بسیاری را صرف نگهداری آن‌ها کرد.

آنچه که پیج جونز توصیف می‌کند، نشانگانی است که از یک مجموعه مشکلات فنی و مدیریتی پدید می‌آید. به هرحال، اگر قرار بود همه پروژه‌های ناموفق نرم‌افزاری مورد کالبد شکافی قرار گیرند، به احتمال زیاد یک زمینه‌ی مشترک در آنها به چشم می‌خورد، «**ضعف مدیریت پروژه**».

طیف مدیریتی

مدیریت پروژه‌های نرم‌افزاری در چهار بخش افراد[1]، محصول[2]، فرآیند[3] و پروژه[4] انجام می‌گردد. ترتیب بخش‌های مطرح شده داری اهمیت است و اختیاری نیست. در مدیریت مؤثر پروژه‌های نرم‌افزاری، بر چهار مورد زیر تأکید می‌شود، به عبارت دیگر در مدیریت پروژه‌های نرم‌افزاری چهار بخش زیر باید به شکلی کارآمد و مناسب مدیریت گردد:

۱- افراد

در یک بیان ساده مدیریت افراد یعنی شناخت و ارتقاء افراد مناسب. عامل انسانی یکی از مهمترین حوزه‌های مدیریت پروژه است. پرورش منابع انسانی نرم‌افزاری پرانگیزه و با مهارت بالا از دهه‌ی ۱۹۶۰ مورد بحث بوده است. مدیری که فراموش کند کار مهندسی نرم‌افزار، یک تلاش کاملا انسانی است، هرگز در مدیریت پروژه‌های نرم‌افزاری موفق نخواهد شد. یک مدیر پروژه باید بتواند با انتخاب افراد شایسته، تیم تولید نرم‌افزار را به عالی‌ترین شکل ممکن سازماندهی کند. همچنین یک مدیر پروژه باید بتواند علاوه بر انتخاب افراد شایسته، در طول پروژه انگیزه لازم را برای انجام کار در افراد ایجاد کرده و ارتباطات و وظایف آنها را نیز سازماندهی کند. عامل «منابع

[1] People
[2] Product
[3] Process
[4] Project

انسانی» چنان اهمیت دارد که «مؤسسه مهندسی نرم‌افزار[1]»، «مدل بلوغ قابلیت‌های انسانی[2]» را جهت انتخاب و ارتقا منابع انسانی معرفی کرده‌است. در این مدل بر مواردی همچون، توانایی فردی، روحیه همکاری و رقابت، نحوه آموزش، خصوصیات محیط کار و مدیریت بهره‌وری افراد تاکید شده‌است. نتیجه اینکه مدیر پروژه جهت موفقیت در مدیریت مناسب پروژه‌های نرم‌افزاری باید مدام و به طور پیوسته در جهت جذب، ایجاد انگیزه و ارتقاء عامل انسانی سازمان نرم‌افزاری کوشا باشد. تیم پروژه برای اینکه موثر واقع شود، باید به گونه‌ای سازماندهی شود که مهارت‌ها و توانایی‌های تک تک افراد را به حداکثر برساند. و این وظیفه رهبر تیم است.

طی مطالعه‌ای که توسط IEEE منتشر شده‌است، از معاونان مهندسی سه شرکت فناوری بزرگ پرسیده شد که مهمترین عامل در موفقیت یک پروژه‌ی نرم‌افزاری چیست. پاسخ آنها به شرح زیر بود:

معاون اول: به گمان اگر قرار باشد یک چیز را به عنوان مهمترین عامل در محیط خودمان انتخاب کنیم، می‌گویم آن یک چیز ابزارها نیست، بلکه افراد است.

معاون دوم: مهمترین عامل موفقیت ما در این پروژه، داشتن افراد زرنگ بود...به نظر من موارد دیگر خیلی اهمیت ندارند... مهمترین کاری که برای پروژه باید انجام بدهید، گزینش منابع انسانی مناسب است... موفقیت سازمان تولید نرم‌افزار، خیلی، خیلی به توانایی جمع‌کردن افراد خبره بستگی دارد.

معاون سوم: تنها قانونی که من دارم، حصول اطمینان از داشتن افراد نخبه و خبره است. و اینکه افراد خبره تربیت کنم و اینکه محیطی فراهم کنم که افراد مناسب در آن بتوانند به کار و تولید مشغول شوند.

در واقع، این یک بیانیه قانع‌کننده درباره اهمیت افراد در فرآیند تولید نرم‌افزار است. با این حال، همه ما از معاون شرکت، که مهندس ارشد است، تا دون‌پایه‌ترین کارمندان، اغلب آنگونه که باید، قدردان منابع انسانی نیستیم. مدیران چنین استدلال می‌کنند که پرسنل در درجه اول اهمیت قرار دارند، ولی آنچه در عمل انجام می‌دهند، خلاف گفته‌هایشان است. در این بخش به بررسی ذی‌نفع‌هایی[3] خواهیم پرداخت که در فرآیند تولید نرم‌افزار مشارکت دارند.

۱- **مدیران ارشد[4]:** به تعریف اهداف تجاری پروژه می‌پردازند و تاثیر چشمگیری بر نتایج پروژه دارند.

۲- **مدیران پروژه[5]:** برنامه‌ریزی، ایجاد انگیزه و سازماندهی متخصصین تیم نرم‌افزاری بر عهده مدیران فنی پروژه است. نحوه سازماندهی تیم نرم‌افزاری جلوتر شرح داده می‌شود.

---

[1] Software Engineering Institute (SEI)
[2] People Capability Maturity Model (People-CMM)
[3] Stakeholders
[4] Senior Managers
[5] Project Managers

۳- **متخصصین**[1]**:** اجرای فرآیند تولید نرم‌افزار بر عهده متخصصین تیم نرم‌افزاری است. افرادی که با داشتن مهارت‌های فنی لازم، قادر به پیاده‌سازی پروژه نرم‌افزاری خواهند بود.

۴- **مشتری**[2]**:** تعیین نیازمندی‌های پروژه نرم‌افزاری بر عهده مشتری است.

۵- **کاربران نهایی**[3]**:** استفاده نهایی از نرم‌افزار توسط کاربران نهایی صورت می‌گیرد.

توجه: سه گروه اول در سمت بخش تولید نرم‌افزار و دو گروه دوم در سمت کارفرما قرار دارد.

توجه: عدم توجه کافی به مدیریت افراد، ریسک مدیریت افراد را ایجاد خواهد کرد.

رهبران تیم

یکی از احساسات واقعاً رضایت بخش اینست که یک پروژه را در زمینه کاری خود با موفقیت رهبری کرده و به پایان برسانید. علاوه بر پاداش‌هایی که موفقیت در پروژه، بر سر شما و افراد زیردستتان سرازیر می‌کند، این امر برای شما کارنامه‌ای عالی رقم زده و امتیاز و احترام زیادی را در بین همکاران برایتان به همراه می‌آورد. با وجود اینکه رهبری یک گروه مزایای خود را دارد، اما نباید فراموش کرد که قدرت بیشتر مسئولیت بیشتری نیز با خود می‌آورد. زیرا این شما هستید که باید تیم را مدیریت و رهبری کنید بدین معنا که میان افراد هماهنگی، ارزیابی و کنترل و تلاش را اشاعه دهید. و در زمان مقتضی توجه افراد را به هدف، گاهی زمان و گاهی نتیجه عملکرد معطوف کنید. مدیریت مناسب باعث تقویت انگیزه و اطمینان اعضای تیم گشته و موجب اطمینان خاطر آنها جهت تلاش، صرف زمان و انرژی روی اهداف می‌شود.

مدیریت پروژه یک فعالیت مبتنی بر خصوصیات ذاتی مدیریتی است. در واقع نمی‌توان گفت یک فرد تنها به صرف داشتن دانش فنی می‌تواند مدیر کارآمدی هم باشد. از این جمله این مطلب استنباط می‌گردد که یک فرد خبره به لحاظ داشتن دانش فنی می‌تواند به عنوان یک شخصیت متخصص فعالیت فنی داشته باشد، اما اگر همین فرد خصوصیات ذاتی مدیریتی را دارا نباشد، برای مدیریت پروژه شخصیت مناسبی نخواهد بود.

جری واینبرگ[4] سه ویژگی مهم رهبران تیم را به صورت زیر بیان می‌کند:

۱- **انگیزه**[5]: رهبر تیم باید این توانایی را داشته باشد تا بتواند بالاترین میزان انگیزه را در افراد تیم ایجاد کند، تا توانایی افراد در بالاترین سطح ممکن آشکار گردد.

۲- **سازماندهی**[6]: رهبر تیم باید این توانایی را داشته باشد تا بتواند، تیم را به گونه‌ای هدایت

---

[1] Practitioners
[2] Customers
[3] End User
[4] Jerry Weinberg
[5] Motivation
[6] Organization

و سازماندهی کند، تا هدف پروژه از یک مفهوم انتزاعی به یک محصول عملیاتی بدل گردد.

۳- **ایده‌ها یا نوآوری**[1]: رهبر تیم باید این توانایی را داشته باشد تا بتواند، بالاترین میزان خلاقیت را در افراد تیم ایجاد کند، تا توانایی افراد در خلاقیت و نوآوری در بالاترین سطح ممکن آشکار گردد.

واینبرگ معتقد است که رهبران پروژه‌های موفق از شیوه‌های حل مساله برای مدیریت استفاده می‌کنند. یعنی، مدیر پروژه نرم‌افزاری باید توجه خود را به درک مساله‌ای که قرار است حل شود، معطوف کند، جریان ایده‌ها را مدیریت کند و در عین حال همه افراد تیم را آگاه کند (در گفتار و بسیار مهم‌تر از آن، در عمل) که کیفیت، حرف اول را می‌زند و جای سهل‌انگاری ندارد.

به قول حماسه سرای نامی ایران، فردوسی:

«بزرگی سراسر به گفتار نیست دو صد گفته چون نیم کردار نیست»

تیم نرم‌افزاری

منظور از تیم نرم‌افزاری یا سازمان اجرایی پروژه، تخصیص و نحوه چیدمان نیروی انسانی موجود برای تولید نرم‌افزار است. سازماندهی‌های مختلفی برای تیم‌های نرم‌افزاری وجود دارد. اینکه کدام ساختار برای برای پروژه بهتر می‌باشد، کاملا بستگی به مساله‌ای دارد که با آن روبرو هستیم. در واقع اینکه چه ساختار تیمی برای پروژه مناسب‌تر است، وابسته به شرایط مختلفی است که در زیر به آنها اشاره کرده‌ایم:

- میزان دشواری مساله‌ای که قرار است حل شود.
- میزان کار محاسبه شده توسط برآوردها برحسب تعداد خطوط کد (KLOC) یا نقاط عملکرد (FP)
- مدت زمان کنار هم ماندن اعضای تیم (طول عمر تیمی)
- میزان قابلیت پیمانه‌ای کردن مساله
- کیفیت و قابلیت اطمینان مورد نیاز برای سیستمی که قرار است ساخته شود.
- مهلت زمانی برای تحویل پروژه
- میزان ارتباطات میان بخش‌های مختلف پروژه

الگوهای سازماندهی[2]

انتخاب مناسب نوع سازماندهی بر اساس شرایط پروژه، کارآمدی نحوه سازماندهی را به ارمغان خواهد آورد. به طور کلی دیدگاه کنستانتین برای سازماندهی پروژه‌های نرم‌افزاری وجود

[1] Ideas or Innovation
[2] Organizational Paradigms

دارد، که در ادامه به بررسی آن می‌پردازیم:

دیدگاه کنستانتین[1]

در دیدگاه کنسانتین چهار ساختار تیمی مختلف برای سازماندهی تیم‌های نرم‌افزاری بیان شده است.

۱- الگوی تصادفی[2]

در گذشته به این الگو، **مدل** غیرمتمرکز دموکراتیک (Democratic Decentralized-DD) نیز گفته می‌شد. غیرمتمرکز است چون خرد جمعی و ارتباطات میان اعضای گروه وجود دارد، دموکراتیک است زیرا مدیر و رئیس دائمی ندارد. در این مدل، تیم ساختار ضعیفی دارد و نتایج کار به خلاقیت و توانایی تک تک اعضای تیم وابسته است. این روش زمانی کاربرد دارد که نیاز به استفاده از خلاقیت و توانایی ذهنی همه اعضای گروه و ایجاد یک خرد جمعی برای **حل یک مساله دشوار** باشد و نه **تصمیم‌گیری و توافقات**. به دلیل عدم وجود یک مدیر و به تبع عدم کنترل اعضای تیم، در صورتی که کارکردی منظم موردنیاز باشد، چنین تیمی کارآمد نخواهد بود.

در این ساختار، مدیر و رئیس ثابت و دائمی برای پروژه وجود ندارد. در عوض افرادی برای مدت کوتاه به عنوان هماهنگ‌کننده کار[3] و نه مدیر برای دوره‌های کوتاه زمانی منتسب می‌شوند، سپس پس از اتمام دوره جای خود را به یکی دیگر از اعضای تیم می‌دهند. ارتباط بین اعضای تیم به صورت افقی (هم‌سطح) می‌باشد یعنی نگاه بالا به پایین میان اعضای تیم وجود ندارد به عبارت دیگر اعضای تیم به یکدیگر برتری مقامی ندارند و تصمیمات در مورد مسائل و مشکلات براساس توافق و خرد جمعی اتخاذ می‌شود. فرد هماهنگ‌کننده به هیچ‌عنوان رفتار مدیر و رئیس را نخواهد داشت و صرفا وظیفه هماهنگی اعضای تیم را بر عهده دارد. در طول کار بنا به شرایط کار و تخصص اعضای گروه، یک هماهنگ کننده متناسب با کارهای پیش‌رو انتخاب می‌گردد و هماهنگی کارهای دیگری را بر عهده می‌گیرد.

این روش برای پروژه‌هایی مناسب است که یا حل مساله آن **بسیار پیچیده و دشوار** است یا نمونه‌های مشابه آن قبلاً توسط همین تیم انجام نشده است. زیرا یافتن راه حل مساله توسط همه اعضای تیم انجام می‌شود. در واقع در مواقعی که پروژه نیاز به راه حل‌های خلاقانه و نوآوری دارد، الگوی تصادفی می‌تواند بسیار کارآمد باشد، در واقع در این روش احتمال موفقیت برای نوآوری کارهای جدید بیشتر است. الگوی تصادفی برای یافتن راه حل مساله به دلیل وجود خرد جمعی مناسب و برای توافقات و تصمیم‌گیری به دلیل عدم کنترل مدیران نامناسب است.

از آنجاکه در این روش راه حل مساله توسط همه اعضای تیم بررسی می‌شود، و خرد جمعی و

---

[1] Constantine
[2] Random Paradigm
[3] Task Coordinator

ارتباطات میان اعضای تیم بسیار زیاد است، بنابراین کارکرد تیم در انجام کارها کند می‌باشد.

۲- الگوی بسته[1]

در گذشته به این الگو، **مدل متمرکز کنترل‌شده** (Controlled Centralized-CC) یا **برنامه‌نویس ارشد** (Chief Programmer) نیز گفته می‌شد. برنامه‌نویس ارشد همان مدیر تیم بود. متمرکز است چون خرد جمعی و ارتباطات میان اعضای تیم بسیار ناچیز است، کنترل شده است زیرا مدیر و رئیس دائمی دارد. در این مدل، تیم در راستای یک سلسله مراتب سنتی از مسئولیت‌ها سازمان‌دهی می‌شود. برنامه‌ریزی و هماهنگ‌سازی داخلی تیم و حل مشکلات سطح بالا بر عهده‌ی مدیر تیم است. و انجام کارهای فنی بر عهده اعضای تیم است. ارتباط بین مدیر و اعضای تیم به صورت عمودی است.

به دلیل وجود یک مدیر و به تبع کنترل اعضای تیم، در صورتی که کارکردی منظم موردنیاز باشد، چنین تیمی کارآمد خواهد بود.

این روش برای پروژه‌هایی مناسب است که یا حل مساله آن ساده است یا نمونه‌های مشابه آن قبلاً توسط همین تیم انجام شده است. زیرا یافتن راه حل مساله توسط تنها یک نفر (مدیر تیم) و بدون کمک دیگر اعضای تیم انجام می‌شود. در مقابل در مواقعی که پروژه نیاز به راه حل‌های خلاقانه و نوآوری دارد، الگوی بسته نمی‌تواند چندان کارآمد باشد، در واقع در این روش احتمال موفقیت برای نوآوری کارهای جدید کمتر است. الگوی بسته برای یافتن راه حل مساله به دلیل عدم وجود خرد جمعی نامناسب و برای توافقات و تصمیم‌گیری به دلیل کنترل مدیران مناسب است.

از آنجاکه در این روش راه حل مساله توسط تنها یک نفر و به شکل متمرکز کنترل‌شده بررسی می‌شود، و خرد جمعی و ارتباطات میان اعضای تیم بسیار ناچیز است، بنابراین کارکرد تیم در انجام کارها سریع می‌باشد.

۳- الگوی باز[2]

در گذشته به این الگو، **مدل** غیرمتمرکز کنترل شده (Controlled Decentralized-CD) نیز گفته می‌شد. غیرمتمرکز است چون خرد جمعی و ارتباطات میان گروه‌های داخل تیم وجود دارد، کنترل شده است زیرا مدیر اصلی و مدیران میانی دارد. الگوی باز، مزایای الگوی بسته و تصادفی را با هم ترکیب می‌کند. در این مدل، تیم در راستای یک سلسله مراتب سنتی از مسئولیت‌ها به تعدادی گروه سازمان‌دهی می‌شود. پروژه دارای یک مدیر اصلی است که هماهنگی وظایف و هماهنگ‌سازی گروه‌ها را بر عهده دارد. همچنین، مدیریت وظایف سطوح پایین‌تر بر عهده‌ی مدیران میانی پروژه است که برنامه‌ریزی و حل مشکلات سطح بالا را در گروه‌ها بر عهده دارند.

---

[1] Closed Paradigm
[2] Open Paradigm

در سطح آخر سلسله مراتب نیز انجام کارهای فنی بر عهده اعضای گروه است. ارتباط بین مدیر اصلی، مدیران میانی و اعضای تیم به صورت عمودی و ارتباط میان اعضای گروه‌ها به صورت افقی است. حل مسأله و تصمیم‌گیری یک کار تیمی است اما وظیفه پیاده‌سازی راه‌حل‌ها، بین زیرگروه‌ها توسط مدیر اصلی تقسیم و توزیع می‌شود.

به دلیل وجود مدیر اصلی و مدیران میانی و به تبع کنترل اعضای تیم، در صورتی که کارکردی منظم موردنیاز باشد، چنین تیمی کارآمد خواهد بود.

این روش برای پروژه‌هایی مناسب است که یا حل مساله آن پیچیده و دشوار است یا نمونه‌های مشابه آن قبلاً توسط همین تیم انجام نشده است. زیرا یافتن راه حل مساله توسط همه اعضای تیم انجام می‌شود. در واقع در مواقعی که پروژه نیاز به راه حل‌های خلاقانه و نوآوری دارد، الگوی باز می‌تواند بسیار کارآمد باشد، در واقع در این روش احتمال موفقیت برای نوآوری کارهای جدید بیشتر است. الگوی باز برای یافتن راه حل مساله به دلیل وجود خرد جمعی مناسب و برای توافقات و تصمیم‌گیری نیز به دلیل کنترل مدیران مناسب است.

از آنجاکه در این روش راه حل مساله توسط همه اعضای تیم بررسی می‌شود، و خرد جمعی و ارتباطات میان اعضای تیم زیاد است، بنابراین کارکرد تیم در انجام کارها کند می‌باشد.

۴- الگوی همزمان[1]

این الگو، بر قطعه قطعه کردن مساله به شیوه‌ای طبیعی و سازمان‌دهی اعضای تیم برای کار روی هر یک از این قطعات تاکید دارد، به گونه‌ای که میان اعضای تیم ارتباط چندانی برقرار نیست. به بیان دیگر در مواقعی‌که برنامه قابلیت پیمانه‌پذیری دارد، الگوی همزمان می‌تواند مورد استفاده قرار گیرد.

تا به اینجا مدیریت افراد بررسی شد حال در ادامه به بررسی مراحل بعدی مدیریت یعنی مدیریت محصول، مدیریت فرآیند و مدیریت پروژه می‌پردازیم:

۲- محصول

در یک بیان ساده مدیریت محصول یعنی شناخت دقیق بخش‌های مختلف و مرتبط با محصول مورد نظر مشتری. این شناخت از محصول، در فعالیت ارتباطات آغاز می‌گردد. پیش از آنکه بتوان برای پروژه در فعالیت برنامه‌ریزی، برنامه‌ریزی کرد، ابتدا باید اهداف[2] و محدوده[3] عملکردی محصول مشخص شود. تعیین اهداف و حوزه پروژه در واقع همان شناخت نیازمندی‌های مشتری است، بدون آنکه نحوه پیاده‌سازی آن مورد توجه قرار بگیرد. در مدیریت محصول باید راه‌حل‌های مختلف جهت رسیدن به بهترین راه حل با توجه با امکانات و محدودیت‌های مرتبط با پروژه همچون مهلت زمانی و بودجه مورد نظر، در نظر گرفته شود، همچنین محدودیت‌های فنی و

[1] Synchronous Paradigm
[2] Objective
[3] Scope

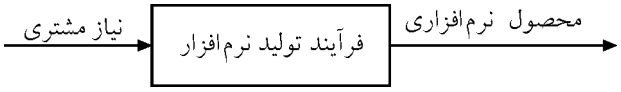
مدیریتی که محصول را تحت تاثیر قرار می‌دهد، مشخص شود. بدون این اطلاعات نمی‌توان برآوردی منطقی از هزینه‌ها، زمان‌بندی و ریسک‌های پروژه ارائه کرد. بدین منظور سازنده نرم‌افزار و مشتری باید به طور متناوب یکدیگر را ملاقات کنند تا اهداف و محدوده عملکردی محصول به طور دقیق مشخص گردد.

توجه: عدم توجه کافی به مدیریت محصول، ریسک مدیریت محصول را ایجاد خواهد کرد.

۳- فرآیند

پس از شناخت محصول، باید یک مدل فرآیند تولید نرم‌افزار جهت اجرای مراحل تولید نرم‌افزار انتخاب گردد. در یک بیان ساده مدیریت فرآیند یعنی انتخاب مدل فرآیند تولید نرم‌افزار مناسب و متناسب با ماهیت پروژه نرم‌افزاری و نظارت بر نحوه اجرای فعالیت‌های چارچوبی و چتری همروند با پیشرفت پروژه نرم‌افزاری. در واقع مدیریت فرآیند ضمن انتخاب مدل فرآیند تولید مناسب، می‌بایست اقدامات و وظایف لازم جهت تکمیل پروژه را نیز مشخص کند. همچنین نتایج مورد انتظار هر فعالیت را در فرآیند تولید نرم‌افزار تعیین کند، و پس از اتمام هر فعالیت، اقدام به بررسی و کنترل نتایج فعالیت مورد نظر کند که آیا نتایج مورد انتظار برآورده شده است یا خیر.

هر پروژه‌ی نرم‌افزاری، چه بزرگ و چه کوچک مراحلی را طی می‌نماید که در طی آن مجموعه‌ای از نیازمندی‌های مشتری به یک محصول نرم‌افزاری تبدیل می‌گردد. الگو و قالبی که چگونگی طی مراحل مختلف یک پروژه را تعریف می‌نماید، اصطلاحاً **فرآیند تولید نرم‌افزار** نامیده می‌شود. شکل زیر فرآیند تولید نرم‌افزار و ورودی و خروجی آن را نشان می‌دهد:



همانطور که ملاحظه می‌نمایید، ورودی این فرآیند، نیاز یا خواسته‌های مشتری و خروجی آن، یک محصول نرم‌افزاری است. یک فرآیند تولید در یک پروژه به ما می‌گوید که برای دستیابی به هدف مطلوب که همان تولید یک فرآورده‌ی نرم‌افزاری با کیفیت مطلوب است، چه کسی[1]، چه کاری را[2]، چه موقع[3] و چگونه[4] باید انجام دهد، در واقع، بدون داشتن تعریف مشترکی از فرآیند تولید نرم‌افزار، هماهنگی انجام کار تیمی در یک پروژه‌ی نرم‌افزاری، امکان‌پذیر نخواهد بود.

توجه نمایید که مدل فرآیندی که انتخاب می‌کنیم دقیقاً به نرم‌افزاری که تولید می‌کنیم بستگی دارد. ممکن است که یک مدل فرآیند برای تولید سیستم الکترونیکی هواپیما مناسب باشد، در حالی که مدل فرآیندی دیگر، برای ایجاد یک وب‌سایت استفاده شود. پس انتخاب مدل فرآیند

[1] Who
[2] What
[3] When
[4] How

براساس ماهیت نرم‌افزار صورت می‌گیرد.

به طور کلی فعالیت‌های چارچوبی مرتبط با فرآیند تولید نرم‌افزار صرف‌نظر از اندازه، پیچیدگی پروژه و زمینه‌ی کاربردی آن و مستقل از متدولوژی ساخت‌یافته و شیءگرا به پنج فعالیت ارتباط، برنامه‌ریزی، مدل‌سازی (تحلیل و طراحی)، ساخت (پیاده‌سازی و تست) و استقرار تقسیم می‌شود، به بیان دیگر فعالیت‌ها در هر دو دسته‌ی متدولوژی ساخت‌یافته و شیءگرا همین‌ها خواهند بود، اما کارهایی که در هر فعالیت در متدولوژی ساخت‌یافته و شیءگرا انجام می‌شود شباهت‌ها و تفاوت‌هایی خواهد داشت.

انجام درست فعالیت‌های چارچوبی تا حدود بسیار زیادی تحت کنترل انسان می‌باشد ولی برای تولید موفق یک پروژه‌ی نرم‌افزاری به تنهایی کافی نیستند. زیرا در وادی زندگی علاوه بر قوانین انسانی، قوانین طبیعی نیز وجود دارند، اگر فعالیت‌های چارچوبی درست بودند، اما اطلاعات موجود در هارد دیسک به دلیل عوامل طبیعی از بین رفتند، چه کار کنیم، اگر فعالیت‌های چارچوبی درست بودند، اما به دلیل ماهیت انسانی بودن یک انسان و عوامل طبیعی همچون مرگ و میر و زلزله، مدیران و همکاران خود را از دست دادیم، چه کار کنیم و واقعاً اگر همه‌ی موارد تحت کنترل انسان برای رسیدن به موفقیت درست بودند، عوامل طبیعی که خارج از کنترل انسان هستند را چه کار کنیم ...

در وادی مهندسی نرم‌افزار، توسط فعالیت‌های چتری، عوامل طبیعی خارج از کنترل انسان و هر آنچه مربوط به موفقیت پروژه باشد، نظارت و تحت کنترل دقیق قرار می‌گیرد، در واقع فعالیت‌های چارچوبی فرآیند تولید نرم‌افزار توسط تعدادی از فعالیت‌های چتری تکمیل می‌شود. به طور کلی، فعالیت‌های چتری در سرتاسر یک پروژه‌ی نرم‌افزاری به کار برده می‌شوند و به تیم نرم‌افزاری کمک می‌کنند تا پیشرفت، کیفیت، تغییر و ریسک را کنترل کنند. به بیان دیگر فعالیت‌های چتری بر محقق شدن خصوصیات پروژه‌های موفق نرم‌افزاری (بازه‌ی زمانی از قبل برنامه‌ریزی شده، بودجه‌ای از قبل پیش‌بینی شده و با صرف کمترین هزینه و دقیقاً مطابق با نیازمندی‌های واقعی کاربران) در حین انجام فعالیت‌های چارچوبی، تأکید می‌کند.

توجه: عدم توجه کافی به مدیریت فرآیند، ریسک مدیریت فرآیند را ایجاد خواهد کرد.

۴- پروژه

در یک بیان ساده مدیریت پروژه به معنی **تهیه و اجرای برنامه‌ریزی** است. برنامه‌ریزی در انجام هر کاری باعث مدیریت بهتر فرآیند تکمیل آن کار می‌شود. به عبارت دیگر برای انجام مناسب هر کاری، راهی جز برنامه‌ریزی نداریم. پس از مدیریت اولیه افراد، محصول و فرآیند، نوبت به مدیریت پروژه می‌رسد. همچنین ترتیب زمانی روال انجام کارهای آتی، بر اساس برآوردها، تخمین‌ها و اندازه‌گیری‌های انجام شده بر روی کارهای قبلی، در فعالیت زمان‌بندی، به عنوان بخشی از فعالیت برنامه‌ریزی مشخص می‌گردد. پروژه باید با برآوردی مناسب نسبت به میزان کار، زمان لازم برای انجام کار، هزینه لازم برای انجام کار، ریسک‌ها، توانایی‌ها، امکانات و محدودیت‌ها در یک مهلت زمانی مشخص شده، زمان‌بندی شود، و این یعنی برنامه‌ریزی.

برنامه‌ریزی[1] یعنی هنر حرکت از مبدأ موجود به مقصد مطلوب برای رسیدن به نتیجه‌ای مطلوب براساس خواسته‌های مورد نیاز در یک زمان مشخص. برنامه‌ریزی فرآیند چگونگی رسیدن به اهداف، در چارچوب امکانات، توانایی‌ها و محدودیت‌هاست. برنامه‌ریزی، فرآیندی برای رسیدن به اهداف است. برنامه‌ریزی به پیش‌بینی آینده و ساختن آینده تا حدودی قابل تصور کمک می‌کند. برنامه‌ریزی آن پلی است بین آنجایی‌که هستیم و آنجایی‌که می‌خواهیم برویم. برنامه‌ریزی به آینده می‌نگرد. برنامه‌ریزی یعنی اندیشیدن از پیش.

در ادامه به بیان داستان کوتاهی می‌پردازیم که طی پنج دهه‌ی گذشته، ده‌ها هزار بار برای سازندگان نرم‌افزار اتفاق افتاده است:

در اواخر دهه‌ی ۱۹۶۰ مهندس جوانی با چشم‌های روشن انتخاب شد تا برای یک کاربرد صنعتی خاص برنامه‌ای بنویسد. دلیل انتخاب او ساده بود. او در گروه فنی خودش تنها کسی بود که در یک سمینار برنامه‌نویسی کامپیوتری شرکت کرده بود. او به زیر و بم زبان اسمبلی و فرترن کاملا آشنا بود، ولی هیچ چیزی درباره‌ی مهندسی نرم‌افزار، برنامه‌ریزی، زمان‌بندی و پیگیری پروژه نمی‌دانست.

رئیس، جزوات لازم را همراه با توصیفی از آنچه باید انجام شود، به او داد و به او اطلاع داد که برای انجام کار دو ماه وقت دارد.

او جزوه‌ها را خواند، رویکردی برای خودش در نظر گرفت و شروع به کدنویسی کرد. بعد از دو هفته، رئیس او را به دفترش خواند تا از روند امور بپرسد.

مهندس جوان با همان شور و جوانی گفت: «عالی. این کار از آنچه فکر می‌کردم بسیار آسان‌تر بود. احتمالا ۷۵٪ کار تمام شده‌است.»

رئیس لبخندی زد و مهندس جوان را تشویق کرد که به کار ادامه دهد. قرار شد هفته‌ی دیگر دوباره همدیگر را ببینند.

یک هفته‌ی بعد، رئیس دوباره او را به دفترش فراخواند:

«کجای کار هستیم؟» جوان گفت: «همه چیز خوب پیش می‌رود، ولی من به چند مساله‌ی کوچک برخوردم که البته خیلی زود بر طرف می‌شوند و دوباره روی روال می‌افتم.»

رئیس پرسید: «مهلت داده شده کافی هست؟»

مهندس جوان گفت: «مشکلی نیست. ۹۰٪ کار تمام است.»

اگر فقط چند سال در دنیای نرم‌افزار کار کرده باشید، می‌توانید این داستان را تمام کنید. حتما تعجب نخواهید کرد اگر بدانید که مهندس جوان تا آخر مدت مقرر برای پروژه در همان ۹۰٪ باقی ماند و تازه یک ماه بعد، آن هم با کمک دیگران توانست پروژه را کامل کند. اگر کنجکاوید که بدانید آن جوان چه کسی بوده است، کسی نبود، جز، **راجر اس. پرسمن.**

«هر تلاشی منجر به نتیجه‌ای مطلوب نمی‌گردد، بلکه این تلاشی مطلوب است که منجر به نتیجه‌ای

[1] Planning

مطلوب می‌گردد.»

لازمه‌ی تلاش مطلوب، برنامه‌ریزی است. برنامه‌ریزی می‌تواند اجرای هر کار پیچیده‌ای را ساده‌تر سازد. هر کار مهندسی مستلزم برنامه‌ریزی می‌باشد. مهندسی نرم‌افزار نیز مانند هر فعالیت مهندسی دیگری، نیازمند برنامه‌ریزی است. فعالیت برنامه‌ریزی، برنامه‌ای را برای فعالیت‌های مختلف بخش‌های مختلف فرآیند تولید نرم‌افزار پایه‌ریزی می‌کند. این فعالیت، وظیفه‌های فنی که باید هدایت شوند، ریسک‌هایی که محتمل می‌شوند (مانند عدم شناسایی برخی نیازمندی‌ها، از دست دادن داده‌ها و مدیران)، منابع مورد نیاز، واحدهای کاری که باید ایجاد شوند و برنامه زمان‌بندی[1] برای کارها را تشریح می‌کند. زمان‌بندی به عنوان بخشی از فعالیت‌های برنامه‌ریزی به به تعیین زمان و تعیین تقدم و تاخر رسیدگی به کارها یا وظیفه‌ها بر اساس مهلت زمانی می‌پردازد.

در یک پروژه نرم‌افزاری، هنگامی نتیجه پروژه موفقیت‌آمیز خواهد بود که فعالیت برنامه‌ریزی کارآمد باشد و فعالیت برنامه‌ریزی هنگامی کارآمد خواهد بود که فعالیت زمان‌بندی کارآمد باشد یعنی منجر به تحویل به موقع پروژه در مهلت زمانی گردد، و فعالیت زمان‌بندی نیز زمانی کارآمد خواهد بود که فعالیت‌های برآورد میزان کار، زمان لازم برای انجام کار، هزینه لازم برای انجام کار و مدیریت ریسک بر اساس امکانات، توانایی‌ها و محدودیت‌ها کارآمد باشد.

در یک پروژه نرم‌افزاری، هنگامی نتیجه پروژه ناموفقیت‌آمیز خواهد بود که فعالیت برنامه‌ریزی ناکارآمد باشد و فعالیت برنامه‌ریزی هنگامی ناکارآمد خواهد بود که فعالیت زمان‌بندی ناکارآمد باشد یعنی منجر به عدم تحویل به موقع پروژه در مهلت زمانی گردد، و فعالیت زمان‌بندی نیز زمانی ناکارآمد خواهد بود که فعالیت‌های برآورد میزان کار، زمان لازم برای انجام کار، هزینه لازم برای انجام کار و مدیریت ریسک بر اساس امکانات، توانایی‌ها و محدودیت‌ها ناکارآمد باشد. به بیان دیگر ناکارآمدی در انجام فعالیت‌های برآورد میزان کار، زمان لازم برای انجام کار، هزینه لازم برای انجام کار و مدیریت ریسک بر اساس امکانات، توانایی‌ها و محدودیت‌ها منجر به ناکارآمدی فعالیت زمان‌بندی می‌گردد و ناکارآمدی در فعالیت زمان‌بندی به معنی عدم تحویل به موقع پروژه در مهلت زمانی منجر به ناکارآمدی فعالیت برنامه‌ریزی می‌گردد و ناکارآمدی در فعالیت برنامه‌ریزی منجر به نتیجه ناموفقیت‌آمیز پروژه در یک پروژه نرم‌افزاری خواهد شد.

تهیه و تدوین برنامه‌ریزی به معنی مشخص‌کردن چگونگی و نحوه انجام کار و ارائه زمان‌بندی مناسب بر اساس مهلت زمانی مشخص شده از سوی کارفرما، بر عهده مدیر پروژه است. ارائه زمان‌بندی مناسب زمانی میسر خواهد بود که مدیر پروژه به شناخت زیادی از میزان کار، زمان لازم برای انجام کار، هزینه لازم برای انجام کار و ریسک‌های پروژه رسیده باشد و شناخت زیاد از میزان کار، زمان لازم برای انجام کار، هزینه لازم برای انجام کار و ریسک‌های پروژه زمانی میسر خواهد بود که مدیر پروژه به برآورد، تخمین و اندازه‌گیری مناسبی از میزان کار، زمان لازم برای انجام کار و هزینه لازم برای انجام کار و ریسک‌های پروژه فعلی بر اساس

[1] Scheduling

پروژه‌های انجام شده قبلی رسیده باشد.

همچنین **اجرا و نظارت بر برنامه‌ریزی** تهیه شده جهت حرکت به سمت مقصد مورد نظر بر اساس زمان‌بندی تعیین‌شده بر عهده مدیر پروژه است.

برای مثال برای تولید یک کتاب جهت انتشار در یک مهلت زمانی برای نمونه تا نمایشگاه کتاب در یک انتشارات باید برنامه‌ریزی صورت گیرد. در گام ابتدایی می‌بایست تهیه و تدوین برنامه‌ریزی توسط مدیر پروژه صورت گیرد. این برنامه‌ریزی به معنی مشخص‌کردن چگونگی و نحوه انجام کار می‌بایست بر اساس امکانات، توانایی‌ها و محدودیت‌های انتشارات انجام گردد. بنابراین پس از شناخت لازم و کافی از ماهیت و مشخصات چاپ کتاب بر اساس برآورد، تخمین و اندازه‌گیری مناسب از میزان کار، زمان لازم برای انجام کار، هزینه لازم برای انجام کار و ریسک‌های مربوط به چاپ کتاب از تجربه‌های به دست آمده از چاپ کتب قبلی، می‌بایست در گام بعدی زمان‌بندی کار، بر اساس مهلت زمانی انتشار کتاب توسط مدیریت پروژه ارائه گردد. برای‌مثال، در فعالیت برنامه‌ریزی، در گام نحوه تبلیغات می‌بایست نحوه تبلیغات از سوی رسانه‌ها مشخص گردد، رسانه‌هایی همچون رادیو و تلویزیون یا روزنامه‌های پرتیراژ، همچنین در گام بعد یعنی تولید و چاپ کتاب می‌بایست نحوه چاپ مشخص گردد، یعنی چاپ دیجیتال باشد یا چاپ افست، که در صورت داشتن بودجه کافی و تیراژ بالا، چاپ افست انتخاب می‌گردد. به هر حال چگونگی و نحوه انجام کار بر اساس میزان کار، زمان لازم برای انجام کار، هزینه لازم برای انجام کار، توانایی‌ها، محدودیت‌ها و مهلت زمانی می‌بایست مشخص گردد و این یعنی برنامه‌ریزی که بر اساس مهلت زمانی، زمان‌بندی نیز شده است. ریسک‌هایی هم در کار وجود دارد، مثل افزایش قیمت کاغذ، باید مراقب باشیم ریسک‌ها به زمان‌بندی کار صدمه وارد نکند، برای مثال در مدیریت ریسک افزایش قیمت کاغذ، پس از شناسایی این ریسک برای مقابله با آن می‌توان زودتر اقدام به تهیه کاغذ مورد نیاز نمود. اگر ریسک‌ها از قبل شناسایی نشوند، پس از وقوع در حین کار، منجر به بحران می‌گردند، که باید با هوشمندی مدیریت بحران مناسب انجام گردد، تا به هر شکلی که شده است، زمان‌بندی صدمه نبیند. همه این‌ها بر عهده مدیریت پروژه است. تهیه و تدوین برنامه‌ریزی بر عهده مدیر پروژه است، اجرا و نظارت بر اجرای برنامه‌ریزی نیز بر عهده مدیر پروژه است. موفقیت کار در گرو نحوه رهبری مدیریت پروژه است، در همه فرازها و فرودها...

توجه: اغلب، پس از تهیه و تدوین برنامه‌ریزی، برنامه‌ها در مرحله اجرا بر اساس زمان‌بندی مشخص شده پیش نمی‌روند، زیرا برنامه‌ریزی بر اساس یک برآورد مناسب از میزان کار، زمان لازم برای انجام کار، هزینه لازم برای انجام کار، ریسک‌ها، توانایی‌ها، امکانات و محدودیت‌ها زمان‌بندی نشده‌است. موثرترین شیوه برآورد، ثبت عملکرد گذشته در پروژه‌های قبلی است، گذشته را ثبت کنید، برای برآورد مناسب و بهبود عملکرد در آینده.

توجه: قانون ۹۰-۹۰ در پروژه‌هایی که برنامه‌ریزی آنها بر اساس یک برآورد مناسب از میزان کار، زمان لازم برای انجام کار، هزینه لازم برای انجام کار، ریسک‌ها، توانایی‌ها، امکانات و محدودیت‌ها، زمان‌بندی نشده‌است بیان می‌کند که ۹۰ درصد بخش ابتدایی کار، ۹۰ درصد از زمان

و هزینه انجام کار را به خود اختصاص می دهد. و ۱۰ درصد بخش انتهایی کار نیز، مجددا ۹۰ درصد دیگر از زمان و هزینه انجام کار را به خود اختصاص می دهد.

توجه: به عنوان مدیر پروژه، باید بدانید چه چیزهایی ممکن است به خطا برود تا بتوانید آنها را کنترل کنید، و این یعنی آگاهی و مدیریت...

توجه: عدم توجه کافی به مدیریت پروژه، ریسک مدیریت پروژه را ایجاد خواهد کرد.

توجه: زمانی شروع به انجام یک پروژه سرانجام خوش و موفقی خواهد داشت که، هر یک از مراحل **مدیریت افراد**، **مدیریت محصول**، **مدیریت فرآیند** و **مدیریت پروژه** در همه مراحل انجام پروژه از ابتدا تا انتها، به عالی‌ترین شکل ممکن، پیوسته و بدون توقف مدیریت، رهبری و هدایت گردد.

اصول W^5HH

بوهم[1] در یک مقاله راهکارهایی را برای برنامه‌ریزی به معنی مشخص‌کردن چگونگی و نحوه انجام کارها بر اساس یک برآورد مناسب از میزان کار، زمان لازم برای انجام کار، هزینه لازم برای انجام کار، ریسک‌ها، توانایی‌ها، امکانات و محدودیت‌ها در یک زمان‌بندی و مهلت زمانی مشخص شده، ارائه کرده است. در این راهکار مراحلی برای مشخص‌شدن کارهای پروژه و ایجاد یک تعریف درست و دقیق از پروژه، پیشنهاد شده‌است که اهداف پروژه، نقاط عطف پروژه، زمان‌بندی‌ها، مسئولیت‌ها، روش‌های فنی و مدیریتی و منابع موردنیاز را دربر می‌گیرد و منجر به هدایت درست و صحیح پروژه می‌شود و برای تمامی پروژه‌های نرم‌افزاری بدون توجه به اندازه و پیچیدگی‌آن قابل استفاده است. بوهم این پیشنهاد را اصول W^5HH نام‌گذاری کرده‌است، که مجموعه‌ای از سوالات است، که مدیر پروژه را برای شناخت دقیق پروژه و برنامه‌ریزی برای انجام کار پروژه راهنمایی می‌کند.

اصل W^5HH با هفت سوال بیان شده‌است که با حروف W و H آغاز می‌شوند.

Why –۱

چرا این سیستم تولید می‌گردد؟

تمام ذی‌نفعان باید دلایل قانع‌کننده برای تولید یک سیستم را مورد ارزیابی قرار دهند. باید دقیق و واضح مشخص شود که زمان و هزینه لازم برای انجام کار صرفه و توجیه اقتصادی دارد یا ندارد.

What –۲

چه کارهایی باید انجام شود؟

مجموعه کارهای لازم برای انجام پروژه لیست می‌گردد.

[1] Boehm

When –۳

چه موقع کارها باید انجام شود؟

زمان‌بندی هر یک از کارها بر اساس مهلت زمانی مشخص می‌گردد.

Who –۴

چه کسی مسئول چه کاری می‌باشد؟

نقش و مسئولیت هر یک از اعضای تیم پروژه برای انجام هریک از کارها و بخش‌های مختلف پروژه مشخص می‌گردد.

Where –۵

چه جایگاهی برای هریک از ذی‌نفعان وجود دارد؟

از نگاه سلسله مراتبی سازمانی، جایگاه و سلسله مراتب ذی‌نفعان (مشتری و سازنده) مشخص می‌گردد.

How –۶

کار از لحاظ فنی و مدیریتی چگونه انجام خواهد شد؟

پس از شناخت کارها در مرحله دوم، برنامه‌ریزی به معنی مشخص‌کردن چگونگی و نحوه انجام کارها بر اساس یک برآورد مناسب از میزان کار، زمان لازم برای انجام کار، هزینه لازم برای انجام کار، ریسک‌ها، توانایی‌ها، امکانات و محدودیت‌ها در یک زمان‌بندی و مهلت زمانی مشخص شده انجام می‌گردد.

How Many –۷

میزان کار، زمان و هزینه لازم برای انجام کار چقدر است؟

تشخیص میزان کار، زمان لازم برای انجام کار و هزینه لازم برای انجام کار و ریسک‌های پروژه زمانی میسر خواهد بود که مدیر پروژه به برآورد، تخمین و اندازه‌گیری مناسبی از میزان کار، زمان لازم برای انجام کار و هزینه لازم برای انجام کار و ریسک‌های پروژه فعلی بر اساس پروژه‌های انجام شده قبلی رسیده باشد.

برنامه‌ریزی پروژه‌های نرم‌افزاری

برای تحقق خواسته و هدف، قبل از اقدام به استفاده از توان فیزیکی و انجام کار، باید با اقدام به برنامه‌ریزی به حد کافی از توانایی‌های ذهنی استفاده شود. برنامه‌ریزی در حقیقت پاسخ به دو سوال زیر است:

۱- کجا می‌خواهیم برویم؟

۲- چگونه می‌خواهیم برویم؟

همانطور که پیش‌تر نیز گفتیم، در یک پروژه نرم‌افزاری، هنگامی نتیجه پروژه موفقیت‌آمیز خواهد بود که فعالیت برنامه‌ریزی کارآمد باشد و فعالیت برنامه‌ریزی هنگامی کارآمد خواهد بود

که فعالیت زمان‌بندی کارآمد باشد یعنی منجر به تحویل به موقع پروژه در مهلت زمانی گردد، و فعالیت زمان‌بندی نیز زمانی کارآمد خواهد بود که فعالیت‌های برآورد میزان کار، زمان لازم برای انجام کار، هزینه لازم برای انجام کار و مدیریت ریسک بر اساس امکانات، توانایی‌ها و محدودیت‌ها کارآمد باشد.

هنگامی که نیاز مبرم به نرم‌افزار حتمی شده‌است، طرف‌های ذی‌نفع جمع شده‌اند، مهندسان نرم‌افزار آماده‌ی شروع به کار هستند و پروژه در شرف آغاز است، همچنان مساله‌ی مهمی در میان است، اینکه کار چگونه پیش خواهد رفت؟

پاسخ در یک کلام، **برنامه‌ریزی** است. بدون شک، کار با برنامه‌ریزی پیش خواهد رفت.

برنامه‌ریزی پروژه‌های نرم‌افزاری شامل فعالیت‌های «**برآورد میزان کار**»، «**برآورد زمان لازم برای انجام کار**»، «**برآورد هزینه لازم برای انجام کار**»، «**مدیریت ریسک**» و «**زمان‌بندی**» است، که تهیه و نظارت بر اجرای آن بر عهده «**مدیر پروژه**» می‌باشد.

در فعالیت برآورد فعالیت‌هایی انجام می‌شود تا مشخص شود که برای ساخت یک محصول یا سیستم نرم‌افزاری چه میزان کار، زمان و هزینه لازم است. این کار توسط مدیران پروژه‌ی نرم‌افزاری بر اساس نیازمندی‌ها و ریسک‌های پروژه پیش‌رو و با استفاده از داده‌های تاریخی موجود در پروژه‌های پیشین انجام می‌گردد.

آیا بدون دانستن مقدار کاری که باید انجام شود، زمانی که باید صرف شود و هزینه‌ای که باید خرج شود، ریسک‌های ممکن و مهلت زمانی به انجام کاری مبادرت می‌ورزید؟

البته که خیر، بنابراین انجام برآورد میزان کار، زمان لازم برای انجام کار، هزینه لازم برای انجام کار، مدیریت ریسک و زمان‌بندی منطقی به نظر می‌رسد. که انجام آنها یعنی **برنامه‌ریزی**.

لازمه ارائه‌ی زمان‌بندی درست، برآورد درست میزان کار، زمان لازم برای انجام کار، هزینه لازم برای انجام کار و شناخت درست ریسک‌های کار است. زمانی از **فرد بروکز**، پرسیده شد که پروژه‌های نرم‌افزاری چگونه از زمان‌بندی عقب می‌افتند؟ پاسخ او ساده و بسیار عمیق بود: «**روزی از روزها**»

زیرا صدها کار کوچک باید انجام شود تا هدف و خواسته‌ای بزرگ‌تر محقق گردد. برخی از این کارها، خارج از جریان اصلی پروژه قرار می‌گیرند اما می‌توان آنها را بدون نگرانی درباره‌ی تاثیری که بر تاریخ تکمیل پروژه می‌گذارند، به انجام رساند. ولی برخی دیگر از کارها در مسیر «**بحرانی**» قرار می‌گیرند، که اگر این کارهای بحرانی از زمان‌بندی عقب بیافتند، آنگاه تاریخ تکمیل کل پروژه نیز به مخاطره می‌افتد.

استیو مک کانل در باب زمان‌بندی گفته است: «**زمان‌بندی‌های بیش از حد خوش‌بینانه به زمان‌بندی‌های واقعا کوتاه منجر نمی‌شوند، بلکه نتیجه‌ی آنها زمان‌بندی‌های بلندمدت‌تر است.**»

هنگام زمان‌بندی، کارها باید به چند بخش تقسیم شوند، همچنین به وابستگی‌های میان بخش‌ها توجه شود و به هر بخشی وظیفه، کار و زمان لازم تخصیص داده شود. همچنین مسئولیت‌ها، پیامدها و نقاط عطف نیز باید مورد توجه قرار گیرند.

هنگامی‌که یک مدل فرآیند تولید نرم‌افزار مناسب و متناسب با ماهیت پروژه‌ی نرم‌افزاری انتخاب کرده‌اید، وظایف مهندسی نرم‌افزار را که باید انجام شوند، مشخص کرده‌اید، میزان کار، زمان لازم برای انجام کار، هزینه لازم برای انجام کاری را که باید انجام شود و تعداد افراد لازم را برآورد کرده‌اید، مهلت زمانی را می‌دانید و حتی ریسک‌ها را در نظر گرفته‌اید، اکنون زمان متصل کردن نقاط به یکدیگر فرا رسیده‌است. یعنی باید شبکه‌ای از وظایف مهندسی‌نرم‌افزار ایجاد کنید که به کمک آنها بتوانید کار را سر وقت انجام دهید. زمانی‌که این شبکه را ایجاد کردید، باید مسئولیت‌های هر وظیفه را تعیین کنید، از انجام آنها اطمینان حاصل کنید و با تحقق پیدا کردن ریسک‌ها آنها را بر این شبکه تطبیق دهید، این یعنی زمان‌بندی و پیگیری پروژه.

به منظور ساخت یک سیستم کامپیوتری، وظایف مهندسی نرم‌افزار فراوانی به موازات هم انجام می‌شود و نتیجه‌ی کار انجام شده طی یک وظیفه ممکن است بر کارهایی که قرار است در وظیفه‌ای دیگر انجام شود، اثری بنیادی بگذارد. این وابستگی‌های متقابل را بدون یک زمان‌بندی به سختی می‌توان درک کرد. ضمن اینکه **ارزیابی روند پیشرفت کار** در پروژه‌های نرم‌افزاری بدون یک زمان‌بندی مشروح عملا غیرممکن است. فعالیت‌های چارچوبی مربوط به فرآیند تولید نرم‌افزار، برای بخش‌های مختلف محصولی که باید ساخته‌شود، پالایش، شفاف و واضح می‌شوند، میزان کار و زمان لازم برای انجام کار به هر بخش به عنوان وظیفه تخصیص داده می‌شود و شبکه‌ای از فعالیت‌های مربوط به هر بخش به عنوان وظیفه به شیوه‌ای ایجاد می‌شود که تیم نرم‌افزاری بتواند در مهلت زمانی مقرر، محصول نهایی را تحویل دهد.

نتیجه اینکه مقدمه برنامه‌ریزی، برآورد میزان کار، زمان لازم برای انجام کار، هزینه لازم برای انجام کار است، در ادامه تشخیص ریسک‌ها و در نهایت ارائه زمان‌بندی است. زیرا برای ارائه زمان‌بندی، باید برآورد میزان کار، زمان لازم برای انجام کار، هزینه لازم برای انجام کار و ریسک‌ها مشخص باشد. در ادامه این فصل به تشریح «**نحوه برآورد**»، «**نحوه تشخیص ریسک‌ها**» و در نهایت «**نحوه زمان‌بندی**» می‌پردازیم.

نحوه برآورد (تخمین)

برآورد میزان کار، زمان لازم برای انجام کار و هزینه لازم برای انجام کار در پروژه‌های آتی بر اساس مقادیر و اندازه‌های پروژه‌های پیشین انجام می‌گردد. بنابراین برای به‌دست آوردن برآورد میزان کار، زمان لازم برای انجام کار و هزینه لازم برای انجام کار برای پروژه‌های آتی، باید پروژه‌های پیشین را **اندازه‌گیری** نمود. جدای از اینکه این داده‌های تاریخی اندازه‌گیری شده در پروژه‌های پیشین در برآورد میزان کار، زمان لازم برای انجام کار و هزینه لازم برای انجام کار مورد استفاده قرار می‌گیرد، این داده‌های تاریخی مطابق واژه‌ی تاریخ دیرینیان، می‌تواند جهت پند و اندرز نیز در پروژه‌های آتی کاربرد داشته باشد. تاریخ مفهومی انتزاعی است و ناظر به وقایع گذشته است. مجموعه‌ی حوادث فرهنگی، طبیعی، اجتماعی، اقتصادی و سیاسی و رویدادهایی است که در گذشته و در زمان و مکان زندگی انسان ها و در رابطه با آن ها رخ داده است. این رویدادها شامل اموری می شود از قبیل کردارها و دستاوردهای مادی و معنوی بشر و هرآنچه که گفته،

اندیشیده و عمل کرده است. نرم‌افزارهای پیشین نیز تاریخ دارند، مجموعه‌ی حوادث و رویدادهایی است که در گذشته و در زمان و مکان تکامل نرم‌افزارها و در رابطه با آن‌ها رخ داده است. این رویدادها شامل اموری می‌شود از قبیل دستاوردهای نرم‌افزار و هرآنچه که انجام داده است. اگر این داده‌های تاریخی مربوط به پروژه‌های پیشین اندازه‌گیری شوند، معیار و ملاک ایده‌آلی برای بهبود کیفیت «**فرآیند**» و «**پروژه**» در پروژه‌های آتی خواهد بود. بنابراین ثبت و اندازه‌گیری این داده‌های تاریخی مربوط به پروژه‌های پیشین دو کاربرد اساسی دارد، یکی برای برآورد میزان کار، زمان لازم برای انجام کار و هزینه لازم برای انجام کار در پروژه‌های آتی و دیگری به عنوان **معیاری** برای بهبود کیفیت «**فرآیند**» و «**پروژه**» در پروژه‌های آتی.

اندازه‌ها و معیارها (متریک‌ها) در نرم‌افزار

معیارهای پروژه و فرآیند تولید نرم‌افزار، مقادیری کمی یا عددی هستند که این امکان را فراهم می‌کنند تا از کارایی فرآیند تولید نرم‌افزار و پروژه‌هایی که از این فرآیند تولید نرم‌افزار به عنوان چارچوب استفاده می‌کنند دیدی به‌دست آید. سپس این داده‌ها تحلیل می‌شوند، با میانگین‌های گذشته مقایسه می‌شوند و مورد ارزیابی قرار می‌گیرند تا مشخص شود که آیا کیفیت و بهره‌وری بهبود یافته‌است یا خیر.

اگر اندازه‌گیری انجام نشود و معیارهایی حاصل نگردد، قضاوت‌ها ممکن است فقط بر اساس ارزیابی‌های ذهنی و نه عینی باشد. با اندازه‌گیری می‌توان روندها را چه خوب و چه بد رصد کرد، برآوردهای کارآمدتری ایجاد کرد و با گذشت زمان بهبودهایی را نیز رقم زد. روند کار بدین نحو است که پروژه‌های مشابه پروژه فعلی که در گذشته ایجاد شده‌اند اندازه‌گیری می‌شوند، سپس با اندازه‌گیری‌های انجام شده بر روی پروژه مشابه قبلی، مبنای برآورد مناسبی از میزان کار، زمان لازم برای انجام کار، هزینه لازم برای انجام کار برای پروژه فعلی فراهم می‌گردد. با ثبت عملکرد گذشته، می‌توان فرصت برآورد و بهبود آینده را فراهم کرد. اندازه‌گیری پروژه‌های فعلی، بر اساس اندازه‌گیری پروژه‌های مشابه قبلی انجام می‌گیرد. این اندازه‌گیری‌های پروژه‌های قبلی، برای اندازه‌گیری پروژه فعلی، در یک عبارت ساده یعنی «**برآورد**» برآورد میزان کار، زمان لازم برای انجام کار، هزینه لازم برای انجام کار از گذشته برای آینده.

اندازه‌گیری با فراهم آوردن سازوکاری برای ارزیابی عینی، ما را قادر می‌سازد تا دیدی از فرآیند تولید نرم‌افزار و پروژه به‌دست آوریم. لرد کلوین زمانی گفته بود:

«**هنگامی‌که می‌توانید آنچه را که درباره‌اش حرف می‌زنید، اندازه‌گیری و بر حسب اعداد بیان کنید، چیزی درباره‌ی آن می‌دانید، ولی اگر نتوانید اندازه‌گیری کنید، اگر نتوانید آنرا بر حسب اعداد بیان کنید، دانش شما ناقص خواهد بود. این ممکن است شروع دانش باشد، ولی به سختی می‌توان گفت که به مرحله‌ی علم دست پیدا کرده‌اید.**»

از اندازه‌گیری می‌توان در سرتاسر پروژه‌ی نرم‌افزاری برای کمک به برآورد، کنترل کیفیت، ارزیابی بهره‌وری و کنترل پروژه استفاده کرد. سرانجام اینکه، مهندسان نرم‌افزار می‌توانند از اندازه‌گیری برای کمک به ارزیابی کیفیت محصولات کاری و کمک به تصمیم‌گیری‌های تاکتیکی به

موازات پیشرفت پروژه استفاده کنند.

در حیطه فرآیند نرم‌افزار و پروژه‌هایی که با استفاده از این فرآیند اجرا می‌شوند، دغدغه اصلی تیم نرم‌افزاری، معیارهای بهره‌وری و کیفیتی است. برای برآورد و به تبع برنامه‌ریزی دقیق‌تر کارهای فعلی، گذشته کارهای مشابه از نظر بهره‌وری و کیفیت مورد توجه قرار می‌گیرد. در واقع اطلاعات بهروه‌وری و کیفیتی پروژهای قبلی، به پروژهای مشابه فعلی تعمیم پیدا می‌کند.

اندازه‌گیری پروژه‌های مشابه پیشین، یک ابزار مدیریتی است که اگر درست انجام شود، به مدیر پروژه دیدی درست از پروژه فعلی می دهد و در نتیجه، به مدیر پروژه و تیم نرم‌افزاری کمک می‌کند تا تصمیم‌های درستی بگیرند که در نهایت به موفقیت پروژه منجر شود. «اندازه‌گیری‌ها و برآوردها به شما این امکان را می‌دهند که بدانید کی بخندید و کی گریه کنید!»

پارک، گوترت و فلوراک در راهنمایی که برای اندازه‌گیری نرم‌افزار نوشته‌اند، دلایل لزوم اندازه‌گیری را چنین بیان‌کرده‌اند:

۱- **مشخص‌کردن** فرآیند، محصول، منابع و برنامه‌ریزی‌های لازم برای انجام پروژه بر اساس اندازه‌گیری‌ها

۲- **ارزیابی** روند پیشرفت کارها براساس اندازه‌گیری‌ها و برنامه‌ریزی‌ها

۳- **پیش‌بینی** آینده کارها، اندازه‌گیری‌های به‌دست آمده در پروژه‌های قبلی، مبنایی برای پیش‌بینی کارهای پروژه فعلی خواهد بود.

۴- بهبود کارها، توسط انجام کارهای مهم و غیرفوری، قبل از آنکه هزینه‌های سنگین تحمیل شود، این بهبودها امکان‌پذیر است، زیرا اندازه‌گیری‌ها، ارزیابی‌ها و برنامه‌ریزی‌ها مشخص است.

اندازه‌گیری نرم‌افزار (software measurement)

اندازه‌گیری، اساس تمامی رشته‌های مهندسی است و مهندسی نرم‌افزار نیز از این قاعده مستثنی نیست. به طور کلی اندازه‌گیری نرم‌افزار به دو طبقه کلی اندازه‌گیری کمیت نرم‌افزار و اندازه‌گیری کیفیت نرم‌افزار تقسیم می‌گردد.

اندازه‌گیری کمیت نرم‌افزار

اندازه‌گیری کمیت نرم‌افزار به دو روش اندازه‌گرا یعنی مبتنی بر سایز (Size-Oriented) و عملکردگرا یعنی مبتنی بر کارکرد (Function-Oriented) انجام می‌گردد.

روش اندازه‌گرا (Size-Oriented)

در روش اندازه‌گرا، اندازه کمیت نرم‌افزار به شکل مستقیم (direct measure) و از طریق شمارش تعداد خطوط کد برنامه یعنی Line of Code یا LOC محاسبه می‌گردد. یک نرم‌افزار علاوه بر اندازه اصلی تعداد خطوط کد برنامه در روش اندازه‌گیری اندازه‌گرا، اندازه‌های فرعی دیگری همچون هزینه برای هر خط کد، تعداد خطاها (error) و تعداد نقص‌ها (defect) دارد. که

در آن error تعداد خطاهای یافت شده پیش از تحویل نرم‌افزار به مشتری و defect تعداد نقایص یافت‌شده پس از تحویل به مشتری است. حال اگر نسبت این اندازه‌های فرعی به اندازه اصلی یعنی در هر ۱۰۰۰ خط کد برنامه KLOC تناسب‌گیری و محاسبه گردد، آنگاه حاصل این تناسبات در یک نرم‌افزار خاص بازهم یک **اندازه (measure)** خواهد بود. **اما** اگر نسبت این اندازه‌های فرعی به اندازه اصلی یعنی در هر ۱۰۰۰ خط کد برنامه KLOC تناسب‌گیری و محاسبه گردد، آنگاه حاصل این تناسبات در مقام **مقایسه** با نرم‌افزاری دیگر یک **معیار (metric) اندازه‌گرا** خواهد بود. این معیار جهت مقایسه، بررسی و ارزیابی فرآیند تولید نرم‌افزار در گذشته دو یا چند نرم‌افزار مورد استفاده قرار می‌گیرد. این مقایسه **گذشته** منجر به پندگیری و به تبع بهبود فرآیند تولید نرم‌افزار در **آینده** می‌گردد.

در جدول زیر، اندازه‌های اصلی و فرعی دو پروژه به نام‌های A و B که در گذشته ایجاد شده‌اند، قرار گرفته است:

نام پروژه	تعداد خطوط کد برنامه (LOC)	هزینه (تومان)	خطا (error)	نقص (defect)
A	21000	300000000	440	64
B	14000	340000000	700	40

حال اگر نسبت این اندازه‌های فرعی به اندازه اصلی یعنی در هر ۱۰۰۰ خط کد برنامه KLOC تناسب‌گیری و محاسبه گردد، آنگاه جدول زیر را خواهیم داشت:

نام پروژه	تعداد خطوط کد برنامه (LOC)	هزینه برای هر خط کد (تومان)	خطا در هر هزار خط کد (error)	نقص در هر هزار خط کد (defect)
A	21000	14285	20.95	3.04
B	14000	24285	29.16	1.66

توجه: مطابق نتایج جدول فوق واضح است که در مقام مقایسه و معیار (metric) اندازه‌گرا برای دو تیم A و B، تیم B در کشف خطاهای برنامه موفق‌تر بوده است.

روش عملکردگرا (Function-Oriented)

در روش عملکردگرا، اندازه کمیت نرم‌افزار به شکل غیرمستقیم (indirect measure) و از طریق رابطه FP محاسبه می‌گردد. FP سرواژه‌ی عبارت Function Point و به معنی نقطه عملکرد یا امتیاز کارکرد می‌باشد. در این روش چون نمی‌توان عملکرد نرم‌افزار را مستقیماً اندازه‌گیری کرد، باید آن را به طور غیرمستقیم و از طریق اندازه‌گیری‌های مستقیم دیگر به دست آورد. نقطه عملکرد با استفاده از یک فرمول تجربی مبتنی بر اندازه‌گیری‌های قابل شمارش (مستقیم) از مقادیر دامنه اطلاعاتی (information domain) و ضریب وزنی پیچیدگی (complexity) نرم‌افزار، به دست می‌آید.

نقطه عملکرد (FP) از رابطه‌ی زیر محاسبه می‌گردد:

$$FP = \text{تعداد کل} \times \left[۰/۶۵ + ۰/۰۱ \times \sum (F_i)\right]$$

مقادیر دامنه اطلاعاتی پروژه به شیوه زیر تعیین می‌شود:

۱- **تعداد ورودی‌های برنامه:** شمارش تعداد ورودی‌های مختلف برنامه خواه ورودی‌های کاربر یا ورودی‌های حاصل از برنامه‌های دیگر. این ورودی‌ها اگر داده‌های کاربردی برنامه باشند درون فایل‌های منطقی داخلی برنامه (پایگاه داده‌ها) برای نرم‌افزارهای بانک اطلاعات و اگر داده‌های کنترلی برنامه باشند درون فایل‌های واسط خارجی برنامه (حسگرهای خارجی یا زیرسیستم‌های خارجی) برای نرم‌افزاری بی‌درنگ که مجهز به حسگرها هستند، قرار می‌گیرد.

۲- **تعداد خروجی‌های برنامه:** شمارش تعداد خروجی‌های مختلف برنامه که توسط برنامه در سیستم‌های بانک اطلاعات (پایگاه داده‌ها) یا سیستم‌های بی‌درنگ برای کاربر آماده می‌شود. که شامل نتایج حاصل از پرس و جوهای برنامه و پیغام‌های خطای برنامه است.

۳- **تعداد پرس و جوهای برنامه:** شمارش تعداد پرس و جوهای مختلف برنامه که منجر به بازیابی داده‌های کاربردی از فایل‌های منطقی داخلی برنامه (پایگاه داده‌ها) برای نرم‌افزارهای بانک اطلاعات و یا بازیابی داده‌های کنترلی از فایل‌های واسط خارجی برنامه (حسگرهای خارجی یا زیرسیستم‌های خارجی) برای نرم‌افزاری بی‌درنگ می‌شود.

۴- **تعداد فایل‌های منطقی داخلی برنامه:** شمارش تعداد فایل‌های منطقی مختلف داخلی برنامه، این فایل‌های منطقی داخلی در سیستم‌های بانک اطلاعات مورد استفاده قرار می‌گیرند، که همان فایل‌های پایگاه داده‌ها هستند که جهت ذخیره‌سازی ورودی‌های برنامه و یا بازیابی پرس و جوهای برنامه استفاده می‌شوند. رمز عبور برای ورود به برنامه داخل فایل‌های منطقی داخلی نگهداری می‌شود.

۵- **تعداد فایل‌های واسط خارجی برنامه:** شمارش تعداد فایل‌های واسط مختلف خارجی برنامه، این فایل‌های واسط خارجی در سیستم‌های بی‌درنگ جهت نگهداری و پیکربندی تنظیمات حسگرها مورد استفاده قرار می‌گیرند، امروزه نرم‌افزارهای بی درنگ طیف وسیعی از برنامه‌های کامپیوتری را پوشش می‌دهند. در نرم‌افزارهای بی‌درنگ باید خروجی و پاسخ نهایی در یک زمان مشخص و از پیش تعیین شده حاصل شود. در این نرم‌افزارها، زمان نقشی کلیدی ایفا می‌کند و زمان پاسخ باید به موقع و تضمین شده باشد. نرم‌افزارهای بی‌درنگ معمولاً به عنوان یک دستگاه کنترلی در یک کاربرد خاص (مثلاً صنعتی) به کار گرفته می‌شوند. در این نرم‌افزارها دیر پاسخ دادن به همان بدی پاسخ ندادن است. در این نوع نرم‌افزارها هدف اصلی طراحان، پاسخگویی سریع (در مهلت تعیین شده) به رویدادها و درخواست‌ها می‌باشد و راحتی کاربران و بهره‌وری منابع در درجه‌های بعدی اهمیت، قرار دارند.

انسان در وادی زندگی نیازهای گوناگونی دارد، یکی از نیازهای اساسی انسان، نیاز به امنیت است. اما گاهاً، ممکن است در معرض عوامل محیطی و بیرونی و یا حتی درونی، امنیت انسان در

شرایط هشیاری یا ناهشیاری به مخاطره بیفتد. بنابراین نیاز است تا مکانیزمی همواره هوشیار و همیشه بیدار و با اشراف لحظه به لحظه مخاطرات پیرامون انسان را رصد و تحت کنترل خود قرار دهد تا در موقع لزوم و به صورت آنی، بی‌درنگ، در لحظه و در زمان حقیقی و واقعی (تا دیر نشده) با تهدید مقابله کند، نرم‌افزارهای بی‌درنگ این نگهبان همیشه هوشیار و همیشه بیدار هستند. مانند نرم‌افزارهای ترمز اتومبیل، کنترل ضربان قلب اتاق بیهوشی، کنترل فشار کابین هواپیما، دستگاه فتوکپی، SafeHome و...

توجه: هر نرم‌افزاری که توسط سنسور و حسگر، عالم خارج را شنود و مورد ارزیابی قرار می‌دهد، تا در موقع لزوم و در زمان واقعی، حقیقی و تا دیر نشده عکس‌العمل نشان دهد، یک نرم‌افزار بی‌درنگ است.

توجه: فایل‌های واسط خارجی برنامه شامل هم محلی برای نگهداری تنظیمات مربوط به حسگرها و هم محلی برای ورود داده کنترلی به زیرسیستم جهت پایش و پاسخ سریع و بی‌وقفه به رویدادهای محیط پیرامون خود است.

توجه: فایل‌های منطقی داخلی برنامه و فایل‌های واسط خارجی برنامه درون مخزن داده‌ها نگهداری می‌شوند.

پس از تعیین مقادیر دامنه اطلاعاتی پروژه (information domain)، باید جدول زیر تکمیل گردد.

			ضرایب وزنی				
			پیچیده	متوسط	ساده		تعداد	دامنه اطلاعاتی
	=		۶	۴	۳	×		تعداد ورودی‌های برنامه
	=		۷	۵	۴	×		تعداد خروجی‌های برنامه
	=		۶	۴	۳	×		تعداد پرس و جوهای برنامه
	=		۱۵	۱۰	۷	×		تعداد فایل‌های منطقی داخلی برنامه
	=		۱۰	۷	۵	×		تعداد فایل‌های واسط خارجی برنامه
								تعداد کل

هدف از تکمیل جدول فوق، تعیین مقدار تعداد کل (count total) است، زیرا این مقدار پارامتری از رابطه نقطه عملکرد (امتیاز کارکرد) است.

در جدول فوق مقدار ضریب وزنی مشخص می‌کند که هریک از عناصر دامنه اطلاعاتی (information domain) به چه ضریب وزنی، پیچیدگی (complexity) را به برنامه تحمیل می‌کنند.

سازمان‌هایی که روش نقطه عملکرد را به کار می‌برند، ملاک‌هایی برای تعیین میزان ضریب وزنی عناصر دامنه اطلاعاتی (information domain) به شکل ساده، متوسط و پیچیده را مشخص می‌کنند. شمارش کل، حاصل جمع تعداد همه عناصر دامنه اطلاعاتی (information domain) با احتساب ضرایب وزنی آن‌ها است که از جدول مذکور به دست می‌آید.

تنها قدم باقی‌مانده جهت محاسبه نقطه عملکرد (FP) تعیین مقدار $\sum(F_i)$ است. در واقع هر F_i عددی بین ۰ تا ۵ است که در پاسخ به چهارده پرسش متفاوت زیر تعیین می‌شود.

مقادیر F_i:

	بی‌تأثیر	ضعیف	متوسط	خوب	عالی	بسیار عالی
مقادیر F_i	۰	۱	۲	۳	۴	۵

سؤالات مربوط به F_iها:

۱- آیا سیستم نیاز به داشتن نسخه پشتیبان و ترمیم دارد؟

۲- آیا ارتباطات داده‌ای موردنیاز است؟

۳- آیا عملیات پردازش توزیع شده وجود دارند؟

۴- آیا کارایی اهمیت دارد؟

۵- آیا سیستم در محیط عملیاتی موجود و طبق نظر مشتری اجرا خواهد شد؟

۶- آیا سیستم به ورودی داده‌های آنلاین نیاز دارد؟

۷- آیا ورود داده‌های آنلاین نیاز به تراکنش ورودی دارد تا در عملیات‌ها یا صفحه نمایش‌های چندگانه ایجاد گردد؟

۸- آیا فایل‌های اصلی به صورت آنلاین بروزرسانی می‌شوند؟

۹- آیا ورودی‌ها، خروجی‌ها، فایل‌ها و درخواست‌ها پیچیده‌اند؟

۱۰- آیا پردازش داخلی پیچیده است؟

۱۱- آیا کد طراحی شده قابل استفاده مجدد می‌باشد؟

۱۲- آیا تغییرات و نصب در طراحی در نظر گرفته شده است؟

۱۳- آیا سیستم طوری طراحی شده است که بتوان آن را چند بار در سازمان‌های متفاوت نصب کرد؟

۱۴- آیا نرم‌افزار طوری طراحی شده است که امکان تغییرات را بدهد و کاربر به سادگی از آن استفاده نماید؟

توجه: با استفاده از مقیاسی که از صفر (عدم اهمیت یا لزوم اجرا) تا ۵ (مطقاً ضروری) تغییر می‌کند، به هریک از پرسش‌های فوق پاسخ داده می‌شود.

توجه: اگر در پاسخ به پرسش «آیا کد طراحی شده قابل استفاده مجدد می‌باشد؟» مقدار ۵ برای F_i در نظر گرفته شود، یعنی برنامه در حداکثر قابلیت استفاده مجدد قرار دارد که پیچیدگی برنامه افزایش می‌یابد.

توجه: به هر F_i مقدار فاکتور تعدیل (Value Adjustment Factor) گفته می‌شود. حداکثر مقدار $\sum(F_i)$ برابر ۷۰ یعنی ($۱۴\times۵=۷۰$) و حداقل مقدار آن برابر ۰ یعنی ($۱۴\times۰=۰$) است. زمانیکه مقدار $\sum(F_i)$ برابر مقدار متوسط خود یعنی ۳۵ باشد، مقدار نقطه عملکرد (FP) برابر با تعداد کل (Count Total) می‌شود.

مطابق رابطه نقطه عملکرد (FP) داریم:

$$FP = \text{تعداد کل} \times \left[۰/۶۵+۰/۰۱\times\sum(F_i)\right]$$

$$FP = \text{تعداد کل} \times \left[۰/۶۵+۰/۰۱\times۳۵\right]$$

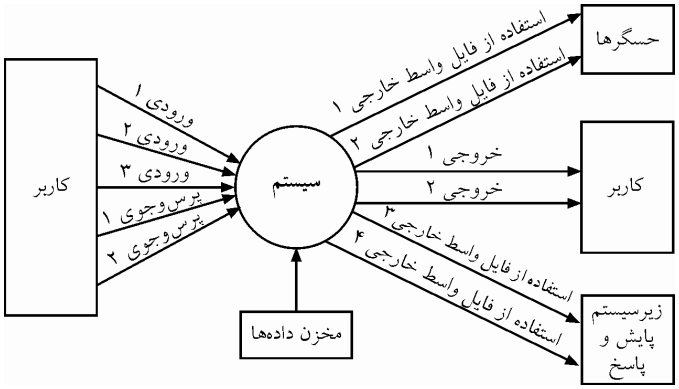
$$FP = \text{تعداد کل} \times \left[۰/۶۵+۰/۳۵\right]$$

$$FP = \text{تعداد کل} \times \left[۱\right]$$

$$FP = \text{تعداد کل}$$

توجه: مقادیر ثابت در رابطه نقطه عملکرد (FP) و ضرایب وزنی که در شمارش دامنه اطلاعاتی (information domain) به کار برده می‌شوند، به طور تجربی به دست آمده‌اند.

مثال: مدل تحلیل زیر را در نظر بگیرید:



مقادیر ضریب وزنی مربوط به عناصر دامنه اطلاعاتی (information domain)، به صورت زیر است:

دامنه اطلاعاتی	تعداد	ضریب وزنی	ضریب وزنی ×تعداد
ورودی‌های برنامه	3	3	9
خروجی‌های برنامه	2	4	8
پرس و جوهای برنامه	2	3	6
فایل‌های منطقی داخلی برنامه	1	7	7
فایل‌های واسط خارجی برنامه	4	5	20
تعداد کل			50

توجه: می‌توان گفت که هر یک عدد فایل منطقی داخلی 2.5 برابر نسبت به ورودی‌های برنامه و پرس و جوهای برنامه، پیچیدگی برنامه را افزایش می‌دهد.

توجه: فرض کنید مقدار $\sum(F_i)$ برابر 46 باشد، یعنی محصولی با پیچیدگی معتدل.

بنابراین مقدار نقطه عملکرد (FP) برای مدل فوق به صورت زیر محاسبه می‌گردد:

$$FP = \text{تعداد کل} \times \left[۰/۶۵ + ۰/۰۱ \times \sum(F_i)\right]$$

$$FP = ۵۰ \times \left[۰/۶۵ + ۰/۰۱ \times ۴۶\right] = ۵۶$$

یک نرم‌افزار علاوه بر اندازه اصلی نقطه عملکرد (FP) در روش اندازه‌گیری عملکردگرا، اندازه‌های فرعی دیگری همچون هزینه برای هر FP، تعداد خطاها (error) و تعداد نقص‌ها (defect) دارد. که در آن error تعداد خطاهای یافت شده پیش از تحویل نرم‌افزار به مشتری و defect تعداد نقایص یافت‌شده پس از تحویل به مشتری است. حال اگر نسبت این اندازه‌های فرعی به اندازه اصلی یعنی در هر FP تناسب‌گیری و محاسبه گردد، آنگاه حاصل این تناسبات در یک نرم‌افزار خاص بازهم یک **اندازه (measure)** خواهد بود. **اما** اگر نسبت این اندازه‌های فرعی به اندازه اصلی یعنی در هر FP تناسب‌گیری و محاسبه گردد، آنگاه حاصل این تناسبات در مقام **مقایسه** با نرم‌افزاری دیگر یک **معیار (metric) عملکردگرا** خواهد بود. این معیار جهت مقایسه، بررسی و ارزیابی فرآیند تولید نرم‌افزار در گذشته دو یا چند نرم‌افزار مورد استفاده قرار می‌گیرد. این مقایسه **گذشته** منجر به پندگیری و به تبع بهبود فرآیند تولید نرم‌افزار در **آینده** می‌گردد.

در جدول زیر، اندازه‌های اصلی و فرعی دو پروژه به نام‌های A و B که در گذشته ایجاد شده‌اند، قرار گرفته است:

نام پروژه	مقدار نقطه عملکرد (FP)	هزینه (تومان)	خطا (error)	نقص (defect)
A	710	300000000	44	6
B	540	340000000	70	4

حال اگر نسبت این اندازه‌های فرعی به اندازه اصلی یعنی در هر FP تناسب‌گیری و محاسبه گردد، آنگاه جدول زیر را خواهیم داشت:

نام پروژه	مقدار نقطه عملکرد (FP)	هزینه برای هر FP (تومان)	خطا در هر FP (error)	نقص در هرFP (defect)
A	710	422535	0.06	0.008
B	540	629629	0.12	0.007

توجه: مطابق نتایج جدول فوق واضح است که در مقام مقایسه و معیار (metric) عملکردگرا برای دو تیم A و B، تیم B در کشف خطاهای برنامه موفق‌تر بوده است.

امتیاز ویژگی (Feature Point)

مقدار نقطه عملکرد یا امتیاز عملکرد (Function Point) برای محاسبه‌ی درجه‌ی پیچیدگی سیستم‌های تجاری بی‌درنگ و مدیریت بانک اطلاعات مورد استفاده قرار می‌گیرد، در نرم‌افزارهای مدیریت بانک اطلاعات حجم داده‌ها بالا و حجم محاسبات پایین است. همانطور که پیش‌تر گفتیم در محاسبه‌ی مقدار **امتیاز عملکرد** دامنه‌ی اطلاعاتی پروژه شامل ورودی‌های برنامه، خروجی‌های برنامه، پرس و جوهای برنامه، فایل‌های منطقی داخلی برنامه و فایل‌های واسط خارجی برنامه مورد شمارش قرار می‌گیرند. اما مقدار **امتیاز ویژگی** برای محاسبه‌ی درجه‌ی پیچیدگی سیستم‌های علمی و مهندسی مورد استفاده قرار می‌گیرد، نرم‌افزارهای علمی و مهندسی برای کاربردهایی با محاسبات پیچیده و سنگین مورد استفاده قرار می‌گیرند، مانند نرم‌افزارهای محاسبات ریاضی (مثل ضرب ماتریس‌ها و معکوس‌سازی ماتریس‌ها)، رمزگشایی یک متن، علوم زمین‌شناسی، ستاره‌شناسی و کنترل سیستم‌های صنعتی. در این نوع نرم‌افزارها حجم داده‌ها پایین و حجم محاسبات بالا است. در این نوع نرم‌افزاها برای محاسبه‌ی مقدار امتیاز ویژگی علاوه بر دامنه‌ی اطلاعاتی پروژه، **تعداد الگوریتم‌های برنامه** نیز مورد شمارش قرار می‌گیرند. نتیجه اینکه روش **امتیاز ویژگی** (Feature Point) جهت اندازه‌گیری سیستم‌های علمی و مهندسی که پیچیدگی الگوریتمی آنها بسیار بالاست مورد استفاده قرار می‌گیرد.

توجه: معیارهای مبتنی بر تعداد خطوط برنامه (روش اندازه‌گرا) و معیارهای مبتنی بر نقاط عملکرد (روش عملکردگرا)، پیشگویی‌های نسبتاً دقیقی برای حجم فعالیت‌ها و هزینه توسعه نرم‌افزارهای آتی درآینده به شمار می‌روند.

تبدیل LOC به FP و FP به LOC

نسبت بین LOC و FP وابسته به زبان‌های برنامه‌نویسی است.

مثال: فرض کنید برنامه‌ای با زبان C++ و assembly پیاده‌سازی شده است. اگر پیاده‌سازی این برنامه ۳۲۰۰۰ خط کد به زبان C++ و ۴۸۰۰ خط کد به زبان assembly نیاز داشته باشد، مقدار Function Point کل این برنامه چقدر است؟

مقدار متوسط LOC در زبان‌های C++ و assembly به ازای یک FP به صورت زیر است:

	C++	assembly
LOC	64	320

مطابق مقادیر فوق برای محاسبه‌ی Function Point کل برنامه داریم:

$$FP_{Total} = FP_{C++} + FP_{assembly} = \frac{۳۲۰۰۰}{۶۴} + \frac{۴۸۰۰}{۳۲۰} = ۵۰۰ + ۱۵ = ۵۱۵$$

اندازه‌گیری کیفیت نرم‌افزار

اندازه‌گیری کیفیت نرم‌افزار در چهار بخش صحت (Correctness)، قابلیت نگهداری

(Maintainability)، یکپارچگی یا جامعیت یا تمامیت (Integrity) و قابلیت استفاده (Usability) انجام می‌گردد.

معیارهای کیفیت نرم‌افزار

هدف اصلی مهندسی نرم‌افزار، تولید سیستم، برنامه‌ی کاربردی یا محصولی با کیفیت بالاست. برای رسیدن به این هدف مهندسان نرم‌افزار باید روش‌هایی کارآمد را همراه با ابزارهای ویژه در فرآیند نرم‌افزار مورد استفاده قرار دهند. همچنین، یک مهندس نرم‌افزار توانا، اگر به کیفیت بالا اهمیت می‌دهد، باید روند توسعه نرم‌افزار را به طور مدون اندازه‌گیری نماید.

یک مهندس نرم‌افزار، برای تأمین کیفیت نرم‌افزار باید به اندازه‌گیری و ارزیابی کیفیت تحلیل، مدل‌های طراحی، کد و موارد تست در طول فرآیند توسعه نرم‌افزار بپردازد.

اندازه‌گیری کیفیت

اگرچه برای تعیین کیفیت نرم‌افزار اندازه‌گیری‌های مختلفی در نظر گرفته شده است، اما اندازه‌گیری‌هایی بعد از تحویل نرم‌افزار مورد توجه قرار می‌گیرند و از اهمیت بیشتری برخوردارند:

۱- صحت (Correctness)

در بخش صحت، اندازه کیفیت نرم‌افزار به شکل مستقیم (direct measure) و از طریق رابطه Defects per KLOC محاسبه می‌گردد.

یک برنامه باید درست کار کند وگرنه برای کاربران خود ارزشی ندارد. صحت، حدی از کارکرد نرم‌افزار برای انجام صحیح وظایف آن است. متداول‌ترین اندازه‌گیری برای صحت، تعداد نقایص به ازای هر KLOC (هزار خط کد) است. منظور از نقص (defect)، همان خطاهایی است که پس از تحویل نرم‌افزار و توسط مشتری شناسایی و مشخص می‌شود. بدیهی است که هرچه تعداد نقص‌های برنامه کمتر باشد، مشتری رضایت بیشتری از برنامه خواهد داشت.

۲- قابلیت نگهداری (Maintainability)

نگهداری نرم‌افزار از هر فعالیت دیگر مهندسی نرم‌افزار بیشتر کار می‌برد. قابلیت نگهداری، سادگی تصحیح یک برنامه در صورت وقوع خطا، تغییر محیط عملیاتی و تغییر نیازمندی‌های مشتری، می‌باشد. راهی برای اندازه‌گیری مستقیم قابلیت نگهداری وجود ندارد، بنابراین باید از اندازه‌گیری‌های غیرمستقیم استفاده شود.

یک معیار ساده مبتنی بر زمان «میانگین زمان لازم برای تغییر» (MTTC: Mean Time-To-Change) است، که برابر با مدت زمانی که طول می‌کشد تا درخواست تغییر، تحلیل، طراحی، پیاده‌سازی، تست، انجام و سپس نسخه‌ی تغییریافته میان کاربران توزیع گردد. هر چه MTTC کمتر باشد قابلیت نگهداری بالاتر است. به طور میانگین، برنامه‌هایی که قابل نگهداری هستند، از برنامه‌هایی که قابل نگهداری نیستند، MTTC کمتری دارند.

یک معیار مبتنی بر هزینه «ریخت و پاش یا ضایعات» (Spoilage) است که توسط «هیتاچی»

ابداع شده است و عبارتند از هزینه‌ای که صرف رفع نواقصی می‌شود که پس از ارایه سیستم به کاربران شناسایی می‌شوند.

۳- یکپارچگی یا جامعیت یا تمامیت (Integrity)

این صفت مقاومت و ایستادگی سیستم را در برابر حملات امنیتی اندازه‌گیری می‌کند. این حملات روی هر مؤلفه نرم‌افزار (برنامه‌ها، داده‌ها و مستندات) صورت می‌پذیرد. برای اندازه‌گیری یکپارچگی، مقدار دو صفت زیر را باید تعیین کرد:

الف) تهدید (Threat)

احتمال وقوع حمله‌ای از یک نوع خاص در زمان معین است، که می‌توان آن را از روی شواهد تجربی تخمین زد یا به دست آورد.

ب) امنیت (Security)

احتمال دفع حمله‌ای از یک نوع خاص است که آن را نیز از روی شواهد تجربی می‌توان تخمین زد یا به دست آورد. بنابراین یکپارچگی سیستم را به صورت زیر می‌توان تعریف کرد:

$$\text{یکپارچگی} = \sum[1 - (\text{تهدید} \times (1 - \text{امنیت}))]$$

که در آن تهدید و امنیت روی همه‌ی انواع حملات جمع بسته می‌شوند.

مثال: اگر در سیستمی احتمال حمله به آن برابر ۲۰ درصد و احتمال دفع حمله در مکانیزم امنیتی آن برابر ۴۰ درصد باشد، و فقط همین حمله برای سیستم محتمل باشد، مقدار یکپارچگی این سیستم چقدر است؟

پاسخ: مطابق رابطه یکپارچگی داریم:

$$\text{یکپارچگی} = [1 - (۰/۲ \times ۰/۶)] = [1 - ۰/۱۲] = ٪۸۸$$

مثال: اگر در سیستمی احتمال حمله به آن برابر ۱۰۰درصد و احتمال دفع حمله در مکانیزم امنیتی آن برابر ۱۰۰ درصد باشد، و فقط همین حمله برای سیستم محتمل باشد، مقدار یکپارچگی این سیستم چقدر است؟

پاسخ: مطابق رابطه یکپارچگی داریم:

$$\text{یکپارچگی} = [1 - (1 \times ۰)] = [1 - ۰] = ٪۱۰۰$$

مثال: اگر در سیستمی احتمال حمله به آن برابر۰ درصد و احتمال دفع حمله در مکانیزم امنیتی آن برابر ۱۰۰ درصد باشد، و فقط همین حمله برای سیستم محتمل باشد، مقدار یکپارچگی این سیستم چقدر است؟

پاسخ: مطابق رابطه یکپارچگی داریم:

$$\text{یکپارچگی} = [1 - (۰ \times ۰)] = [1 - ۰] = ٪۱۰۰$$

نتیجه اینکه اگر **سپردفاعی** یعنی امنیت ۱۰۰درصد باشد، مستقل از اینکه تهدید به چه میزان باشد، یکپارچگی ۱۰۰درصد است. به قول سون تزو، ژنرال چینی که ۲۵۰۰ سال قبل می‌زیسته است: «اگر دشمن و خودتان را بشناسید، دیگر لازم نیست از صد تیر هم هراس داشته باشید.»

۴- قابلیت استفاده یا سهولت کاربرد (Usability)

عبارت «سهولت استفاده» (User Friendliness) در بحث‌های محصولات نرم‌افزاری به وفور مشاهده می‌شود. اگر استفاده از برنامه‌ای آسان نباشد سرنوشت آن شکست خواهد بود، حتی اگر اعمالی که انجام می‌دهد، ارزشمند و به درستی پیاده‌سازی شده‌باشد. و این یعنی توجه به برآورده‌سازی نیازمندی‌های وظیفه‌مندی و عدم توجه به برآورده‌سازی نیازمندی‌های غیروظیفه‌مندی که از جنس مزه سیستم است. قابلیت استفاده، کوشش برای تعیین کمّی سهولت استفاده برحسب چهار خصوصیت قابل اندازه‌گیری است:

الف) مهارت فیزیکی و هوش لازم برای فراگیری سیستم
ب) متوسط زمان لازم برای یادگیری سیستم
ج) میزان بهره‌وری سیستم، وقتی یک کاربر معمولی از آن استفاده می‌کند.
د) نظرسنجی از کاربران سیستم

کارآیی رفع نقص

کارآیی رفع نقص یا DRE که سرواژه‌ی عبارت Defect Removal Efficiency می‌باشد، DRE **اندازه‌ای** برای کنترل فعالیت‌های مربوط به تضمین کیفیت در روند فرآیند تولید نرم‌افزار است. به بیان دیگر DRE **اندازه‌ای** است برای نشان دادن توانایی فعالیت‌های تضمین و کنترل کیفیت که در طی فرآیند به کار گرفته شده‌است. اگر این **اندازه** محاسبه گردد، آنگاه حاصل این اندازه در مقام **مقایسه** با گذشته یک فعالیت از فرآیند تولید نرم‌افزار یا کل فعالیت‌های پروژه یک **معیار (metric) اندازه‌گرا** خواهد بود. این معیار جهت مقایسه، بررسی و ارزیابی یک فعالیت از فرآیند تولید نرم‌افزار یا کل فعالیت‌های پروژه با گذشته همان پروژه یا پروژه **دیگر** مورد استفاده قرار می‌گیرد. این مقایسه **گذشته** منجر به پندگیری و به تبع بهبود فرآیند تولید نرم‌افزار در **آینده** می‌گردد. بنابراین DRE یک معیار کیفیتی است که هم در سطح پروژه و هم در سطح بهبود فرآیند تولید نرم‌افزار مفید واقع می‌شود.

هنگامی که DRE برای کل پروژه در نظر گرفته شود، به شیوه‌ی زیر تعریف می‌شود:

$$DRE = \frac{E}{E + D}$$

که در آن Error یا E تعداد خطاهای یافت‌شده پیش از تحویل نرم‌افزار به مشتری و Defect یا D تعداد نقایص یافت‌شده پس از تحویل به مشتری است.

توجه: مقدار ایده‌آل برای DRE برابر با یک است. یعنی اینکه هیچ نقصی پس از تحویل نرم‌افزار به مشتری یافت نشود. یعنی تمام خطاهای برنامه قبل از تحویل به مشتری کشف شده

باشد، که این امر منجر به این می‌شود تعداد نقایص یافت‌شده پس از تحویل به مشتری برابر صفر گردد. (D = ۰)

توجه: با افزایش E، احتمال کاهش مقدار نهایی D بیشتر می‌شود، زیرا خطاها قبل از تحویل نرم‌افزار به مشتری شناسایی شده‌اند. به بیان دیگر خطاها قبل از اینکه به عنوان نقص آشکار شوند، کشف و برطرف شده‌اند. DRE به عنوان یک اندازه و معیار، تیم توسعه را تشویق می‌کند تا در کشف خطاها قبل از تحویل نرم‌افزار به مشتری کوشاتر، دقیق‌تر و فنی‌تر باشد. زیرا اگر این خطاها توسط تیم توسعه نرم‌افزار کشف (uncover) نگردد، آنگاه در آینده توسط مشتری کشف می‌گردد و این یعنی ایجاد یک محصول نرم‌افزاری که مورد رضایت‌مندی مشتری قرار نمی‌گیرد.

مثال: یک تیم نرم‌افزاری، نسخه‌ی نخست نرم‌افزار خود را به مشتری تحویل می‌دهد. این مشتری، طی ماه نخست استفاده، ۸ نقص کشف می‌کند. تیم نرم‌افزاری پیش از تحویل، ۲۴۲ خطا طی مرورهای فنی رسمی و تمامی وظایف فعالیت آزمون یافته است. DRE کل برای پروژه پس از یک ماه استفاده چقدر است؟

پاسخ: هنگامی‌که DRE برای کل پروژه در نظر گرفته شود، به شیوه‌ی زیر تعریف می‌شود:

$$DRE = \frac{E}{E + D}$$

که در آن E تعداد خطاهای یافت‌شده پیش از تحویل نرم‌افزار به مشتری برابر ۲۴۲ و D تعداد نقایص یافت‌شده پس از تحویل به مشتری برابر ۸ است. بنابراین مقدار DRE مطابق رابطه فوق، به صورت زیر محاسبه می‌گردد:

$$DRE = \frac{E}{E + D} = \frac{۲۴۲}{۲۴۲ + ۸} = \frac{۲۴۲}{۲۵۰} = ۰/۹۶۸$$

همچنین DRE می‌تواند در تمامی مراحل فعالیت‌های چارچوبی در فرآیند تولید نرم‌افزار قابل استفاده باشد. بهتر است در انتهای هر مرحله از فعالیت‌های چارچوبی (ارتباطات، برنامه‌ریزی، مدل‌سازی (تحلیل و طراحی)، ساخت (پیاده‌سازی و تست) و استقرار) بازبینی‌های فنی رسمی (formal technical review) با هدف کشف خطاهای رخ داده در آن مرحله انجام شود. علت این موضوع این است که با انتشار هر خطا به مرحله‌ی بعد، باید هزینه‌ی بیشتر صرف شناسایی و رفع آن گردد. برای‌مثال اگر خطایی در مرحله مدل‌سازی رخ داده و در همان مرحله کشف، شناسایی و مرتفع نگردد، به مراتب هزینه‌ی کشف، شناسایی و رفع آن در مرحله‌ی ساخت بیشتر است.

هنگامی‌که DRE برای داخل پروژه، یعنی بخشی از پروژه یا فعالیت‌های فرآیند تولید نرم‌افزار در نظر گرفته شود، به شیوه‌ی زیر تعریف می‌شود:

$$DRE_i = \frac{E_i}{E_i + E_{i+۱}}$$

که در آن E_i تعداد خطاهای یافت‌شده در فعالیت مرحله iام است که توسط تیم بازبینی در همان مرحله اصلاح و مرتفع شده‌است و $E_{i+۱}$ تعداد خطاهای یافت‌شده مربوط به فعالیت مرحله

iام است که توسط تیم بازبینی در همان مرحله اصلاح و مرتفع نشده‌است و کشف آن تا مرحله i+1ام به تعویق افتاده‌است.

توجه: مقدار ایده‌آل برای DRE_i برابر با یک است. یعنی اینکه هیچ خطایی از فعالیت مرحله‌ی قبلی، پس از ورود به فعالیت مرحله‌ی بعدی یافت نشود. یعنی تمام خطاهای یک فعالیت قبل از ورود به فعالیت بعدی کشف شده باشد، که این امر منجر به این می‌شود تعداد خطاهای یافت‌شده از فعالیت مرحله‌ی قبلی، پس ورود به فعالیت مرحله‌ی بعدی برابر صفر گردد. $(E_{i+1}=0)$

توجه: با افزایش E_i، احتمال کاهش مقدار E_{i+1} بیشتر می‌شود، زیرا خطاها قبل از ورود به مرحله‌ی بعدی در همان مرحله‌ی جاری شناسایی شده‌اند. به بیان دیگر خطاها قبل از اینکه در مرحله‌ی بعدی آشکار شوند، در همان مرحله‌ی جاری کشف و برطرف شده‌اند. DRE_i به عنوان یک اندازه و معیار، تیم توسعه را تشویق می‌کند تا در کشف خطاها در همان مرحله‌ی جاری و قبل از ورود به مرحله‌ی بعدی کوشاتر، دقیق‌تر و فنی‌تر باشد. زیرا اگر این خطاها توسط تیم توسعه نرم‌افزار در همان مرحله‌ی جاری کشف، شناسایی و مرتفع نگردد، آنگاه در مرحله‌ی بعدی با هزینه‌ی بیشتر می‌بایست کشف، شناسایی و مرتفع گردد و این یعنی هدر دادن زمان و منابع.

مثال: در پایان یک پروژه، تعیین شده‌است که ۳۰ خطا طی فعالیت مدل‌سازی و ۱۲ خطا طی فعالیت ساخت کشف شده است که رد آنها تا خطاهای کشف نشده در فعالیت مدل‌سازی گرفته شده‌است. DRE برای فعالیت مدل‌سازی چقدر است؟

هنگامی‌که DRE برای داخل پروژه، یعنی بخشی از پروژه یا فعالیت‌های فرآیند تولید نرم‌افزار در نظر گرفته شود، به شیوه‌ی زیر تعریف می‌شود:

$$DRE_i = \frac{E_i}{E_i + E_{i+1}}$$

که در آن E_i تعداد خطاهای یافت‌شده در فعالیت مرحله مدل‌سازی برابر ۳۰ است که توسط تیم بازبینی در همان مرحله مدل‌سازی اصلاح و مرتفع شده‌است و E_{i+1} تعداد خطاهای یافت‌شده مربوط به فعالیت مدل‌سازی است که توسط تیم بازبینی در همان مرحله مدل‌سازی اصلاح و مرتفع نشده‌است و کشف آن تا مرحله ساخت به تعویق افتاده‌است، که برابر ۱۲ است. بنابراین مقدار DRE_i مطابق رابطه فوق، به صورت زیر محاسبه می‌گردد:

$$DRE_i = \frac{E_i}{E_i + E_{i+1}} = \frac{۳۰}{۳۰+۱۲} = \frac{۳۰}{۴۲} = ۰/۷۱۴$$

برآورد پروژه‌های نرم‌افزاری

انجام برآورد برای پروژه‌های نرم‌افزاری با مجموعه‌ای از فعالیت‌ها آغاز می‌شود که در مجموع، برنامه‌ریزی پروژه نامیده می‌شود. پیش از آنکه بتوان پروژه را آغاز کرد، تیم نرم‌افزاری باید کاری را که قرار است انجام شود، منابع مورد نیاز و زمانی را که از آغاز تا پایان سپری خواهد شد،

برآورد کند. هنگامی که این فعالیت‌ها انجام شوند، تیم نرم‌افزاری باید یک زمان‌بندی برای پروژه وضع کند که وظایف مهندسی نرم‌افزار و نقاط عطف را تعریف کند، تعیین کند چه کسانی مسئول اجرای هر وظیفه هستند و وابستگی‌های بین وظایفی را مشخص کند که تاثیری قوی بر پیشرفت کار دارند.

برنامه‌ریزی، یک تعهد اولیه ایجاد می‌کند، حتی اگر این احتمال باشد که این تعهد اشتباه از آب درآید. هرگاه که برآوردهایی انجام می‌شود، در واقع نگاهی به آینده ایجاد می‌شود، و به تبع همین آینده‌نگری‌ها در برآوردها قدری هم عدم قطعیت قرار خواهد داشت. برآورد علاوه بر علم، هنر نیز محسوب می‌شود. برآورد بستری برای تمامی فعالیت‌های دیگر برنامه‌ریزی پروژه فراهم می‌کند، از آنجا که برنامه‌ریزی پروژه، نقشه‌ی راهی برای مهندسی نرم‌افزار موفق ارائه می‌دهد، عقل حکم می‌کند که کار برآورد با دقت و با حوصله انجام شود. چرا که رویکردهای برآوردی خوب و داده‌های تاریخی مستحکم، نویدبخش دستیابی به واقعیت از میان توقعات غیرممکن هستند. هرچه بیشتر بدانید، بهتر برآورد می‌کنید، بنابراین به موازات پیشرفت پروژه، برآوردهای خود را بهنگام کنید.

فرآیند برنامه‌ریزی پروژه

هدف برنامه‌ریزی پروژه‌ی نرم‌افزاری، فراهم ساختن چارچوبی است که مدیر را قادر به انجام برآوردهای منطقی در خصوص **منابع**، **هزینه‌ها** و **زمان‌بندی** می‌کند. به علاوه، این برآوردها باید بتوانند سناریوهای بدترین حالت و بهترین حالت را مشخص کنند به گونه‌ای که بتوان برای پیامدهای پروژه حد و مرزی قائل شد. اگرچه به طور ذاتی، درجه‌ای از عدم قطعیت وجود دارد، اما تیم نرم‌افزاری به برنامه‌ای روی می‌آورد که در نتیجه‌ی انجام این وظایف بنا نهاده شده باشد. بنابراین، برنامه باید با پیشرفت کار، بهنگام‌سازی شود و بر خواسته‌های جدید تطابق یابد. فرآیند برنامه‌ریزی پروژه شامل مراحل تعیین دامنه‌ی پروژه، برآورد منابع مورد نیاز پروژه، اندازه‌گیری پروژه‌های مشابه گذشته، برآورد نیرو، هزینه و زمان لازم برای انجام پروژه فعلی، زمان‌بندی و مدیریت ریسک می‌باشد.

دامنه نرم‌افزار و امکان‌سنجی

اولین مرحله در فرآیند برنامه‌ریزی پروژه تعیین دامنه‌ی نرم‌افزار است، دامنه‌ی نرم‌افزار کارکردها و خصوصیاتی که باید به کاربر نهایی تحویل داده شود را مشخص می‌کند. دامنه‌ی نرم‌افزار از طریق یکی از دو تکنیک زیر تعریف می‌شود:

۱- تهیه‌ی توصیفی روایی پس از برقراری ارتباط با تمامی ذی‌نفعان

۲- تهیه‌ی مجموعه‌ای از use caseها

کارکردهایی که به یکی از دو روش فوق مشخص می‌شوند مورد ارزیابی و در برخی موارد مورد پالایش قرار می‌گیرد تا پیش از آغاز برآورد، جزئیات بیشتری فراهم آید. زیرا در برآورد هزینه و زمان‌بندی پروژه موثر خواهند بود.

هنگامی‌که دامنه‌ی نرم‌افزار مشخص شد، این سوال مطرح می‌شود که آیا می‌توان نرم‌افزاری ایجاد کرد که دامنه‌ی نرم‌افزار را برآورده سازد؟ آیا اجرای پروژه امکان‌پذیر است؟

این بخش از فرآیند برنامه‌ریزی پروژه، بخش مهمی است که البته اغلب اوقات مورد بی‌توجهی قرار می‌گیرد. بنابراین فقط تعیین دامنه‌ی نرم‌افزار کافی نیست، بلکه باید امکان‌سنجی نیز انجام گردد تا پروژه به سمت سرنوشتی نامشخص سوق پیدا نکند. معیارهایی که برای بررسی امکان‌پذیری انجام پروژه در نظر گرفته می‌شود با چهار بخش زیر مرتبط است:

۱- **فناوری** (Technology): هدف از بررسی امکان‌پذیری فناوری، بررسی این موضوع است که آیا فناوری و فن لازم و کافی برای انجام پروژه وجود دارد یا خیر. به عبارت دیگر، هدف از این بررسی، شناسایی توانایی تیم نرم‌افزاری برای ایجاد سیستم پیشنهاد شده می‌باشد. این شناخت باید شامل بررسی و آگاهی تیم نرم‌افزاری از هدف احتمالی، سخت‌افزار، نرم‌افزار، محیط عملیاتی که استفاده می‌شود، به انضمام اندازه سیستم، میزان پیچیدگی آن و تجربه تیم نرم‌افزاری در ایجاد سیستم‌های مشابه باشد.

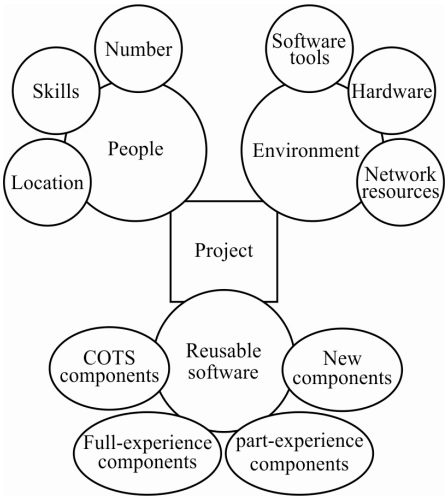
۲- **مالی** (Finance): هدف از بررسی امکان‌پذیری مالی، بررسی این موضوع است که آیا پول لازم و کافی برای انجام پروژه وجود دارد یا خیر. به عبارت دیگر، هدف از این بررسی، تعیین هزینه‌های مالی مربوط به پروژه‌ی نرم‌افزاری که با آن مواجه هستیم، می‌باشد. پروژه‌ای که انجام می‌شود باید از توجیه اقتصادی خوبی برای سازنده و مشتری برخوردار باشد، در غیر اینصورت انجام آن مقرون به صرفه نخواهد بود. در مورد نرم‌افزارهای تجاری نیز باید بررسی شود که آیا بازار توان خرید نرم‌افزار را دارد یا خیر.

۳- **زمان** (Time): هدف از بررسی امکان‌پذیری زمانی، بررسی این موضوع است که آیا زمان لازم و کافی برای انجام پروژه وجود دارد یا خیر. به عبارت دیگر، هدف از این بررسی، بررسی این موضوع است که آیا کلیه‌ی فعالیت‌های مهم پروژه در محدوده‌های زمانی تعیین شده توسط تیم نرم‌افزاری قابل انجام است یا خیر. به عبارت دیگر، باید مشخص کرد که آیا پروژه بر اساس زمان‌بندی پیش خواهد رفت یا خیر. در مورد نرم‌افزارهای تجاری نیز باید بررسی شود که آیا زمان عرضه به بازار قابل رقابت است یا خیر.

۴- **منابع** (Resources): هدف از بررسی امکان‌پذیری منابع، بررسی این موضوع است که آیا منابع لازم و کافی برای انجام پروژه وجود دارد یا خیر.

منابع

دومین مرحله در فرآیند برنامه‌ریزی پروژه برآورد منابع مورد نیاز برای انجام پروژه است، در شکل زیر سه طبقه‌ی اصلی منابع مهندسی نرم‌افزار نشان داده است. این سه طبقه شامل **منابع انسانی** (افراد)، **منابع نرم‌افزاری** (مولفه‌های نرم‌افزاری با قابلیت استفاده مجدد) و **منابع محیطی** (محیط توسعه یا ابزارهای سخت‌افزاری و نرم‌افزاری) می‌باشد. به هریک از این سه منبع، دوایری متصل شده است که بیان می‌کند مدیر پروژه باید به چه خصوصیاتی از هر منبع توجه کند.



منابع انسانی

برنامه‌ریز پروژه باید بر اساس دامنه‌ی پروژه، موقعیت سازمانی (مدیر، مهندس نرم‌افزار ارشد) و مهارت‌های لازم (شبکه، پایگاه داده و برنامه‌نویسی) برای توسعه نرم‌افزار را مشخص کند. برای پروژه‌های نسبتا کوچک (در حد چند نفر- ماه) تنها یک نفر ممکن است تمامی وظایف مهندسی نرم‌افزار را انجام دهد و بسته به نیاز با متخصصان مشورت کند. برای پروژه‌های بزرگتر، منابع انسانی ممکن است به لحاظ جغرافیایی در چند مکان متفاوت پراکنده باشند. بنابراین موقعیت مکانی هر یک از منابع انسانی باید مشخصص گردد. **تعداد** افراد لازم برای یک پروژه‌ی نرم‌افزاری را می‌توان تنها پس از برآورد تلاش و نیروی لازم برای توسعه‌ی نرم‌افزار (بر حسب نفر- ماه) تعیین کرد. نحوه محاسبه برآورد تلاش مورد نیاز در ادامه فصل بررسی می‌شود.

منابع محیطی

منابع محیطی، حامی و پشتیبان پروژه‌ی نرم‌افزاری است. و غالبا SEE که سرواژه‌ی عبارت Software Engineering Environment و به معنی محیط مهندسی نرم‌افزار است، خوانده می‌شود. این محیط شامل سخت‌افزار و نرم‌افزار است. سخت‌افزار، سکویی را فراهم می‌کند که از ابزارها (نرم‌افزارها) برای تولید محصولات کاری که به عنوان دستاوردهای فرآیند مهندسی نرم‌افزار هستند، حمایت کند.

منابع نرم‌افزاری (مولفه‌های نرم‌افزاری با قابلیت استفاده مجدد)

زمان و هزینه‌ی فرآیند تولید نرم‌افزار براساس مدل توسعه‌ی مبتنی بر مؤلفه به دلیل استفاده از قطعات آماده (قابلیت استفاده مجدد) کاهش چشمگیری دارد. در واقع یک قطعه با قابلیت استفاده‌ی مجدد یک بار ساخته می‌شود و بارها و بارها استفاده می‌شود و این یعنی سودآوری!

از نظر قابلیت استفاده مجدد، مولفه‌ها به چهار گروه زیر تقسیم می‌شوند:

- مولفه‌های آماده یا پیش‌ساخته (Off-the-shelf components)

مولفه‌های آماده مولفه‌هایی هستند که قبلا در پروژه‌های پیشین و یا شرکت سازنده‌ی دیگری توسعه داده شده‌اند، و به طور کامل با مولفه‌ی مورد نیاز فعلی تطابق دارند. مانند مولفه‌ی فروشگاه‌ساز ووکامرس در ورد پرس. اگر مولفه‌ای توسط شرکت دیگری توسعه داده شده است و در پروژه‌های پیشین ایجاد نشده است، اما در پروژه فعلی کاربرد دارد، بهتر است مولفه‌ی مورد نظر از شرکت سازنده خریداری شود، زیرا در اغلب موارد به دلایل اقتصادی هزینه‌ی خرید از هزینه‌ی توسعه‌ی مولفه کمتر است. که مقرون به صرفه هم هست و این یعنی سودآوری! این مولفه‌ها به COTS که سرواژه‌ی عبارت Commercial Off-The-Shelf می‌باشد نیز موسوم است.

- مولفه‌های مشابه با تجربیات قبلی (Full-experience components)

مولفه‌های مشابه مولفه‌هایی هستند که قبلا در پروژه‌های پیشین توسعه داده شده‌اند، و به طور بسیار زیاد با مولفه‌های مورد نیاز فعلی شباهت دارند. تیم توسعه در تبدیل مولفه‌های مشابه به مولفه‌های مورد نیاز باتجربه است، زیرا اختلاف کمی مابین مولفه‌های مشابه و مولفه‌های مورد نیاز فعلی وجود دارد. نتیجه اینکه تیم توسعه در اصلاح مولفه‌های مشابه، با تجربه است، بنابراین با اصلاحات بسیار کم، مولفه‌های مشابه می‌توانند با مولفه‌های مورد نیاز فعلی تطابق پیدا کنند و از همان در پروژه‌ی فعلی استفاده شود. که مقرون به صرفه هم هست و این یعنی سودآوری!

- مولفه‌های نسبتا مشابه (Partial-experience components)

مولفه‌های نسبتا مشابه مولفه‌هایی هستند که قبلا در پروژه‌های پیشین توسعه داده شده‌اند، و به طور بسیار کمی با مولفه‌های مورد نیاز فعلی شباهت دارند. تیم توسعه در تبدیل مولفه‌های نسبتا مشابه به مولفه‌های مورد نیاز بی‌تجربه است، زیرا اختلاف زیادی مابین مولفه‌های نسبتا مشابه و مولفه‌های مورد نیاز فعلی وجود دارد. نتیجه اینکه تیم توسعه در اصلاح مولفه‌های نسبتا مشابه، بی‌تجربه است، بنابراین با اصلاحات بسیار زیاد، مولفه‌های نسبتا مشابه می‌توانند با مولفه‌های مورد نیاز فعلی تطابق پیدا کنند اما این کار بسیار پُرریسک و پُرهزینه است که راه‌حل بهتر ساخت مولفه‌ی مورد نظر فعلی از ابتدا است. در غیر اینصورت هزینه‌ی تبدیل مولفه‌های نسبتا مشابه به مولفه‌های مورد نیاز فعلی از هزینه‌ی ساخت مولفه‌ی مورد نظر فعلی از ابتدا بیشتر می‌شود. که مقرون به صرفه هم نیست و این یعنی ...!

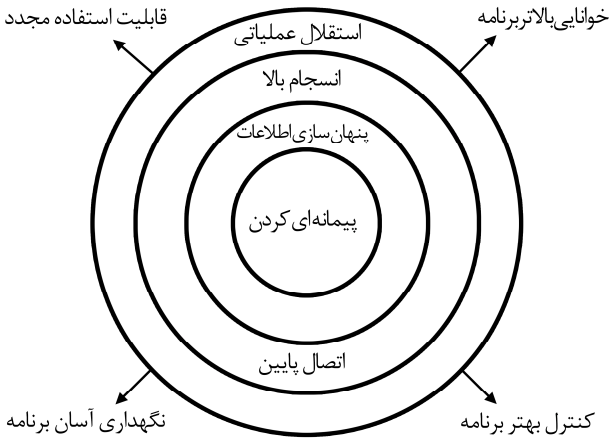
- مولفه‌های جدید (New Components)

مولفه‌های جدید مولفه‌هایی هستند که قبلا در پروژه‌های پیشین و یا شرکت سازنده‌ی دیگری توسعه داده نشده‌اند، و به طور کامل با مولفه‌های آماده، مشابه و نسبتا مشابه تفاوت دارند. بنابراین باید توسط تیم توسعه، ایجاد شوند. اما نه فقط ایجاد، بلکه ایجاد با یک طراحی ایده‌آل، در این شرایط است که هم حال پروژه فعلی‌تان خوب خواهد شد و هم حال پروژه آتی‌تان.

توجه: در یک طراحی ایده‌آل، هدف، محقق کردن بالاترین سطح انسجام (Cohesion بالا) داخل پیمانه‌های برنامه و کمترین سطح اتصال (Coupling پایین) مابین پیمانه‌های برنامه است.

توجه: پیمانه‌ای‌کردن، بسته‌بندی، برقراری پنهان‌سازی اطلاعات، انسجام بالا، اتصال پایین و برقراری استقلال عملیاتی، به عنوان اصول و معیارهای طراحی معماری مطلوب، منجر به **خوانایی بالاتر برنامه**، **کنترل بهتر برنامه**، **قابلیت استفاده مجدد** مولفه‌های (پیمانه‌های) برنامه جاری در برنامه‌های آتی و **نگهداری آسان برنامه** جاری می‌گردد.

شکل زیر گویای مطلب است:



برآورد اندازه‌ی نرم‌افزار

برآورد اندازه، نیروی مورد نیاز و هزینه‌های مربوط به پروژه و همچنین زمان لازم برای انجام کار فعالیت علمی دقیقی به شمار نمی‌آید. زیرا این برآوردها به متغییرهایی همچون نبوغ افراد، محیط نرم‌افزار و ماهیت خود پروژه وابسته و متصل است. اما به هر حال این سوال مطرح است که اندازه‌ی نرم‌افزاری را که قرار است بسازیم، چگونه تعیین کنیم؟

برآورد پروژه‌های نرم‌افزاری نیز همانند حل مسأله است. در اکثر موارد، مسأله‌ای که باید حل شود پیچیده‌تر از آن است که بتوان به یکباره به آن پرداخت. از همین رو مسأله تجزیه شده و به صورت یک سری مسائل کوچک‌تر بدل می‌گردد. برآورد پروژه‌های نرم‌افزاری نیز به همین شیوه باید انجام شود. یعنی برآورد از جزء به کل به شیوه تجزیه پروژه به بخش‌های مختلف آن. بنابراین جهت برآورد اندازه‌ی نرم‌افزار، می‌توان از روش‌های **مبتنی بر تجزیه** استفاده کرد. اما پوتنام (Putnam) و مایرز (Myers) روش‌های دیگری را نیز معرفی کرده‌اند که قبل از بیان روش‌های مبتنی بر تجزیه، به بررسی روش‌های پیشنهادی آنها می‌پردازیم.

برآورد اندازه نرم‌افزار با استفاده از روش‌های پوتنام و مایرز

پوتنام و مایرز چهار روش متفاوت برای تعیین اندازه نرم‌افزار پیشنهاد می‌کنند:

۱- تعیین اندازه به روش منطق فازی

در این روش با استفاده از تکنیک استدلال تقریبی که اساس منطق فازی را تشکیل می‌دهد

اندازه‌ای برای نرم‌افزار برآورد می‌شود. مدیر پروژه برای استفاده از این روش باید نوع نرم‌افزار را شناسایی کند، بزرگی آن را در مقیاس کیفی تعیین نماید و سپس این بزرگی را در گستره‌ی اصلی پالایش کند.

۲- تعیین اندازه به روش مؤلفه‌های استاندارد

هر نرم‌افزار متشکل از چند «مؤلفه استاندارد» است که در یک حیطه کاربردی خاص، کاربرد دارد. برای مثال، مؤلفه‌های استاندارد برای یک سیستم اطلاعاتی شامل زیرسیستم‌ها، پیمانه‌ها، صفحات نمایش و گزارش‌ها است. بنابراین در گام اول و ابتدایی باید مشخص شود که نرم‌افزار فعلی به چه مولفه‌های استانداردی نیاز دارد. همچنین در گام دوم بر اساس داده‌های تاریخی و پروژه‌های پیشین و گذشته باید مشخص شود که هر مولفه استاندارد به طور متوسط چه اندازه‌ای دارد. در گام سوم نیز باید مشخص شود که تعداد مولفه‌های استاندارد برنامه چقدر است. و در گام چهارم و انتهایی با استفاده از سه گام قبل، اندازه برنامه برآورد می‌شود.

برای‌مثال، در یک برنامه به زبان کوبول، نیاز به ۱۰ گزارش مختلف است و داده‌های تاریخی نشان می‌دهد که در زبان کوبول هر گزارش به ۳۰۰ خط کد (LOC) نیاز دارد، بنابراین برآورد می‌شود که اندازه برنامه آتی در بخش فقط گزارشات حدود ۳۰۰۰ خط کد (LOC) خواهد بود.

۳- تعیین اندازه به روش اندازه‌گیری تغییرات

این روش برای وقتی مورد استفاده قرار می‌گیرد که پروژه‌ی فعلی از مولفه‌های مشابه پیشین می‌تواند استفاده کند، اما جهت تطابق با پروژه فعلی باید تغییراتی در مولفه‌های مشابه پیشین اعمال شود. در این شرایط مدیر پروژه **فقط** می‌بایست اندازه تغییرات برای هریک از مولفه‌ها را برآورد کند. مثل افزودن کد، تغییر دادن کد و حذف کد. هدف از برآورد اندازه نرم‌افزار، برآورد نیروی مورد نیاز و هزینه‌های مربوط به پروژه و همچنین زمان لازم برای انجام کار است. بنابراین زمانی که از مولفه‌های مشابه پیشین استفاده مجدد می‌شود، نیاز به برآورد کل اندازه مولفه نیست، زیرا نیرو، هزینه و زمانی صرف ساخت بخش‌های بدون تغییر نمی‌شود. بلکه نیرو، هزینه و زمان فقط صرف اصلاح بخش‌های دستخوش تغییر می‌شود.

۴- تعیین اندازه به روش تعداد نقاط عملکرد

در این روش اندازه نرم‌افزار به کمک رابطه FP برآورد می‌گردد.

برآورد اندازه نرم‌افزار با استفاده از روش تجزیه

در برآورد اندازه نرم‌افزار با استفاده از روش تجزیه، ایده تقسیم و غلبه مورد استفاده قرار می‌گیرد. برآورد اندازه نرم‌افزار با استفاده از روش تجزیه به دو فرم زیر انجام می‌شود:

- برآورد اندازه نرم‌افزار با استفاده از روش تجزیه بر اساس LOC

در برآورد اندازه نرم‌افزار با استفاده از روش تجزیه بر اساس LOC، تجزیه از مولفه اصلی برنامه شروع می‌شود و سپس مولفه اصلی برنامه به مولفه‌های فرعی و فرعی‌تر تجزیه می‌شود. در ادامه مطابق روش اندازه‌گیری اندازه‌گرا (Size-Oriented) اندازه هریک از مولفه‌های فعلی مورد

نیاز، در پروژه‌های پیشین و گذشته اندازه‌گیری می‌شود و نتیجه این **اندازه‌گیری** به عنوان **برآورد** برای هریک از مولفه‌های پروژه فعلی در نظر گرفته می‌شود. و درنهایت کل برآوردهای مولفه‌ها جهت ایجاد برآورد کلی پروژه فعلی با هم جمع می‌شوند. در روش اندازه‌گرا، اندازه نرم‌افزار به شکل مستقیم (direct measure) و از طریق شمارش تعداد خطوط کد برنامه یعنی Line of Code یا LOC محاسبه می‌گردد. داده‌های تاریخی از سه منظر خوشبینانه (Optimistic)، محتمل‌ترین (most possible) و بدبینانه (Pessimistic) مورد برآورد قرار می‌گیرد و سپس میانگین برآورد این سه منظر بر اساس وزن هریک مطابق رابطه‌ی زیر محاسبه می‌شود:

$$AVG(S) = \frac{(۱\times S_{optimistic}) + (۴\times S_{most\ possible}) + (۱\times S_{Pessimistic})}{۶}$$

توجه: منظر محتمل‌ترین بیشترین اهمیت در محاسبه میانگین برآورد را دارد، بنابراین به شکل تجربی ضریب ۴ به آن اختصاص داده شده است.

مثال: با فرض مشخص بودن داده‌های تاریخی از سه منظر خوشبینانه(Optimistic) و برابر مقدار ۴۶۰۰ خط کد، محتملترین(most possible) و برابر مقدار ۶۹۰۰ خط کد و بدبینانه(Pessimistic) و برابر مقدار ۸۶۰۰ خط کد برای مولفه‌ی فرعی۱ مورد نظر، میانگین برآورد مولفه‌ی فرعی۱ چقدر است؟

پاسخ: مطابق رابطه‌ی زیر داریم:

$$AVG(S_۱) = \frac{(۱\times S_{optimistic}) + (۴\times S_{most\ possible}) + (۱\times S_{Pessimistic})}{۶}$$

پس از جایگذاری مقادیر داده‌های تاریخی بر اساس رابطه‌ی فوق داریم:

$$= \frac{(۱\times ۴۶۰۰) + (۴\times ۶۹۰۰) + (۱\times ۸۶۰۰)}{۶} = ۶۸۰۰$$

اگر فرض کنیم میانگین برآورد مولفه‌ی فرعی۱ بخشی از کل پروژه باشد با فرض مشخص بودن میانگین برآوردهای مولفه‌های فرعی دیگر، آنگاه برآورد کلی تعداد خطوط کد پروژه به صورت زیر خواهد بود:

مولفه‌های فرعی	میانگین تعداد خط برآورد شده (LOC)
مولفه فرعی۱	**6800**
مولفه فرعی۲	**2300**
مولفه فرعی۳	**5300**
مولفه فرعی۴	**3350**
مولفه فرعی۵	**4950**
مولفه فرعی۶	**2100**
مولفه فرعی۷	**8400**
برآورد کلی تعداد خطوط کد پروژه	**33200**

- برآورد اندازه نرم‌افزار با استفاده از روش تجزیه بر اساس FP

در برآورد اندازه نرم‌افزار با استفاده از روش تجزیه بر اساس FP، برنامه به دامنه اطلاعاتی (information domain) خود تجزیه می‌شود. در ادامه مطابق روش اندازه‌گیری عملکردگرا (Function-Oriented) تعداد هریک از دامنه‌های اطلاعاتی مورد نیاز، در پروژه‌های پیشین و گذشته شمارش می‌شود و نتیجه این **شمارش** به عنوان **برآورد** برای هریک از دامنه‌های اطلاعاتی فعلی در نظر گرفته می‌شود. و درنهایت کل برآوردهای شمارش شده مربوط به تعداد دامنه اطلاعاتی پس از ضرب شدن در ضریب وزنی هر یک از دامنه‌های اطلاعاتی جهت ایجاد برآورد تعداد کل پروژه فعلی با هم جمع می‌شوند. در روش عملکردگرا، اندازه نرم‌افزار به شکل غیرمستقیم (indirect measure) و از طریق رابطه FP محاسبه می‌گردد. FP سرواژه‌ی عبارت Function Point و به معنی نقطه عملکرد یا امتیاز کارکرد می‌باشد. در این روش چون نمی‌توان عملکرد نرم‌افزار را مستقیماً اندازه‌گیری کرد، باید آن را به طور غیرمستقیم و از طریق اندازه‌گیری های مستقیم دیگر به دست آورد. نقطه عملکرد با استفاده از یک فرمول تجربی مبتنی بر اندازه گیری‌های قابل شمارش (مستقیم) از مقادیر دامنه اطلاعاتی (information domain) و ضریب وزنی پیچیدگی (complexity) نرم‌افزار، به دست می‌آید.

نقطه عملکرد (FP) از رابطه‌ی زیر محاسبه می‌گردد:

$$FP = \text{تعداد کل} \times \left[۰/۶۵ + ۰/۰۱ \times \sum (F_i)\right]$$

مقادیر دامنه اطلاعاتی پروژه به شیوه زیر تعیین می‌شود:

۱- تعداد ورودی‌های برنامه

۲- تعداد خروجی‌های برنامه

۳- تعداد پرس و جوهای برنامه

۴- تعداد فایل‌های منطقی داخلی برنامه

۵- تعداد فایل‌های واسط خارجی برنامه

پس از تعیین مقادیر دامنه اطلاعاتی پروژه (information domain)، باید جدول زیر تکمیل گردد.

		ضرایب وزنی						
		پیچیده		متوسط		ساده		تعداد	دامنه اطلاعاتی
	=	۶		۴	×	۳			تعداد ورودی‌های برنامه
	=	۷		۵	×	۴			تعداد خروجی‌های برنامه
	=	۶		۴	×	۳			تعداد پرس و جوهای برنامه
	=	۱۵		۱۰	×	۷			تعداد فایل‌های منطقی داخلی برنامه
	=	۱۰		۷	×	۵			تعداد فایل‌های واسط خارجی برنامه
									تعداد کل

هدف از تکمیل جدول فوق، تعیین مقدار تعداد کل (count total) است، زیرا این مقدار پارامتری از رابطه نقطه عملکرد (امتیاز کارکرد) است.

تنها قدم باقی‌مانده جهت محاسبه‌ی نقطه عملکرد (FP) تعیین مقدار $\sum (F_i)$ است. در واقع هر F_i عددی بین ۰ تا ۵ است که در پاسخ به چهارده پرسش متفاوت که پیش‌تر به آن پرداختیم، تعیین می‌شود.

مقادیر F_i:

	بی‌تأثیر	ضعیف	متوسط	خوب	عالی	بسیار عالی
مقادیر F_i	۰	۱	۲	۳	۴	۵

داده‌های تاریخی از سه منظر خوشبینانه (Optimistic)، محتمل‌ترین (most possible) و بدبینانه (Pessimistic) مورد برآورد قرار می‌گیرد و سپس میانگین برآورد این سه منظر بر اساس وزن هریک مطابق رابطه‌ی زیر محاسبه می‌شود:

$$AVG(S) = \frac{(۱\times S_{optimistic}) + (۴\times S_{most\ possible}) + (۱\times S_{Pessimistic})}{۶}$$

توجه: منظر محتمل‌ترین بیشترین اهمیت در محاسبه میانگین برآورد را دارد، بنابراین به شکل تجربی ضریب ۴ به آن اختصاص داده شده است.

مثال: با فرض مشخص بودن داده‌های تاریخی از سه منظر خوشبینانه (Optimistic) و برابر مقدار ۲۰، محتملترین (most possible) و برابر مقدار ۲۲ و بدبینانه (Pessimistic) و برابر مقدار ۳۰ برای تعداد دامنه اطلاعاتی ورودی‌های برنامه، میانگین برآورد تعداد دامنه اطلاعاتی ورودی‌های برنامه چقدر است؟

پاسخ: مطابق رابطه‌ی زیر داریم:

$$AVG(S_i) = \frac{(۱\times S_{optimistic}) + (۴\times S_{most\ possible}) + (۱\times S_{Pessimistic})}{۶}$$

پس از جایگذاری مقادیر داده‌های تاریخی بر اساس رابطه‌ی فوق داریم:

$$= \frac{(۱\times ۲۰) + (۴\times ۲۲) + (۱\times ۳۰)}{۶} = ۲۳$$

اگر فرض کنیم میانگین برآورد تعداد دامنه اطلاعاتی ورودی‌های برنامه بخشی از کل دامنه‌های اطلاعاتی پروژه باشد با فرض مشخص بودن میانگین برآوردهای تعداد دامنه‌های اطلاعاتی دیگر، آنگاه برآورد تعداد کل پروژه به صورت زیر خواهد بود:

دامنه اطلاعاتی	تعداد خوشبینانه	تعداد محتملترین	تعداد بدبینانه	میانگین برآورد	ضریب وزنی متوسط	ضریب وزنی × میانگین برآورد
ورودی‌های برنامه	20	22	30	23	4	92
خروجی‌های برنامه	12	14	22	15	5	75
پرس و جوهای برنامه	16	22	28	22	4	88
فایل‌های منطقی داخلی برنامه	4	3	2	3	10	30
فایل‌های واسط خارجی برنامه	5	4	3	4	7	28
تعداد کل						313

همچنین فرض کنید مقدار $\sum(F_i)$ برابر 46 باشد، یعنی محصولی با پیچیدگی معتدل. بنابراین مقدار **برآورد نقطه عملکرد** ($FP_{estimated}$) برای پروژه فوق به صورت زیر محاسبه می‌گردد:

$$FP_{estimated} = \text{تعداد کل} \times \left[۰/۶۵ + ۰/۰۱ \times \sum(F_i)\right]$$

$$FP_{estimated} = ۳۱۳ \times \left[۰/۶۵ + ۰/۰۱ \times ۴۶\right] = ۳۴۷$$

برآورد نیرو، هزینه و زمان

پس از برآورد اندازه برنامه، نوبت به محاسبه نیروی مورد نیاز، هزینه‌های مربوط به توسعه پروژه و همچنین زمان لازم برای انجام کار می‌رسد. سه روش برای برآورد نیرو، هزینه و زمان وجود دارد که در ادامه مورد بررسی قرار می‌گیرند.

برآورد نیرو و هزینه بر اساس فرآیند تولید نرم‌افزار

پس از برآورد اندازه نرم‌افزار با استفاده از روش تجزیه بر اساس LOC، نوبت به برآورد نیروی مورد نیاز و هزینه‌های مربوط به توسعه پروژه می‌رسد. همانطور که گفتیم در برآورد اندازه نرم‌افزار با استفاده از روش تجزیه بر اساس LOC، تجزیه از مولفه اصلی برنامه شروع می‌شود و سپس مولفه اصلی برنامه به مولفه‌های فرعی و فرعی‌تر تجزیه می‌شود. در ادامه مطابق روش اندازه‌گیری اندازه‌گرا(Size-Oriented) اندازه هریک از مولفه‌های فعلی مورد نیاز، در پروژه‌های پیشین و گذشته اندازه‌گیری می‌شود و نتیجه این **اندازه‌گیری** به عنوان **برآورد** برای هریک از مولفه‌های پروژه فعلی در نظر گرفته می‌شود. و درنهایت کل برآوردهای مولفه‌ها جهت ایجاد برآورد کلی پروژه فعلی با هم جمع می‌شوند، مطابق جدول زیر:

مولفه‌های فرعی	میانگین تعداد خط برآورد شده(LOC)
مولفه فرعی۱	**6800**
مولفه فرعی۲	**2300**
مولفه فرعی۳	**5300**
مولفه فرعی۴	**3350**
مولفه فرعی۵	**4950**
مولفه فرعی۶	**2100**
مولفه فرعی۷	**8400**
برآورد کلی تعداد خطوط کد پروژه	**33200**

اما در برآورد نیروی مورد نیاز و هزینه‌های مربوط به توسعه پروژه، بر اساس **فرآیند تولید نرم‌افزار**، ملاک برآورد، فقط برآورد کلی تعداد خطوط کد پروژه و به تبع فقط برآورد فعالیت پیاده‌سازی نیست. بلکه ملاک برآورد، برآورد کل فعالیت‌های چارچوبی (ارتباط، برنامه‌ریزی، مدل‌سازی و ساخت) است. هنگامی‌که مولفه‌های برنامه و فعالیت‌های چارچوبی مشخص شد، نیروی مورد نیاز برای انجام هر یک از فعالیت‌های چارچوبی بر حسب نفر – ماه برای هر یک از مولفه‌های برنامه بر اساس داده‌های تاریخی و پروژه‌های پیشین و گذشته برآورد می‌شود. جدول زیر گویای مطلب است:

مولفه‌های فرعی	ارتباطات	برنامه‌ریزی	تحلیل ریسک	مدل‌سازی		ساخت		جمع کل
				تحلیل	طراحی	پیاده‌سازی	تست	
مولفه فرعی۱	0.01	0.01	0.01	0.5	2.5	0.4	5.00	8.43
مولفه فرعی۲	0.02	0.02	0.02	0.75	4.00	0.6	2.00	7.41
مولفه فرعی۳	0.02	0.02	0.02	0.5	4.00	1.00	3.00	8.56
مولفه فرعی۴	0.05	0.05	0.05	0.5	3.00	1.00	1.50	6.15
مولفه فرعی۵	0.05	0.05	0.05	0.5	3.00	0.75	1.50	5.9
مولفه فرعی۶	0.05	0.05	0.05	0.25	2.00	0.5	1.50	4.4
مولفه فرعی۷	0.05	0.05	0.05	0.5	2.00	0.5	2.00	5.15
جمع کل	0.25	0.25	0.25	3.5	20.5	4.75	16.5	46.00

جمع کل حاصل، برآوردی از نیروی لازم برای انجام کل پروژه است. که مقدار آن برابر ۴۶ نفر در ماه (PM: person per month) شده است. بر اساس میانگین حقوق ماهیانه به مبلغ ۱۰ میلیون تومان برای هر نفر، برآورد هزینه کل پروژه به صورت زیر محاسبه می‌شود:

برآورد هزینه کل پروژه = ۴۶×۱۰۰۰۰۰۰۰ = ۴۶۰۰۰۰۰۰۰

در صورت تمایل می‌توان میزان حقوق را برای هر یک از فعالیت‌های چارچوبی جداگانه تعیین و محاسبه نمود.

برآورد نیرو و هزینه با استفاده از رابطه‌ی بهره‌وری

در برآورد نیرو و هزینه با استفاده از رابطه‌ی بهره‌وری، ایده تقسیم و غلبه مورد استفاده قرار می‌گیرد. برآورد نیرو و هزینه با استفاده از رابطه‌ی بهره‌وری به دو فرم زیر انجام می‌شود:

- برآورد نیرو و هزینه با استفاده از روش تجزیه بر اساس LOC

پس از برآورد اندازه نرم‌افزار با استفاده از روش تجزیه بر اساس LOC، نوبت به برآورد نیروی مورد نیاز و هزینه‌های مربوط به توسعه پروژه می‌رسد. همانطور که گفتیم در برآورد اندازه نرم‌افزار با استفاده از روش تجزیه بر اساس LOC، تجزیه از مولفه اصلی برنامه شروع می‌شود و سپس مولفه اصلی برنامه به مولفه‌های فرعی و فرعی‌تر تجزیه می‌شود. در ادامه مطابق روش اندازه‌گیری اندازه‌گرا (Size-Oriented) اندازه هریک از مولفه‌های فعلی مورد نیاز، در پروژه‌های پیشین و گذشته اندازه‌گیری می‌شود و نتیجه این **اندازه‌گیری** به عنوان **برآورد** برای هریک از مولفه‌های پروژه فعلی در نظر گرفته می‌شود. و درنهایت کل برآوردهای مولفه‌ها جهت ایجاد برآورد کلی پروژه فعلی با هم جمع می‌شوند، مطابق جدول زیر:

مولفه‌های فرعی	میانگین تعداد خط برآورد شده(LOC)
مولفه فرعی۱	**6800**
مولفه فرعی۲	**2300**
مولفه فرعی۳	**5300**
مولفه فرعی۴	**3350**
مولفه فرعی۵	**4950**
مولفه فرعی۶	**2100**
مولفه فرعی۷	**8400**
برآورد کلی تعداد خطوط کد پروژه	**33200**

در برآورد نیروی مورد نیاز و هزینه‌های مربوط به توسعه‌ی پروژه، با استفاده از رابطه‌ی بهره‌وری، ملاک برآورد، برآورد کلی تعداد خطوط کد پروژه (LOC) و به تبع برآورد فعالیت پیاده‌سازی است. هنگامی‌که مولفه‌های برنامه و اندازه‌ی هر مولفه مشخص شد، مقدار **فاکتور**

بهره‌وری (productivity) بر اساس داده‌های تاریخی و پروژه‌های پیشین و گذشته برآورد می‌شود. مقدار فاکتور بهره‌وری برابر با نسبت اندازه‌ی پروژه (LOC) به نیروی لازم (PM: person per month) برای توسعه‌ی پروژه است، به صورت زیر:

$$\text{productivity} = \frac{\text{LOC}}{\text{PM}}$$

برای برآورد مقدار **فاکتور بهره‌وری**، برای پروژه‌ی **فعلی**، ابتدا باید پروژه‌های گذشته و پیشینی که از نظر زمینه‌ی کاری، میزان پیچیدگی و میزان دقت مورد نیاز، شبیه به پروژه‌ی فعلی است مشخص شوند. سپس مقدار فاکتور بهره‌وری پروژه‌های گذشته و پیشین از روی اندازه و نیروی مورد نیاز آنها محاسبه می‌شود. در نهایت میانگین مقادیر بهره‌وری پروژه‌های گذشته و پیشین محاسبه می‌گردد و به عنوان برآورد بهره‌وری پروژه‌ی **فعلی** معرفی می‌گردد.

مثال: برآورد کلی تعداد خطوط کد پروژه فعلی برابر مقدار LOC ۳۳۲۰۰ **و همچنین میانگین مقادیر فاکتور بهره‌وری پروژه‌های گذشته و پیشین مشابه با پروژه فعلی برابر مقدار** $\frac{\text{LOC}}{\text{PM}}$ ۷۹۰ **است، برآورد نیروی لازم (PM: person per month) برای توسعه‌ی پروژه فعلی چقدر است؟**

پاسخ: مطابق رابطه‌ی بهره‌وری داریم:

$$\text{productivity} = \frac{\text{LOC}}{\text{PM}}$$

پس از جایگذاری مقادیر داده‌های تاریخی بر اساس رابطه‌ی فوق داریم:

$$\text{PM} = \frac{\text{LOC}}{\text{productivity}} = \frac{۳۳۲۰۰}{۷۹۰} = ۴۲ \text{ (person per month)}$$

بنابراین نیروی لازم برای توسعه‌ی پروژه برابر با مقدار ۴۲ نفر در ماه است. با فرض میانگین حقوق ماهیانه به مبلغ ۱۰ میلیون تومان برای هر نفر، برآورد هزینه‌ی کل پروژه به صورت زیر محاسبه می‌شود:

$$\text{Total Cost} = \text{PM} \times \text{Salary} = ۴۲ \times ۱۰۰۰۰۰۰۰ = ۴۲۰۰۰۰۰۰۰$$

و در نهایت برآورد هزینه‌ی هر خط کد از پروژه به صورت زیر محاسبه می‌شود:

$$\text{Partial Cost} = \frac{\text{Total Cost}}{\text{LOC}} = \frac{۴۲۰۰۰۰۰۰۰}{۳۳۲۰۰} = ۱۲۶۵۰$$

- برآورد نیرو و هزینه با استفاده از روش تجزیه بر اساس FP

پس از برآورد اندازه نرم‌افزار با استفاده از روش تجزیه بر اساس FP، نوبت به برآورد نیروی مورد نیاز و هزینه‌های مربوط به توسعه پروژه می‌رسد. همانطور که گفتیم در برآورد اندازه نرم‌افزار با استفاده از روش تجزیه بر اساس FP، برنامه به دامنه اطلاعاتی (information domain) خود تجزیه می‌شود. در ادامه مطابق روش اندازه‌گیری عملکردگرا (Function-Oriented) تعداد هریک

از دامنه‌های اطلاعاتی مورد نیاز، در پروژه‌های پیشین و گذشته شمارش می‌شود و نتیجه این **شمارش** به عنوان **برآورد** برای هریک از دامنه‌های اطلاعاتی فعلی در نظر گرفته می‌شود. و درنهایت کل برآوردهای شمارش شده مربوط به تعداد دامنه اطلاعاتی پس از ضرب شدن در ضریب وزنی هر یک از دامنه‌های اطلاعاتی جهت ایجاد برآورد تعداد کل پروژه فعلی با هم جمع می‌شوند.

نقطه عملکرد (FP) از رابطه‌ی زیر محاسبه می‌گردد:

$$FP = \text{تعداد کل} \times \left[۰/۶۵ + ۰/۰۱ \times \sum(F_i)\right]$$

با فرض مشخص بودن میانگین برآوردهای تعداد دامنه‌های اطلاعاتی پروژه فعلی، آنگاه برآورد تعداد کل پروژه به صورت زیر خواهد بود:

دامنه اطلاعاتی	تعداد خوشبینانه	تعداد محتملترین	تعداد بدبینانه	میانگین برآورد	ضریب وزنی متوسط	ضریب وزنی × میانگین برآورد
ورودی‌های برنامه	20	22	30	23	4	92
خروجی‌های برنامه	12	14	22	15	5	75
پرس و جوهای برنامه	16	22	28	22	4	88
فایل‌های منطقی داخلی برنامه	4	3	2	3	10	30
فایل‌های واسط خارجی برنامه	5	4	3	4	7	28
تعداد کل						313

همچنین فرض کنید مقدار $\sum(F_i)$ برابر 46 باشد، یعنی محصولی با پیچیدگی معتدل.

بنابراین مقدار **برآورد نقطه عملکرد** ($FP_{estimated}$) برای پروژه فوق به صورت زیر محاسبه می‌گردد:

$$FP_{estimated} = \text{تعداد کل} \times \left[۰/۶۵ + ۰/۰۱ \times \sum(F_i)\right]$$

$$FP_{estimated} = ۳۱۳ \times \left[۰/۶۵ + ۰/۰۱ \times ۴۶\right] = ۳۴۷$$

در برآورد نیروی مورد نیاز و هزینه‌های مربوط به توسعه‌ی پروژه، با استفاده از رابطه‌ی بهره‌وری، ملاک برآورد، برآورد کلی مقدار FP و به تبع برآورد فعالیت پیاده‌سازی است. هنگامی‌که مقدار FP مشخص شد، مقدار **فاکتور بهره‌وری** (productivity) بر اساس داده‌های تاریخی و پروژه‌های پیشین و گذشته برآورد می‌شود. مقدار فاکتور بهره‌وری برابر با نسبت اندازه‌ی پروژه (FP) به نیروی لازم (PM: person per month) برای توسعه‌ی پروژه است، به صورت زیر:

$$\text{productivity} = \frac{\text{FP}}{\text{PM}}$$

برای برآورد مقدار **فاکتور بهره‌وری**، برای پروژه‌ی **فعلی**، ابتدا باید پروژه‌های گذشته و پیشینی که از نظر زمینه‌ی کاری، میزان پیچیدگی و میزان دقت مورد نیاز، شبیه به پروژه‌ی فعلی است مشخص شوند. سپس مقدار فاکتور بهره‌وری پروژه‌های گذشته و پیشین از روی اندازه و نیروی مورد نیاز آنها محاسبه می‌شود. در نهایت میانگین مقادیر بهره‌وری پروژه‌های گذشته و پیشین محاسبه می‌گردد و به عنوان برآورد بهره‌وری پروژه‌ی **فعلی** معرفی می‌گردد.

مثال: برآورد کلی مقدار FP پروژه فعلی برابر مقدار FP ۳۴۷ و همچنین میانگین مقادیر فاکتور بهره‌وری پروژه‌های گذشته و پیشین مشابه با پروژه فعلی برابر مقدار $\frac{\text{FP}}{\text{PM}}$ ۶ است، برآورد نیروی لازم (PM: person per month) برای توسعه‌ی پروژه فعلی چقدر است؟

پاسخ: مطابق رابطه‌ی بهره‌وری داریم:

$$\text{productivity} = \frac{\text{FP}}{\text{PM}}$$

پس از جایگذاری مقادیر داده‌های تاریخی بر اساس رابطه‌ی فوق داریم:

$$\text{PM} = \frac{\text{FP}}{\text{productivity}} = \frac{۳۴۷}{۶} = ۵۸ \text{ (person per month)}$$

بنابراین نیروی لازم برای توسعه‌ی پروژه برابر با مقدار ۵۸ نفر در ماه است. با فرض میانگین حقوق ماهیانه به مبلغ ۱۰ میلیون تومان برای هر نفر، برآورد هزینه‌ی کل پروژه به صورت زیر محاسبه می‌شود:

$$\text{Total Cost} = \text{PM} \times \text{Salary} = ۵۸ \times ۱۰۰۰۰۰۰۰ = ۵۸۰۰۰۰۰۰۰$$

و در نهایت برآورد هزینه‌ی هر FP از پروژه به صورت زیر محاسبه می‌شود:

$$\text{Partial Cost} = \frac{\text{Total Cost}}{\text{FP}} = \frac{۵۸۰۰۰۰۰۰۰}{۳۴۷} = ۱۶۷۱۴۶۹$$

برآورد نیرو، هزینه و زمان بر اساس مدل‌های تجربی

برآورد نیرو، هزینه و زمان بر اساس مدل‌های تجربی با استفاده از تحلیل رگرسیون (Regression Analysis) روی داده‌های جمع‌آوری شده از پروژه‌های نرم‌افزاری پیشین و گذشته به‌دست می‌آید. رگرسیون یعنی بازگشت. یعنی پیش‌بینی و بیان تغییرات یک متغیر بر اساس اطلاعات متغیر دیگر. واژه‌ی رگرسیون (Regression) از لحاظ لغوی در فرهنگ لغت به معنی پسروی، برگشت و بازگشت است. اما از دید آمار و ریاضیات به مفهوم بازگشت به یک مقدار متوسط یا میانگین به کار می‌رود.

مثال: رابطه ی بین قد و وزن انسان‌ها را در نظر بگیرید. همه می‌دانیم که این رابطه یک رابطه

مستقیم ریاضی و صد درصدی نیست که لزوما هر که قد بلندتری داشته باشد وزن بیشتری داشته باشد، اما می‌توان گفت که با احتمال قابل قبولی افراد با قد بلندتر، وزن بیشتری نیز دارند. در اینجا پیش‌بینی وزن از روی قد و بیان ارتباط بین این متغیر با روش آماری رگرسیون خطی صورت می‌پذیرد که این رابطه را به صورت کمی به ما نشان می‌دهد.

رگرسیون را با معادله ی رگرسیون بیان می‌کنند. در مثال فوق معادله ی رگرسیون خطی می‌تواند به صورت زیر باشد:

متغیر وزن = متغیر قد × a + b

در مدل‌های برآورد نرم‌افزارهای کامپیوتری برای پیش‌بینی و برآورد نیروی لازم برای توسعه‌ی پروژه، به عنوان تابعی از LOC یا FP، از روابطی استفاده می‌شود که به طور تجربی به‌دست می‌آیند. در همه‌ی مدل‌های تجربی برآورد نیروی لازم (E) برای توسعه‌ی پروژه تابعی از اندازه‌ی برنامه (بر حسب LOC یا FP) است. داده‌های تجربی که اکثر مدل‌های برآورد را پشتیبانی می‌کنند از تعداد محدودی از پروژه‌های نمونه به‌دست می‌آیند. همچنین، هیچ مدل برآوردی نیست که برای تمامی انواع نرم‌افزارها و تمامی محیط‌های توسعه‌ی نرم‌افزار، مناسب باشد. بنابراین باید از نتایج به دست آمده از مدل‌های تجربی با دید باز استفاده کرد. نتیجه اینکه روابط تجربی وابسته به نوع پروژه هستند، پس در انجام برآورد، انتخاب این روابط باید آگاهانه و متناسب با نوع پروژه باشد.

ساختار کلی مدل‌های برآورد تجربی به صورت زیر است:

$$E = A + B \times (e_v)^C$$

که A، B و C ثابت‌هایی هستند که به طور تجربی به‌دست می‌آیند، E نیروی لازم برای توسعه پروژه بر حسب نفر در ماه (PM: person per month) و e_v متغییر برآورد بر حسب LOC یا FP است. علاوه بر پارامترهای فوق، اغلب مدل‌های برآورد، شکلی از مولفه‌های تنظیم پروژه را دارند که به کمک آن می‌توان E را به وسیله‌ی سایر خصوصیات مانند پیچیدگی مساله، تجربه‌ی کارمندان و محیط توسعه‌ی پروژه محاسبه نمود. مانند مدل والستون و فلیکس که مقدار E بر حسب LOC محاسبه می‌گردد، به صورت زیر:

$$E = ۵/۲ \times (KLOC)^{۰/۹۱}$$

و یا مانند مدل آلبرشت و گافنی که مقدار E بر حسب FP محاسبه می‌گردد، به صورت زیر:

$$E = -۹۱/۴ + ۰/۳۵۵ \times FP$$

بررسی اجمالی روابط مختلف مدل‌های تجربی نشان می‌دهد که هر کدام از آنها به ازای مقادیر یکسانی از LOC یا FP، نتایج متفاوتی را ایجاد می‌کنند. دلیل آن هم روشن است، زیرا مدل‌های برآورد باید برای نیازهای محلی، نرمال شوند.

در ادامه به بررسی دو مدل برآورد نیرو، هزینه و زمان بر اساس مدل‌های تجربی می‌پردازیم:

- مدل کوکومو۲ (COCOMO II Model)

واژه‌ی COCOMO سرواژه عبارت Construcive Cost Model و به معنی مدل هزینه‌ی

ساخت است. مدل اولیه COCOMO که توسط آقای بری بوهم[1] ارائه شده و در کتاب اقتصاد مهندسی نرم‌افزار منتشر شده است، کاربرد بسیار زیادی برای برآورد نرم‌افزار در صنعت نرم‌افزار پیداکرده است. بعدها این مدل به مرور زمان تکامل یافت و یک مدل جامع‌تری برای برآورد نرم‌افزار به‌نام COCOMO II ارائه شد. دقت کنید که فعالیت برآورد نرم‌افزار فقط در ابتدای توسعه‌ی نرم‌افزار قرار ندارد، بلکه با انجام برآوردها در طول توسعه‌ی نرم‌افزار نیز فعالیت برآورد نرم‌افزار وجود دارد که دقت و کیفیت آن افزایش پیدا می‌کند. بنابراین مدل COCOMO II نیز فعالیت برآورد را در سه محدوده‌ی زمانی متفاوت سازماندهی می‌کند:

۱- مدل ترکیبی کاربردی

این مدل در طول مراحل اولیه‌ی مهندسی نرم‌افزار به کار می‌رود. خصوصاً زمانی که نمونه سازی واسط‌های کاربر، ارتباط نرم‌افزار با محیط پیرامون و ارزیابی تکنولوژی‌های قابل استفاده مورد بررسی قرار می‌گیرند.

۲- مدل مراحل اولیه طراحی

این مدل هنگامی به کار می‌رود که نیازمندی‌ها به ثبات و پایداری رسیده باشند و طراحی اولیه معماری نرم‌افزار ایجاد شده باشد.

۳- مدل مرحله بعد از معماری

در طول ساخت نرم‌افزار به کار می‌رود.

همانند تمام مدل‌های برآورد، این مدل نیز، نیازمند اطلاعات مربوط به اندازه می‌باشد. سه روش برای اندازه‌گیری وجود دارد، که عبارتند از:

۱- LOC یا Line of code یا تعداد خطوط کد

۲- FP یا Function Point یا نقاط عملکرد

۳ – OP یا Object Ponit یا نقاط شیء

این مدل بیشتر از روش OP یا Object Ponit استفاده می‌کند در این مدل، اندازه کمیت نرم‌افزار به شکل غیرمستقیم (indirect measure) و از طریق رابطه‌ی OP محاسبه می‌گردد. OP سرواژه‌ی عبارت Object Point و به معنی نقطه شیء یا امتیاز شیء می‌باشد. در این روش چون نمی‌توان عملکرد نرم‌افزار را مستقیماً اندازه‌گیری کرد، باید آن را به طور غیرمستقیم و از طریق اندازه‌گیری‌های مستقیم دیگر به دست آورد. نقطه شیء با استفاده از یک فرمول تجربی مبتنی بر اندازه‌گیری‌های قابل شمارش (مستقیم) از مقادیر دامنه اشیاء و ضریب وزنی پیچیدگی (complexity) نرم‌افزار، به دست می‌آید.

نقطه شیء (OP) از رابطه‌ی زیر محاسبه می‌گردد:

[1] Boehm

OP = تعداد کل

مقادیر دامنه اشیاء پروژه به شیوه زیر تعیین می‌شود:

۱- صفحات نمایش (در واسط کاربری)

۲- گزارش‌ها

۳- مولفه‌های مورد نیاز در ساخت نرم‌افزار فعلی

پس از تعیین مقادیر دامنه اشیاء پروژه، باید جدول زیر تکمیل گردد.

دامنه اشیاء	تعداد	ضرایب وزنی			ضریب وزنی ×تعداد
		ساده	متوسط	پیچیده	
صفحات نمایش		1	2	3	
گزارش‌ها		2	5	8	
مولفه‌ها				10	
تعداد کل					

هدف از تکمیل جدول فوق، تعیین مقدار تعداد کل (count total) است، زیرا این مقدار برابر OP است.

در جدول فوق مقدار ضریب وزنی مشخص می‌کند که هریک از عناصر دامنه اشیاء به چه ضریب وزنی، پیچیدگی (complexity) را به برنامه تحمیل می‌کنند.

سازمان‌هایی که روش نقطه شیء را به کار می‌برند، ملاک‌هایی برای تعیین میزان ضریب وزنی عناصر دامنه اشیاء به شکل ساده، متوسط و پیچیده را مشخص می‌کنند. تعداد کل، حاصل جمع تعداد همه عناصر دامنه اشیاء با احتساب ضرایب وزنی آن‌ها است که از جدول مذکور به دست می‌آید.

مثال: مقادیر ضریب وزنی مربوط به عناصر دامنه اشیاء را در یک پروژه‌ی فرضی به صورت زیر در نظر بگیرید:

دامنه اشیاء	تعداد	ضریب وزنی	ضریب وزنی ×تعداد
صفحات نمایش	3	3	9
گزارش‌ها	2	8	16
مولفه‌ها	2	10	20
تعداد کل			45

مقدار نقطه شیء (OP) برای پروژه‌ی فرضی فوق به صورت زیر محاسبه می‌گردد:

OP = تعداد کل

OP = 45

توجه: هنگامی‌که تولید مبتنی بر مولفه یا قابلیت استفاده مجدد از نرم‌افزارهای پیشین، مد نظر باشد، درصد استفاده‌ی مجدد (reuse%) در پروژه‌ی فعلی برآورد می‌شود و از درصد کل مولفه‌ها کسر می‌شود، زیرا برای اشیائی که قابلیت استفاده مجدد دارند و قبلا تولید شده‌اند، هزینه و نیرویی صرف نمی‌شود.

توجه: مؤلفه یک قطعه‌ی آماده، کاربردی و پیاده‌سازی شده است که دارای واسط لازم جهت اتصال با سایر قطعات و استقرار در بخشی از یک سیستم عملیاتی می‌باشد.

توجه: زمان و هزینه‌ی فرآیند تولید نرم‌افزار براساس مدل توسعه‌ی مبتنی بر مؤلفه به دلیل استفاده از قطعات آماده (قابلیت استفاده مجدد) کاهش چشمگیری دارد. در واقع یک قطعه با قابلیت استفاده‌ی مجدد یک بار ساخته می‌شود و بارها و بارها استفاده می‌شود و این یعنی سودآوری!

با توجه به مطالب فوق، می‌توان NOP یا New Object Ponit یا نقاط شیء جدید را از رابطه‌ی زیر محاسبه نمود:

$$NOP = OP \times \left(\frac{۱۰۰ - \%reuse}{۱۰۰}\right)$$

برای به‌دست آوردن برآورد نیروی مورد نیاز و هزینه‌های مربوط به توسعه‌ی پروژه بر اساس مقدار NOP محاسبه شده، باید یک آهنگ بهره‌وری (Productivity) به دست آورد.

در برآورد نیروی مورد نیاز و هزینه‌های مربوط به توسعه‌ی پروژه، با استفاده از رابطه‌ی بهره‌وری، ملاک برآورد، برآورد کلی مقدار NOP و به تبع برآورد فعالیت پیاده‌سازی است. هنگامی‌که مقدار NOP مشخص شد، مقدار **فاکتور بهره‌وری** (productivity) بر اساس داده‌های تاریخی و پروژه‌های پیشین و گذشته برآورد می‌شود. مقدار فاکتور بهره‌وری برابر با نسبت اندازه‌ی پروژه (NOP) به نیروی لازم (PM: person per month) برای توسعه‌ی پروژه است، به صورت زیر:

$$productivity = \frac{NOP}{PM}$$

برای برآورد مقدار **فاکتور بهره‌وری**، برای پروژه‌ی **فعلی**، ابتدا باید پروژه‌های گذشته و پیشینی که از نظر زمینه‌ی کاری، میزان پیچیدگی و میزان دقت مورد نیاز، شبیه به پروژه‌ی فعلی است مشخص شوند. سپس مقدار فاکتور بهره‌وری پروژه‌های گذشته و پیشین از روی اندازه و نیروی مورد نیاز آنها محاسبه می‌شود. در نهایت میانگین مقادیر بهره‌وری پروژه‌های گذشته و پیشین محاسبه می‌گردد و به عنوان برآورد بهره‌وری پروژه‌ی **فعلی** معرفی می‌گردد.

مثال: برآورد کلی مقدار OP پروژه فعلی برابر مقدار OP ۴۵، و درصد قابلیت استفاده مجدد یا reuse% برابر ۲۰ درصد و همچنین میانگین مقادیر فاکتور بهره‌وری پروژه‌های گذشته و پیشین مشابه با پروژه فعلی برابر مقدار $\frac{NOP}{PM}$ ۴ است، برآورد نیروی لازم (PM: person per month) برای توسعه‌ی پروژه فعلی چقدر است؟

پاسخ: مقدار NOP یا New Object Ponit یا نقاط شیء جدید مطابق رابطه‌ی زیر محاسبه می‌شود:

$$NOP = OP \times \left(\frac{۱۰۰ - \%reuse}{۱۰۰} \right)$$

پس از جایگذاری مقدار درصد قابلیت استفاده مجدد یا **%reuse** بر اساس رابطه‌ی فوق داریم:

$$NOP = OP \times \left(\frac{۱۰۰ - \%reuse}{۱۰۰} \right) = ۴۵ \times \left(\frac{۱۰۰ - ۲۰}{۱۰۰} \right) = ۴۵ \times \left(\frac{۸۰}{۱۰۰} \right) = ۳۶$$

همچنین مطابق رابطه‌ی بهره‌وری داریم:

$$productivity = \frac{NOP}{PM}$$

پس از جایگذاری مقادیر داده‌های تاریخی بر اساس رابطه‌ی فوق داریم:

$$PM = \frac{NOP}{productivity} = \frac{۳۶}{۴} = ۹ \text{ (person per month)}$$

بنابراین نیروی لازم برای توسعه‌ی پروژه برابر با مقدار ۹ نفر در ماه است. با فرض میانگین حقوق ماهیانه به مبلغ ۱۰ میلیون تومان برای هر نفر، برآورد هزینه‌ی کل پروژه به صورت زیر محاسبه می‌شود:

$$Total\ Cost = PM \times Salary = ۹ \times ۱۰۰۰۰۰۰۰ = ۹۰۰۰۰۰۰۰$$

و در نهایت برآورد هزینه‌ی هر NOP از پروژه به صورت زیر محاسبه می‌شود:

$$Partial\ Cost = \frac{Total\ Cost}{NOP} = \frac{۹۰۰۰۰۰۰۰}{۳۶} = ۲۵۰۰۰۰۰$$

- معادله‌ی نرم‌افزار

معادله‌ی نرم‌افزار که با نام معادله پوتنام[1] نیز شناخته می‌شود، یک معادله چندمتغیره و پویاست. که در آن فرض می‌شود، کار به طور متوازن در طول مراحل توسعه‌ی نرم‌افزار توزیع می‌شود. معادله‌ی نرم‌افزار، به عنوان یک مدل برآورد از داده‌های هزاران پروژه‌ی نرم‌افزاری به دست آمده است، این معادله، رابطه‌ی بین زمان، هزینه و نیروی لازم برای توسعه‌ی پروژه را نشان می‌دهد. معادله‌ی نرم‌افزار به صورت زیر است:

$$E = PM = \frac{LOC \times B^{۰/۳۳۳}}{P^{۳}} \times \left(\frac{۱}{t^{۴}} \right)$$

E : نیروی لازم جهت توسعه‌ی پروژه برحسب نفر در ماه

[1] Putnam

t : مدت زمان توسعه‌ی پروژه بر حسب ماه

B : ضریب مهارت‌های ویژه‌ی توسعه‌دهندگان، این ضریب با رشد اندازه‌ی برنامه افزایش می‌یابد، زیرا با رشد اندازه‌ی برنامه، نیاز به مهارت‌های بیشتری جهت حفظ انسجام بالا، اتصال پایین، مستندسازی و تضمین کیفیت وجود دارد. مقدار B عددی بین صفر و یک است. برای برنامه‌های کوچک در بازه 5 KLOC تا 15 KLOC مقدار B برابر 0.16 و برای برنامه‌های بزرگ در بازه 70 KLOC به بالا مقدار B برابر 0.39 است.

P : فاکتور بهره‌وری که نشان‌دهنده موارد زیر است:

- تکامل کلی فرآیند و روش‌های مدیریتی
- میزان مبادرت به عادات پسندیده در مهندسی نرم‌افزار
- سطح زبان برنامه‌نویسی مورد استفاده
- وضعیت محیط نرم‌افزار و ابزارهای جانبی
- مهارت‌ها و تجربه تیم نرم‌افزاری
- سادگی و پیچیدگی برنامه‌کاربردی

مطابق معادله‌ی نرم‌افزار هرچه مقدار p بیشتر باشد، توسعه‌ی پروژه به نیروی کمتری نیاز دارد.

زمان‌بندی پروژه

پس از تعیین مدل فرآیند تولید نرم‌افزار، برآوردهای مربوط به میزان کار، نیروی لازم برای انجام کار، زمان لازم برای انجام کار و هزینه لازم برای انجام کار نوبت به زمان‌بندی انجام پروژه می‌رسد. یعنی باید شبکه‌ای از وظایف مهندسی نرم‌افزار ایجاد شود تا کارها به موقع و سر وقت انجام شود. هنگامی که شبکه ایجاد شد، باید به هر وظیفه مسئولیتی نسبت داده شود و از انجام آن اطمینان حاصل شود. مدیر پروژه با استفاده از زمان‌بندی به عنوان یک راهنما، می‌تواند هر مرحله از فرآیند نرم‌افزار را پیگیری و کنترل کند. به بیان دیگر زمان‌بندی پروژه با هدف ایجاد یک زمان‌بندی مناسب و پویا جهت اتمام به موقع پروژه انجام می‌شود. البته پس از تکمیل برنامه‌ی زمان‌بندی، باید به حسن اجرای این برنامه نیز نظارت داشت که به این فعالیت پیگیری یا همان ردیابی[1] پروژه گفته می‌شود. زیرا همانطور که پیش‌تر نیز گفتیم **تهیه و تدوین برنامه‌ریزی** به معنی مشخص‌کردن چگونگی و نحوه انجام کار و ارائه زمان‌بندی مناسب بر اساس مهلت زمانی مشخص شده از سوی کارفرما، بر عهده مدیر پروژه است. همچنین **اجرا و نظارت بر برنامه‌ریزی** تهیه شده جهت حرکت به سمت مقصد مورد نظر بر اساس زمان‌بندی تعیین‌شده نیز بر عهده مدیر پروژه است.

می‌توان گفت عمده‌ترین دلیل زمانبندی پروژه، جلوگیری از تأخیر در تحویل نرم‌افزار است.

زمان‌بندی پروژه فعالیتی است که با توجه به «مهلت زمانی» برای توسعه‌ی پروژه، نیروی

[1] Tracking

انسانی لازم را در میان وظایف موجود توزیع می‌کند. بنابراین زمان‌بندی، هم بر روی توزیع نیرو و هم بر زمان اجرای هر وظیفه نظارت دارد.

بدیهی است که در اولین مرحله‌ی زمان‌بندی، ابتدا باید تمامی وظایف پروژه از طریق روش‌هایی همچون مصاحبه، گفتگو، پرسش‌نامه و نمونه‌سازی، شناسایی شوند.

بعد از شناخت وظایف و وابستگی‌های مابین آن‌ها، باید نیروی انسانی پروژه را مابین وظایف مختلف توزیع کرد. توزیع نیروی کار براساس برآوردها انجام می‌شود. فعالیت توزیع نیروی کار از قانون ۴۰-۲۰-۴۰، پیروی می‌کند. ۴۰ درصد از نیروی پروژه باید به فعالیت‌های مدل تحلیل و مدل طراحی، ۲۰ درصد به فعالیت پیاده‌سازی و ۴۰ درصد نیز به فعالیت تست نرم‌افزار اختصاص یابد.

هنگامی‌که روند حرکت پروژه به هر دلیلی از زمان‌بندی تهیه‌شده بر اساس مهلت زمانی عقب افتاده است، صرف فقط اضافه‌کردن برنامه‌نویسان جدید، راهکار نهایی نخواهد بود، زیرا اگرچه تعداد برنامه‌نویسان پروژه به لطف ورود برنامه‌نویسان جدید افزایش یافته است، اما این برنامه نویسان جدید، نیاز به آموزش و آشنایی با پروژه فعلی دارند. این آموزش‌ها توسط برنامه‌نویسان قدیم انجام می‌گردد، بنابراین همان برنامه‌نویسان قدیم را جهت آموزش برنامه‌نویسان جدید از دست داده‌ایم، که منجر به توقف روند رو به جلوی کنونی نیز می‌شود.

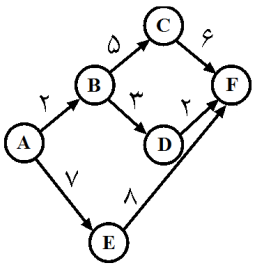
روش‌های زمان‌بندی پروژه

در زمان‌بندی پروژه، باید مشخص شود که هر وظیفه در چه بازه‌ی زمانی باید شروع شود و چه مدت زمان نیاز دارد تا تکمیل شود. تکنیک‌های ارزیابی و بازبینی برنامه[1] (PERT) و روش مسیر بحرانی[2] (CPM) جهت زمان‌بندی پروژه مورد استفاده قرار می‌گیرند. هر دو روش PERT و CPM توسط نمودارهای شبکه‌ای به برنامه‌ریز پروژه امکانات زیر را می‌دهند:

۱- نمایش شبکه وظایف و مسیرهای بحرانی پروژه

۲- برآورد زمان لازم برای انجام وظایف از طریق مدل‌های آماری

۳- محاسبه زمان‌بندی‌های مرزی میان وظایف



یکی از مهم‌ترین مسائل در زمان‌بندی پروژه تعیین وابستگی بین وظایف است. برای شناسایی و نمایش وابستگی بین وظایف از شبکه‌ای به نام شبکه‌ی وظایف[3] استفاده می‌شود. شکل مقابل نشان می‌دهد که وظایف چطور در یک پروژه فرضی پیش‌نیاز یکدیگرند.

[1] Program evaluation and review technique
[2] Critical path Method
[3] Task network

به عنوان مثال وظیفه AB پیش‌نیازی برای دو وظیفه BC و BD است و به همین ترتیب به صورت غیرمستقیم پیش‌نیازی برای دو وظیفه‌ی دیگر CF و DF نیز می‌باشد. بنابراین امکان توسعه‌ی موازی هیچ‌یک از این چهار وظیفه با وظیفه AB وجود ندارد. اعداد روی هر یال می‌تواند مدت زمان لازم برای تکمیل هر وظیفه را مشخص کند. در این مثال فرض کنید که اعداد روی هر یال، تعداد روزهای لازم جهت تکمیل هر وظیفه را نشان می‌دهند.

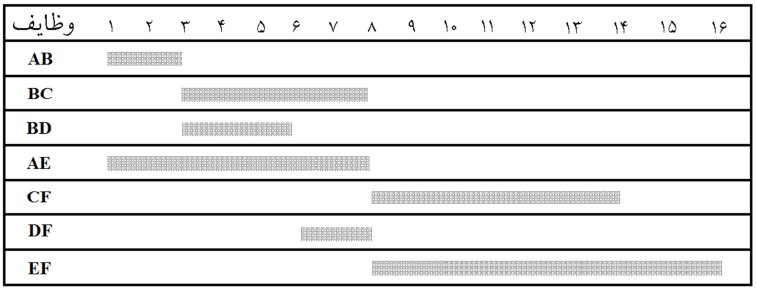
مسیر بحرانی مسیری از گراف شبکه‌ی وظایف است که اگر بخواهیم پروژه از زمان‌بندی خود عقب نماند، باید تمام وظایف روی این مسیر بدون تأخیر انجام شده و در موعد مقرر به اتمام برسند. در مثال بالا حداقل زمان لازم جهت تکمیل پروژه پانزده روز است. این زمان مربوط به مسیر AEF می‌باشد. مسیری که از گره شروع (A) آغاز شده به گره پایان (F) ختم شده است. مجموع وزن یال‌های بقیه مسیرها، از پانزده کمتر است. بنابراین مسیر بحرانی برنامه، مسیر AEF است. حال اگر طبق برنامه‌ریزی، قرار باشد فعالیت BC در ابتدای روز سوم آغاز شود، ولی در واقعیت این وظیفه در ابتدای روز چهارم شروع شود تحویل پروژه یک روز به تأخیر می‌افتد! بنابراین می‌توان گفت که شبکه وظایف که با نمودارهای PERT و CPM نشان داده می‌شوند، ابزاری جهت نمایش وابستگی داخلی بین وظایف است و جهت تعیین مسیر بحرانی به کار می‌رود.

پس از آنکه زمان‌های قطعی و مشخص برای شروع و پایان هر وظیفه تعیین شد، نوبت به مدل‌سازی زمان‌بندی کلی پروژه می‌رسد، نمودار زمانی یا نمودار گانت جهت این مرحله مورد استفاده قرار می‌گیرد.

نمودار گانت یا نمودار زمانی

نمودار گانت نوعی نمودار میله‌ای است که شروع و پایان وظایف یا فعالیت‌های پروژه را نشان می‌دهد. در این نمودار ارتباطات پیش‌نیاز میان وظایف و فعالیت‌ها بهتر نشان داده می‌شود. در این نمودار تمام وظایف در سمت چپ نمودار مشخص می‌گردد و زمان شروع هر وظیفه، مدت زمان انجام آن و زمان خاتمه آن در سمت راست نمودار مدل‌سازی می‌گردد. هنگامی‌که چند خط میله‌ای در یک بازه‌ی زمانی مشترک به صورت موازی رسم می‌شوند، این نتیجه استنباط می‌گردد که در آن بازه‌ی زمانی، وظایف مربوط به آن خطوط به صورت موازی اجرا خواهند شد.

مثال: شکل زیر نمودار گانت مربوط به پروژه قبل را نشان می‌دهد:



توجه: در ادبیات زمان‌بندی پروژه گاهی به جای استفاده از کلمه «**وظیفه**»، از عبارت «**ساختار توقف کار**» (WBS) [1] نیز استفاده می‌شود که همان معنای وظیفه را خواهد داشت.

پیگیری زمان‌بندی پروژه

زمان‌بندی پروژه، نقشه‌ی راهنمایی برای مدیر پروژه فراهم می‌آورد. اگر زمان‌بندی پروژه درست تهیه شده باشد، وظایف و نقاط عطفی را که باید به موازات پیشرفت پروژه پیگیری و کنترل شوند، تعیین می‌کند. پیگیری زمان‌بندی پروژه به روش‌های زیر انجام می‌شود:

- برپایی نشست‌های ادواری درباره‌ی وضعیت پروژه، که در آن اعضای هر تیم، پیشرفت‌ها و مشکلات پروژه را گزارش می‌کنند.
- ارزیابی نتایج کلیه‌ی بازبینی‌ها که در سرتاسر فرآیند مهندسی نرم‌افزار اجرا می‌شوند.
- بررسی اینکه آیا نقاط عطف پروژه یا فازهای پروژه (Milestone) در تاریخ زمان‌بندی شده به پایان رسیده‌اند یا خیر. هر Milestone مجموعه‌ای از وظایف مرتبط به یکدیگر را نشان می‌دهد که می‌توانند به عنوان یک فاز از پروژه در نظر گرفته شوند.
- مقایسه‌ی تاریخ شروع واقعی با تاریخ شروع برنامه‌ریزی شده برای هریک از وظایف پروژه
- برگزاری جلسات رسمی با سازندگان نرم‌افزار، به منظور بررسی ارزیابی شخصی آنها از پیشرفت زمانی پروژه و مشکلات حاضر در مسیر
- استفاده از تحلیل ارزش حاصله (EVA) به منظور ارزیابی کمی پیشرفت پروژه

در دنیای واقعی، مدیران پروژه از تمامی این روش‌های پیگیری زمان‌بندی پروژه استفاده می‌کنند. در ادامه روش تحلیل ارزش حاصله مورد بررسی قرار می‌گیرد.

تحلیل ارزش حاصله (EVA: Earned Value Analysis)

تحلیل ارزش حاصله یک تکنیک کمّی برای سنجش پیشرفت، به موازات انجام وظایف تخصیص داده شده در زمان‌بندی پروژه است. بیان ساده‌تر، ارزش حاصله، میزانی از پیشرفت است. این میزان باعث سنجش «درصد کامل شدن» پروژه با استفاده از تحلیل کمّی، (به جای حدسیات) می‌شود.

برای تعیین ارزش حاصله، مراحل زیر اجرا می‌شود:

۱- بودجه‌ی کار زمان‌بندی شده (BCWS: Budgeted Cost of Work Scheduled)

بودجه‌ی کار زمان‌بندی شده توسط فعالیت برآورد برای هر وظیفه‌ی مهندسی نرم‌افزار (برحسب نفر ساعت یا نفر روز) محاسبه می‌گردد. بنابراین، $BCWS_i$ کار برنامه‌ریزی شده برای وظیفه‌ی کاری i است. برای تعیین پیشرفت در یک نقطه‌ی مفروض از زمان‌بندی پروژه، مقدار BCWS، حاصل جمع مقادیر $BCWS_i$ برای همه‌ی وظایف کاری است که باید تا آن نقطه از زمان‌بندی کامل شده باشند.

---

[1] Work breakdown structure

۲- بودجه تکمیل کل پروژه (BAC)

بودجه تکمیل پروژه از حاصل جمع کلیه‌ی مقادیر $BCWS_i$ برای وظایف کاری بدست می‌آید.

$$BAC = \sum (BCWS_i)$$

۳- محاسبه مقدار هزینه‌ی کار انجام‌شده (BCWP: Budget Cost of Work Performed)

BCWP برابر حاصل جمع مقادیر BCWS برای همه وظایف کاری که طبق زمان‌بندی پروژه انجام شده‌اند.

توجه: تفاوت BCWS و BCWP در این است که اولی نشانگر بودجه‌ی فعالیت‌هایی است که انجام آنها برنامه‌ریزی شده است و دومی هزینه‌ی فعالیت‌هایی را نشان می‌دهد که واقعاً انجام شده‌اند.

حال با داشتن مقادیر BCWS، BAC و BCWP می‌توان شاخص‌های مهم پیشرفت پروژه را محاسبه کرد:

شاخص کارایی زمان‌بندی $SPI = \frac{BCWP}{BCWS}$

واریانس زمان‌بندی $SV = BCWP - BCWS$

SPI نشان‌دهنده‌ی کارآیی زمان‌بندی پروژه بود و برای ارزیابی اجرای برنامه به کار می‌رود.

توجه: چنانچه این شاخص (SPI) بزرگتر از یک باشد نشانگر این است که پروژه زودتر از زمان‌بندی اولیه اجرا شده است و اگر کوچکتر از یک باشد نشان می‌دهد که کار انجام شده دیرتر از برآورد اولیه اجرا شده است و اگر $SPI = ۱$ باشد پروژه مطابق زمان‌بندی اولیه اجرا شده است.

SV شاخص مطلقی از انحراف نسبت به زمان‌بندی برنامه‌ریزی شده است:

درصد زمان‌بندی شده برای کامل شدن پروژه $= \frac{BCWS}{BAC}$

توجه: مقدار فوق نشانگر درصد کاری است که باید در زمان t انجام شده باشد.

درصد کامل شده $= \frac{BCWP}{BAC} \times ۱۰۰$

توجه: این مقدار، شاخص کمی از «درصد کامل شدن» پروژه در نقطه‌ای مفروض در زمان t است.

هزینه‌ی واقعی کار انجام شده (ACWP) را نیز می‌توان محاسبه کرد. مقدار ACWP، مجموع فعالیت واقعی صرف شده برای کارهایی است که تا آن نقطه در زمان‌بندی پروژه کامل شده‌اند. بنابراین می‌توان مقادیر زیر را محاسبه نمود:

شاخص کارآیی هزینه $CPI = \frac{BCWP}{ACWP}$

واریانس هزینه $CV = BCWP - ACWP$

توجه: اگر $CPI = ۱$ باشد، پروژه مطابق برآورد هزینه‌ی اولیه پیش رفته است.

توجه: CV، شاخص مطلقی از صرفه‌جویی در هزینه‌ها (در مقابل هزینه‌های برنامه‌ریزی شده)

یا کمبود بودجه در مرحله خاصی از پروژه است.

توجه: به طور کلی «تحلیل مقدار حاصل» (EVA) مانند یک رادار، مشکلات زمان‌بندی پروژه را پیش از وقوع روشن می‌سازد. بدین ترتیب مدیر پروژه قادر خواهد بود تا پیش از بحرانی شدن اوضاع، عمل مناسب را پیش گیرد.

مدیریت ریسک

برای انجام پروژه‌های بزرگ نرم‌افزاری لازم است تحلیل ریسک انجام شود، اما اکثر مدیران پروژه‌های نرم‌افزاری آن را به طور سطحی انجام می‌دهند، یا اصلاً انجام نمی‌دهند. زمانی که صرف شناسایی، تحلیل و مدیریت ریسک می‌شود، سبب می‌شود تا مشکلات کمتری در حین ایجاد پروژه رخ دهد.

به قول سون تزو، ژنرال چینی که ۲۵۰۰ سال قبل می‌زیسته است: «اگر دشمن و خودتان را بشناسید، دیگر لازم نیست از صد تیر هم هراس داشته باشید.»

ریسک یک مشکل بالقوه است، یعنی ممکن است رخ دهد یا ندهد. ولی نتیجه هر چه باشد، بهتر است احتمال وقوع آن شناسایی، میزان تأثیر آن برآورد و یک برنامه برای مقابله با آن وضع شود.

رابرت چارته در کتاب خود با عنوان تحلیل و مدیریت ریسک، یک تعریف مفهومی از ریسک ارائه داده است:

ریسک به اتفاقات آینده مربوط می‌شود. دیگر، کاری به امروز و دیروز ندارد. زیرا آنچه را که امروز برداشت می‌کنیم، دیروز کاشته‌ایم. یک برداشت خوب، حاصل یک کاشت و داشت خوب است.

در طول توسعه‌ی یک پروژه، همواره ریسک‌هایی وجود دارد که خصوصیات پروژه‌های موفق نرم‌افزاری (زمان، هزینه و نیازمندی‌ها) را تهدید می‌کند.

از نگاه مشتری یک پروژه موفق نرم‌افزاری، پروژه‌ای است که بر اساس سه خصوصیت اساسی زیر تولید گردد:

۱- بازه‌ی زمانی از قبل برنامه‌ریزی شده (بازه‌ی زمانی مشخص)

۲- بودجه‌ای از قبل پیش‌بینی شده و با صرف کمترین هزینه (مقرون به صرفه)

۳- دقیقاً مطابق با نیازمندی‌های واقعی کاربران (کیفیت مطلوب)

بنابراین هرآنچه باعث تهدید خصوصیات فوق گردد، ریسک محسوب می‌گردد.

در مدیریت ریسک با اتخاذ تصمیمات مناسب هم باید به فکر کاهش احتمال ریسک بود، و هم به فکر مقابله با وقوع ریسک‌های احتمالی بود.

استراتژی‌های ریسک

به طور کلی استراتژی‌های ریسک به دو نوع پس‌کنشی و پیش‌کنشی تقسیم می‌شود:

۱- ریسک پس کنشی (reactive)

در این نوع ریسک که موسوم به روش «جنگ با آتش» نیز می‌باشد تا زمانی که خطری پیش

نیامده است هیچ عملی انجام نمی‌شود و به محض بروز مشکل، تیم نرم‌افزار سعی در رفع سریع آن می‌کند.

اگر این روش با شکست مواجه شود آنگا باید «مدیریت بحران» انجام شود زیرا پروژه در خطر واقعی قرار گرفته است. ریسک پس‌کنشی موسوم به ریسک واکنشی نیز می‌باشد.

۲- ریسک پیش کنشی (proactive)

در این نوع ریسک، ریسک‌های بالقوه شناسایی و احتمال وقوع و درجه تأثیر آنها، مورد بررسی قرار می‌گیرد. سپس از نظر درجه‌ی اهمیت طبقه‌بندی می‌گردند. در نهایت، تیم نرم‌افزاری طراحی را برای مدیریت ریسک ایجاد می‌کند.

ریسک‌های نرم‌افزاری

هر ریسک دو خصوصیت زیر را شامل می‌شود.

عدم قطعیت (Uncertainty): ریسک ممکن است اتفاق بیافتد و ممکن است اتفاق نیافتد. وقوع ریسک همواره بین صفر و یک است ($0 < P_{Risk} < 1$). یعنی هیچ ریسکی با احتمال ۱۰۰٪ وجود ندارد. ریسک با احتمال ۱۰۰٪ دیگر یک ریسک نیست، بلکه یک محدودیت (constraint) برای پروژه محسوب می‌گردد.

خسارت (Loss): اگر ریسک به وقوع بپیوندد یعنی به واقعیت تبدیل شود، آنگاه پیامدهای ناخواسته‌ای از جنس خسارت با خود به همراه خواهد داشت.

جهت مدیریت ریسک‌ها، ابتدا باید آنها را شناسایی و سپس مدیریت کرد. به طور کلی ریسک‌های نرم‌افزاری به سه دسته زیر طبقه‌بندی می‌شوند:

۱- ریسک‌های پروژه‌ای

اگر ریسک‌های پروژه‌ای اتفاق بیافتد، آنگاه زمان‌بندی پروژه عقب می‌افتد و هزینه افزایش می‌یابد. بنابراین ریسک پروژه‌ای برنامه‌ریزی و زمان‌بندی پروژه را تهدید می‌کند و منجر به دوباره کاری و افزایش هزینه‌ها می‌گردد. مانند زمان‌بندی نامناسب، عدم مدیریت و سازماندهی نیروی انسانی لازم، عدم درک کافی نیازمندی‌های مشتری و عدم برآورد و تخمین مناسب برای پروژه. کلیه‌ی ریسک‌هایی که خصوصیات پروژه‌های موفق نرم‌افزاری (زمان، هزینه و نیازمندی‌ها) را تهدید کند، جزو دسته‌ی ریسک‌های پروژه‌ای قرار دارند.

۲- ریسک‌های فنی

اگر ریسک‌های فنی اتفاق بیافتد، آنگاه پیاده‌سازی مشکل و یا غیرممکن می‌شود. بنابراین ریسک فنی کیفیت محصول نهایی و سر موعد تحویل دادن نرم‌افزار را تهدید می‌کند. ریسک‌های فنی، مشکلات بالقوه‌ی تحلیل، طراحی، پیاده‌سازی، ایجاد واسط، اعتبارسنجی و نگهداری را نمایان می‌کنند. به‌علاوه، ابهام در مشخصات، عدم قطعیت فنی، بی‌استفاده شدن فنی، و فناوری پیشرو نیز از عوامل ریسک به شمار می‌روند. ریسک‌های فنی بدین دلیل رخ می‌دهند که حل مسأله از آنچه

تصور می‌شده، دشوارتر است.

۳- ریسک‌های تجاری

ریسک‌های تجاری عملی بودن ساخت نرم‌افزار را تهدید می‌کنند. به بیان دیگر ریسک‌های تجاری شامل تهدیداتی می‌شوند که نتایج مورد انتظار محصول را به مخاطره انداخته و گاه امکان تکمیل پروژه را تهدید می‌کند. پنج نمونه از عمده‌ترین ریسک‌های تجاری به صورت زیر است:

الف) ریسک بازاری (market risk): ساخت محصولی عالی و درست که هیچ‌کس خواهان آن نیست. مانند ایجاد بازی برای آتاری.

ب) ریسک راهبردی (strategic risk): ساخت محصولی که دیگر در راهبرد تجاری سازمان خریدار نگنجد.

ج) ریسک فروش (sales risk): اگر ریسک فروش اتفاق بیافتد، آنگاه محصولی تولید می‌شود که قسمت فروش نمی‌داند آنرا چگونه بفروشد.

د) ریسک مدیریتی (management risk): اگر ریسک مدیریتی اتفاق بیافتد، آنگاه پروژه حمایت مدیران رده بالا را از دست می‌دهد. به دلیل تغییر نقطه تمرکز مدیران و یا تغییر مدیران.

هـ) ریسک بودجه‌ای (budget risk): از دست دادن وعده‌های تأمین منابع مالی و نیروی انسانی.

شناسایی ریسک

پس از معرفی انواع ریسک‌ها، باید به شناسایی و تعیین میزان تاثیر هر ریسک پرداخت. شناسایی ریسک‌ها یک کوشش سیستماتیک در جهت مشخص‌کردن چیزهایی است که برنامه‌ریزی پروژه شامل برآوردها و زمان‌بندی را تهدید می‌کنند. مدیر پروژه با شناسایی ریسک‌های قابل پیش‌بینی و شناخته‌شده، نخستین گام به سمت پرهیز از آنها در صورت امکان، و کنترل آنها در صورت لزوم بر می‌دارد. برخی ریسک‌ها، ریسک‌های عمومی (Generic risks) هستند و برخی ریسک‌ها، ریسک‌های اختصاصی (Product specific risks) یعنی ریسک مختص محصول و پروژه هستند. ریسک‌های عمومی برای همه پروژه‌های نرم‌افزاری تهدیدی بالقوه به شمار می‌روند. برای شناسایی ریسک‌های عمومی پروژه، داده‌های تاریخی پروژه‌های گذشته مورد بررسی قرار می‌گیرد. اما ریسک‌های اختصاصی را فقط کسانی می‌توانند شناسایی کنند که درک درستی از فناوری، افراد و محیط خاص پروژه مورد نظر دارند. برای شناسایی ریسک‌های خاص پروژه، برنامه‌ریزی پروژه و بیان حوزه کاربرد نرم‌افزار مورد بررسی قرار می‌گیرد و پاسخی برای این سوال به دست می‌آید: «کدام ویژگی‌های این محصول ممکن است برنامه‌ریزی ما را مورد تهدید قرار دهد؟» برای‌مثال اندازه بزرگ برنامه، ریسک نرسیدن به زمان‌بندی را با خود به همراه دارد.

یک روش برای شناسایی ریسک‌ها ایجاد فهرست کنترلی از آیتم‌های ریسک است. این فهرست کنترلی را می‌توان برای شناسایی ریسک‌ها و مورد توجه قراردادن زیرمجموعه‌ای از ریسک‌های شناخته‌شده و قابل پیش‌بینی به کار گرفت.

تعیین مقدار در معرض ریسک قرار گرفتن (Risk Exposure)

در پیش‌بینی ریسک، که برآورد ریسک نیز خوانده می‌شود، کوشش می‌شود که ریسک از دو لحاظ مورد سنجش قرار گیرد. اول بررسی **احتمال** به واقعیت پیوستن ریسک و دوم میزان **تاثیر** و پیامدهای مرتبط با ریسک در صورت وقوع ریسک. برآورد این دو مقدار برای هر ریسک مبتنی بر تجربیات تیم توسعه و داده‌های تاریخی پروژه‌های گذشته است. به این نحو که از تک تک اعضای تیم، به شیوه‌ی **پرسش‌نامه و نوبت‌چرخشی**، آن‌قدر نظرخواهی می‌شود تا اینکه مقدار **احتمال ریسک** و **تاثیر ریسک** به تدریج شروع به همگرایی کند. هنگامی که ریسک‌های پروژه مشخص شد، لیست ریسک‌ها در یک جدول با عنوان جدول ریسک (risk table) بر اساس احتمال ریسک و تاثیر ریسک، مرتب‌سازی می‌شود. ریسک‌های با احتمال بالا و تاثیر زیاد در بالای جدول جمع می‌شود و ریسک‌های با احتمال پایین و تاثیر کم در انتهای جدول قرار می‌گیرند. به این ترتیب اولویت‌بندی مرتبه‌ی اول ریسک‌ها انجام می‌شود. مدیر پروژه جدول ریسک حاصل را بررسی می‌کند و یک خط برش یا مرزی (Cutoff line) مشخص می‌کند. این خط برش که به صورت افقی در نقطه‌ای از جدول کشیده می‌شود، نشان می‌دهد که فقط ریسک‌های واقع در بالای این خط برش، شایسته توجه بیشتر هستند. ریسک‌هایی که زیر این خط برش قرار می‌گیرند، دوباره ارزیابی می‌شوند تا اولویت‌بندی مرتبه دوم برای آنها صورت گیرد. نتیجه اینکه رسیدگی به تمامی ریسک‌های پروژه غیر ممکن است، و بهتر است به ریسک‌هایی که بالای خط برش هستند، رسیدگی شود. همچنین احتمال ریسک‌ها در طول پروژه ثابت نیست، بلکه متغییر است. که بسته به شرایط پروژه ریسک‌ها در طول پروژه در جدول ریسک جا به جا و اولویت‌بندی می‌شوند. جدول ریسک شامل ستون‌های عنوان ریسک، احتمال ریسک، تاثیر ریسک و طرح RMMM شامل مراحل کاهش (Mitigation)، نظارت (Monitoring) و مدیریت (Management) بر ریسک (Risk) می‌باشد. این طرح موسوم به RMMM می‌باشد که در ادامه بررسی می‌کنیم. جدول زیر گویای مطلب است:

عنوان ریسک	احتمال ریسک	تاثیر ریسک	RMMM
برآورد کمتر از میزان واقعی	60%	2	
عدم استفاده مجدد لازم	70%	2	
مقاومت کاربران نهایی در برابر سیستم	40%	3	
کافی نبودن مهلت تحویل	50%	2	
از دست رفتن سرمایه	40%	1	
تغییر دادن خواسته‌ها توسط مشتری	80%	2	
بی‌تجربگی کارمندان	30%	2	
بالا بودن تغییر و تحول کارمندان	70%	2	
...	...	...	...
...	...	...	...

توجه: ارزش تاثیرات ریسک در جدول ریسک فوق به صورت زیر است:

۱- فاجعه‌بار
۲- بحرانی
۳- کم اهمیت
۴- قابل چشم‌پوشی

یک ریسک که تاثیر زیاد ولی احتمال وقوع اندکی دارد، نباید مقدار چشمگیری از زمان مدیریت ریسک را به خود اختصاص دهد، اما ریسک‌های با تاثیر بالا و احتمال وقوع متوسط تا زیاد و ریسک‌های با تاثیر اندک و احتمال وقوع بالا باید مقدار زیادی از زمان مدیریت ریسک به آن اختصاص داده شود.

به منظور کمّی‌سازی اثرات ریسک‌های پروژه، برای هر ریسک شناسایی شده، فاکتوری به نام Risk Exposure محاسبه می‌گردد. میزان کلی در معرض ریسک قرار گرفتن، RE، با استفاده از رابطه‌ی زیر تعیین می‌شود:

$$RE = P \times C$$

که در آن P احتمال وقوع ریسک و C میزان هزینه‌ای است که در صورت وقوع ریسک، پروژه خسارت خواهد دید.

توجه: در رابطه‌ی فوق ارزش **تاثیر ریسک** از جنس هزینه و در پارامتر C قرار می‌گیرد.

مثال: اگر احتمال ریسک ۸۰٪ و تأثیر ریسک بر حسب هزینه برابر ۵۰۰۰ واحد پولی باشد، میزان در معرض ریسک قرار گرفتن (Risk Exposure) برابر است با:

پاسخ:

مطابق رابطه‌ی RE داریم:

$$RE = P \times C$$

بنابراین داریم:

$$RE = ۰/۸ \times ۵۰۰۰ = ۴۰۰۰$$

مثال: تیم مدیریت پروژه، یکی از ریسک‌های موجود در ساخت برنامه‌ی کاربردی را به شکل زیر تعریف کرده است:

شناسایی ریسک: تنها ۷۰٪ قطعات (Components) نرم‌افزاری که برای استفاده مجدد برنامه‌ریزی شده بودند، در برنامه‌ی کاربردی استفاده خواهند شد یعنی به واقع مورد استفاده مجدد قرار می‌گیرند، علت این ریسک می‌تواند به دلیل عدم تطابق احتمالی واسط‌های مولفه‌ها و نقاط اتصالی باشد. باقی‌مانده کارکردهای نرم‌افزار باید کدنویسی و پیاده‌سازی شوند.

احتمال ریسک: ۸۰٪

تأثیر ریسک: می‌خواستیم از ۶۰ قطعه نرم‌افزاری (Software Component) در پروژه برای ساخت برنامه کاربردی استفاده کنیم. هر قطعه نرم‌افزاری ۱۰۰ خط کد می‌باشد و هزینه نوشتن هر خط کد

برنامه ۲۰۰۰۰ تومان است. برای این ریسک، میزان در معرض ریسک قرار گرفتن (Risk Exposure) برابر است با:

پاسخ:

مطابق رابطه‌ی RE داریم:

$$RE = P \times C$$

برای محاسبه‌ی مقدار RE، ابتدا باید مقدار C، یعنی هزینه‌ی خسارت وارده در هنگام وقوع ریسک، محاسبه شود. اگر ریسک شناسایی شده به وقوع بپیوندد، آنگاه تیم نرم‌افزاری، می‌بایست به ناچار، ۳۰٪ از ۶۰ مولفه یعنی مولفه‌هایی که قابل استفاده مجدد بودند را دوباره پیاده‌سازی کند، که می‌شود ۱۸ مولفه که باید از نو ساخته شود. همچنین هزینه‌ی ساخت هر مولفه به طور تقریبی به صورت زیر است:

هزینه‌ی ساخت هر مولفه = ۱۰۰ × ۲۰۰۰۰ = ۲۰۰۰۰۰۰

هزینه‌ی ساخت هر مولفه در رابطه‌ی فوق به طور تقریبی است، زیرا اطلاعات آماری بوده و محاسبات به صورت برآورد انجام می‌شود.

بنابراین هزینه‌ی کل خسارت وارده به پروژه یعنی ساخت مجدد ۱۸ مولفه، به صورت زیر خواهد بود:

$$C = ۱۸ \times ۲۰۰۰۰۰۰ = ۳۶۰۰۰۰۰۰$$

در نهایت بر اساس رابطه‌ی RE داریم:

$$RE = P \times C = ۰/۸ \times ۳۶۰۰۰۰۰۰ = ۲۸۸۰۰۰۰۰۰$$

توجه: اگر برآوردی از احتمال وقوع ریسک یعنی P و میزان هزینه‌ای که در صورت وقوع ریسک، پروژه خسارت خواهد دید یعنی C، در دست باشد، RE را می‌توان برای هر یک از ریسک‌های موجود در جدول ریسک محاسبه کرد. مقدار کلی RE برای همه‌ی ریسک‌های بالای خط برش (مرزی) در جدول ریسک، می‌تواند آگاهی برای تنظیم برآورد هزینه‌ی نهایی یک پروژه فراهم آورد. همچنین می‌توان از آن برای پیش‌بینی افزایش احتمالی منابع انسانی استفاده کرد که در نقاط گوناگون از زمان‌بندی پروژه، مورد نیاز هستند.

توجه: تیم پروژه باید جدول ریسک را در بازه‌های زمانی منظم مورد بازبینی قرار دهد و هر ریسک را دوباره ارزیابی کند تا تعیین کند که شرایط جدید چه زمانی باعث تغییر احتمال و تاثیر آن می‌شود. به عنوان نتیجه‌ای از این فعالیت، ممکن است لازم باشد ریسک‌های جدیدی به جدول افزوده شود، برخی ریسک‌ها از آن حذف شوند و موقعیت نسبی ریسک‌های موجود تغییر یابد.

طرح RMMM

ستون RMMM در جدول ریسک، به چهار واژه‌ی **کاهش** (Mitigation)، **نظارت** (Monitoring) ، **مدیریت** (Management) و **ریسک** (Risk) اشاره دارد. و به این معنی است که برای تمامی ریسک‌های بالای خط برش (مرزی) سه اقدام **کاهش**، **نظارت** و **مدیریت** باید انجام شود.

اقدام اول- کاهش ریسک (Mitigation): در این اقدام مدیر پروژه باید تلاش کند، علت ریسک را شناسایی و آنرا در بهترین حالت حذف کند. در واقع احتمال وقوع ریسک را به مقدار صفر در بهترین حالت کاهش دهد. و این یعنی جلوگیری و اجتناب از ریسک (risk avoidance). در واقعیت در بسیاری از ریسک‌ها فقط می‌توان احتمال ریسک را کاهش داد و جلوگیری کامل از آن عملا شدنی نیست.

برای مثال فرض کنید تغییر و تحول زیاد در کارمندان، به عنوان یک ریسک برای پروژه در نظر گرفته می‌شود. و بر اساس تجربه و نبوغ مدیریت، احتمال ریسک این تغییر و تحول زیاد و حدود 70% درصد برآورد شده است که نسبتا بالاست. همچنین درجه‌ی تاثیر ریسک بحرانی لحاظ شده‌است. یعنی تغییر و تحول زیاد، تاثیری بحرانی بر هزینه و زمان‌بندی پروژه دارد. بنابراین باید در حوزه‌ی تمرکز مدیر پروژه جهت انجام فعالیت‌های RMMM قرار بگیرد. که می‌بایست برای کاهش این ریسک، اقداماتی جهت کاهش تغییر و تحول در نظر گرفت. ازمیان مراحل احتمالی، به موارد زیر جهت **کاهش ریسک**، می‌توان اشاره کرد:

- ملاقات با کارمندان فعلی، جهت تعیین علل نارضایتی احتمالی آنها، مثل شرایط کاری نامناسب و حقوق پایین.
- کوشش برای ایجاد رضایت در کارمندان از طریق حذف و یا کاهش علل نارضایتی قبل از شروع پروژه، مثل افزایش حقوق.
- سازماندهی تیم‌های پروژه به نحوی که اطلاعات مربوط به هر فعالیت توسعه به طور گسترده پراکنده شود.
- تعریف استانداردهای مستندسازی و برقراری راهکارهایی برای حصول اطمینان از توسعه‌ی مستندات، به شیوه‌ی زمان‌بندی شده.
- انجام مرورهای موازی برای تمامی کارها.
- قراردادن کارمندان پشتیبان برای هر کدام از متخصصان موثر و پراهمیت.

اقدام دوم- نظارت ریسک (Monitoring): در این اقدام به موازات پیشرفت پروژه، فعالیت‌های نظارت ریسک آغاز می‌شود. احتمال وقوع ریسک‌ها به دلیل انجام اقدام کاهش ریسک و یا تغییرات پروژه تغییر می‌کند. بنابراین در اقدام نظارت ریسک می‌بایست کاهش یا افزایش احتمال وقوع ریسک‌ها و به تبع علل آن نیز مورد نظارت قرار گیرد. به عبارت دیگر کارآمدی و کیفیت اقدام کاهش ریسک و نتایج حاصل از آن در اقدام نظارت ریسک به طور مداوم رصد، کنترل و پایش می‌شود.

همانطور که گفتیم تغییر و تحول زیاد در کارمندان، به عنوان یک ریسک برای پروژه در نظر گرفته می‌شود. که می‌بایست برای کاهش این ریسک، اقداماتی جهت کاهش تغییر و تحول در نظر گرفت. که این اقدامات باید در مرحله‌ی کاهش ریسک انجام شود. حال در مرحله‌ی نظارت ریسک باید کارآمدی و کیفیت اقدام کاهش ریسک و نتایج حاصل از آن به طور مداوم رصد، کنترل و پایش شود. ازمیان مراحل احتمالی، به موارد زیر جهت **نظارت ریسک**، می‌توان اشاره

کرد:

- ارزیابی رفتار افراد تیم در زمانی‌که فشارهای کاری پروژه بالاست.
- بررسی همبستگی مابین اعضای تیم
- بررسی روابط شخصی مابین اعضای تیم
- بررسی مسائل بالقوه ناشی از حقوق و مزایا
- بررسی دیگر فرصت‌های شغلی که در داخل و خارج از سازمان برای افراد پروژه در دسترس است.

علاوه بر پایش عوامل ذکر شده در بالا، مدیر پروژه باید موثر بودن مراحل کاهش ریسک را نیز پایش کند. برای‌مثال، یک مرحله‌ی کاهش ریسک که در بالا ذکر شد، به تعریف «استانداردهای مستندسازی و برقراری راهکارهایی برای حصول اطمینان از توسعه‌ی مستندات، به شیوه‌ی زمان‌بندی شده» اختصاص داده شده است. این راهکار برای حصول اطمینان از پیوستگی، در صورت خروج یک فرد مهم از پروژه است. مدیر پروژه باید مستندات را به دقت پایش کند تا مطمئن شود که همگی می‌توانند روی پای خود بایستند و در صورتی که فرد تازه واردی مجبور شد در میانه‌ی پروژه به تیم ملحق شود، بتوانند اطلاعات لازم را در اختیار وی قرار دهند.

اقدام سوم- مدیریت ریسک (Management): در این اقدام، پروژه در وضعیتی تصور می‌شود که فعالیت‌های کاهش ریسک با شکست مواجه شده‌است و ریسک به وقوع پیوسته است. در این شرایط، مدیر پروژه راه کارهایی جهت کنترل واقعه مشخص می‌کند. به این کار برنامه‌ریزی احتیاطی (contingency planning) نیز گفته می‌شود.

همانطور که گفتیم تغییر و تحول زیاد در کارمندان، به عنوان یک ریسک برای پروژه در نظر گرفته می‌شود. که می‌بایست برای کاهش این ریسک، اقداماتی جهت کاهش تغییر و تحول در نظر گرفت. که این اقدامات باید در مرحله‌ی کاهش ریسک انجام شود. همچنین در مرحله‌ی نظارت ریسک باید کارآمدی و کیفیت اقدام کاهش ریسک و نتایج حاصل از آن به طور مداوم رصد، کنترل و پایش شود. حال اگر این ریسک به وقوع بپیوندد یعنی به جایی می‌رسیم که پروژه به خوبی در جریان است و چند نفر از کارمندان اعلام می‌کنند که می‌خواهند دست از کار بکشند اگر فعالیت کاهش ریسک دنبال شده باشد، نیروی پشتیبان در دسترس است، اطلاعات مستندسازی شده است و آگاهی‌های لازم به افراد تیم داده شده است، ازمیان مراحل احتمالی، به موارد زیر جهت **مدیریت ریسک**، می‌توان اشاره کرد:

- تمرکز منابع بر روی عملیاتی که کارمندان آن کامل هستند و زمان‌بندی مجدد پروژه تا افراد تازه واردی که به تیم اضافه شده‌اند، بتوانند خودشان را برسانند.
- از افرادی که قصد خروج از پروژه را دارند درخواست شود تا هفته‌های آخر حضور خود را صرف انتقال آگاهی‌ها کنند. این ممکن است شامل اطلاع‌رسانی ویدیویی، تهیه مستندات توضیحی و یا ملاقات با اعضای دیگری باشد که در تیم باقی می‌مانند.

شایان ذکر است که مراحل RMMM باعث افزایش هزینه‌ی پروژه و تحمیل هزینه‌های اضافی به پروژه می‌گردد. برای مثال، صرف وقت جهت تامین نیروی پشتیبان برای هر یک از افراد فنی

مهم، هزینه بر می‌دارد. بنابراین، بخشی از مدیریت ریسک، ارزیابی موقعیتی است که، افزایش مزایا به واسطه‌ی مراحل RMMM، هزینه‌ی مقرون به صرفه‌ای ندارد، به عبارت دیگر مزایای RMMM نمی‌تواند هزینه‌ی اجرای آنرا توجیه کند. هنگامی بهتر است از RMMM استفاده کرد که مزایای حاصل از RMMM بیشتر از هزینه‌ی اجرای آن باشد.

قابلیت اطمینان نرم‌افزار

قابلیت اطمینان یک برنامه‌ی کامپیوتری، عنصر مهمی از کیفیت کلی آن به شمار می‌رود. اگر برنامه‌ای به دفعات از اجرا باز بماند، عوامل کیفیتی دیگر نرم‌افزار، اهمیت خود را از دست خواهند داد.

قابلیت اطمینان به زبان آماری، به عنوان «احتمال عملکرد خالی از شکست یک برنامه‌ی کامپیوتری در محیطی مشخص برای یک زمان معین» تعریف می‌شود.

برای مثال، اگر برنامه X در عرض هشت ساعت پردازش دارای قابلیت اطمینان ۹۶٪ باشد و در طول این هشت ساعت، برنامه ۱۰۰ بار اجرا شود، این برنامه احتمالاً ۹۶ بار از ۱۰۰ بار را درست کار کرده است. بنابر تعریف ارائه شده قابلیت اطمینان نرم‌افزار را می‌توان با استفاده از اطلاعات آماری به صورت **مستقیم** اندازه‌گیری نمود.

اندازه‌گیری قابلیت اطمینان

برای اندازه‌گیری قابلیت اطمینان می‌توان از معیار ساده MTBF که سرواژه‌ی عبارت Mean Time Between Failures و به معنی **متوسط زمان بین دو خرابی** است، استفاده کرد که مطابق رابطه‌ی زیر است:

MTBF=MTTF+MTTR

MTTF سرواژه‌ی عبارت Mean Time To Failure و به معنی **متوسط فاصله‌ی زمانی تا خرابی دوباره** است و MTTR سرواژه‌ی عبارت Mean Time To Repair و به معنی **متوسط زمان رفع خرابی است**. زمانی که یک خرابی رخ می‌دهد، ابتدا مدت زمانی طول می‌کشد تا خرابی تعمیر شود (MTTR) و سپس مدت زمانی طول می‌کشد تا خرابی دوباره رخ دهد (MTTF) و این یعنی MTBF که به معنی **متوسط زمان بین دو خرابی است**. هرچه مقدارMTBF افزایش یابد، تعداد خرابی‌ها در واحد زمان کاهش می‌یابد و این یعنی افزایش قابلیت اطمینان نرم‌افزار.

بسیاری از پژوهشگران معتقدند MTBF به مراتب مفیدتر از تعداد نقایص به ازای هر KLOC ($\frac{\text{Defect}}{\text{KLOC}}$) یا تعداد نقایص به ازای هر FP ($\frac{\text{Defect}}{\text{FP}}$) است. زیرا نقص‌هایی ممکن است پس از ۳۰۰۰۰ ساعت مصرف CPU آشکار شوند.

اندازه‌گیری قابلیت دسترسی

قابلیت دسترسی نرم‌افزار به معنی احتمال کار کردن برنامه طبق خواسته‌ها در یک بازه‌ی زمانی مشخص می‌باشد که به صورت زیر تعریف می‌شود:

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \times \%۱۰۰$$

و یا به طور ساده‌تر داریم:

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTBF}} \times \%۱۰۰$$

توجه: اندازه‌ی قابلیت اطمینان (MTBF) به یک اندازه به MTTR و MTTF حساس است. اما اندازه‌ی قابلیت دسترسی به MTTR بیش‌تر حساس می‌باشد.

الگوهای طراحی یا Design Patterns

برای هر کدام از ما پیش آمده که در مواجهه با یک مساله طراحی از خود بپرسیم که آیا کسی برای این مساله راهکاری پیدا کرده است؟ پاسخ تقریبا همیشه مثبت است! حل مساله، یافتن راهکار است.

الگوهای طراحی یا Design Patterns به عنوان راهکار در حل مساله، شما را از «اختراع دوباره چرخ» مصون نگاه می‌دارد؛ الگوهای طراحی به عنوان راهکار در حل مساله؛ اگر به طور موثر استفاده شود، همان نتایج موفق را مجددا تکرار می کند.

الگوی غیرمولد (non generative) مساله را تعریف می کند اما راهکار ارائه نمی دهد؛ اما الگوی مولد (generative) مساله را تعریف می کند و راهکار حل مساله را نیز ارائه می دهد.

الگوی طراحی را می‌توان «یک قاعده‌ی سه بخشی دانست که واسط میان یک حیطه یا حوزه معین (Context)، یک مساله (Problem) و یک راهکار یا راه حل (Solution) را بیان می‌کند.»

برای طراحی نرم‌افزار، حیطه به طراح نرم‌افزار این امکان را می‌دهد تا محیطی را که مساله در آن جای دارد، درک کند و دریابد چه راهکاری ممکن است در این محیط مناسب باشد. برای مثال کیفیت خانه برای برآورده ساختن آرامش و نیازهای مشتری، مهم است. اما شیوه‌های رسیدن به این مهم در شهرهای شمالی و جنوبی شباهت‌ها و تفاوت‌هایی دارد. به طور مثال در شهرهای شمالی بارندگی به وفور وجود دارد، پس روش‌ها و ابزارهای ساختمانی باید به گونه‌ای انتخاب شوند که در برابر بارندگی مقاوم باشند. مثل روش سقف شیروانی با استفاده از ابزار سفال! بنابراین محیط عملیاتی یک مساله، بر ارائه‌ی راهکار برای حل آن مساله تاثیر دارد.

به دو روش می‌شود به نتیجه رسید:

روش آزمون و خطا شده توسط دیگران که ابهام ندارد و روش آزمون و خطا توسط خودمان که ممکن است در مسیر دچار ابهام شویم.

به طور کلی محیط عملیاتی، مجموعه‌ای از خواسته‌ها، محدودیت‌ها و قید و بندها، بر روی درک خود صورت مساله و سپس ارائه راهکار برای حل مساله تاثیر دارد.

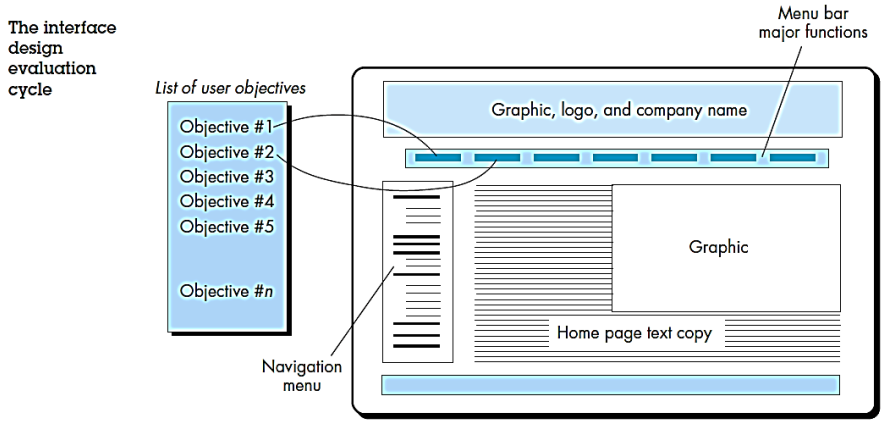
دقت کنید که اغلب مسائل چندین راهکار برای حل مساله دارند، ولی تنها یکی از آنهاست که در حیطه‌ی مساله موجود، بسیار مناسب است و می‌تواند موثر و نتیجه‌گرا واقع شود.

همه‌ی نرم‌افزارها به رفاه و کیفیت زندگی انسان کمک می‌کنند، بنابراین بهترین الگوهای طراحی، الگوهایی هستند که بر زیبایی و رفاه بیشتر برای انسان تاکید دارند.

از الگوهای طراحی می‌توان برای حل مسائل مربوط به طراحی داده، طراحی معماری، طراحی مولفه و طراحی واسط استفاده کرد.

الگوی طراحی صفحات وب

مدل مسیریابی (Navigation)، در مدل‌سازی و طراحی صفحات «**مبتنی بر وب**» کاربرد عمده دارد.



اگر این احتمال وجود دارد که کاربران در مکان‌ها و سطوح گوناگونی از سلسله مراتب وب سایت و محتوا وارد برنامه‌ی تحت وب شوند، حتما هر صفحه با ویژگی‌های گشت و گذار یا مسیریابی (**Navigation**) باید طوری طراحی شود که کاربر را به سایر نقاط مورد نظر دیگر، هدایت و حمایت کند. بهترین سفر، آن است که با چند گام انجام شود، فاصله‌ی میان کاربر و هدفش باید کوتاه باشد.

تست‌های فصل نهم

۱- تیم «الف» پیش از تحویل نرم‌افزار به کاربر نهایی در طی فرایند توسعه‌ی نرم‌افزار ۴۵۰ خطا (error) یافته است. تیم «ب» ۲۰۰ خطا یافته است. کدام‌یک از گزینه‌های زیر به ترتیب یک measure و metric مناسب برای مقایسه‌ی efficiency دو تیم در برطرف نمودن خطاها معرفی می‌نماید؟
(مهندسی IT – دولتی ۸۴)

۱) نسبت خطاها به FP (Function Point) و نسبت خطاها به مجموع خطاها و کاستی‌های (defects) نرم‌افزار
۲) نسبت کاستی‌ها (defects) به FP (Function Point) و نسبت خطاها به نفر – ماه هر تیم
۳) نسبت کاستی‌ها (defects) به KLOC و نسبت کاستی‌ها به FP نرم‌افزار
۴) نسبت خطاها به نفر – ماه هر تیم و نسبت خطاها به KLOC نرم‌افزار

۲- با فرض مشخص بودن پارامترهای زیر در یک پروژه‌ی نرم‌افزاری، تعیین نمایید چه مدت (سال) طول می‌کشد تا پروژه‌ی مفروض به انجام برسد؟ **(مهندسی IT – آزاد ۸۴)**

- تعداد خطوط کد = ۲۰۰۰۰ – میزان تلاش (نفر – سال) = ۴۴/۸۱
- فاکتور مهارت‌های ویژه = ۰/۷ – پارامتر بهره‌وری = ۵۰۰۰

۱) ۱ سال ۲) ۲ سال ۳) ۲/۵۱ سال ۴) ۱/۶۵ سال

۳- نرم‌افزار مورد استفاده برای کنترل یک دستگاه ماکروفر نیاز به ۳۲۰۰۰ خط کد زبان C++ و ۴۸۰۰ خط کد زبان اسمبلی دارد. اگر تعداد میانگین خط کد برنامه برای ساخت یک Function Point برای زبان C++ برابر LOC/FP 64 باشد و این تعداد میانگین برای زبان اسمبلی برابر LOC/FP 320 باشد، تعداد Function Point نرم‌افزار کنترل دستگاه ماکروفر چه میزان است؟ **(مهندسی IT – دولتی ۸۵)**

۱) ۳۸۴ ۲) ۵۱۵ ۳) ۲۰۴۸۰ ۴)

۴- اگر MTBF مدت زمان میانگین بین خرابی، MTTF، مدت زمان میانگین تا وقوع خرابی و MTTR مدت زمان میانگین تا رفع خرابی باشد، میزان در دسترس بودن (Availability) نرم‌افزار چگونه براساس این پارامترها محاسبه می‌گردد؟ **(مهندسی IT – دولتی ۸۵)**

۱) $\text{Availability} = [\text{MTTF} / \text{MTTR}] * ۱۰۰\%$
۲) $\text{Availability} = [\text{MTTF} / \text{MTBF}] * ۱۰۰\%$
۳) $\text{Availability} = [\text{MTTF} / \text{MTTR} + \text{MTTF}] * ۱۰۰\%$
۴) $\text{Availability} = [\text{MTTR} / \text{MTTF} + \text{MTBF}] * ۱۰۰\%$

۵- به منظور کمی‌سازی اثرات ریسک‌های پروژه، برای هر ریسک شناسایی شده، فاکتوری به نام Risk Exposure محاسبه می‌گردد. تیم مدیریت پروژه، یکی از ریسک‌های موجود در ساخت یک برنامه کاربردی را به شکل زیر تعریف کرده است:

شناسایی ریسک: تنها ۷۰٪ قطعات (Components) نرم‌افزاری که برای استفاده مجدد برنامه‌ریزی شده بودند، در برنامه کاربردی استفاده خواهند شد. باقی‌مانده کارکردهای نرم‌افزار باید کدنویسی و پیاده‌سازی شوند.
احتمال رخداد ریسک: ۸۰٪
تأثیر ریسک: می‌خواستیم از ۶۰ قطعه نرم‌افزاری (Software Component) در پروژه برای ساخت برنامه کاربردی استفاده کنیم. هر قطعه نرم‌افزاری ۱۰۰ خط کد می‌باشد و هزینه نوشتن هر خط کد برنامه ۱۴۰۰ تومان است. برای این ریسک، فاکتور Risk Exposure، کدام‌یک از گزینه‌های زیر می‌باشد؟ (مهندسی IT – دولتی ۸۵)

۱) ۲۰۱۶۰۰۰ ۲) ۵۸۸۰۰۰۰ ۳) ۲۵۲۰۰۰۰ ۴) ۴۷۰۴۰۰۰

۶- متداول‌ترین Measure برای بررسی درستی (Correctness) یک برنامه کدام‌یک از گزینه‌های زیر است؟ (مهندسی IT – دولتی ۸۵)

۱) MTTR ۲) MTTC
۳) تعداد Defect ۴) تعداد Defect در هر KLOC

۷- به عنوان مدیر یک پروژه در یک شرکت معروف تولیدکننده محصولات نرم‌افزاری مشغول به فعالیت هستید. یکی از محصولات این شرکت، یک واژه‌پرداز (Word Processor) است. وظیفه‌ی تیم شما، ساخت نگارش نسل بعدی این واژه‌پرداز است. رقابت در بازار شدید است و زمان‌بندی فشرده‌ای برای توسعه‌ی این نرم‌افزار توسط مدیریت ارشد به شما اعلام شده است. کدام ساختار تیمی و مدل فرآیند را در تولید این محصول استفاده می‌کنید؟ (مهندسی IT – دولتی ۸۵)

۱) غیرمتمرکز کنترل شده (Controlled Decentralized) – RAD
۲) متمرکز کنترل شده (Controlled Centralized) – آبشاری (Waterfall)
۳) غیرمتمرکز دموکراتیک (Democratic Decentralized) – پیچشی (Spiral)
۴) متمرکز کنترل شده (Controlled Centralized) – افزایشی (Incremental)

۸- در کدام‌یک از الگوهای سازمانی ارائه شده توسط کنستانتین (جهت تیم‌های مهندسی نرم‌افزار) بر نوآوری و کشفیات فنی تأکید نمی‌گردد؟ (مهندسی IT – آزاد ۸۵)

۱) الگوی بسته ۲) الگوی تصادفی ۳) الگوی باز ۴) الگوی هم‌زمان

۹- در کدام‌یک از روش‌های تعیین اندازه‌ی نرم‌افزار که توسط پوتنام و مایزر ارائه گردید از تکنیک‌های استدلال تقریبی استفاده می‌شود؟ (مهندسی IT – آزاد ۸۵)

۱) تعیین اندازه نقاط عملکرد ۲) تعیین اندازه تغییرات
۳) تعیین اندازه به روش منطق فازی ۴) تعیین اندازه مؤلفه‌های استاندارد

۱۰- ریسک راهبردی در کدام‌یک از گروه‌های ریسک نرم‌افزاری قرار دارد؟ (مهندسی IT – آزاد ۸۵)

۱) ریسک فنی ۲) ریسک تجاری ۳) ریسک پروژه‌ای ۴) هیچ‌کدام

۱۱- کدام‌یک از عبارت‌های زیر نادرست است؟ (مهندسی IT – دولتی ۸۶)

۱) امکان وجود دو مسیر بحرانی در نمودار pert وجود دارد.

۲) نمودار gantt به صورت میله‌ای و نمودار pert به صورت شبکه‌ای است.

۳) در نمودار pert تأکید روی تعیین مسیر بحرانی است، در حالی که در نمودار gantt می‌توان نحوه‌ی تخصیص منابع را به روشنی دید.

۴) نمودار gantt روابط پیش‌نیازی میان فعالیت‌ها را بهتر نمایش می‌دهد در حالی که در نمودار pert توازی فعالیت‌ها به روشنی دیده می‌شود.

۱۲- بهترین ساختار تیمی برای یک پروژه بسیار مشکل و با درجه‌ی سختی بالا کدام است؟ (مهندسی IT – دولتی ۸۶)

۱) Chief Programmer ۲) Controlled Centralized

۳) Controlled Decentralized ۴) Democratic Decentralized

۱۳- کدام‌یک از عبارات زیر صحیح نمی‌باشد؟ (مهندسی IT – دولتی ۸۶)

۱) نسبت document بر KLOC عاملی جهت میزان اهمیت Umbrella Task می‌باشد.

۲) نسبت Defect بر KLOC را عاملی جهت اندازه Correctness می‌دانیم.

۳) کاهش MTTC نقش اصلی را در Maintainability ایفا می‌کند.

۴) کاهش زمان آموزش در Usability مؤثر است.

۱۴- کدام‌یک از مؤلفه‌های زیر جزء ارکان اصلی مدیریت پروژه محسوب نمی‌گردد؟ (مهندسی IT – آزاد ۸۶)

۱) کیفیت ۲) افراد ۳) فرآیند ۴) محصول

۱۵- کدام عبارت صحیح است؟ (مهندسی IT – دولتی ۸۷)

۱) هنگامی که برنامه نوشته و اجرا شد دیگر کار تمام است.

۲) راه‌هایی وجود دارد تا بتوان کیفیت محصول نرم‌افزاری را قبل از اجرا نیز ارزیابی کرد.

۳) یک توضیح کلی از اهداف برای آغاز نوشتن برنامه کافی است، می‌توان جزئیات را بعداً اضافه نمود.

۴) اگر یک پروژه از برنامه زمان‌بند خود عقب بیافتد، می‌توان با افزودن برنامه‌نویسان خبره مشکل را حل کرد.

۱۶- تصور کنید اهداف مهندسی نرم‌افزار را با علائم MUN (برای برطرف کردن نیازهای کاربر)، LCP (برای هزینه‌ی پایین تولید)، HP (برای کارایی بالا)، P (برای انتقال‌پذیری بیشتر)، LCM (برای هزینه پایین نگهداری)، HR (برای قابلیت اعتماد بالا) و DOT (برای تحویل به موقع) نشان‌گذاری می‌کنیم. اولویت این اهداف برای نرم‌افزار یک سیستم عامل مطابق کدام‌یک از موارد زیر (از چپ به راست) باید اولویت‌گذاری شود؟ (مهندسی IT – دولتی ۸۷)

۱) DOT, HR, MUN, HP, P, LCP, LCM

۲) P, HR, LCM, HP, MUN, LCP, DOT

۳) MUN, HR, LCM, HP, LCP, DOT, P
۴) MUN, LCM, HR, HP, P, LCP, DOT

۱۷- کدام مورد، به عنوان معیار اندازه‌گیری و تخمین اندازه‌ی نرم‌افزار مورد استفاده قرار می‌گیرد؟ (مهندسی IT – دولتی ۸۸)

۱) تعداد ورودی (Input Number)
۲) مورد کاربرد (Use Case)
۳) خط برنامه (Line-of-Code)
۴) عملکرد (Function Point)

۱۸- کدام‌یک از فعالیت‌ها جزو فعالیت‌های واحد تضمین کیفیت در تولید نرم‌افزار موردنظر نمی‌باشد؟ (مهندسی IT – دولتی ۸۸)

۱) آنالیز و طراحی سیستم با RUP
۲) برنامه‌ریزی با MS Project
۳) تخمین هزینه با روش COCOMOII
۴) تخمین ریسک با استفاده از استراتژی RMMM

۱۹- کدام‌یک از رابطه‌ها برای ارزیابی اجرای برنامه در تولید سیستم مورد استفاده می‌باشد؟ (مهندسی IT – دولتی ۸۸)

۱) شاخص هزینه تعریف شده CPI=BCWP/ACWP
۲) شاخص اجرای برنامه کار تعریف شده SPI=BCWP/BCWS
۳) مجموعه هزینه انجام کارهای تعریف شده $BAC = \sum (BCWS_k)$
۴) هر سه مورد صحیح می‌باشد.

۲۰- «تصمیم‌گیری بر مبنای ارتباطات و توافقات» از مشخصه‌های کدام‌یک از روش‌های سازمانی موجود برای تیم‌های مهندسی نرم‌افزار است؟ (مهندسی IT – دولتی ۸۹)

۱) روش باز
۲) روش بسته
۳) روش تصادفی
۴) روش همزمان

۲۱- در یک سازمان مجری توسعه‌ی سیستم‌های اطلاعاتی، وظیفه‌ی مدیریت پروژه بر عهده‌ی شما گذاشته شده است. وظیفه‌ی شما ایجاد یک برنامه کاربردی کاملاً مشابه با برنامه‌های کاربردی دیگری است که تیم شما قبلاً ایجاد نموده است با این تفاوت که این پروژه بزرگتر و پیچیده‌تر است. خواسته‌های مشتری به طور کامل توسط وی مستندسازی شده و در اختیار شما قرار دارد. چه ساختار تیمی و کدام مدل(های) فرایند نرم‌افزار را انتخاب خواهید کرد؟ (مهندسی IT – دولتی ۸۹)

۱) ساختار تیمی غیرکنترلی و آزاد (DD) و مدل افزایشی (Incremental)
۲) ساختار تیمی کنترل شده متمرکز (CC) و مدل توسعه سریع (RAD)
۳) ساختار تیمی کنترل شده غیرمتمرکز (CD) و مدل افزایشی یا حلزونی (Spiral)
۴) ساختار تیمی کنترل شده متمرکز (CC) و مدل توسعه‌ی نرم‌افزار مبتنی بر مؤلفه (CBD)

۲۲- اگر احتمال ریسک ۶۰٪ و تأثیر ریسک برحسب هزینه برابر با ۲۵۰۰ واحد پولی باشد، میزان در معرض ریسک قرار گرفتن (Risk Exposure) برابر است با: (مهندسی IT – دولتی ۸۹)

۱) ۱۵۰۰
۲) ۱۰۰۰
۳) ۴۱۶۷
۴) هیچ‌کدام

۲۳- فرض کنید نرخ سود سالانه ۷ درصد باشد، اگر در سامانه‌ای سیصد میلیون ریال سرمایه‌گذاری شود، پس از هفت سال ارزش این سرمایه برابر با کدام‌یک از مقادیر زیر خواهد بود؟

(مهندسی IT – دولتی ۹۱)

۱) $(۳۰۰\times(۱/۰۷)^{۷})$ میلیون ریال ۲) $(۳۰۰\times۷\times۰/۰۷)$ میلیون ریال

۳) $(((۳۰۰\times۱/۰۷)\times۷)+۳۰۰)$ میلیون ریال ۴) $(۳۰۰+۰/۰۷)^{۷}$ میلیون ریال

۲۴- تعیین مسیر بحرانی، در کدام‌یک از روش‌های زیر یک فعالیت اصلی است؟ (مهندسی IT – دولتی ۹۲)

۱) روش آزمون مسیرهای پایه ۲) روش آزمون مبتنی بر سناریو

۳) روش ارزیابی و بازبینی برنامه (PERT) ۴) روش مدل‌سازی نهاد – رابطه (ER)

۲۵- دو روش زمان‌بندی پروژه، قابل اعمال در توسعه‌ی نرم‌افزار کدام است؟ (مهندسی IT – دولتی ۹۴)

۱) PERT و WBS ۲) PERT و CPM

۳) CPM و WBS ۴) CPM، WBS و PERT

۲۶- روش سازماندهی باز در مورد مدیریت تیم مهندسی نرم‌افزار مزایای کدام یک از روش‌های سازماندهی را در خود دارد؟ (مهندسی IT – دولتی ۹۵)

۱) روش تصادفی و روش هم‌زمان

۲) روش بسته و روش تصادفی

۳) روش هم‌زمان و روش کنترل شده غیرمتمرکز

۴) روش کنترل شده غیرمتمرکز و روش تصادفی

۲۷- روش محاسبه‌ی اندازه سیستم Feature Point برای استفاده در مورد چه نوع سیستم‌هایی توصیه می‌شود؟ (مهندسی IT – دولتی ۹۵)

۱) سیستم‌هایی که بزرگ هستند.

۲) سیستم‌هایی که توزیع شده هستند.

۳) سیستم‌هایی که با تکنولوژی‌های جدید تولید می‌شوند.

۴) سیستم‌هایی که پیچیدگی الگوریتمی آنها بسیار بالاست.

۲۸- کدام‌یک از جملات زیر در مورد اندازه‌ی کارآیی زدودن نقایص (Defect Removal Efficiency) درست است؟ (مهندسی IT – دولتی ۹۵)

۱) از DRE می‌توان در طی پروژه برای ارزیابی توانایی تیم در تولید هرچه سریع‌تر سیستم استفاده کرد.

۲) مقدار ایده‌آل برای DRE نزدیک صفر است که نشان می‌دهد تقریبا هیچ نقصی در نرم‌افزار پیدا نشده‌است.

۳) از DRE پس از اتمام پروژه و برای برنامه‌ریزی جهت تامین نیروی انسانی پروژه‌های بعدی می‌توان استفاده کرد.

۴) DRE اندازه‌ای است برای نشان دادن توانایی فعالیت‌های تضمین و کنترل کیفیت که در طی فرآیند به کار گرفته شده‌است.

۲۹- اگر ریسک‌های فنی اتفاق بیافتد چه خطری ممکن است بروز کند؟ (مهندسی IT – دولتی ۹۵)

۱) زمان‌بندی پروژه عقب می‌افتد و هزینه افزایش می‌یابد.
۲) پیاده‌سازی مشکل و یا غیرممکن می‌شود.
۳) محصولی تولید می‌شود که قسمت فروش نمی‌داند آنرا چگونه بفروشد.
۴) پروژه حمایت مدیران رده بالا را از دست می‌دهد.

۳۰- اگر پروژه‌ای در شرایط اضطراری باشد، مجموعه وظایف شامل چه وظایفی باید باشد؟ (مهندسی IT – دولتی ۹۵)

۱) وظایف عمومی فرآیند، وظایف حمایتی، وظایف تضمین کیفیت
۲) اضطراری بودن شرایط پروژه تاثیری در انتخاب مجموعه وظایف ندارد.
۳) وظایف عمومی فرآیند، وظایف ضروری برای تضمین کیفیت، مستندسازی و مرور پس از تحویل
۴) در صورت اضطراری بودن شرایط، مدیر پروژه می‌تواند هر وظیفه‌ای را که ضروری نمی‌داند حذف کند.

۳۱- کدام‌یک از کمیت‌های زیر نوعا در امکان‌سنجی اقتصادی (مالی) پروژه‌های ایجاد نرم‌افزار محاسبه می‌شود؟ (مهندسی IT – دولتی ۹۶)

۱) ارزش خالص فعلی
۲) تراز تجاری سازمان
۳) زمان پاسخ بازار
۴) سرمایه در گردش سازمان

۳۲- در کدام فعالیت، زودترین و دیرترین زمان‌های شروع و اتمام برای فعالیت‌های پروژه تعیین می‌شوند؟ (مهندسی IT – دولتی ۹۶)

۱) تحلیل ریسک نیازمندی‌ها
۲) تحلیل مسیر بحرانی
۳) تخصیص مسئولیت‌ها
۴) تخصیص منابع

۳۳- توصیف «یک قاعده سه- بخشی که رابطه‌ی بین یک حوزه (Context)، یک مساله (Problem) و یک راه حل (Solution) را بیان می‌کند»، مربوط به کدام مورد است؟ (مهندسی IT – دولتی ۹۷)

۱) معماری MVC
۲) داستان کاربر (User Story)
۳) الگوی طراحی (Design Pattern)
۴) اصل جایگزینی لیسکوف (Liskov Substitution Principle)

۳۴- مدل مسیریابی (Navigation)، در مدل‌سازی کدام‌یک از انواع سیستم‌ها کاربرد عمده دارد؟ (مهندسی IT – دولتی ۹۷)

۱) بحرانی
۲) کنترل خطا
۳) کنترل پروژه
۴) مبتنی بر وب

۳۵- فرض کنید MTTF و MTTR به ترتیب نشان‌دهنده متوسط زمان تا وقوع خرابی و متوسط زمان ترمیم یا رفع خرابی برای یک نرم افزار باشند. در این خصوص کدام مورد درست است؟
(مهندسی IT – دولتی ۹۸)

۱) معیار قابلیت اعتماد (Reliability) بیشتر به MTTR حساس است.
۲) معیار در دسترس بودن (Availability) بیشتر به MTTF حساس است.
۳) معیار قابلیت اعتماد (Reliability) به یک اندازه به MTTF و MTTR حساس است.
۴) معیار در دسترس بودن (Availability) به یک اندازه به MTTF و MTTR حساس است.

۳۶- در زمان‌بندی تکالیف (Tasks) در یک پروژه، تکالیفی که در مسیر بحرانی (Critical Path) قرار دارند، دارای چه خصوصیتی هستند؟ **(مهندسی IT – دولتی ۹۹)**

۱) تخصیص منابع به آنها مشکل‌تر است.
۲) تاخیر آنها باعث تاخیر کل پروژه می‌شود.
۳) زمان انجام آنها با روش‌های سنتی قابل تخمین نیست.
۴) بروز خطا در آنها باعث انتشار خطا به سایر تکالیف مسیر بحرانی می‌شود.

پاسخ تست‌های فصل نهم

۱- گزینه (۱) صحیح است.

همان معیار بازدهی رفع نقص (DRE) است.

$$DRE = \frac{E}{E + D}$$

۲- گزینه (۱) صحیح است.

$$E = \left[loc \times B^{۰/۳۳۳} / p \right]^{۳} \times \left(\frac{۱}{t^{۲}} \right) \quad \begin{matrix} E = ۴۴/۸۱ \\ B = ۰/۷ \\ P = ۵۰۰۰ \\ LOC = ۲۰۰۰ \end{matrix} \quad \rightarrow t = ۱$$

۳- گزینه (۲) صحیح است.

$$\left.\begin{matrix} \frac{LOC}{FP_{C++}} = ۶۴ \Rightarrow \frac{۳۲۰۰۰}{FP_{C++}} = ۶۴ \Rightarrow FP_{C++} = ۵۰۰ \\ \frac{LOC}{FP_{Assembly}} = ۳۲۰ \Rightarrow \frac{۴۸۰۰}{FP_{Assembly}} = ۳۲۰ \Rightarrow FP_{Assembly} = ۱۵ \end{matrix}\right\} \Rightarrow ۵۱۵$$

۴- گزینه (۲) صحیح است.

توجه کنید که MTBF=MTTR+MTTF است و لذا گزینه سوم نادرست است زیرا باید در مخرج کسر از پرانتز استفاده می‌شد.

۵- گزینه (۱) صحیح است.

از ۶۰ مؤلفه نرم‌افزاری اگر فقط ۷۰٪ آنها قابل استفاده باشند ۱۸ مؤلفه را باید مجدداً ساخت چون هر قطعه ۱۰۰ خط کد دارد ۱۸×۱۰۰ خط داریم هزینه هر خط نیز ۱۴۰۰ تومان است لذا ۱۸۰۰×۱۴۰۰ و بنابراین هزینه کل برابر است با ۲۵۲۰۰۰۰ تومان است.

طبق رابطه RE=P*C که P احتمال و C هزینه ریسک است داریم:

$$RE = ۰/۸ \times ۲۵۲۰۰۰۰ = ۲۰۱۶۰۰۰$$

۶- گزینه (۴) صحیح است.

درستی یک برنامه نرم‌افزاری برحسب تعداد نقص‌ها در هر هزار خط کد تعیین می‌شود.

۷- گزینه (۴) صحیح است.

روش‌های غیرمتمرکز معمولاً به زمان بیشتری برای به اتمام رساندن پروژه نیاز دارند لذا گزینه‌های اول و سوم نادرست هستند. با توجه به اینکه رقابت در بازار شدید است باید نسخه‌ای را سریعاً به بازار تحویل دهیم لذا گزینه‌ی چهارم صحیح است.

۸- گزینه (۱) صحیح است.

در الگوی بسته هنگام ساخت نرم‌افزاری خوب عمل می‌نماید که نیاز به کارهای روتین دارد نه کارهایی با نوآوری و کشفیات فنی.

۹- گزینه (۳) صحیح است.

سنگ بنای منطق فازی استفاده از تکنیک‌های استدلال تقریبی است.

۱۰- گزینه (۲) صحیح است.

به طور کلی پنج ریسک تجاری مهم عبارتند از:

۱- ریسک بازاری؛ ۲- ریسک راهبردی؛ ۳- ریسک فروش؛ ۴- ریسک مدیریتی؛ ۵- ریسک بودجه‌ای

۱۱- گزینه (۳) صحیح است.

نمودار گانت، نوعی نمودار میله‌ای است که شروع و پایان فعالیت‌های پروژه را نشان می‌دهد. در این نمودار، نحوه‌ی تخصیص منابع نمایش داده نمی‌شود پس گزینه‌ی ۳ درست است. در نمودار Pert توالی فعالیت‌ها و مسیر بحرانی نمایش داده می‌شود.

۱۲- گزینه (۴) صحیح است.

برای پروژه‌های بسیار بزرگ نیاز است که از مدیریت نامتمرکز برای مدیریت فعالیت‌ها استفاده کنیم.

۱۳- گزینه (۱) صحیح است.

فعالیت‌های چتری برای مدیریت بهتر فرآیند تولید نرم‌افزار است.

۱۴- گزینه (۱) صحیح است.

ارکان اصلی مدیریت پروژه عبارتند از: افراد، محصول، فرآیند و پروژه.

۱۵- گزینه (۲) صحیح است.

به کمک روش‌های Verification & Validation می‌توان کیفیت نرم‌افزار را مورد تحلیل قرار داد. مثلاً از روش‌های بازرسی (Inspection) و تحلیل ایستا استفاده نمود.

۱۶- گزینه (۳) صحیح است.

با توجه به اینکه تعیین و شناسایی نیازمندی‌ها بهترین روش در فرآیند توسعه نرم‌افزار است پس گزینه اول و دوم نادرست است. همچنین نگهداری و تکامل نیز از گام‌های مهم فرآیند توسعه می‌باشد.

۱۷- گزینه (۱) صحیح است.

در اندازه‌گیری نرم‌افزار (Software Measurement) ورودی‌های نرم‌افزار عامل مهمی در تخمین می‌باشند.

۱۸- گزینه (۱) صحیح است.

زیرا آنالیز و طراحی جزء مراحل اولیه مهندسی نرم‌افزار می‌باشد نه مراحل تضمین کیفیت. در مرحله‌ی تضمین کیفیت، زمان‌بندی و برنامه‌ریزی، تخمین هزینه و ریسک انجام می‌شود.

۱۹- گزینه (۲) صحیح است.

The budget cost of work performed = BCWP

هزینه‌ی بودجه‌بندی شده برای کار انجام شده

The budget cost of work scheduled = BCWS

مجموع هزینه‌ی برآوردی زمانبندی شده برای تحقق فعالیت‌ها

Schedule performance Index=SPI

شاخص عملکرد زمان‌بندی پروژه

SPI نشان‌دهنده‌ی عملکرد زمان‌بندی پروژه بوده و برای ارزیابی اجرای برنامه به کار می‌رود که از تقسیم ارزش کسب شده بر ارزش زمان‌بندی به دست می‌آید:

SPI=BCWP/BCWS

چنانچه این شاخص بزرگتر از یک باشد نشانگر این است که پروژه زودتر از زمان‌بندی اولیه اجرا شده است و اگر کوچکتر از یک باشد نشان می‌دهد که کار انجام شده دیرتر از برآورد اولیه اجرا شده است و اگر SPI=۱ باشد پروژه مطابق زمان‌بندی اولیه اجرا شده است.

۲۰- گزینه (۱) صحیح است.

در الگوی باز، تلاش می‌شود ساختار تیم به گونه‌ای باشد که کارها از طریق همکاری بین افراد با حداکثر ارتباطات انجام شود. ضمناً تصمیم‌گیری‌ها مبتنی بر اجماع است.

۲۱- گزینه (۴) صحیح است.

چون این برنامه مشابه برنامه انجام شده قبلی است پس جزو مسائل دشوار نمی‌باشد پس نیازی به همفکری آزاد (دموکراتیک) نمی‌باشد. همچنین با تکنیک مبتنی بر مؤلفه، از مؤلفه‌های آماده برنامه کاربردی قبلی می‌توان استفاده مجدد نمود.

۲۲- گزینه (۱) صحیح است.

میزان کلی در معرض ریسک قرار گرفتن از رابطه‌ی زیر محاسبه می‌شود:

RE=P*C

که در آن P احتمال وقوع یک ریسک است و C هزینه‌ی اضافی در صورت وقوع ریسک می‌باشد.

RE = ۶۰*۲۵۰۰/۱۰۰ = ۱۵۰۰

۲۳- گزینه (۱) صحیح است.

توجه: این سؤال مربوط به مباحث مطرح شده در درس اقتصاد مهندسی رشته مهندسی فناوری

اطلاعات در مقطع کارشناسی می‌باشد. $A = P(1+i)^t$

i = نرخ سود

t = مدت زمان به سال

P = مبلغ سرمایه‌گذاری

A = اصل و فرع سرمایه‌گذاری

فرض کنید در پایان سال اول سرمایه‌گذاری هستید، سود حاصل از سرمایه‌گذاری پس از گذشت یک سال به شرح زیر می‌باشد:

سود حاصل از سرمایه‌گذاری $۳۰۰/۰۰۰/۰۰۰ \times \%۷ = ۲۱/۰۰۰/۰۰۰$

اصل و فرع سرمایه‌گذاری در پایان سال اول $۳۰۰/۰۰۰/۰۰۰ + ۲۱/۰۰۰/۰۰۰ = ۳۲۱/۰۰۰/۰۰۰$

حال فرض کنید سرمایه‌گذاری شما تا پایان سال دوم نیز ادامه دارد. در این صورت خواهیم داشت:

$۳۲۱/۰۰۰/۰۰۰ \times \%۷ = ۲۲/۴۷۰/۰۰۰$

$۳۲۱/۰۰۰/۰۰۰ + ۲۲/۴۷۰/۰۰۰ = ۳۴۳/۴۷۰/۰۰۰$

مطابق فرمول در پایان سال دوم داریم:

$۳۰۰/۰۰۰/۰۰۰ \times (۱+\%۷)^۲ = ۳۴۳/۴۷۰/۰۰۰$

بنابراین در پایان سال هفتم داریم:

$۳۰۰/۰۰۰/۰۰۰ (۱+\%۷)^۷ = ۴۸۱/۷۳۴/۴۰۰$

۲۴- گزینه (۳) صحیح است.

گزینه اول نادرست است زیرا، تست مسیرهای پایه مرتبط با تست جعبه سفید می‌باشد.

گزینه دوم نادرست است زیرا، تست مبتنی بر سناریو، به معنی انجام تست براساس سناریوها و نیازمندی‌های مشتری است و ارتباطی به مسیر بحرانی ندارد.

گزینه سوم درست است زیرا، روش‌های ارزیابی و بازبینی برنامه (PERT) و مسیر بحرانی (CPM) دو روش زمان‌بندی برای تولید پروژه‌های نرم‌افزاری هستند. مدل ریاضی این نمودارها گراف است، که به برنامه‌ریز پروژه امکانات زیر را می‌دهد:

- نمایش توالی فعالیت‌ها و مسیرهای بحرانی پروژه
- برآورد زمان لازم برای انجام وظایف از طریق مدل‌های آماری
- محاسبه زمان‌بندی‌های مرزی میان وظایف

۲۵- گزینه (۲) صحیح است.

روش‌های ارزیابی و بازبینی برنامه (PERT) و مسیر بحرانی (CPM) دو روش زمان‌بندی برای تولید پروژه‌های نرم‌افزاری هستند. مدل ریاضی این نمودارها گراف است، که به برنامه‌ریز پروژه امکانات زیر را می‌دهد:

- نمایش توالی فعالیت‌ها و مسیرهای بحرانی پروژه
- برآورد زمان لازم برای انجام وظایف از طریق مدل‌های آماری

• محاسبه زمان‌بندی‌های مرزی میان وظایف

۲۶- گزینه (۲) صحیح است.

انتخاب مناسب نوع سازماندهی بر اساس شرایط پروژه، کارآمدی نحوه سازماندهی را به ارمغان خواهد آورد. به طور کلی دیدگاه کنستانتین برای سازمان‌دهی پروژه‌های نرم‌افزاری وجود دارد، که در ادامه به بررسی آن می‌پردازیم:

دیدگاه کنستانتین (Constantine)

در دیدگاه کنسانتین چهار ساختار تیمی مختلف برای سازماندهی تیم‌های نرم‌افزاری بیان شده است.

۱- الگوی تصادفی (Random Paradigm)

در گذشته به این الگو، **مدل** غیرمتمرکز دموکراتیک (Democratic Decentralized-DD) نیز گفته می‌شد. غیرمتمرکز است چون خرد جمعی و ارتباطات میان اعضای گروه وجود دارد، دموکراتیک است زیرا مدیر و رئیس دائمی ندارد. در این مدل، تیم ساختار ضعیفی دارد و نتایج کار به خلاقیت و توانایی تک تک اعضای تیم وابسته است. این روش زمانی کاربرد دارد که نیاز به استفاده از خلاقیت و توانایی ذهنی همه اعضای گروه و ایجاد یک خرد جمعی برای **حل یک مساله دشوار** باشد و نه **تصمیم‌گیری و توافقات**. به دلیل عدم وجود یک مدیر و به تبع عدم کنترل اعضای تیم، در صورتی که کارکردی منظم موردنیاز باشد، چنین تیمی کارآمد نخواهد بود.

در این ساختار، مدیر و رئیس ثابت و دائمی برای پروژه وجود ندارد. در عوض افرادی برای مدت کوتاه به عنوان هماهنگ‌کننده کار(Task Coordinator) و نه مدیر برای دوره‌های کوتاه زمانی منتسب می‌شوند، سپس پس از اتمام دوره جای خود را به یکی دیگر از اعضای تیم می‌دهند. ارتباط بین اعضای تیم به صورت افقی (هم‌سطح) می‌باشد یعنی نگاه بالا به پایین میان اعضای تیم وجود ندارد به عبارت دیگر اعضای تیم به یکدیگر برتری مقامی ندارند و تصمیمات در مورد مسائل و مشکلات براساس توافق و خرد جمعی اتخاذ می‌شود. فرد هماهنگ‌کننده به هیچ‌عنوان رفتار مدیر و رئیس را نخواهد داشت و صرفا وظیفه هماهنگی اعضای تیم را بر عهده دارد. در طول کار بنا به شرایط کار و تخصص اعضای گروه، یک هماهنگ کننده متناسب با کارهای پیش‌رو انتخاب می‌گردد و هماهنگی کارهای دیگری را بر عهده می‌گیرد.

این روش برای پروژه‌هایی مناسب است که یا حل مساله آن **بسیار پیچیده و دشوار** است یا نمونه‌های مشابه آن قبلاً توسط همین تیم انجام نشده است. زیرا یافتن راه حل مساله توسط همه اعضای تیم انجام می‌شود. در واقع در مواقعی که پروژه نیاز به راه حل‌های خلاقانه و نوآوری دارد، الگوی تصادفی می‌تواند بسیار کارآمد باشد، در واقع در این روش احتمال موفقیت برای نوآوری کارهای جدید بیشتر است. الگوی تصادفی برای یافتن راه حل مساله به دلیل وجود خرد جمعی مناسب و برای توافقات و تصمیم‌گیری به دلیل عدم کنترل مدیران نامناسب است.

از آنجاکه در این روش راه حل مساله توسط همه اعضای تیم بررسی می‌شود، و خرد جمعی و

ارتباطات میان اعضای تیم بسیار زیاد است، بنابراین کارکرد تیم در انجام کارها کند می‌باشد.

۲- الگوی بسته (Closed Paradigm)

در گذشته به این الگو، **مدل متمرکز کنترل‌شده** (Controlled Centralized-CC) یا **برنامه‌نویس ارشد** (Chief Programmer) نیز گفته می‌شد. برنامه‌نویس ارشد همان مدیر تیم بود. متمرکز است چون خرد جمعی و ارتباطات میان اعضای تیم بسیار ناچیز است، کنترل شده است زیرا مدیر و رئیس دائمی دارد. در این مدل، تیم در راستای یک سلسله مراتب سنتی از مسئولیت‌ها سازمان‌دهی می‌شود. برنامه‌ریزی و هماهنگ‌سازی داخلی تیم و حل مشکلات سطح بالا بر عهده‌ی مدیر تیم است. و انجام کارهای فنی بر عهده اعضای تیم است. ارتباط بین مدیر و اعضای تیم به صورت عمودی است.

به دلیل وجود یک مدیر و به تبع کنترل اعضای تیم، در صورتی که کارکردی منظم موردنیاز باشد، چنین تیمی کارآمد خواهد بود.

این روش برای پروژه‌هایی مناسب است که یا حل مساله آن ساده است یا نمونه‌های مشابه آن قبلاً توسط همین تیم انجام شده است. زیرا یافتن راه حل مساله توسط تنها یک نفر(مدیر تیم) و بدون کمک دیگر اعضای تیم انجام می‌شود. در مقابل در مواقعی که پروژه نیاز به راه حل‌های خلاقانه و نوآوری دارد، الگوی بسته نمی‌تواند چندان کارآمد باشد، در واقع در این روش احتمال موفقیت برای نوآوری کارهای جدید کمتر است. الگوی بسته برای یافتن راه حل مساله به دلیل عدم وجود خرد جمعی نامناسب و برای توافقات و تصمیم‌گیری به دلیل کنترل مدیران مناسب است.

از آنجاکه در این روش راه حل مساله توسط تنها یک نفر و به شکل متمرکز کنترل‌شده بررسی می‌شود، و خرد جمعی و ارتباطات میان اعضای تیم بسیار ناچیز است، بنابراین کارکرد تیم در انجام کارها سریع می‌باشد.

۳- الگوی باز (Open Paradigm)

در گذشته به این الگو، **مدل** غیرمتمرکز کنترل شده (Controlled Decentralized-CD) نیز گفته می‌شد. غیرمتمرکز است چون خرد جمعی و ارتباطات میان گروه‌های داخل تیم وجود دارد، کنترل شده است زیرا مدیر اصلی و مدیران میانی دارد. الگوی باز، مزایای الگوی بسته و تصادفی را با هم ترکیب می‌کند. در این مدل، تیم در راستای یک سلسله مراتب سنتی از مسئولیت‌ها به تعدادی گروه سازمان‌دهی می‌شود. پروژه دارای یک مدیر اصلی است که هماهنگی وظایف و هماهنگ‌سازی گروه‌ها را بر عهده دارد. همچنین، مدیریت وظایف سطوح پایین‌تر بر عهده‌ی مدیران میانی پروژه است که برنامه‌ریزی و حل مشکلات سطح بالا را در گروه‌ها بر عهده دارند. در سطح آخر سلسله مراتب نیز انجام کارهای فنی بر عهده اعضای گروه است. ارتباط بین مدیر اصلی، مدیران میانی و اعضای تیم به صورت عمودی و ارتباط میان اعضای گروه‌ها به صورت افقی است. حل مسأله و تصمیم‌گیری یک کار تیمی است اما وظیفه پیاده‌سازی راه‌حل‌ها، بین زیرگروه‌ها توسط مدیر اصلی تقسیم و توزیع می‌شود.

به دلیل وجود مدیر اصلی و مدیران میانی و به تبع کنترل اعضای تیم، در صورتی که کارکردی منظم موردنیاز باشد، چنین تیمی کارآمد خواهد بود.

این روش برای پروژه‌هایی مناسب است که یا حل مساله آن پیچیده و دشوار است یا نمونه‌های مشابه آن قبلاً توسط همین تیم انجام نشده است. زیرا یافتن راه حل مساله توسط همه اعضای تیم انجام می‌شود. در واقع در مواقعی که پروژه نیاز به راه حل‌های خلاقانه و نوآوری دارد، الگوی باز می‌تواند بسیار کارآمد باشد، در واقع در این روش احتمال موفقیت برای نوآوری کارهای جدید بیشتر است. الگوی باز برای یافتن راه حل مساله به دلیل وجود خرد جمعی مناسب و برای توافقات و تصمیم‌گیری نیز به دلیل کنترل مدیران مناسب است.

از آنجاکه در این روش راه حل مساله توسط همه اعضای تیم بررسی می‌شود، و خرد جمعی و ارتباطات میان اعضای تیم زیاد است، بنابراین کارکرد تیم در انجام کارها کند می‌باشد.

۴- الگوی همزمان (Synchronous Paradigm)

این الگو، بر قطعه قطعه کردن مساله به شیوه‌ای طبیعی و سازمان‌دهی اعضای تیم برای کار روی هر یک از این قطعات تاکید دارد، به گونه‌ای که میان اعضای تیم ارتباط چندانی برقرار نیست. به بیان دیگر در مواقعی‌که برنامه قابلیت پیمانه‌پذیری دارد، الگوی همزمان می‌تواند مورد استفاده قرار گیرد.

صورت سوال به این شکل است:

روش سازماندهی باز در مورد مدیریت تیم مهندسی نرم‌افزار مزایای کدام یک از روش‌های سازماندهی را در خود دارد؟

۱) روش تصادفی و روش هم‌زمان

گزینه اول پاسخ سوال نیست، زیرا الگوی باز، مزایای الگوی **بسته** و **تصادفی** را با هم ترکیب می‌کند و در خود دارد.

۲) روش بسته و روش تصادفی

گزینه دوم پاسخ سوال است، زیرا الگوی باز، مزایای الگوی **بسته** و **تصادفی** را با هم ترکیب می‌کند و در خود دارد.

۳) روش هم‌زمان و روش کنترل شده غیرمتمرکز

گزینه سوم پاسخ سوال نیست، زیرا الگوی باز، مزایای الگوی **بسته** و **تصادفی** را با هم ترکیب می‌کند و در خود دارد. دقت کنید که روش کنترل شده غیرمتمرکز همان الگوی باز است.

۴) روش کنترل شده غیرمتمرکز و روش تصادفی

گزینه چهارم پاسخ سوال نیست، زیرا الگوی باز، مزایای الگوی بسته و تصادفی را با هم ترکیب می‌کند و در خود دارد. دقت کنید که روش کنترل شده غیرمتمرکز همان الگوی باز است.

۲۷- گزینه (۴) صحیح است.

مقدار نقطه عملکرد یا امتیاز عملکرد (Function Point) برای محاسبه‌ی درجه‌ی پیچیدگی

سیستم‌های تجاری بی‌درنگ و مدیریت بانک اطلاعات مورد استفاده قرار می‌گیرد، در نرم‌افزارهای مدیریت بانک اطلاعات حجم داده‌ها بالا و حجم محاسبات پایین است. در محاسبه‌ی مقدار **امتیاز عملکرد** دامنه‌ی اطلاعاتی پروژه شامل ورودی‌های برنامه، خروجی‌های برنامه، پرس و جوهای برنامه، فایل‌های منطقی داخلی برنامه و فایل‌های واسط خارجی برنامه مورد شمارش قرار می‌گیرند. اما مقدار **امتیاز ویژگی** برای محاسبه‌ی درجه‌ی پیچیدگی سیستم‌های علمی و مهندسی مورد استفاده قرار می‌گیرد، نرم‌افزارهای علمی و مهندسی برای کاربردهایی با محاسبات پیچیده و سنگین مورد استفاده قرار می‌گیرند، مانند نرم‌افزارهای محاسبات ریاضی (مثل ضرب ماتریس‌ها و معکوس‌سازی ماتریس‌ها)، رمزگشایی یک متن، علوم زمین‌شناسی، ستاره‌شناسی و کنترل سیستم‌های صنعتی. در این نوع نرم‌افزارها حجم داده‌ها پایین و حجم محاسبات بالا است. در این نوع نرم‌افزاها برای محاسبه‌ی مقدار امتیاز ویژگی علاوه بر دامنه‌ی اطلاعاتی پروژه، **تعداد الگوریتم‌های برنامه** نیز مورد شمارش قرار می‌گیرند. نتیجه اینکه روش **امتیاز ویژگی** (Feature Point) جهت اندازه‌گیری سیستم‌های علمی و مهندسی که پیچیدگی الگوریتمی آنها بسیار بالاست مورد استفاده قرار می‌گیرد.

صورت سوال به این شکل است:

روش محاسبه‌ی اندازه سیستم Feature Point برای استفاده در مورد چه نوع سیستم‌هایی توصیه می‌شود؟

۱) سیستم‌هایی که بزرگ هستند.

گزینه اول پاسخ سوال نیست، زیرا روش **امتیاز ویژگی** (Feature Point) جهت اندازه‌گیری سیستم‌های علمی و مهندسی که پیچیدگی الگوریتمی آنها بسیار بالاست مورد استفاده قرار می‌گیرد.

۲) سیستم‌هایی که توزیع شده هستند.

گزینه دوم پاسخ سوال نیست، زیرا روش **امتیاز ویژگی** (Feature Point) جهت اندازه‌گیری سیستم‌های علمی و مهندسی که پیچیدگی الگوریتمی آنها بسیار بالاست مورد استفاده قرار می‌گیرد.

۳) سیستم‌هایی که با تکنولوژی‌های جدید تولید می‌شوند.

گزینه سوم پاسخ سوال نیست، زیرا روش **امتیاز ویژگی** (Feature Point) جهت اندازه‌گیری سیستم‌های علمی و مهندسی که پیچیدگی الگوریتمی آنها بسیار بالاست مورد استفاده قرار می‌گیرد.

۴) سیستم‌هایی که پیچیدگی الگوریتمی آنها بسیار بالاست.

گزینه چهارم پاسخ سوال است، زیرا روش **امتیاز ویژگی** (Feature Point) جهت اندازه‌گیری سیستم‌های علمی و مهندسی که پیچیدگی الگوریتمی آنها بسیار بالاست مورد استفاده قرار می‌گیرد.

۲۸- گزینه (۴) صحیح است.

کارآیی رفع نقص یا DRE که سرواژه‌ی عبارت Defect Removal Efficiency می‌باشد، **اندازه‌ای** برای کنترل فعالیت‌های مربوط به تضمین کیفیت در روند فرآیند تولید نرم‌افزار است. به بیان دیگر DRE **اندازه‌ای** است برای نشان دادن توانایی فعالیت‌های تضمین و کنترل کیفیت که در طی فرآیند به کار گرفته شده‌است. اگر این **اندازه** محاسبه گردد، آنگاه حاصل این اندازه در مقام **مقایسه** با گذشته یک فعالیت از فرآیند تولید نرم‌افزار یا کل فعالیت‌های پروژه یک **معیار (metric) اندازه‌گرا** خواهد بود. این معیار جهت مقایسه، بررسی و ارزیابی یک فعالیت از فرآیند تولید نرم‌افزار یا کل فعالیت‌های پروژه با گذشته همان پروژه یا پروژه **دیگر** مورد استفاده قرار می‌گیرد. این مقایسه **گذشته** منجر به پندگیری و به تبع بهبود فرآیند تولید نرم‌افزار در **آینده** می‌گردد. بنابراین DRE یک معیار کیفیتی است که هم در سطح پروژه و هم در سطح بهبود فرآیند تولید نرم‌افزار مفید واقع می‌شود.

صورت سوال به این شکل است:

کدام‌یک از جملات زیر در مورد اندازه‌ی کارآیی زدودن نقایص (Defect Removal Efficiency) درست است؟

۱) از DRE می‌توان در طی پروژه برای ارزیابی توانایی تیم در تولید هرچه سریع‌تر سیستم استفاده کرد.

گزینه اول پاسخ سوال نیست، زیرا ارزیابی توانایی تیم در تولید هرچه سریع‌تر سیستم توسط مقایسه با زمان‌بندی پروژه امکان‌پذیر است.

۲) مقدار ایده‌آل برای DRE نزدیک صفر است که نشان می‌دهد تقریبا هیچ نقصی در نرم‌افزار پیدا نشده‌است.

گزینه دوم پاسخ سوال نیست، زیرا مقدار ایده‌آل برای DRE برابر یک است.

۳) از DRE پس از اتمام پروژه و برای برنامه‌ریزی جهت تامین نیروی انسانی پروژه‌های بعدی می‌توان استفاده کرد.

گزینه سوم پاسخ سوال نیست، زیرا از اندازه‌گیری و برآورد پروژه‌های پیشین می‌توان برای برنامه‌ریزی جهت تامین نیروی انسانی پروژه‌های بعدی استفاده کرد.

۴) اندازه‌ای است برای نشان دادن توانایی فعالیت‌های تضمین و کنترل کیفیت که در طی فرآیند به کار گرفته شده‌است.

گزینه چهارم پاسخ سوال است.

۲۹- گزینه (۲) صحیح است.

صورت سوال به این شکل است:

اگر ریسک‌های فنی اتفاق بیافتد چه خطری ممکن است بروز کند؟

۱) زمان‌بندی پروژه عقب می‌افتد و هزینه افزایش می‌یابد.

گزینه اول پاسخ سوال نیست، زیرا اگر ریسک‌های پروژه‌ای اتفاق بیافتد، آنگاه زمان‌بندی پروژه عقب می‌افتد و هزینه افزایش می‌یابد.

۲) پیاده‌سازی مشکل و یا غیرممکن می‌شود.

گزینه دوم پاسخ سوال است، زیرا اگر ریسک‌های فنی اتفاق بیافتد، آنگاه پیاده‌سازی مشکل و یا غیرممکن می‌شود.

۳) محصولی تولید می‌شود که قسمت فروش نمی‌داند آنرا چگونه بفروشد.

گزینه سوم پاسخ سوال نیست، زیرا اگر ریسک فروش اتفاق بیافتد، آنگاه محصولی تولید می‌شود که قسمت فروش نمی‌داند آنرا چگونه بفروشد.

۴) پروژه حمایت مدیران رده بالا را از دست می‌دهد.

گزینه چهارم پاسخ سوال نیست. زیرا اگر ریسک مدیریتی اتفاق بیافتد، آنگاه پروژه حمایت مدیران رده بالا را از دست می‌دهد.

۳۰- گزینه (۳) صحیح است.

صورت سوال به این شکل است:

اگر پروژه‌ای در شرایط اضطراری باشد، مجموعه وظایف شامل چه وظایفی باید باشد؟

۱) وظایف عمومی فرآیند، وظایف حمایتی، وظایف تضمین کیفیت

گزینه اول پاسخ سوال نیست، زیرا در شرایط اضطراری پروژه، نیاز به عکس العمل سریع است، در این شرایط فعالیت‌های چارچوبی(عمومی) به طور کامل انجام می‌شود، اما جهت حفظ کیفیت پروژه فقط وظایف ضروری از فعالیت‌های چتری(حمایتی) همچون تضمین کیفیت نرم‌افزار و مستندسازی انجام می‌شود و نه همه وظایف حمایتی.

۲) اضطراری بودن شرایط پروژه تاثیری در انتخاب مجموعه وظایف ندارد.

گزینه دوم پاسخ سوال نیست، زیرا اضطراری بودن شرایط پروژه تاثیری در انتخاب مجموعه وظایف دارد.

۳) وظایف عمومی فرآیند، وظایف ضروری برای تضمین کیفیت، مستندسازی و مرور پس از تحویل

گزینه سوم پاسخ سوال است، زیرا در شرایط اضطراری پروژه، نیاز به عکس العمل سریع است، در این شرایط فعالیت‌های چارچوبی(عمومی) به طور کامل انجام می‌شود، اما جهت حفظ کیفیت پروژه فقط وظایف ضروری از فعالیت‌های چتری همچون تضمین کیفیت نرم‌افزار و مستندسازی انجام می‌شود. همچنین پس از تحویل پروژه مرور و بازبینی جهت نگهداری نرم‌افزار انجام می‌شود.

۴) در صورت اضطراری بودن شرایط، مدیر پروژه می‌تواند هر وظیفه‌ای را که ضروری نمی‌داند حذف کند.

گزینه چهارم پاسخ سوال نیست. زیرا در صورت اضطراری بودن شرایط، مدیر پروژه نمی‌تواند

هر وظیفه‌ای را که ضروری نمی‌داند حذف کند.

۳۱- گزینه (۱) صحیح است.

هنگامی‌که دامنه‌ی نرم‌افزار مشخص شد، این سوال مطرح می‌شود که آیا می‌توان نرم‌افزاری ایجاد کرد که دامنه‌ی نرم‌افزار را برآورده سازد؟ آیا اجرای پروژه امکان‌پذیر است؟

این بخش از فرآیند برنامه‌ریزی پروژه، بخش مهمی است که البته اغلب اوقات مورد بی‌توجهی قرار می‌گیرد. بنابراین فقط تعیین دامنه‌ی نرم‌افزار کافی نیست، بلکه باید امکان‌سنجی نیز انجام گردد تا پروژه به سمت سرنوشتی نامشخص سوق پیدا نکند. معیارهایی که برای بررسی امکان‌پذیری انجام پروژه در نظر گرفته می‌شود با چهار بخش زیر مرتبط است:

۱- **فناوری** (Technology): هدف از بررسی امکان‌پذیری فناوری، بررسی این موضوع است که آیا فناوری و فن لازم و کافی برای انجام پروژه وجود دارد یا خیر.

۲- **مالی** (Finance): هدف از بررسی امکان‌پذیری مالی، بررسی این موضوع است که آیا پول لازم و کافی برای انجام پروژه وجود دارد یا خیر. به عبارت دیگر، هدف از این بررسی، تعیین هزینه‌های مالی مربوط به پروژه‌ی نرم‌افزاری که با آن مواجه هستیم، می‌باشد. پروژه‌ای که انجام می‌شود باید از توجیه اقتصادی خوبی برای سازنده و مشتری برخوردار باشد، در غیر اینصورت انجام آن مقرون به صرفه نخواهد بود. در مورد نرم‌افزارهای تجاری نیز باید بررسی شود که آیا بازار توان خرید نرم‌افزار را دارد یا خیر.

۳- **زمان** (Time): هدف از بررسی امکان‌پذیری زمانی، بررسی این موضوع است که آیا زمان لازم و کافی برای انجام پروژه وجود دارد یا خیر.

۴- **منابع** (Resources): هدف از بررسی امکان‌پذیری منابع، بررسی این موضوع است که آیا منابع لازم و کافی برای انجام پروژه وجود دارد یا خیر.

صورت سوال به این شکل است:

کدام‌یک از کمیت‌های زیر نوعا در امکان‌سنجی اقتصادی (مالی) پروژه‌های ایجاد نرم‌افزار محاسبه می‌شود؟

۱) ارزش خالص فعلی

گزینه اول پاسخ سوال است، زیرا وقتی مدیری باید پروژه‌های مختلف را با یکدیگر مقایسه کند و تصمیم بگیرد کدام پروژه را انجام دهد، از روش ارزش خالص فعلی (NPV: Net present value) استفاده می‌کند. ارزش خالص فعلی که اغلب به آن NPV گفته می‌شود، ابزار تحلیلگران مالی است. در محاسبه‌ی ارزش خالص فعلی، ارزش زمانی پول در نظر گرفته می‌شود و جریان‌های نقدی آینده براساس ارزش پول امروز بیان می‌شود. همچنین، با محاسبه‌ی ارزش خالص فعلی، رقم دقیقی به دست می‌آید که مدیران می‌توانند با استفاده از آن، مبلغ سرمایه‌گذاری‌شده‌ی نقدی اولیه را به‌آسانی با ارزش فعلی بازگشت سرمایه مقایسه کنند. هروقت شرکتی بخواهد از ارزش پول امروز برای بیان بازدهی آینده استفاده کند، ارزش فعلی

خالص، روش مناسب و مطمئنی خواهد بود. اگر ارزش خالص فعلی منفی باشد، پروژه‌ی موردنظر پروژه‌ی خوبی نخواهد بود و درنهایت موجب ازدست‌رفتن پول‌های نقد در آن کسب و کار خواهد شد. ولی اگر مثبت باشد، باید پروژه را انجام دهید. هرچه عدد مثبت به‌دست‌آمده بزرگ‌تر باشد، سودی که عاید شرکت می‌شود نیز بیشتر خواهد بود. نتیجه اینکه ارزش خالص فعلی در امکان‌سنجی اقتصادی (مالی) پروژه‌های ایجاد نرم‌افزار مورد استفاده قرار می‌گیرد.

ارزش خالص فعلی تفاضل بین ارزش فعلی جریانات نقدی ورودی و ارزش فعلی جریان نقدی خروجی است. ارزش خالص فعلی، در علم اقتصاد مهندسی، یکی از روش‌های استاندارد ارزیابی طرح‌های اقتصادی است. در این روش، جریان نقدینگی (درآمدها و هزینه‌ها) بر پایه زمان وقوع (درآمد یا هزینه) به نرخ روز تنزیل می‌شود. به این ترتیب در جریان نقدینگی، ارزش زمان انجام هزینه و به دست‌آمدن درآمد نیز لحاظ می‌گردد. ارزش خالص فعلی در محاسبات اقتصادی، اقتصاد مهندسی، بودجه کشورها و مباحث اقتصاد خرد واقتصاد کلان، تجارت و صنعت به‌طور گسترده‌ای به‌کار می‌رود.

در روش ارزش خالص فعلی، ابتدا تمامی درآمدها و هزینه‌ها بسته به اینکه در چه زمانی به وقوع خواهند پیوست، با نرخ تورم مناسبی طبق رابطه‌ی زیر تنزیل می‌شوند:

$$\frac{R_t}{(1+i)^t}$$

در این رابطه t زمان انجام هزینه یا واقع شدن درآمد، i نرخ تورم و R_t مقدار کمی درآمد و هزینه بر اساس جریان نقدینگی است. سپس با تفاضل هزینه‌های تبدیل شده از درآمدهای تبدیل شده، عدد خالصی به‌دست خواهد آمد که به آن NPV گفته می‌شود. اگر این عدد مثبت باشد، طرح سودآور و قابل قبول بوده و اگر منفی باشد، طرح زیان‌ده و غیر قابل اجرا (از نظر اقتصادی) است و این یعنی امکان‌سنجی.

یکی از اصلی‌ترین کاربردهای ارزش خالص فعلی، مطالعات اقتصاد مهندسی و ارزیابی توجیه فنی و اقتصادی پروژه‌ها است. به عنوان مثال اگر یک کارخانه برای ایجاد خط تولید محصول جدیدی پیش‌بینی کند که در سال اول نیاز به ۱۰۰ میلیون تومان هزینه سرمایه (جهت راه‌اندازی خط تولید) داشته باشد و خط تولید تا پایان سال به بهره‌برداری برسد و طی ۶ سال بعدی، از محل فروش محصول تولیدی، درآمدی برابر سالانه ۳۰ میلیون تومان ایجاد شود و همچنین هزینه‌های جاری تولید و عرضه آن محصول (مانند مواد اولیه، آب و برق و تلفن، دستمزد، حمل و نقل، بازاریابی، ...) سالانه ۵ میلیون تومان باشد و نرخ تورم سالانه نیز ۱۰٪ منظور شود، نحوه محاسبه ارزش خالص فعلی به صورت زیر خواهد بود:

سال	جریان نقدینگی	ارزش فعلی	ارزش فعلی تجمعی
سال جاری t =۰	$\frac{-۱۰۰۰۰۰}{(۱+۰/۱۰)^{۰}}$	۱۰۰- میلیون تومان	۱۰۰- میلیون تومان
سال اول t =۱	$\frac{۳۰۰۰۰-۵۰۰۰}{(۱+۰/۱۰)^{۱}}$	۲۳ میلیون تومان	۷۷- میلیون تومان
سال دوم t =۲	$\frac{۳۰۰۰۰-۵۰۰۰}{(۱+۰/۱۰)^{۲}}$	۲۱ میلیون تومان	۵۶- میلیون تومان
سال سوم t =۳	$\frac{۳۰۰۰۰-۵۰۰۰}{(۱+۰/۱۰)^{۳}}$	۱۹ میلیون تومان	۳۷- میلیون تومان
سال چهارم t =۴	$\frac{۳۰۰۰۰-۵۰۰۰}{(۱+۰/۱۰)^{۴}}$	۱۷ میلیون تومان	۲۰- میلیون تومان
سال پنجم t =۵	$\frac{۳۰۰۰۰-۵۰۰۰}{(۱+۰/۱۰)^{۵}}$	۱۵ میلیون تومان	۵- میلیون تومان
سال ششم t =۶	$\frac{۳۰۰۰۰-۵۰۰۰}{(۱+۰/۱۰)^{۶}}$	۱۴میلیون تومان	۹+ میلیون تومان

۲) تراز تجاری سازمان

گزینه دوم پاسخ سوال نیست، زیرا تراز تجاری سازمان، تفاوت میان ارزش پولی واردات و صادرات خروجی در طی یک دوره معین در اقتصاد سازمانی است. اگر صادرات بیشتر از واردات باشد مقدار مثبت به نام مازاد تجاری مشخص می‌شود. اگر واردات بیشتر از صادرات باشد مقدار منفی به نام کسری تجاری یا شکاف تجاری مشخص می‌شود.

۳) زمان پاسخ بازار

گزینه سوم پاسخ سوال نیست، زیرا زمان پاسخ بازار مربوط به زمان مناسب ارائه محصول به بازار است. به عبارت دیگر زمان پاسخ بازار مربوط به بهترین زمان برای معرفی یک محصول جدید به بازار است. اگر در حال کار بر روی یک محصول یا سرویس جدید هستید و قصد دارید آن را در روزها یا ماه‌های آینده به بازار معرفی کنید، بدانید که اجرای درست این معرفی که خود می تواند یک پروژه‌ی جدید قلمداد گردد، تاثیر مهمی در موفقیت کسب و کار شما خواهد داشت. هر روز در سراسر دنیا شرکت های نوپا، کسب و کارهای کوچک، متوسط و بزرگ، محصول یا سرویس جدید خود را به بازار معرفی می کنند. تعدادی از آن ها آنقدر سر و صدا به پا می کند که

اغلب ما از آن خبردار می شویم، برای مثال وقتی یک مدل جدید آیفون به بازار معرفی می شود. یا وقتی گوگل یا فیسبوک امکانات جدیدی را به سرویس هایشان اضافه می کنند. در دنیای کسب و کارهای اینترنتی معرفی یک محصول جدید به بازار یا Launch که به معنی روانه‌کردن است، اگر بدون برنامه ریزی و دانش لازم انجام شود، کاری دشوار و دلهره‌آور برای ارائه کننده آن محصول خواهد بود. صنعت وب روز به روز در حال تغییر و پذیرای محصولات جدید است. رقابت آنقدر فشرده است که کارآفرینان مجبور هستند روش‌ها و ایده‌های جدید و خلاقانه‌ای را برای معرفی محصولاتشان و رساندن صدای خود به گوش مخاطبان هدف ابداع کنند. تکامل، در واقع یک نردبان است نه یک مسیر بسته و باید پله‌های آن را یکی یکی بالا رفت. پس به محض اینکه احساس کردید که محصولتان امکانات و ویژگی‌های اساسی را دارد و از کیفیت قابل قبول و مناسبی برخوردار است، آن را به بازار عرضه کنید. هر چه زودتر محصول را به دست مشتریان هدف برسانید، امکان گرفتن سریعتر نظرات مشتریان را دارید. در نتیجه می‌توانید تغییرات مورد نیاز را زودتر و با هزینه‌ی کمتر در محصول یا خدمات خود اعمال کنید. در آینده وقت برای اضافه کردن امکانات و ویژگی‌های فرعی هست و سر فرصت می‌توانید کارهای باقیمانده را انجام دهید. با توجه به اینکه پس از لانچ تعداد زیادی نظر، پیشنهاد و انتقاد دریافت می‌کنید، پس انجام کارهای باقیمانده بر اساس این نظرات و توام با بهینه‌سازی وضعیت موجود رخ خواهد داد که این، موجب انجام بهتر کارها و در نتیجه بالا رفتن کیفیت نسخه بعدی محصول خواهد شد. بسیاری از محصولات موفق ایرانی و خارجی در هنگام لانچ از این تاکتیک استفاده کرده‌اند. برای مثال وقتی میل‌چیمپ (MailChimp) سرویس بازاریابی ایمیلی و مدیریت لیست‌های ایمیل خود را عرضه کرد بسیاری از امکاناتی که امروز دارد مانند قالب‌های ایمیل و RSS-to-Email را نداشت. و یا یک مثال مشهورتر، اینکه در اولین نسخه سیستم عامل گوشی‌های همراه آیفون (iPhone) اگرچه امکان جستجو در اینترنت وجود داشت اما امکان جستجوی لیست شماره‌های مخاطبان (Contacts) وجود نداشت. همچنین امکان copy-paste دو سال پس از ارائه اولین مدل آیفون در نسخه‌های بعدی سیستم عامل آن ارائه شد!

پس عقب انداختن انجام کارهایی که در حال حاضر اهمیت و ارزش خاصی برای مشتری ندارند به شما این امکان را می‌دهد که بر روی مسائل و نیازهای مهمتر تمرکز و کار کنید. نباید گذاشت کیفیت قربانی سرعت در لانچ شود. اینکه یک محصول در هنگام لانچ کمترین تعداد امکانات را داشته باشد اشکالی ندارد، اما اگر امکانات و ویژگی‌هاییش بی‌کیفیت باشد، به اعتبار سازندگانش به شدت لطمه خواهد زد و موفقیت آن را به شدت تحت تاثیر منفی قرار خواهد داد.

۴) سرمایه در گردش سازمان

گزینه چهارم پاسخ سوال نیست. زیرا سرمایه در گردش، مجموعه مبالغی است، که در دارایی‌های جاری یک شرکت، سرمایه‌گذاری می‌شود. اگر بدهی‌های جاری از دارایی‌های جاری یک شرکت کسر گردد، سرمایه در گردش خالص به دست می‌آید. مدیریت سرمایه در گردش عبارت است، از تعیین حجم و ترکیب منابع و مصارف سرمایه در گردش، به نحوی که

ثروت سهام‌داران افزایش یابد. دارایی‌های جاری دارایی‌هایی هستند که به صورت معقول و منطقی (نه خوش‌بینانه و نه بدبینانه) در طی سال مالی جاری یا دوره مالی مورد نظر ما، قابل تبدیل به پول نقد باشند. پس پول قطعاً جزو دارایی‌های جاری است .سرمایه گذاری‌های کوتاه مدتی که به سادگی قابل فروش و تبدیل به نقدینگی هستند هم جزو دارایی‌های جاری محسوب می‌شوند .موجودی انبار هم در صورتی که در افق زمانی کوتاه مدت قابل فروش و تبدیل به پول نقد باشد در گروه دارایی‌های جاری طبقه بندی می‌شود .حسابهای دریافتنی (مطالباتی که در کوتاه مدت بتوان آنها را به پول نقد تبدیل کرد و حتی ممکن است اسناد تجاری دقیق از آنها موجود نباشد) هم شکل دیگری از دارایی‌های جاری محسوب می‌شوند. به دارایی‌های جاری، سرمایه در گردش هم گفته می‌شود. اما به خاطر داشته باشید که یک شرکت علاوه بر دارایی‌های جاری، بدهی‌های جاری هم دارد. به همان شیوه فوق می‌توان بدهی‌های جاری را هم تعریف کرد. بدهی‌هایی که به صورت معقول و منطقی (نه خوش‌بینانه و نه بدبینانه) در طی سال مالی جاری یا دوره مالی مورد نظر ما، قابل تبدیل به پول نقد باشند. به همین دلیل بهتر است به سرمایه در گردش خالص توجه داشته باشیم. یعنی: اختلاف بین دارایی‌های جاری و بدهی‌های جاری. اگر بخواهیم به زبان ساده‌تر بگوییم، سرمایه در گردش خالص، میزان واقعی دارایی‌های نقد یا نزدیک به نقد یک سازمان است. چیزی که هزینه‌ها و نیازهای روزمره یک شرکت را تامین می‌کند. آیا تا به حال با کسی مواجه شده‌اید که خانه گرانقیمتی داشته باشد اما به علت ورشکستگی یا هر مسئله دیگری از پرداخت بدیهی‌های روزمره خود ناتوان باشد؟ همیشه به خاطر داشته باشیم که حتی ممکن است با منفی بودن سرمایه در گردش، همچنان یک کسب و کار تا مدتی به کار خود ادامه دهد. اما چنین وضعیتی در بلندمدت پایدار نخواهد بود. گاهی مدیران در گفتگوهای روزمره خود سرمایه در گردش و سرمایه در گردش خالص را به جای یکدیگر به کار می‌برند. بنابراین بهتر است در جلسه‌ها نسبت به مفهومی که واقعاً در ذهن طرف مقابل است حساس باشیم.

۳۲- گزینه (۲) صحیح است.

صورت سوال به این شکل است:

در کدام فعالیت، زودترین و دیرترین زمان‌های شروع و اتمام برای فعالیت‌های پروژه تعیین می‌شوند؟

۱) تحلیل ریسک نیازمندی‌ها

گزینه اول پاسخ سوال نیست، زیرا فعالیت ارتباطات یا مهندسی نیازمندی‌ها منجر به تهیه لیست نیازمندی‌های مشتری و به تبع تحلیل ریسک نیازمندی‌ها می‌گردد. ابزارهای تشخیص برای تهیه لیست نیازمندی‌های مشتری به پنج شکل «**گفتگو**»، «**مشاهده**»، «**پرسش‌نامه**»، «**مکانیزم نمونه‌سازی دورانداختنی**» و «**مکانیزم نمونه‌سازی تکاملی**» می‌باشد. فعالیت ارتباطات از طریق ارتباط با مشتری توسط ارتباط‌گر و ابزارهای مطرح شده انجام می‌گردد.

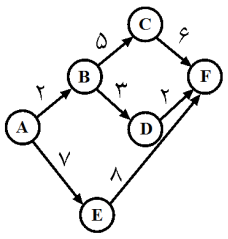
۲) تحلیل مسیر بحرانی

گزینه دوم پاسخ سوال است، زیرا در تحلیل مسیر بحرانی، زودترین و دیرترین زمان‌های

شروع و اتمام برای فعالیت‌های پروژه تعیین می‌شوند. اینکه چه فعالیتی‌هایی از فعالیت‌های پروژه زودتر از همه شروع و اتمام شده‌اند و چه فعالیتی‌هایی از فعالیت‌های پروژه دیرتر از همه شروع و اتمام شده‌اند. این اطلاعات بر روی نمودارهای CPM و PERT مربوط به تحلیل مسیر بحرانی قابل مشاهده است. همچنین زودترین زمان شروع و اتمام برای یک فعالیت زمانیست که فعالیت‌های قبلی و جاری آن فعالیت زودتر از زمان تعیین شده تمام شده باشند. و دیرترین زمان شروع و اتمام برای یک فعالیت زمانیست که زمان‌بندی پروژه به تعویق نیافتد.

در زمان‌بندی پروژه، باید مشخص شود که هر وظیفه در چه بازه‌ی زمانی باید شروع شود و چه مدت زمان نیاز دارد تا تکمیل شود. تکنیک‌های ارزیابی و بازبینی برنامه[1] (PERT) و روش مسیر بحرانی[2] (CPM) جهت زمان‌بندی پروژه مورد استفاده قرار می‌گیرند. هر دو روش PERT و CPM توسط نمودارهای شبکه‌ای به برنامه‌ریز پروژه امکانات زیر را می‌دهند:

۱- نمایش شبکه وظایف و مسیرهای بحرانی پروژه
۲- برآورد زمان لازم برای انجام وظایف از طریق مدل‌های آماری
۳- محاسبه زمان‌بندی‌های مرزی میان وظایف



یکی از مهم‌ترین مسائل در زمان‌بندی پروژه تعیین وابستگی بین وظایف است. برای شناسایی و نمایش وابستگی بین وظایف از شبکه‌ای به نام شبکه‌ی وظایف[3] استفاده می‌شود. شکل مقابل نشان می‌دهد که وظایف چطور در یک پروژه فرضی پیش‌نیاز یکدیگرند.

به عنوان مثال وظیفه AB پیش‌نیازی برای دو وظیفه BC و BD است و به همین ترتیب به صورت غیرمستقیم پیش‌نیازی برای دو وظیفه‌ی دیگر CF و DF نیز می‌باشد. بنابراین امکان توسعه‌ی موازی هیچ‌یک از این چهار وظیفه با وظیفه AB وجود ندارد. اعداد روی هر یال می‌تواند مدت زمان لازم برای تکمیل هر وظیفه را مشخص کند. در این مثال فرض کنید که اعداد روی هر یال، تعداد روزهای لازم جهت تکمیل هر وظیفه را نشان می‌دهند.

مسیر بحرانی مسیری از گراف شبکه‌ی وظایف است که اگر بخواهیم پروژه از زمان‌بندی خود عقب نماند، باید تمام وظایف روی این مسیر بدون تأخیر انجام شده و در موعد مقرر به اتمام برسند. در مثال بالا حداقل زمان لازم جهت تکمیل پروژه پانزده روز است. این زمان مربوط به مسیر AEF می‌باشد. مسیری که از گره شروع (A) آغاز شده به گره پایان (F) ختم شده است. مجموع وزن یال‌های بقیه مسیرها، از پانزده کمتر است. بنابراین مسیر بحرانی برنامه، مسیر AEF

[1] Program evaluation and review technique
[2] Critical path Method
[3] Task network

است. حال اگر طبق برنامه‌ریزی، قرار باشد فعالیت BC در ابتدای روز سوم آغاز شود، ولی در واقعیت این وظیفه در ابتدای روز چهارم شروع شود تحویل پروژه یک روز به تأخیر می‌افتد! بنابراین می‌توان گفت که شبکه وظایف که با نمودارهای PERT و CPM نشان داده می‌شوند، ابزاری جهت نمایش وابستگی داخلی بین وظایف است و جهت تعیین مسیر بحرانی به کار می‌رود.

پس از آنکه زمان‌های قطعی و مشخص برای شروع و پایان هر وظیفه تعیین شد، نوبت به مدل‌سازی زمان‌بندی کلی پروژه می‌رسد، نمودار زمانی یا نمودار گانت جهت این مرحله مورد استفاده قرار می‌گیرد.

۱) تخصیص مسئولیت‌ها

گزینه سوم پاسخ سوال نیست، زیرا پس از تعیین مدل فرآیند تولید نرم‌افزار، برآوردهای مربوط به میزان کار، نیروی لازم برای انجام کار، زمان لازم برای انجام کار و هزینه لازم برای انجام کار نوبت به زمان‌بندی انجام پروژه می‌رسد. یعنی باید شبکه‌ای از وظایف مهندسی نرم‌افزار ایجاد شود تا کارها به موقع و سر وقت انجام شود. هنگامی که شبکه ایجاد شد، باید به هر وظیفه مسئولیتی نسبت داده شود و از انجام آن اطمینان حاصل شود. مدیر پروژه با استفاده از زمان‌بندی به عنوان یک راهنما، می‌تواند هر مرحله از فرآیند نرم‌افزار را پیگیری و کنترل کند. به بیان دیگر زمان‌بندی پروژه با هدف ایجاد یک زمان‌بندی مناسب و پویا جهت اتمام به موقع پروژه انجام می‌شود.

۲) تخصیص منابع

گزینه چهارم پاسخ سوال نیست. زیرا تخصیص منابع پس از برآوردهای نیرو، هزینه و زمان لازم برای انجام کار انجام می‌شود. زمان‌بندی پروژه فعالیتی است که با توجه به «مهلت زمانی» برای توسعه‌ی پروژه، نیروی انسانی لازم را در میان وظایف موجود توزیع می‌کند. بنابراین زمان بندی، هم بر روی توزیع نیرو و هم بر زمان اجرای هر وظیفه نظارت دارد. بدیهی است که در اولین مرحله‌ی زمان‌بندی، ابتدا باید تمامی وظایف پروژه از طریق روش‌هایی همچون مصاحبه، گفتگو، پرسش‌نامه و نمونه‌سازی، شناسایی شوند. بعد از شناخت وظایف و وابستگی‌های مابین آن‌ها، باید نیروی انسانی پروژه را مابین وظایف مختلف توزیع کرد. توزیع نیروی کار براساس برآوردها انجام می‌شود. فعالیت توزیع نیروی کار از قانون ۴۰-۲۰-۴۰، پیروی می‌کند. ۴۰ درصد از نیروی پروژه باید به فعالیت‌های مدل تحلیل و مدل طراحی، ۲۰ درصد به فعالیت پیاده‌سازی و ۴۰ درصد نیز به فعالیت تست نرم‌افزار اختصاص یابد.

۳۳- گزینه (۳) صحیح است.

توصیف «یک قاعده سه- بخشی که رابطه‌ی بین یک حوزه (Context)، یک مساله (Problem) و یک راه حل (Solution) را بیان می‌کند»، مربوط به الگوهای طراحی یا Design Patterns است.

۳۴- گزینه (۴) صحیح است.

مدل مسیریابی (Navigation)، در مدل‌سازی و طراحی صفحات مبتنی بر وب کاربرد عمده دارد.

۳۵- گزینه (۳) صحیح است.

برای اندازه‌گیری قابلیت اطمینان یا اعتماد(Reliability) می‌توان از معیار ساده MTBF که سرواژه‌ی عبارت Mean Time Between Failures و به معنی **متوسط زمان بین دو خرابی** است، استفاده کرد که مطابق رابطه‌ی زیر است:

MTBF=MTTF+MTTR

MTTF سرواژه‌ی عبارت Mean Time To Failure و به معنی **متوسط فاصله‌ی زمانی تا خرابی دوباره** است و MTTR سرواژه‌ی عبارت Mean Time To Repair و به معنی **متوسط زمان رفع خرابی است.** زمانی که یک خرابی رخ می‌دهد، ابتدا مدت زمانی طول می‌کشد تا خرابی تعمیر شود (MTTR) و سپس مدت زمانی طول می‌کشد تا خرابی دوباره رخ دهد (MTTF) و این یعنی MTBF که به معنی **متوسط زمان بین دو خرابی است.** هرچه مقدارMTBF افزایش یابد، تعداد خرابی‌ها در واحد زمان کاهش می‌یابد و این یعنی افزایش قابلیت اطمینان نرم‌افزار.

قابلیت دسترسی نرم‌افزار به معنی احتمال کار کردن برنامه طبق خواسته‌ها در یک بازه‌ی زمانی مشخص می‌باشد که به صورت زیر تعریف می‌شود:

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \times \%۱۰۰$$

و یا به طور ساده‌تر داریم:

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTBF}} \times \%۱۰۰$$

توجه: اندازه‌ی قابلیت اطمینان (MTBF) به یک اندازه به MTTR و MTTF حساس است. اما اندازه‌ی قابلیت دسترسی به MTTR بیش‌تر حساس می‌باشد.

۳۶- گزینه (۲) صحیح است.

صورت گزینه‌ها به صورت زیر است:

۱) تخصیص منابع به آنها مشکل‌تر است.

گزینه اول پاسخ سوال نیست، زیرا در زمان‌بندی پروژه، باید مشخص شود که هر وظیفه در چه بازه‌ی زمانی باید شروع شود و چه مدت زمان نیاز دارد تا تکمیل شود. تکنیک‌های ارزیابی و بازبینی برنامه (PERT) و روش مسیر بحرانی (CPM) جهت زمان‌بندی پروژه مورد استفاده قرار می‌گیرند. در نمودارهای PERT و CPM توالی فعالیت‌ها، مسیربحرانی و تخصیص منابع لحاظ می‌شود.

۲) تاخیر آنها باعث تاخیر کل پروژه می‌شود.

گزینه دوم پاسخ سوال است، زیرا مسیر بحرانی مسیری از گراف شبکه‌ی وظایف است که اگر بخواهیم پروژه از زمان‌بندی خود عقب نماند، باید تمام وظایف یا تکالیف (Tasks) روی این مسیر بدون تأخیر انجام شده و در موعد مقرر به اتمام برسند. در غیر اینصورت تاخیر آنها باعث تاخیر کل پروژه می‌شود.

۳) زمان انجام آنها با روش‌های سنتی قابل تخمین نیست.

گزینه سوم پاسخ سوال نیست، زیرا زمان انجام وظایفی که در مسیر بحرانی قرار دارند یا قرار ندارند همواره توسط روش‌های مختلف برآورد و تخمین (سنتی و غیرسنتی) قابل تخمین و اندازه‌گیری است.

۴) بروز خطا در آنها باعث انتشار خطا به سایر تکالیف مسیر بحرانی می‌شود.

گزینه چهارم پاسخ سوال نیست، زیرا بروز خطا و انتشار آن به بخش‌های مختلف پروژه به کیفیت طراحی و میزان اتصال (Coupling) مابین پیمانه‌ها بستگی دارد. و به توالی وظایف در مسیر بحرانی ارتباطی ندارد.

۱۰ متدولوژی RUP

مقدمه

متدولوژی RUP گنجینه‌ای است ارزشمند از راهکارها و تجارب موفق در مهندسی و تولید نرم‌افزار، در واقع قالب و چارچوبی است برای تعریف فرآیندهای مهندسی و تولید سیستم‌های پیچیده‌ای مانند نرم‌افزار. این چارچوب فرایند به وسیله‌شرکت رشنال در طول بیش از دو دهه تحقیق و بررسی ایجاد گردیده است. این شرکت در سال ۲۰۰۳ رسماً توسط شرکت IBM خریداری شد و در حال حاضر، مالکیت RUP (به عنوان یک محصول) در اختیار شرکت IBM می‌باشد.

RUP رویکردی است منظم و دارای دیسیپلین، برای تخصیص مسئولیت‌ها و مدیریت آنها در یک سازمان یا تیم تولیدکننده سیستم‌های نرم‌افزاری. البته، RUP الگویی را در اختیار مهندسین و مدیران قرار می‌دهد که قابل تعمیم و گسترش به طیف وسیعی از پروژه‌ها همچون سیستم‌های اطلاعاتی، صنعتی، بلادرنگ، ارتباطات راه دور و حتی پروژه‌های تولید محصولات غیرنرم‌افزاری باشد.

هدف این فرآیند عبارت است از: تولید یک محصول دارای کیفیت مطلوب، در یک چارچوب زمانی و هزینه‌ای قابل پیش‌بینی، که تأمین‌کننده نیازهای کاربران نهایی‌اش باشد.

فرایند یکپارچه رشنال (RUP: Rational Unifed Process)

متدولوژی RUP سه مفهوم و معنای تا حدی متفاوت را دربرمی‌گیرد:

۱- RUP روشی برای توسعه نرم‌افزار است.

این روش، دارای ویژگی‌های برجسته‌ای مانند تکرارشونده (Iterative) بودن، تمرکز بر معماری و مبتنی بر موارد کاربرد (یا به عبارت ساده‌تر، مبتنی بر خواسته‌های مشتری) می‌باشد.

۲- RUP یک فرآیند خوب تعریف شده و خوش ساختار است.

در RUP به روشنی مشخص می‌شود که هر کسی چه مسئولیتی دارد و چگونه و چه هنگام

باید این مسئولیت را انجام دهد.

نقش‌ها (Roles) فعالیت‌ها (Activites)، دستاوردها (Artifacts) و جریان‌های کار یا ترتیب و توالی فعالیت‌ها (Work Flows) تعریف شده در RUP، عناصر اصلی یک فرآیند (یعنی چه کسی، چه کاری، چگونه و چه موقع) را تعریف و تبیین می‌نماید. RUP ساختار مناسبی برای کنار هم گذاشتن این مؤلفه‌ها فراهم نموده است.

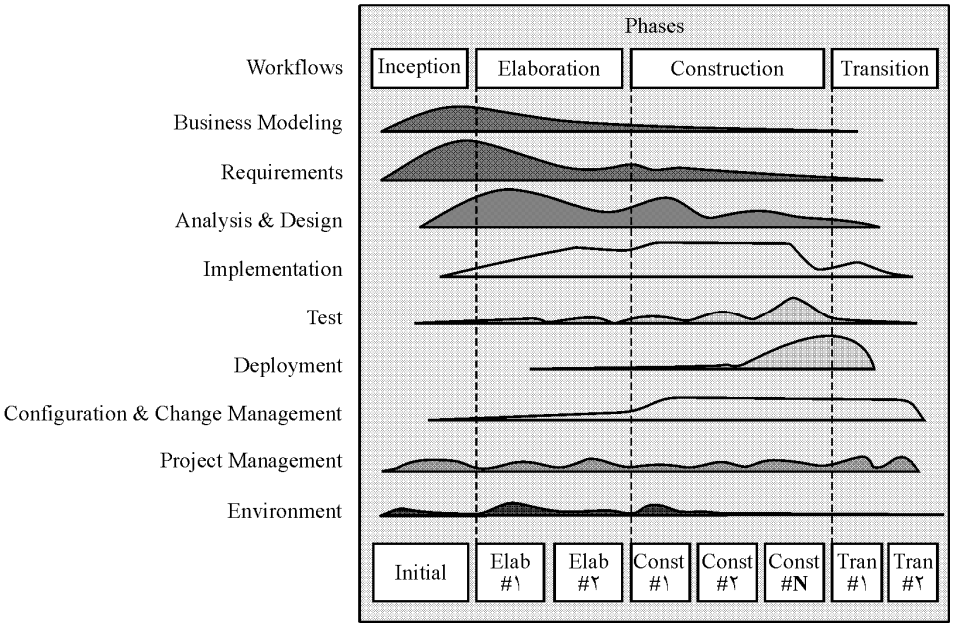
۳- RUP چارچوبی با قابلیت سفارشی‌سازی را فراهم می‌کند.

از این منظر، RUP یک فرآیند نیست که بتوان آن را مستقیماً به عنوان قالب تعریف یک پروژه تولید نرم‌افزار به کار گرفت، بلکه مفهومی است بسیار فراتر و جامع‌تر. در واقع RUP، مانند یک میز پر از غذاهای متنوع در یک رستوران می‌باشد. مسلماً، هیچ‌یک از مهمان‌های این رستوران قادر نخواهند بود همه‌ی غذاها را میل نمایند. در عوض، هر کس با توجه به ذائقه‌اش گلچینی از غذاها را انتخاب می‌نماید. RUP بانک دانش بزرگی از راهکارها و تجارب موفق برای شرایط و طیف گسترده‌ای از پروژه‌های مختلف، فراهم نموده است. هر پروژه‌ای با توجه به نیازها، محدودیت‌ها و امکانات خاصش، به گونه‌ای متفاوت از این بانک دانش استفاده می‌نماید. RUP به عنوان چارچوب فرآیند، دارای قابلیت سفارشی‌سازی و پیکربندی برای طیف وسیعی از پروژه‌ها می‌باشد. به کمک RUP، می‌توان فرایندی مناسب برای تولید یک محصول نرم‌افزاری بسیار کوچک (پروژه‌ای در ابعاد ۱ نفر و ۲ هفته زمان) یا یک پروژه نرم‌افزاری بزرگ (پروژه‌ای در ابعاد ۱۰ هزار نفر و ۵ سال زمان) را تعریف و با موفقیت اجرا نمود. این ویژگی که آن را مقیاس‌پذیری (Scalability) می‌نامند، یکی دیگر از ویژگی‌های کم‌نظیر RUP محسوب می‌شود.

توجه: کتاب‌ها و مقالات زیادی در رابطه با RUP نوشته شده است. اما کامل‌ترین و جامع‌ترین منبع اطلاعاتی مرتبط با RUP، در قالب یک محصول پیاده‌سازی شده با تکنولوژی وب و روی یک لوح فشرده، تولید شده است. این محصول که توسط شرکت رشنال (و اکنون IBM) عرضه می‌شود، شامل کلیه‌ی راهنمایی‌ها، مثال‌ها، تعاریف، الگوها و قالب‌هایی است که سرتاسر چرخه‌ی تولید محصولات نرم‌افزاری (یعنی از زمان شروع پروژه تا انتهای کار و تحویل کامل محصول به دست آمده) را تحت پوشش قرار می‌دهد.

معماری RUP

در شکل زیر معماری RUP نشان داده شده است. همان‌گونه که در این شکل ملاحظه می‌شود، این فرآیند دارای ساختاری دوبعدی است:



دو بعد تشکیل‌دهنده ساختار متعامد RUP عبارتند از:

۱- ساختار دینامیک

ساختار دینامیک RUP، بعد افقی نشان داده شده در شکل فوق و بیانگر ساختار پویا و ملاحظات مرتبط با زمان در فرآیند می‌باشد. در این بعد، ملاحظاتی مانند چرخه‌های توسعه (یا چرخ‌های تولید)، فازها، تکرارها و نقاط تصمیم‌گیری کلیدی مطرح می‌باشد. این مفاهیم در کنار هم، چرخه‌ی عمر یک پروژه نرم‌افزاری (پروژه‌ی تولید یک محصول نرم‌افزاری) را تعریف می‌نمایند.

توجه: RUP یک روش ساختاربندی شده برای تولید تکرارشونده فراهم می‌کند که یک پروژه را به چهار فاز تقسیم می‌کند که در ادامه به توضیح این فازها خواهیم پرداخت.

۲- ساختار استاتیک (محتوایی)

ساختار RUP دارای یک بعد عمودی نیز می‌باشد که بیانگر ساختار استاتیک یا محتوایی آن است. در این بعد، توصیفی از چگونگی دسته‌بندی و سازماندهی عناصر محتوایی فرآیند یعنی، فعالیت‌ها (Activities)، راهنمایی‌ها (Guidelines)، دستاوردها (Artifacts) و نقش (Roles) در قالب دیسیپلین‌ها یا جریان‌های منظم و منطقی ارائه شده است.

ساختار دینامیک RUP

براساس فرایند RUP، یک پروژه نرم‌افزاری که با هدف تولید یک محصول نرم‌افزاری تعریف و اجرا می‌گردد، از نظر دینامیکی یا بعد زمانی دارای چهار فاز می‌باشد:

۱- فاز آغازین یا شناخت (Inception)
۲- فاز تشریح یا معماری (Elaboration)
۳- فاز ساخت (Construction)
۴- فاز انتقال (Transition)

توجه: اغلب کسانی که برای اولین بار با RUP آشنا می‌شوند، با یک تصویر اشتباه مواجه هستند و آن این است که این فازها را صرفاً تغییر نام همان فازهای فرآیندهای مبتنی بر روش آبشاری تلقی می‌کنند. بنابراین فاز آغازین را فازی می‌دانند که در آن همه‌ی نیازمندی‌های سیستم مشخص می‌شود، در فاز تشریح، یک طراحی سطح بالا انجام می‌دهند، در فاز ساخت، برنامه‌نویسی و پیاده‌سازی سیستم را انجام داده و در نهایت در فاز انتقال، تست سیستم را انجام می‌دهند!

هر چند ممکن است که در ظاهر از تکرارشونده بودن فرآیند صحبت شود ولی در عمل روح آبشاری را می‌توان در فعالیت‌هایشان مشاهده نمود!

توجه: در فرآیند RUP برگشت از یک فاز به فازهای قبلی مجاز نمی‌باشد. در حالی که این خصلت در فرآیندهای مبتنی بر روش آبشاری مجاز می‌باشد.

توجه: در فرآیند RUP تمام فعالیت‌های یک فاز طوری برنامه‌ریزی می‌شود که با توجه به ویژگی‌های پروژه، نتایج مورد انتظار حاصل شود. هر گاه در یک فاز با تکرارهای برنامه‌ریزی شده، نتایج مورد انتظار برآورده نشود، بدون رفتن به فاز بعدی، باید یک تکرار دیگر در همان فاز، در نظر گرفته شود.

توجه: برخلاف فرآیندهای مبتنی بر روش آبشاری، فازهای RUP براساس تقسیم‌بندی فعالیت‌ها برحسب نوعشان برنامه‌ریزی نشده است. با این توصیف، در RUP، فاز خاصی به نام تحلیل یا تست وجود ندارد، بلکه در عوض، در هر فاز که ماهیتی زمانی و مبتنی بر تصمیم‌گیری‌هایی مرتبط با ریسک‌های عمده مطرح در پروژه دارند، برای دستیابی به اهداف تعیین شده، به اندازه‌ی کافی انواع فعالیت‌های مختلف (تحلیل، طراحی، پیاده‌سازی، تست و استقرار که آنها را در نمودار RUP دیسیپلین می‌نامند) انجام می‌شود و پس از آنکه نتایج موردنظر حاصل گردید (نقطه تصمیم‌گیری)، تصمیم‌های لازم درباره‌ی خاتمه یا تکرار آن فاز اتخاذ می‌شود.

توجه: برآمدگی‌های کوه مانند موجود در نمودار متعامد RUP میزان فعالیت‌های مختلف (دیسیپلین‌ها) در هر فاز را نشان می‌دهد.

برای‌مثال در فاز انتقال (Transition) ممکن است کماکان پیاده‌سازی مختصری نیاز باشد و حتی، فعالیت‌های مربوط به تحلیل و طراحی نیز ممکن است انجام شود. اما به طور منطقی در فاز انتقال فعالیت‌های مرتبط با تست و استقرار بیشترین حجم را به خود اختصاص می‌دهند.

فاز آغازین (شناخت)

در این مرحله کلیه نیازمندی‌ها و محدوده سیستم به اندازه‌ی کافی شناخته می‌شوند. به بیان دیگر هدف اصلی این فاز، رفع ریسک‌های کلان سازمانی، ریسک‌های مربوط به درک اهداف و

محدوده پروژه و نیز به دست آوردن اطلاعات کافی برای اطمینان از ادامه پروژه و یا توقف آن می‌باشد.

توجه: بسیاری از محافل آکادمیک و صنعتی کشور، این فاز را فاز شناخت نیز می‌نامند، هر چند این نام‌گذاری اشتباه نیست، اما ما ترجیح دادیم برای جلوگیری از اشتباه شدن آن با اولین فاز روش آبشاری، از واژه آغازین در کنار واژه شناخت استفاده نماییم.

اهداف فاز آغازین (شناخت)

۱- شناخت آنچه که باید ساخته شود.
۲- شناخت عملکرد اصلی سیستم
۳- تعیین حداقل یک راه حل ممکن
۴- شناخت هزینه‌ها، برنامه زمان‌بندی و ریسک‌های مرتبط با پروژه
۵- تصمیم‌گیری درباره این که چه فرآیندی دنبال شود و چه ابزاری به کار گرفته شود.

خروجی‌های فاز آغازین (Artifacts)

۱- دید کلی از نیازمندی‌های پروژه (Vision)
۲- مدل موارد کاربرد اولیه که ۱۰ تا ۲۰ درصد آن تکمیل شده است (Use Case)
۳- برآورد اولیه از ریسک‌ها
۴- برآورد منابع موردنیاز پروژه

فاز تشریح

فاز تشریح به نقطه‌ی تصمیم‌گیری در رابطه با تثبیت معماری ختم می‌گردد. در این فاز، غلبه بر ریسک‌های فنی با تثبیت و مبنا قرار دادن معماری سیستم، امکان‌پذیر می‌گردد. اینکه RUP فاز دوم از چرخه تولید را به تثبیت معماری اختصاص داده است، نشان می‌دهد که تثبیت معماری، نقش کلیدی و بسیار مهمی در موفقیت پروژه دارد.

در این فاز، معماری سیستم با توجه به نیازمندی‌ها، ریسک‌ها و محدودیت‌های هزینه و زمان شکل گرفته و پس از انجام تست‌های لازم، تثبیت می‌گردد.

در طول این فاز، ریسک‌های زیر باید رفع شوند:

۱- ریسک‌های مرتبط با نیازمندی‌ها (اینکه آیا در حال پیاده‌سازی سیستم درستی هستیم یا نه؟)
۲- ریسک‌های مرتبط با معماری (اینکه آیا راهکار مناسبی را می‌سازیم یا نه؟)
۳- ریسک‌های مرتبط با هزینه و زمان (آیا واقعاً طبق برنامه هستیم؟)
۴- ریسک‌های مرتبط با فرآیند، ابزارها و محیط (اینکه آیا فرآیند و ابزار مناسبی برای پروژه در اختیار داریم و یا نه؟)

تنها پس از اطمینان از رفع ریسک‌های فوق، می‌توانیم قدم در فاز بعدی، یعنی فاز ساخت، بگذاریم.

اهداف فاز تشریح

۱- شناخت بیشتر نیازمندی‌ها
۲- طراحی، پیاده‌سازی، اعتبارسنجی و تثبیت معماری
۳- طراحی موارد کاربرد اصلی
۴- طراحی پایگاه داده‌ها
۵- رفع ریسک‌های عمده و بهبود برآوردهای هزینه و زمان پروژه

خروجی‌های فاز تشریح

۱- مدل موارد کاربرد (حداقل ۸۰ درصد کامل شده باشد)
۲- شرح معماری نرم‌افزار
۳- مدل نمونه معماری قابل اجرا
۴- فهرست ریسک‌های اصلاح شده
۵- طرح و برنامه‌ریزی کلی پروژه
۶- راهنمای اولیه کاربران (اختیاری)

فاز ساخت

این فاز، معمولاً طولانی‌ترین و در عین حال قابل انعطاف‌ترین فاز در فرآیند تولید می‌باشد. هدف این مرحله آشکارسازی نیازمندی‌های باقی مانده و تکمیل سیستم براساس معماری است. در واقع در این مرحله آنچه که در مراحل قبلی به شکل مدل ایجاد شده‌اند، به شکل محصولی قابل اجرا، پیاده‌سازی می‌شوند. تمرکز اصلی این مرحله روی ایجاد کد با کیفیت بالا با حداقل هزینه است.

در انتهای این فاز یک سیستم نرم‌افزاری (نسخه بتا) با تمام قابلیت‌های تعریف شده براساس نیازمندی‌ها و چشم‌انداز پروژه جهت استقرار در محیط مشتری و کاربران پیاده‌سازی می‌گردد. البته، ممکن است اشکالات و نواقص پنهانی موجود باشد که در طی تکرارهای فاز بعد (انتقال) رفع می‌گردند. در این فاز بخش عمده‌ای از کارهای پروژه انجام می‌شود. بنابراین بهتر است برای جلوگیری از هزینه‌های اضافی (مالی و زمانی) ناشی از دوباره کاری، با معماری تثبیت شده وارد این فاز شد.

اهداف فاز ساخت

۱- کمینه کردن هزینه‌های تولید و بهره‌گیری مناسب از امکان توسعه به صورت موازی به کمک معماری مبتنی بر مؤلفه
۲- توصیف موارد کاربرد، تشریح سایر نیازمندی‌های باقی مانده، تفصیل بیشتر جزییات طراحی، تکمیل پیاده‌سازی و تست
۳- بررسی آمادگی شرایط محیط مشتری برای انتقال نرم‌افزار

خروجی‌های فاز ساخت

۱- راهنمای کاربران شامل مستندات پشتیبانی و مواد آموزشی.
۲- نسخه‌ای از محصول تحت عنوان نسخه بتا شامل مؤلفه‌های نصب و راه‌اندازی
۳- مدل طراحی کامل شده نرم‌افزار

فاز انتقال

در انتهای این فاز، محصول نرم‌افزاری به طور کامل به محیط مشتری و کاربران انتقال یافته و تمام کاربران نهایی سیستم قادر خواهند بود همه خدمات مورد نیازشان را بدون نیاز به حضور مستمر تولیدکنندگان، از سیستم دریافت نمایند، پروژه به طور کامل بسته شده و سیستم به مرحله نگهداری و تکامل وارد می‌گردد.

فاز قبل، یعنی فاز ساخت با ارائه نسخه‌ای از نرم‌افزار شامل تمام قابلیت‌ها، عملکردها و نیازمندی‌های مورد انتظار مشتری، یعنی نسخه بتا، پایان می‌یابد. این نسخه، شامل ابزارهای لازم برای نصب، مستندات تکمیلی، پشتیبان و محتوای آموزشی می‌باشد. اما واضح است که نسخه بتا، محصول نهایی نیست و هنوز لازم است که تنظیم‌ها و پیکربندی‌های دقیق‌تری روی نیازمندی‌ها، کارایی و به طور کلی کیفیت آن انجام شود.

تمرکز اصلی فعالیت‌ها در فاز انتقال، بر این موضوع است که اطمینان حاصل شود، سیستم نرم‌افزاری تولید شده، به طور کامل نیازهای کاربرانش را برآورده می‌نماید. به طور معمول، در این فاز یک یا دو تکرار برنامه‌ریزی می‌شود که عمدتاً شامل تست محصول به منظور آماده‌سازی آن برای تحویل نهایی و نیز اعمال تنظیمات و اصلاحات جزیی براساس بازخورد دریافت شده از کاربران می‌باشد.

اهداف فاز انتقال

۱- انجام تست بتا به منظور تأیید اینکه سیستم پاسخ‌گوی انتظارات کاربران می‌باشد.
۲- آموزش کاربران و نگهدارندگان سیستم
۳- آماده‌سازی محل استقرار و تبدیل بانک‌های اطلاعاتی عملیاتی
۴- در حالتی که محصول به صورت یک بسته تجاری می‌باشد، آماده شدن برای بسته‌بندی، بازاریابی، توزیع، فروش و آموزش کارکنان مربوطه ضروری می‌باشد.
۵- دستیابی به توافق تمام ذینفعان نسبت به اینکه نسخه‌های تثبیت شده در استقرار، کامل بوده و با معیارها و شرایط ارزیابی مورد بحث در چشم‌انداز پروژه، تطابق دارد.
۶- بهبود کارایی پروژه‌هایی آتی از طریق تجربه‌های کسب شده از پروژه‌های قبلی

خروجی‌های فاز انتقال

۱- تست سیستم به منظور ارزیابی آن در برابر انتظار کاربران
۲- اجرای موازی با سیستم موجود که جایگزین آن خواهد شد.
۳- تبدیل داده‌ها در صورت نیاز

۴- آموزش کاربران و نگهدارندگان سیستم
۵- آماده‌سازی محصول جهت بازاریابی و فروش

ساختار محتوایی RUP

همان‌گونه که گفتیم RUP دارای دو بعد می‌باشد یک بعد پویا (دینامیک) که در آن ملاحظات زمانی، مانند فازها و تکرارها مطرح است و یک بعد ایستا (استاتیک) که در آن مفاهیمی مانند:

- نقش‌ها (Roles)
- فعالیت‌ها (Activities)
- دستاوردها (Artifacts)
- جریان‌های کار (Workflows)
- دیسیپلین‌ها (Disciplines)

و سایر عناصر به کار رفته در توصیف محتوای فرآیند، مطرح می‌باشد.

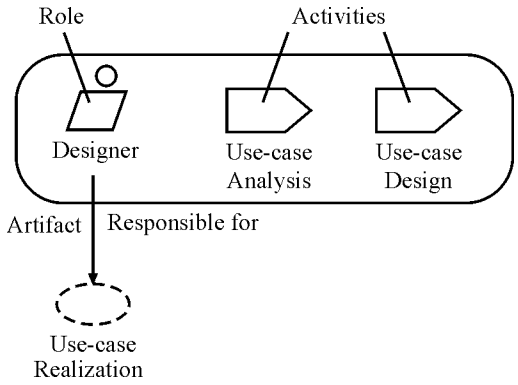
مدل ساختاری RUP

یک فرآیند تولید، با تعریف اینکه چه کسی (Who)، چه کاری را (What)، چه موقع (When) و چگونه (How) برای تولید موفق یک فرآورده‌ی دارای کیفیت مطلوب، باید انجام دهد، زمینه‌ی مناسبی را برای تحقق کار تیمی فراهم می‌نماید. ساختار RUP به عنوان چارچوبی برای فرآیندهای تولید نرم‌افزار، از پنج مؤلفه‌ی اصلی تشکیل شده است:

این مؤلفه‌ها عبارتند از:

- نقش‌ها : معادل مؤلفه «چه کسی» در توصیف یک فرآیند،
- فعالیت‌ها : معادل مؤلفه «چگونه»،
- دستاوردها : معادل «چه چیزی»،
- جریان‌های کار : معادل «چه زمانی»،
- دیسیپلین‌ها : ظرفی برای چهار نوع عنصر قبل.

شکل زیر ارتباط میان نقش‌ها، فعالیت‌ها و دستاوردها در فرآیند RUP را نشان می‌دهد:

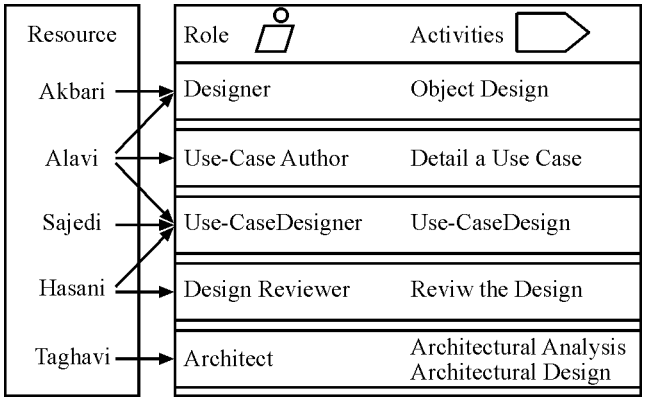


نقش‌ها

نقش، تعریف‌کننده رفتار و مسئولیت‌های یک شخص یا گروهی از افراد است که با هم در یک تیم فعالیت می‌کنند، رفتار در قالب فعالیت‌هایی که یک نقش انجام می‌دهد، بیان می‌شود و هر نقشی با مجموعه‌ای از فعالیت‌های وابسته به هم در ارتباط است. وابستگی فعالیت‌ها در اینجا بدین معناست که مجموعه‌ی خاصی از فعالیت‌ها توسط یک شخص دارای مهارت‌های خاص، به شکل مناسب‌تر و بهتری قابل انجام است. مسئولیت‌های هر نقش معمولاً در ارتباط با دستاوردهای (Artifact) مشخصی که آن نقش ایجاد، ویرایش و کنترل می‌نماید، توصیف می‌شود.

توجه: مفهوم نقش در RUP با عناوین شغلی متفاوت است و در ضمن لزومی ندارد که به تعداد نقش‌ها، افراد متفاوتی حضور داشته باشند. معمولاً یک شخص یک یا چند نقش را به عهده دارد، ضمن آنکه چند شخص نیز می‌توانند یک نقش را داشته باشند.

مثال: در شکل زیر نمونه‌ای از نگاشت‌نقش‌ها به افراد در یک پروژه‌ی نرم‌افزاری نشان داده شده است.



فعالیت‌ها

نقش‌های مختلف در طول فرآیند تولید نرم‌افزار، یکسری فعالیت را انجام می‌دهند. فعالیت، عبارت است از یک واحد کار (Unit of Work) که از یک شخص ایفاکننده یکی از نقش‌های فرآیند، انجام آن انتظار می‌رود و در اثر انجام آن، نتیجه‌ای معنی‌دار برای پروژه حاصل می‌گردد. هر فعالیت، هدف مشخصی دارد که عمدتاً به صورت تولید یا بروزرسانی دستاوردها (Artifacts) مانند بروزرسانی یک مدل، یک کلاس، یا یک طرح می‌باشد. در RUP هر کدام از فعالیت‌ها به یک نقش خاص اختصاص یافته است.

توجه: اغلب فعالیت‌ها در RUP با پیشوند "Activity" مشخص می‌شوند. به عنوان مثال، فعالیت پیدا کردن موارد کاربرد و اکتورها به صورت "Activity: Find use case and actors" می‌باشد.

توجه: هر یک از فعالیت‌ها به مجموعه‌ای از گام‌ها شکسته می‌شوند. با این وصف، کوچک‌ترین کاری که در یک پروژه قابل انجام است، گامی است از یک فعالیت.

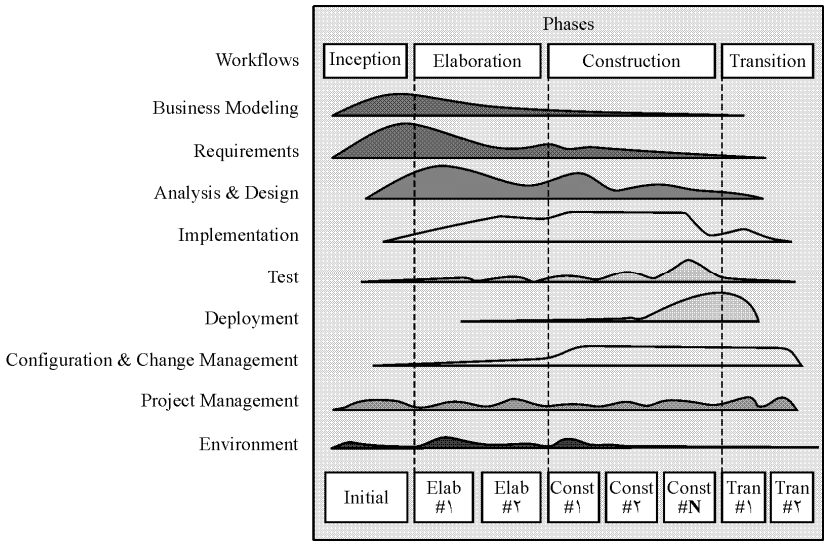
دستاوردها

فعالیت‌های مختلف دارای دستاوردهایی بـه عنـوان ورودی و خروجـی مـی‌باشـند. دسـتاورد، عبارت است از قطعه‌ای از اطلاعات که در طی یک فرآیند، تولید شده، تصحیح می‌شـود یـا مـورد استفاده قرار می‌گیرد. دستاوردها، محصولات قابل لمس پروژه می‌باشند و عبارتند از چیزهـایی کـه **توجه:** دستاوردها در RUP با پیشوند artifact مشخص می‌شوند.

مانند "Artifact: Use Case Model"

دیسیپلین‌ها

دیسیپلین‌ها،گروه‌بندی طبیعی فعالیت‌ها، نقش‌ها و دستاوردهای فرآیند می‌باشند. به بیـان دیگـر دیسیپلین‌ها، ظرف‌هایی هستند برای سازماندهی فعالیت‌های فرآیند. در فرآیند RUP،به طـور پـیش فرض تعداد نه دیسیپلین وجود دارد. این دیسیپلین‌ها بیانگر دسته‌بندی مجموعه نقش‌ها و فعالیت‌ها در قالب گروه‌هایی است که دارای نگرانی، دغدغه‌ها و یا دیدگاه مشترکی می‌باشـند. نـه دیسـیپلین ارائه شده به دو دسته تقسیم می‌شوند، یک دسته شامل شش دیسیپلین فنی و دسته‌ی دیگـر، شـامل سه دیسیپلین پشتیبانی.



دیسیپلین‌های چارچوبی (فنی)

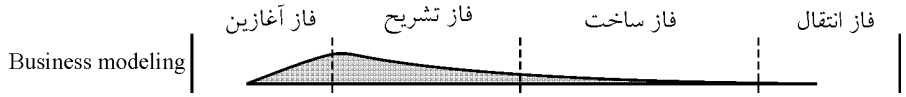
۱- دیسیپلین مدل‌سازی سازمان
۲- دیسیپلین نیازمندی‌ها
۳- دیسیپلین تحلیل و طراحی
۴- دیسیپلین پیاده‌سازی
۵- دیسیپلین تست
۶- دیسیپلین استقرار

دیسیپلین‌های چتری (پشتیبان)

۱- دیسیپلین مدیریت پروژه
۲- دیسیپلین مدیریت پیکربندی و تغییرات
۳- دیسیپلین محیط

نام‌گذاری مجموعه نقش‌ها و فعالیت‌های یکسان در قالب مفهوم دیسیپلین نیز جالب توجه است. می‌دانیم که مفهوم دیسیپلین به معنای نظم و انظباط است. در واقع وجه تسمیه گروه‌بندی‌های یاد شده، این است که مجموعه‌ی نقش‌ها و فعالیت‌هایی که مثلاً در دیسیپلین مدیریت پروژه، گروه‌بندی می‌شوند، دارای نوعی طرز نگرش، دیدگاه انضباط کاری و هدف مشترک می‌باشند که با دیدگاه و اهداف مجموعه نقش‌ها و فعالیت‌های سایر دیسیپلین‌ها، تفاوت اساسی دارد.

۱- دیسیپلین مدل‌سازی سازمان



حجم فعالیت‌های دیسیپلین مدل‌سازی سازمان در طول فازهای مختلف چرخه تولید

اهداف دیسیپلین مدل‌سازی

الف) درک ساختار و پویایی سازمانی که باید سیستم یا سیستم‌های نرم‌افزاری در آن مستقر شوند (سازمان هدف)

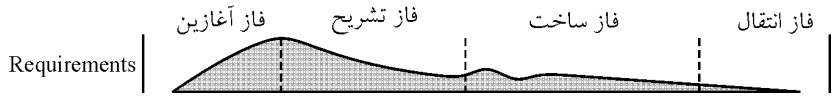
ب) درک مسائل و مشکلات جاری سازمان هدف و شناسایی پتانسیل‌ها و نقاط بهبود

ج) اطمینان از اینکه مشتریان، کاربران نهایی و تولیدکنندگان نرم‌افزار، همگی درک مشترکی از سازمان هدف دارند.

د) استنتاج و استخراج نیازمندی‌های سیستم یا سیستم‌هایی که باید سازمان هدف را در دستیابی به اهدافش، پشتیبانی نمایند.

توجه: برای نیل به این اهداف، دیسیپلین مدل‌سازی سازمان، چگونگی توسعه‌ی یک چشم انداز از سازمان جدید را توصیف نموده و براساس این چشم‌انداز، چگونگی تعریف **فرآیندها، نقش‌ها و مسئولیت‌های** سازمان را در یک مدل از سازمان بیان می‌نماید.

۲- دیسیپلین نیازمندی‌ها



حجم فعالیت‌های دیسیپلین نیازمندی‌ها در طول فازهای مختلف چرخه تولید

اهداف دیسیپلین نیازمندی‌ها

الف) برقراری و حفظ توافق میان مشتری و سایر ذینفعان با تیم تولید درباره‌ی چیستی و چرایی سیستم.

ب) فراهم نمودن درک مناسبی از نیازمندی‌های سیستم برای تولیدکنندگان آن

ج) تعریف و مدیریت مرزهای سیستم (System Boundary)

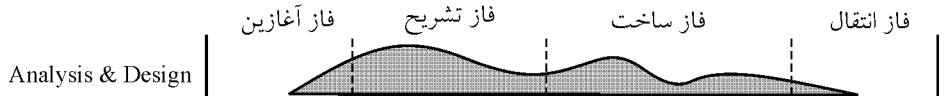
د) فراهم نمودن مبنایی مناسب برای برنامه‌ریزی تکرارهای مختلف (برنامه‌ریزی تکرارها در RUP عمدتاً براساس موارد کاربرد انجام می‌شود)

ه) فراهم نمودن مبنایی مناسب برای تخمین هزینه و زمان تولید سیستم.

ی) تعریف یک واسط کاربر که بر نیازها و اهداف کاربران متمرکز می‌باشد.

توجه: برای دستیابی به این اهداف، دیسیپلین نیازمندی‌ها چگونگی تعریف یک چشم‌انداز برای سیستم و ترجمه‌ی آن به مدل موارد کاربرد و مجموعه مشخصه‌های تکمیلی که جزییات نیازمندی‌های سیستم را در برخواهند داشت، تشریح می‌نماید. این دیسیپلین، چگونگی استفاده از خصیصه‌های نیازمندی‌ها را برای کمک به مدیریت محدوده سیستم و نیز چگونگی مدیریت تغییرات نیازمندی‌ها را تعریف می‌نماید.

۳- دیسیپلین تحلیل و طراحی



حجم فعالیت‌های دیسیپلین تحلیل و طراحی در طول فازهای مختلف چرخه تولید

هدف اصلی دیسیپلین تحلیل و طراحی

هدف اصلی دیسیپلین تحلیل و طراحی، ترجمه نیازمندی‌ها به توصیف‌هایی است که چگونگی پیاده‌سازی سیستم را نشان می‌دهند.

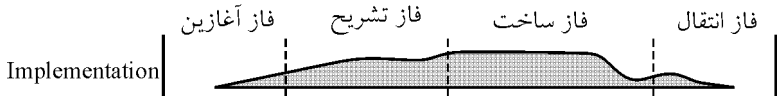
مدل نیازمندی‌ها عمدتاً بیانگر چیستی (What) و نیز شرایط عملکرد (وظیفه‌مندی‌های) سیستم است. در مدل نیازمندی‌ها، سیستم (راه‌حل) به عنوان یک جعبه‌ی سیاه از منظر کاربران و ذینفعان آن، توصیف می‌شود. در مقابل، در مدل‌های تحلیل و طراحی به بررسی چگونگی (How) تحقق نیازمندی‌ها توجه می‌شود.

برای رسیدن به این هدف، ابتدا باید نیازمندی‌ها را به خوبی درک کرده و پس از آن، با انتخاب بهترین استراتژی پیاده‌سازی، نیازمندی‌ها را به طراحی سیستم ترجمه نمود.

در اوایل پروژه، باید یک معماری مستحکم ایجاد شود، به گونه‌ای که طراحی سیستم در چارچوب آن قابل درک بوده و بتوان آن را به راحتی و در چارچوب هزینه‌ها و محدودیت‌های موجود، ساخت و کامل نمود. سپس باید طراحی را با توجه به محیط پیاده‌سازی تطبیق داده و با توجه به ملاحظات کیفی محصول، نظیر کارایی، استحکام، مقیاس‌پذیری و قابلیت گسترش که در

قالب نیازمندی‌های غیروظیفه‌مندی بیان شده‌اند، طراحی را کامل‌تر نمود.

۴- دیسیپلین پیاده‌سازی



حجم فعالیت‌های دیسیپلین پیاده‌سازی در طول فازهای مختلف چرخه تولید

اهداف دیسیپلین پیاده‌سازی

الف) تعریف چگونگی سازماندهی و ساختار کدهای برنامه در قالب زیرسیستم‌های هر لایه از معماری سیستم

ب) پیاده‌سازی کلاس‌ها و اشیاء در قالب یکسری مؤلفه

ج) انجام تست واحد بر روی مؤلفه‌های پیاده‌سازی شده

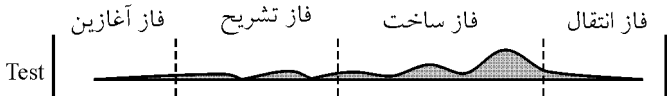
د) یکپارچه‌سازی و مجتمع‌سازی مؤلفه‌ها در قالب یک سیستم قابل اجرا

توجه: به تفاوت معنایی مفاهیم دیسیپلین پیاده‌سازی (Implementation) و فاز ساخت (Construction) و RUP توجه کنید.

بخش عمده‌ی فعالیت‌های دیسیپلین پیاده‌سازی به نوعی به برنامه‌نویسی (Programming) مرتبط می‌باشد. اما در فاز ساخت برای دستیابی به اهداف تعیین شده، فعالیت‌ها یا دیسیپلین‌های دیگر (علاوه بر پیاده‌سازی) نیز دخیل است. بنابراین، پیاده‌سازی یکی از اقدامات مهم و عمده‌ی فاز ساخت است،نه همه‌ی آن.

توجه: در دیسیپلین پیاده‌سازی، مفهوم تست، محدود به تست واحد (Unit Test) برای هر یک از مؤلفه‌ها به صورت جداگانه و مستقل می‌باشد. تست‌های دیگر مانند تست سیستم و تست یکپارچگی در دیسیپلین تست قرار دارند.

۵- دیسیپلین تست



حجم فعالیت‌های دیسیپلین تست در طول فازهای مختلف چرخه تولید

اهداف دیسیپلین تست

الف) یافتن و مستندسازی خطاها، نقایص و مشکلات فرآورده

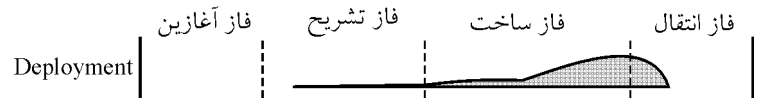
ب) توصیه به داشتن مدیریت روی کیفیت درک شده و مورد قبول از نرم‌افزار

ج) سنجش و ارزیابی فرضیات در نظر گرفته شده در طراحی و توصیف نیازمندی‌ها از طریق

ارائه نمایش‌های عینی و واقعی (ارائه نسخه قابل اجرا و تست شده از نرم‌افزار)
د) تأیید اینکه فرآورده‌ی نرم‌افزاری، مطابق طراحی انجام شده، کار می‌کند.
هـ) تأیید اینکه نیازمندی‌ها به درستی و به صورت مناسب پیاده‌سازی شده‌اند.

توجه: یک تفاوت اساسی میان دیسیپلین تست و سایر دیسیپلین‌های RUP این است که تست ضرورتاً با پیدا کردن و نشان دادن نقاط ضعف و کمبودهای فرآورده نرم‌افزاری در ارتباط می‌باشد. برای دستیابی به این هدف، باید نوعی دیدگاه فلسفه و ذهنیت خاص و متفاوت با دیسیپلین‌های نیازمندی‌ها، تحلیل و طراحی و پیاده‌سازی وجود داشته باشد. در حالی که این سه دیسیپلین بر کامل بودن، سازگاری و هماهنگی و صحت کار متمرکز می‌باشند، دیسیپلین تست بر پیدا کردن اشتباهات، ناسازگاری‌ها، نقایص و کمبودها تأکید دارد.

۶- دیسیپلین استقرار



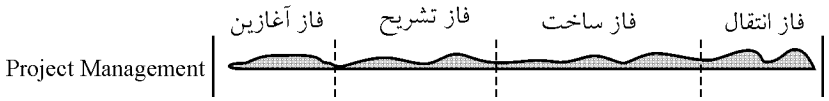
حجم فعالیت‌های دیسیپلین استقرار در طول فازهای مختلف چرخه تولید

اهداف دیسیپلین استقرار

الف) تست نرم‌افزار در محیط مشتری با شرایط عملیاتی نهایی (تست بتا)
ب) بسته‌بندی نرم‌افزار برای تحویل
ج) توزیع نرم‌افزار
د) نصب و راه‌اندازی نرم‌افزار
ه) آموزش کاربران نهایی و نیز آموزش نیروهای بازاریابی و فروش
ی) انتقال و جایگزینی نرم‌افزار موجود و یا تبدیل بانک اطلاعاتی

توجه: اغلب افراد تیم تولید نرم‌افزار با تکمیل پیاده‌سازی سیستم، پیروزی و موفقیت خود را اعلام می‌نماید، اما فراموش می‌شود که رضایت مشتری، تعیین‌کننده پایان موفق تولید یک فرآورده است، نه فقط داشتن یک کامپایل خوب!

۷- دیسیپلین مدیریت پروژه



حجم فعالیت‌های دیسیپلین مدیریت پروژه در طول فازهای مختلف چرخه تولید

اهداف دیسیپلین مدیریت پروژه

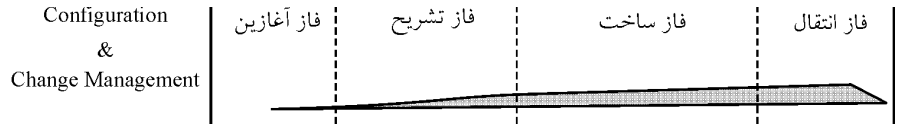
الف) فراهم کردن چارچوبی برای مدیریت پروژه‌های نرم‌افزاری

ب) ارائه رهنمودهای عملی برای برنامه‌ریزی، تخصیص نیروها، اجرا و نظارت بر پروژه‌ها
ج) فراهم نمودن چارچوبی برای مدیریت ریسک

توجه: به طور کلی مدیریت یک پروژه نرم‌افزاری عبارت است از هنر برقراری توازن میان اهداف رقابتی، مدیریت ریسک‌ها و غلبه بر محدودیت‌ها به منظور تحویل فرآورده‌ای نرم‌افزاری که مناسب و مطابق نیازمندی‌های مشتری نهایی می‌باشد.

توجه: در RUP رهنمودهایی برای انجام آسان‌تر و موفقیت‌آمیز فعالیت‌های مرتبط با دیسیپلین مدیریت پروژه فراهم شده است. البته این بدان معنا نیست که به این ترتیب موفقیت پروژه تضمین می‌شود، بلکه رویکرد مناسبی است که به طور محسوس، احتمال موفقیت در تحویل یک فرآورده نرم‌افزاری دارای کیفیت مطلوب را افزایش می‌دهد.

۸- دیسیپلین مدیریت پیکربندی و تغییرات



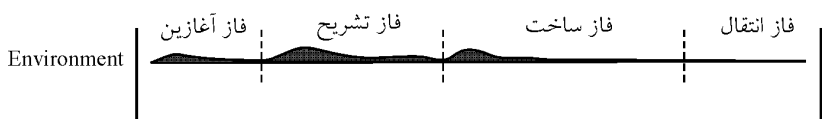
حجم فعالیت‌های دیسیپلین مدیریت پیکربندی و تغییرات در طول فازهای مختلف چرخه تولید

هدف اصلی دیسیپلین مدیریت پیکربندی و تغییرات

هدف دیسیپلین مدیریت پیکربندی و تغییرات، پیگیری و حفظ تمامیت دستاوردهای در حال تکامل در یک پروژه می‌باشد. در طی چرخه تولید، دستاوردهای با ارزش زیادی ایجاد می‌شود. توسعه‌ی این دستاوردها، مستلزم سرمایه‌گذاری و تلاش فراوانی است. بنابراین، این دستاوردها، دارایی‌های ارزشمندی محسوب می‌شوند که باید به خوبی نگهداری شده و امکان استفاده مجدد از آن‌ها فراهم شود. این دستاوردها، دائماً در حال رشد و تکامل می‌باشند و خصوصاً در رویکرد تکرارشونده، بارها و بارها بروزرسانی می‌شوند.

توجه: با وجودی که در RUP، مسئولیت هر دستاورد تنها بر عهده‌ی یک نفر می‌باشد، اما مسلم است که برای نگهداری دستاوردها و سوابق آنها، همیشه نمی‌توان به ذهن و حافظه‌ی اشخاص متکی بود. اعضای تیم پروژه باید قادر باشند به راحتی دستاوردهای مختلف و محل نگهداری آنها را شناسایی نموده، نسخه‌ی مناسبی از یک دستاورد را انتخاب کرده، وضعیت فعلی آن را از روی تاریخچه‌ی مرتبط با آن، تعیین نمایند و همچنین قادر به مشخص کردن مسئول فعلی آن دستاورد باشند.

۹- دیسیپلین محیط



حجم فعالیت‌های دیسیپلین محیط در طول فازهای مختلف چرخه تولید

هدف اصلی دیسیپلین محیط

هدف دیسیپلین محیط، پشتیبانی از سازمان تولیدکننده فرآورده، هم از جنبه فرآیند مناسب برای تولید و هم از نظر ابزارهای لازم برای تولید نرم‌افزار می‌باشد.

این پشتیبانی‌ها عبارتند از:

الف) انتخاب و فراهم کردن ابزارهای مناسب و ضروری

ب) نصب و پیکربندی ابزارهای موردنیاز

ج) پیکربندی فرآیند تولید

د) بهبود فرآیند

ه) ارائه‌ی خدمات فنی برای پشتیبانی از فرآیند، مانند فراهم نمودن زیرساخت فناوری اطلاعات، راهبری ابزارها و حساب‌های شخصی افراد مختلف تیم، تهیه‌ی پشتیبان و موارد مشابه آن.

پیکربندی RUP

چارچوب فرآیند RUP دربرگیرنده‌ی حجم زیادی از راهنمایی‌ها، دستاوردها، فعالیت‌ها و نقش‌ها می‌باشد. از آن جایی که استفاده از تمام این دستاوردها در هیچ پروژه‌ای ممکن نیست، لازم است زیرمجموعه‌ای از RUP را در هر پروژه به عنوان فرآیند و قالب پروژه، انتخاب نماییم.

این کار به وسیله انتخاب یا تولید یک پیکربندی از فرآیند RUP، متناسب با نیازمندی‌ها و الزامات یک پروژه خاص، انجام می‌شود. بدین منظور، می‌توان از یکسری پیکربندی‌های موجود استفاده نموده و یا اینکه یک پیکربندی خاصی را به طور کلی از ابتدا تولید نمود.

جریان کار

جریان کار، یک مدل گرافیکی (بصری) است، برای نمایش جریان منطقی و توالی مجموعه فعالیت‌های هر **دیسیپلین** که منجر به تولید یک نتیجه‌ی ارزشمند می‌شود.

به بیان دیگر جریان کار، مجموعه فعالیت‌ها، نقش‌ها و دستاوردها را در قالب توالی‌هایی که منجر به تولید نتایج ارزشمند می‌شوند، به هم مرتبط می‌سازند. این مدل در زبان مدل‌سازی UML و با استفاده از دیاگرام فعالیت بیان شده است.

تست‌های فصل دهم

۱- توصیف معماری سیستم از فرآورده‌های کدام‌یک از فازهای متدولوژی شیءگرای RUP می‌باشد؟ (مهندسی IT – آزاد ۸۵)

۱) فاز تشریح (Elaboration)
۲) فاز آغازین (Inception)
۳) فاز ساخت (Construction)
۴) فاز انتقال (Transition)

۲- نظم‌های (Disciplines) پشتیبانی در RUP عبارتند از: (مهندسی IT – آزاد ۸۵)

۱) مدل‌سازی کاری، جمع‌آوری نیازمندی‌ها، تحلیل و طراحی
۲) پیاده‌سازی، آزمایش، مدیریت پیکربندی
۳) مدیریت پروژه، مدیریت پیکربندی، محیط
۴) مدیریت پروژه، مدل‌سازی کاری، جمع‌آوری نیازمندی‌ها

۳- قابلیت‌های عملیاتی (Operational Capability) در کدام‌یک از فازهای RUP مورد رسیدگی قرار می‌گیرد؟ (مهندسی IT – آزاد ۸۶)

۱) ساخت (Construction)
۲) آغازین (Inception)
۳) تشریح (Elaboration)
۴) انتقال (Transition)

۴- کدام‌یک از فازهای چهارگانه RUP بر تحلیل و طراحی نرم‌افزار تأکید می‌نماید؟ (مهندسی IT – آزاد ۸۷)

۱) آغازین (Inception)
۲) تشریح (Elaboration)
۳) ساخت (Construction)
۴) انتقال (Transition)

۵- ساختارهای ایستا و پویا در RUP به ترتیب از راست به چپ منعکس‌کننده کدام دیدگاه‌ها می‌باشد؟ (مهندسی IT – آزاد ۸۸)

۱) دیدگاه‌های مدیریتی و دیدگاه‌های تکنیکی
۲) هر دو ساختار به دیدگاه‌های تکنیکی می‌پردازد.
۳) دیدگاه‌های تکنیکی و دیدگاه‌های مدیریتی
۴) دیدگاه مورد کاربری (Use Case) و دیدگاه مؤلفه (Component)

۶- کدام گزینه در خصوص مدیریت پیکربندی نرم‌افزار صحیح نمی‌باشد؟ (مهندسی IT – آزاد ۹۰)

۱) جریان کاری «مدیریت تغییر و پیکربندی» جزء جریان‌های کاری فرآیندی (Process workflows) در RUP محسوب می‌گردد.
۲) مدیریت پیکربندی نرم‌افزار عنصر مهمی از تضمین کیفیت نرم‌افزار است.
۳) مدیریت پیکربندی جهت کنترل خروجی‌های متعدد تولید شده توسط افرادی که روی یک پروژه فعالیت می‌نمایند، ضروری می‌باشد.

۴) مدیریت پیکربندی جهت تشخیص، سازماندهی و کنترل تغییرات در طول چرخه‌ی حیات نرم‌افزار مطرح می‌گردد.

۷- در کدام فاز فرآیند یکپارچه (Unified Process)، معمولا محصولی به نام مبنای قابل اجرای معماری (Executable Architectural Baseline) ایجاد می‌شود؟ (مهندسی IT – دولتی ۹۸)

۱) آغاز (Inception)
۲) انتقال (Transition)
۳) تفصیل (Elaboration)
۴) ساخت (Construction)

۸- در کدام یک از فازهای فرآیند یکپارچه (Unified Process)، محصولی به نام «مبنای قابل اجرای مربوط به معماری» (Executable Architectural Baseline) تولید می‌شود؟ (مهندسی IT – دولتی ۹۹)

۱) آغاز (Inception)
۲) انتقال (Transition)
۳) تفصیل (Elaboration)
۴) ساخت (Construction)

۹- فرض کنید به عنوان یک مهندس نرم‌افزار یک سیستم نرم‌افزاری در اندازه‌ی بسیار بزرگ در سطح کشور را طراحی و پیاده‌سازی می‌کنید. کدام مدل فرآیند (Process Model) را برای این پروژه مناسب می‌دانید؟ (مهندسی IT – دولتی ۹۹)

۱) مدل فازبندی شده (Phased Model)
۲) مدل آبشاری (Waterfall Model)
۳) مدل تکراری (Iterative Model)
۴) مدل موازی (Parallel Model)

۱۰- در کدام فاز از فرآیند یکپارچه (Unified Process)، ریسک‌های پروژه شناسایی و سنجیده می‌شوند؟ (مهندسی IT – دولتی ۱۴۰۰)

۱) آغاز (Inception)
۲) تفصیل (Elaboration)
۳) ساخت (Construction)
۴) انتقال (Transition)

پاسخ تست‌های فصل دهم

۱- گزینه (۱) صحیح است.

شناسایی و تشریح حداقل یک معماری برگزیده از فعالیت‌های فاز آغازین است.

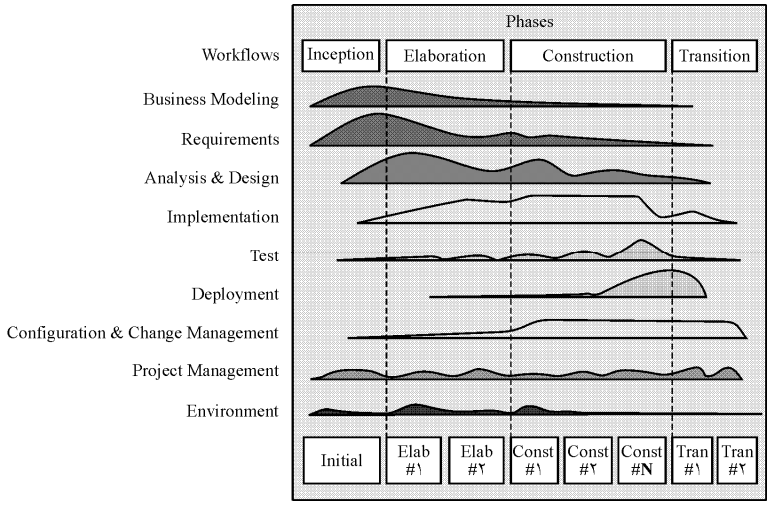
۲- گزینه (۳) صحیح است.

$$\text{Support disciplines} : \begin{cases} \text{project management} & \text{(مدیریت پروژه)} \\ \text{configuration \& change management} & \text{(مدیریت تغییر و پیکربندی)} \\ \text{environment} & \text{(محیط)} \end{cases}$$

۳- گزینه (۱) صحیح است.

محصول آماده است تا به تیم transition منتقل شود جهت بررسی اینکه آیا محصولات به اندازه کافی کامل شده‌اند یا خیر.

۴- گزینه (۲) صحیح است.



۵- گزینه (۳) صحیح است.

ساختار پویا نشان می‌دهد که فرآیند چگونه در قالب چرخه‌ها، فازها و تکرارها بیان شود. اما ساختار ایستا نشان می‌دهد که عناصر فرآیند (فعالیت‌ها، دیسیپلین، فرآورده‌ها و نقش‌ها) چگونه به صورت منطقی دسته‌بندی می‌شوند.

۶- گزینه (۳) صحیح است.

دیسیپلین مدیریت پیکربندی و تغییرات مربوط به دیسیپلین‌های پشتیبان از بعد جریان‌های

کاری (ایستا یا محتوایی) RUP می‌باشد. همچنین مدیریت پیکربندی پارامتر مهمی از تضمین کیفیت نرم‌افزار می‌باشد.

هدف اصلی دیسیپلین مدیریت پیکربندی حفظ، نگهداری و کنترل تغییرات مربوط به دستاوردهای (artifact) مربوط به پروژه می‌باشد و نه کنترل خروجی‌های تولید شده توسط افراد. نسخه‌های مختلف دستاوردهای مهم پروژه، باید تحت کنترل قرار گیرد. با تحول و تکامل دستاوردها، نسخه‌های متعددی از آنها به وجود می‌آید و ضروری است مکانیزم مناسبی برای شناسایی یک دستاورد، نسخه‌های مختلف و نیز تاریخچه تغییرات آن وجود داشته باشد.

بسیاری از دستاوردهای مختلف پروژه به هم وابسته‌اند. لذا بروز یک تغییر، سبب پدید آمدن اثری موج مانند می‌شود. هنگامی که یک دستاورد، دچار تغییر شده و تصحیح می‌گردد، دستاوردهای وابسته به آن نیز باید بازبینی شده و احتمالاً مورد بازبینی و تصحیح قرار گرفته و یا حتی ممکن است لازم باشد که دوباره از ابتدا تولید شوند.

برای مثال یک کد نوشته شده به زبان C++ به یک سری فایل‌های سرآیند وابسته است این کد همچنین به توصیف کلاسی که پیاده‌سازی آن را برعهده دارد، نیز وابسته است.

آخرین نسخه یک دستاورد، اغلب بهترین نسخه آن می‌باشد. اما در برخی از شرایط، موضوع به همین سادگی نیست و لازم است نسخه‌های مختلف از دستاوردها را نگهداری کنیم و به موقع، پایدارترین و بهترین نسخه را که لزوماً آخرین نسخه آن نیست، انتخاب نماییم. علاوه بر این، در بسیاری از موارد لازم است که چندین نفر به طور موازی روی یک دستاورد حاص کار کرده و یا شکل‌های مختلف یک دستاورد در فرآورده‌های مختلفی استفاده شود.

۷- گزینه (۳) صحیح است.

براساس فرایند RUP، یک پروژه نرم‌افزاری که با هدف تولید یک محصول نرم‌افزاری تعریف و اجرا می‌گردد، از نظر دینامیکی یا بعد زمانی دارای چهار فاز می‌باشد:

۱- فاز آغازین یا شناخت (Inception)

۲- فاز تفصیل یا تشریح یا معماری (Elaboration)

۳- فاز ساخت (Construction)

۴- فاز انتقال (Transition)

فاز تشریح به نقطه‌ی تصمیم‌گیری در رابطه با تثبیت معماری ختم می‌گردد. در این فاز، غلبه بر ریسک‌های فنی با تثبیت و مبنا قرار دادن معماری سیستم، امکان‌پذیر می‌گردد. اینکه RUP فاز دوم از چرخه تولید را به تثبیت معماری اختصاص داده است، نشان می‌دهد که تثبیت معماری، نقش کلیدی و بسیار مهمی در موفقیت پروژه دارد. به عبارت دیگر «**فاز تفصیل (Elaboration)، مبنای قابل اجرای معماری (Executable Architectural Baseline) است.**»

در این فاز، معماری سیستم با توجه به نیازمندی‌ها، ریسک‌ها و محدودیت‌های هزینه و زمان شکل گرفته و پس از انجام تست‌های لازم، تثبیت می‌گردد.

۸- گزینه (۳) صحیح است.

براساس فرایند RUP، یک پروژه نرم‌افزاری که با هدف تولید یک محصول نرم‌افزاری تعریف و اجرا می‌گردد، از نظر دینامیکی یا بعد زمانی دارای چهار فاز می‌باشد:

۱- فاز آغازین یا شناخت (Inception)

۲- فاز تفصیل یا تشریح یا معماری (Elaboration)

۳- فاز ساخت (Construction)

۴- فاز انتقال (Transition)

فاز تشریح به نقطه‌ی تصمیم‌گیری در رابطه با تثبیت معماری ختم می‌گردد. در این فاز، غلبه بر ریسک‌های فنی با تثبیت و مبنا قرار دادن معماری سیستم، امکان‌پذیر می‌گردد. اینکه RUP فاز دوم از چرخه تولید را به تثبیت معماری اختصاص داده است، نشان می‌دهد که تثبیت معماری، نقش کلیدی و بسیار مهمی در موفقیت پروژه دارد. به عبارت دیگر «**فاز تفصیل (Elaboration)، مبنای قابل اجرای مربوط به معماری (Executable Architectural Baseline) است.**»

در این فاز، معماری سیستم با توجه به نیازمندی‌ها، ریسک‌ها و محدودیت‌های هزینه و زمان شکل گرفته و پس از انجام تست‌های لازم، تثبیت می‌گردد.

۹- گزینه (۱) صحیح است.

مدل فازبندی شده (Phased Model) مانند RUP بر پایه تکرار و تکامل می‌باشد که مناسب پیاده‌سازی و توسعه‌ی پروژه‌های بسیار بزرگ در سطح کشوری است. RUP یک روش ساختاربندی شده برای تولید تکرارشونده فراهم می‌کند که یک پروژه را به چهار فاز یعنی فاز آغازین یا شناخت (Inception)، فاز تشریح یا معماری (Elaboration)، فاز ساخت (Construction) و فاز انتقال (Transition) تقسیم می‌کند. مدل موازی (Parallel Model) مانند مدل RAD مناسب پروژه‌های کوچک با لیست نیازمندی‌های ثابت و مشخص است.

۱۰- گزینه (۱) صحیح است.

براساس فرایند RUP، یک پروژه نرم‌افزاری که با هدف تولید یک محصول نرم‌افزاری تعریف و اجرا می‌گردد، از نظر دینامیکی یا بعد زمانی دارای چهار فاز می‌باشد:

۱- فاز آغازین یا شناخت (Inception)

۲- فاز تفصیل یا تشریح یا معماری (Elaboration)

۳- فاز ساخت (Construction)

۴- فاز انتقال (Transition)

در فاز آغازین یا شناخت (Inception) کلیه نیازمندی‌ها و محدوده سیستم به اندازه‌ی کافی شناخته می‌شوند. به بیان دیگر هدف اصلی این فاز، رفع ریسک‌های کلان سازمانی (**شناسایی و رفع ریسک‌های پروژه**)، ریسک‌های مربوط به درک اهداف و محدوده پروژه و نیز به دست آوردن اطلاعات کافی برای اطمینان از ادامه پروژه و یا توقف آن می‌باشد.

۱۱ متدولوژی‌های چابک

چابکی در مقابل لَختی و سستی قرار دارد. در یک بیان ساده فِرز و چابک بودن در انجام پروژه‌های نرم‌افزاری به معنی سریع‌بودن در انجام کارها است، اما نه به معنی نادیده گرفتن معیارهای موفق پروژه‌های نرم‌افزاری! (برآورده شدن نیازمندی‌های مشتری، مقرون به صرفه بودن و در زمان مورد انتظار آماده شدن)

پشت مجلل‌ترین اتومبیل هم که باشی، اما چابک نباشی، تصادف ممکن است، مرگبار باشد. چابکی یک هنر است، هنر استفاده از امکانات موجود به عالی‌ترین شیوه‌ی ممکن، هنر انجام صحیح یک راه‌حل، اما به شیوه‌ای هوشمندانه و پُرسرعت، بدون آنکه چیزی از قلم بیفتد.

مهندسان نرم‌افزار، ربات نیستند، آن‌ها تفاوت و شباهت‌هایی دارند، بعضی از آن‌ها نسبت به بعضی دیگر چابک‌تر هستند، و این نشأت گرفته از ماهیت انسانی بودن آن‌ها است، برخی باهوش‌تر و سریع‌تر هستند و این کاملاً مشهود است. در یک بیان ساده، چابکی، کاتالیزور فرآیند تولید نرم‌افزار است. بنابراین، چابکی، مقدمه می‌خواهد. یعنی باید مقدماتی فراهم گردد تا پروژه چابک پیش رود. برای‌مثال سازنده و مشتری هر دو چابک باشند.

پروژه چابک	=	مشتری چابک	+	سازنده چابک

به طور کلی شرایط چابکی و مقدمات چابکی عبارتند از:

مدل فرآیند تولید چابک: این مدل باید براساس رویکرد تکرار و تکامل و مبتنی بر مؤلفه شیءگرا باشد تا سازنده و مشتری روزانه و پیوسته با یکدیگر در ارتباط باشند و همچنین سبب شود تا رضایت مشتری از طریق تحویل نسخه‌های محصول به طور پیوسته و سریع با بالاترین اولویت برآورده گردد. همچنین مدل فرآیند تولید نرم‌افزار، شامل فعالیت‌های چارچوبی به عنوان یک نقشه‌ی حرکت، جهت هدایت و پیشرفت پروژه استفاده می‌گردد، و اینکه هر یک از فعالیت‌های چارچوبی شامل مراحل ارتباط، برنامه‌ریزی، مدل‌سازی (تحلیل و طراحی)، ساخت (پیاده‌سازی و تست) و استقرار به چه شکل و با چه حدی از چابکی انجام گردد، در چابک بودن

روند پیشرفت پروژه مؤثر است و نه ادغام و ترکیب فعالیت‌های چارچوبی همچون ادغام فعالیت‌های طراحی و ساخت.

روش چابک: روش شیءگرایی به دلیل استفاده از مفاهیمی همچنین کلاس به عنوان یک مؤلفه قابل استفاده مجدد، وراثت به عنوان یک عامل بازدارنده از دوباره‌نویسی کد، چندریختی به عنوان یک عامل بالابرنده خوانایی در برنامه، می‌تواند شکل و تمایل چابکی را به خود بگیرد. در روش ساخت‌یافته، چابکی خیلی کم‌تر دیده می‌شود.

توجه کنید که شیءگرایی و مفاهیم کلاس باعث می‌شود تا بتوان مؤلفه‌هایی با طراحی ایده‌آل با همبستگی بالا (Cohesion) و پیوستگی یا اتصال پایین (Coupling) در فعالیت طراحی مربوط به فعالیت‌های چارچوبی ایجاد کرد. هرچه قدر در طراحی مؤلفه‌ها همبستگی بالا و پیوستگی پایین باشد، فرآیند توسعه‌ی نرم‌افزار سریع‌تر خواهد بود و این یعنی چابکی.

ابزار چابک: UML، Nassi–Shneiderman diagram، Warnier-Orr Diagram، Petri net و SysML به دلیل مدل‌سازی خاص و منحصر به فرد خود می‌توانند شکل و شمایل چابکی را به خود بگیرد. مانند تولید کد سریع از مدل‌های طراحی ایجاد شده توسط UML به کمک رشنال رُز.

همانطور که پیش‌تر نیز گفتیم، متدولوژی شیءگرا یا مهندسی نرم‌افزار شیءگرا نظامی است یکپارچه شامل مدل فرآیند شیءگرا (مدرن)، روش شیءگرا (مبتنی بر مفاهیم کلاس، وراثت و چندریختی) و ابزارهای شیءگرا که منجر به ایجاد نرم‌افزاری در بازه‌ی زمانی از قبل برنامه‌ریزی شده، بودجه‌ای از قبل پیش‌بینی شده و دقیقاً مطابق با نیازمندی‌های واقعی مشتری می‌گردد.

توجه: متدولوژی RUP متداول‌ترین نمونه از متدولوژی شیءگرا براساس روش شیءگرا (مبتنی بر مفاهیم کلاس، وراثت و چندریختی)، مدل فرآیند مبتنی بر مؤلفه‌ی شیءگرا با رویکرد تکرار و تکامل و ابزارهای شیءگرا (مثل ابزار مدل‌سازی UML و زبان برنامه‌نویسی C++) می‌باشد.

توجه: نسبت نمونه متدولوژی شیءگرای RUP به متدولوژی شیءگرا، مثل نسبت سیستم عامل ویندوز مدرن به مفاهیم مدرن سیستم عامل است.

اما متدولوژی شیءگرای چابک یا مهندسی نرم‌افزار شیءگرای چابک نظامی است یکپارچه شامل مدل فرآیند شیءگرای چابک (مدرن‌تر و سریع‌تر)، روش شیءگرای چابک (مبتنی بر مفاهیم کلاس، وراثت و چندریختی) و ابزارهای شیءگرای چابک که منجر به ایجاد نرم‌افزاری در بازه‌ی زمانی از قبل برنامه‌ریزی شده، بودجه‌ای از قبل پیش‌بینی شده و دقیقاً مطابق با نیازمندی‌های واقعی مشتری به شکلی چابک می‌گردد. این تعریف، فرآیند تولید پروژه‌های نرم‌افزاری را به سمت موفقیت و چابکی سوق می‌دهد.

توجه: متدولوژی‌های XP[1]، Scrum، Crystal، ASD[2]، DSDM[1] و AUP[2] متداول‌ترین نمونه از

[1] Extreme Programming
[2] Adaptive Software Development

متدولوژی‌های شیءگرای چابک براساس روش شیءگرای چابک (مبتنی بر مفاهیم کلاس، وراثت و چندریختی)، مدل فرآیند مبتنی بر مؤلفه‌ی شیءگرای چابک با رویکرد تکرار و تکامل و ابزارهای شیءگرای چابک (مثل ابزار مدل‌سازی UML، Nassi–Shneiderman diagram، Warnier-Orr Diagram، Petri net، SysML و زبان برنامه‌نویسی C++ و C#) می‌باشد.

توجه: نسبت نمونه متدولوژی‌های شیءگرای چابک XP، Scrum، Crystal، ASD، DSDM و AUP به متدولوژی شیءگرای چابک، مثل نسبت سیستم عامل ویندوز مدرن‌تر و سریع‌تر به مفاهیم مدرن‌تر و سریع‌تر سیستم عامل است.

متدولوژی‌های چابک از طریق یک فرآیند تکرار شونده، به تحویل زودهنگام نرم‌افزار در قالب افزایش‌های مختلف و جلب حداکثری رضایت مشتری می‌پردازند. هر افزایش یک نسخه اجرایی از نرم‌افزار است که در طی فرآیند نرم‌افزار به شکل تکرارشونده به مشتری تحویل می‌شود. با گذشت زمان افزایش‌های نرم‌افزار، کامل و کامل‌تر شده تا به شکل نرم‌افزار نهایی تبدیل شود. در رویکرد مبتنی بر تکرار و تکامل، فرصت یادگیری و بهبود تدریجی در سرتاسر چرخه‌ی تولید فراهم است. بدین ترتیب، در طول پروژه، امکان تصحیح به موقع اشتباهات وجود خواهد داشت. در صورت بروز اشتباه در یک تکرار، امکان جبران آن در تکرار بعدی وجود دارد. رویکرد مبتنی بر تکرار و تکامل به برنامه‌ریزی مستمر و پویا در طول پروژه نیازمند است. رویکرد مبتنی بر تکرار و تکامل در هر تکرار نسخه‌هایی از نرم‌افزار را ارائه می‌دهد که هریک از قبلی کامل‌تر است. به بیان دیگر در مدل‌های تکاملی در هر دور از تکرار نسخه‌ی کامل‌تری از نرم‌افزار تولید می‌شود. در متدولوژی‌های چابک، تحویل سریع‌تر نرم‌افزار ترجیح بیشتری به انجام مراحل تحلیل و طراحی دارد، اگرچه به هیچ وجه نباید از انجام مراحل تحلیل و طراحی صرف نظر کرد. علت اصلی استفاده از متدولوژی‌های چابک این است که نرم‌افزارهای جدید مدام درحال تغییر و تحول هستند، علت این تغییرات مکرر، نیاز نرم‌افزار جهت تطابق با قوانین و نیازهای محیط عملیاتی خودش و یا پیشرفت تکنولوژی است. متدولوژی‌های چابک ضمن ارائه‌ی زودهنگام نرم‌افزار، نسبت به تغییرات نرم‌افزار نیز به دلیل رویکرد تکرار و تکامل بسیار منعطف است. در متدولوژی‌های چابک فعالیت‌های توسعه‌ی نرم‌افزار با جزئیات کمتری نسبت به متدولوژی‌های قدیمی‌تر انجام می‌شود. در واقع تنها وظایف ضروری حاضر در فعالیت‌های اصلی یا framework activity مورد توجه قرار گرفته و از انجام دیگر وظایف غیرضروری چشم پوشی می‌شود. به همین دلیل نام دیگری که برای متدولوژی‌های چابک پیشنهاد می‌شود، مهندسی نرم‌افزار سبک یا software engineering lite است.

در بحث مدیریت منابع انسانی و غیرانسانی نیز، این مدیریت باید چابک باشد، به عبارت دیگر

---

[1] Dynamic System Development Method
[2] Agile Unified Process

پروژه‌های نرم‌افزاری به واسطه فعالیت‌های چارچوبی و فعالیت‌های چتری ایجاد می‌گردند. به بیان دیگر مطابق آنچه پیش از این نیز گفتیم نه تنها فعالیت‌های چارچوبی (فرآیند تولید نرم‌افزار) باید چابک باشد، بلکه فعالیت‌های چتری نیز باید چابک باشد، برای‌مثال یکی از فعالیت‌های چتری، پشتیبان‌گیری از منابع انسانی (مانند مدیران رده بالا) و غیرانسانی (مانند اطلاعات پروژه) است که باید به شکلی چابک و سریع و تا دیر نشده است انجام گردد.

همچنین در مورد تیم‌های متدولوژی‌های چابک باید گفت که این تیم‌ها به صورت خود سازمانده (Self-Organizing) تشکیل شده و اعضای آن‌ها با اختیارات بیشتری نسبت به دیگر تیم‌های نرم‌افزاری عمل می‌کنند. که این خود سازماندهی سبب چابکی تیم‌های کاری می‌گردد. تیم خودسازمانده خودش کنترل کارهایش را برعهده دارد. این تیم خودش به تعهداتش عمل می‌کند و طرح‌هایی برای انجام آنها تعریف می‌کند. هیچ چیز به این اندازه انگیزه را در یک تیم از بین نمی‌برد که آدم دیگری برایش تعیین تکلیف کند و هیچ چیز به اندازه‌ی پذیرش مسئولیت برای تعیین وظایف و انجام تعهدات در تیم ایجاد انگیزه نمی‌کند.

باز هم تأکید می‌کنیم که چابکی، مقدمه می‌خواهد، مقدمات چابکی باید فراهم گردد، تا پیشرفت پروژه چابک باشد، در یک بیان ساده، طوری تیم‌ها و کارها را سازماندهی کنید که همه چیز چابک و سریع باشد و عاملی نتواند روند رو به جلوی پیشرفت پروژه را مختل نماید، به هر دلیلی.

در اقتصاد نوین، پیش‌بینی چگونگی تکامل یافتن یک سیستم کامپیوتری، برای مثال یک برنامه‌ی کاربردی مبتنی بر وب، در گذر زمان، غالبا کاری دشوار است. شرایط بازار به سرعت تغییر می‌کند، نیازهای کاربران نهایی تکامل می‌یابد و تهدیدهای رقابتی بدون هشدار قبلی ظهور می‌کند. در بسیاری شرایط، قادر به تعریف کامل خواسته‌ها قبل از شروع پروژه نخواهید بود. باید به قدر کافی چابک باشید تا بتوانید به یک محیط عملیاتی متغییر پاسخ دهید.

تغییر، هزینه‌بردار است. به ویژه اگر کنترل نشده باشد یا مدیریت ضعیفی بر آن اعمال شود. یکی از بارزترین ویژگی‌های متدولوژی‌های چابک، توانایی آن در کاهش دادن هزینه‌های ناشی از تغییر در سرتاسر فرآیند نرم‌افزار است.

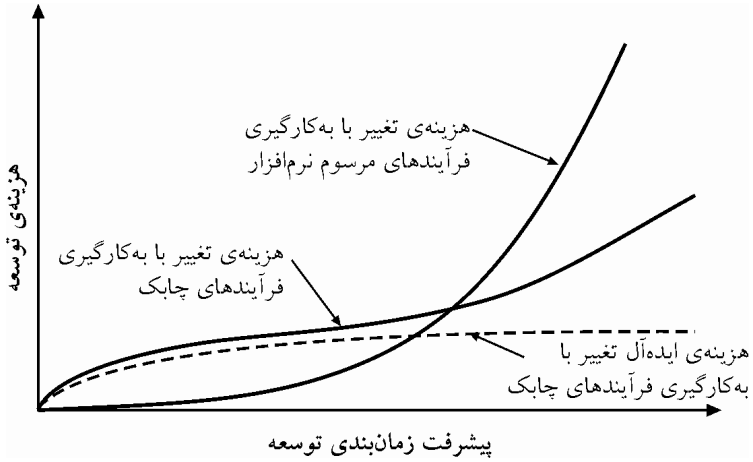
آیا این شرایط بدان معناست که شناخت چالش‌های پدید آمده از یک واقعیت جدید، باعث می‌شود که همه‌ی اصول، مفاهیم، روش‌ها و ابزارهای ارزشمند مهندسی نرم‌افزار را نادیده بگیرید؟ مطقا خیر! مهندسی نرم‌افزار نیز همانند کلیه رشته‌های مهندسی به تکامل خود ادامه می‌دهد و می‌توان آن را طوری تطبیق داد که چالش‌های ناشی از تقاضا برای سرعت را نیز پاسخگو باشد.

این روزها هنگام توصیف یک فرآیند نرم‌افزار مدرن‌تر، **چابکی** واژه‌ای است که به وفور به گوش می‌رسد. تیم چابک تیمی فِرز و چالاک است که قادر است به تغییرات پاسخ مناسب بدهد. تغییر چیزی است که در توسعه نرم‌افزار، بسیار با آن مواجه می‌شویم. تغییرات در نرم‌افزارهایی که درحال ساخته‌شدن هستند، تغییرات در اعضای تیم، تغییرات به دلیل تکنولوژی جدید، و یا هر نوع تغییری که ممکن است بر محصول درحال ساخت یا محصولی که محصول نهایی را ایجاد می‌کند، تاثیرگذار باشند. هرآنچه در نرم‌افزار انجام می‌دهیم باید خود حاوی ویژگی‌هایی برای پشتیبانی از

تغییرات باشد، چیزی که قلب و روح نرم‌افزار است. تیم چابک می‌داند که نرم‌افزار توسط افرادی توسعه می‌یابد که در قالب تیمی کار می‌کنند و مهارت‌های این افراد و توانایی ایشان در **همکاری**، هسته‌ی اصلی موفقیت پروژه است. چابکی، بیشتر خاصیت خود را مرهون دانش نهفته در تیم است و نه در دانش نوشته شده روی کاغذ.

چابکی و هزینه‌های تغییر

عقل سلیم در توسعه نرم‌افزار که چند دهه تجربه پشتیبان آن است، حکایت از آن دارد که هزینه‌ی تغییر به صورت غیرخطی با پیشرفت پروژه افزایش می‌یابد. پاسخگویی به تغییر در فعالیت ارتباطات یعنی هنگامی‌که تیم نرم‌افزاری مشغول جمع‌آوری خواسته‌هاست، نسبتا آسان و با هزینه اندک است. و زمان مورد نیاز برای پاسخ به این نوع تغییرات تاثیری عمیق بر زمان‌بندی پروژه نخواهد داشت. اما اگر چند ماه جلوتر برویم، چطور؟ فرض کنید تیم درحال انجام فعالیت تست است، چیزی که نسبتا در اواخر پروژه رخ می‌هد و ناگهان مشتری درخواست تغییری عمده در قابلیت‌های سیستم را دارد. این تغییر نیاز به اصلاح لیست نیازمندی‌ها، مدل تحلیل، مدل طراحی، پیاده‌سازی و تست دارد، برای‌مثال ساخت سه قطعه جدید و اصلاح و ویرایش پنج قطعه قدیم، و به تبع موردهای تست جدید نیز باید طراحی شوند. در این شرایط هزینه‌ها به سرعت بالا می‌رود و زمان لازم برای حصول اطمینان از اینکه تغییرات بدون برجای گذاشتن اثرات جانبی اعمال شود، قابل چشم پوشی نخواهد بود.



یک فرآیند چابک با رعایت اصول طراحی معماری ایده‌آل (پنهان‌سازی اطلاعات، افزایش انسجام، کاهش اتصال و استقلال عملیاتی) می‌تواند، هزینه تغییر را **تسطیح** کند. به این ترتیب، تیم نرم‌افزاری قادر به پاسخگویی به تغییرات در اواخر پروژه خواهد بود، بدون اینکه ضربه‌ای قابل ملاحظه از نظر زمان و هزینه به پروژه وارد آید. همانطور که پیش‌تر گفتیم، متدولوژی‌های چابک شامل تحویل افزایشی محصول می‌شود. هنگامی‌که تحویل افزایشی با سایر توصیه‌های چابک از

قبیل تست واحد مستمر و برنامه‌نویسی زوجی (دو نفری بر روی یک کامپیوتر) تلفیق شود، زمان و هزینه اعمال تغییرات بیشتر هم کاهش می‌یابد.

فرآیند چابک

فرآیند چابک باید از انطباق‌پذیری برخوردار باشد. ولی انطباق پیوسته و بدون پیشروی از موفقیت چندانی برخوردار نیست. بنابراین، در یک فرآیند نرم‌افزار چابک، باید روند انطباق به طور افزایشی انجام پذیرد. برای دستیابی به انطباق افزایشی و تدریجی، تیم نرم‌افزاری چابک نیاز به بازخورد از مشتری دارد، تا بتواند انطباق‌های لازم را انجام دهد. یک عامل شتاب‌دهنده به دریافت بازخورد مشتریان، نمایش نمونه‌ای اولیه از سیستم عملیاتی به مشتری است. از این رو، راهبرد توسعه افزایشی را باید نهادینه ساخت. نسخه‌های نرم‌افزار یعنی نمونه‌های اولیه قابل اجرا یا بخش‌هایی از سیستم عملیاتی باید در دوره‌های زمانی کوتاه مدت به مشتری تحویل داده شوند تا روند انطباق بتواند همگام با روند تغییرات ادامه یابد. مشتری به کمک این رویکرد مبتنی بر تکرار و تکامل می‌تواند، هر نسخه از نرم‌افزار را مرتبا ارزیابی کند، بازخوردهای لازم را برای تیم نرم‌افزاری فراهم سازد و بر انطباق‌های به عمل آمده جهت پاسخگویی به بازخوردها تاثیر بگذارد.

اصول چابکی

در پیمان چابک (Agile Alliance) دوازده اصل برای کسانی که می‌خواهند به چابکی دست پیدا کنند، تعریف شده است.

۱- جلب رضایت مشتری از طریق تحویل زودهنگام و پیوسته نرم‌افزارهای ارزشمند، بیشترین اولویت را نزد ما دارد.
۲- پذیرا بودن تغییرات در خواسته‌ها حتی در اواخر فرآیند توسعه. که مزیتی رقابتی مابین شرکت‌های سازنده نرم‌افزار است.
۳- تحویل پیوسته‌ی نرم‌افزارهای کاری از دو هفته گرفته تا دو ماه، که بازه‌های زمانی کوتاه‌تر باید در اولویت قرار داده شوند.
۴- دست‌اندرکاران و افراد تجاری باید در سرتاسر پروژه هر روز باهم کار کنند.
۵- سپردن پروژه به افراد باانگیزه، فراهم‌سازی محیط و پشتیبانی مورد نیاز آنها و اطمینان‌کردن به آنها در انجام کارها.
۶- اثربخش‌ترین و موثرترین روش انتقال اطلاعات به درون و بیرون تیم توسعه، گفتگوی رو در رو است.
۷- نرم‌افزاری که به درستی کار کند، میزان اصلی در سنجش پیشرفت کاری است.
۸- فرآیندهای چابک، توسعه پایدار را ارتقا می‌بخشند. سازندگان و مشتریان باید قادر به حفظ سرعت ثابت در مسیر پیشرفت کار باشند.
۹- توجه پیوسته به اعتلای فنی و طراحی معماری ایده‌آل (پنهان‌سازی اطلاعات، افزایش انسجام، کاهش اتصال و استقلال عملیاتی)، باعث افزایش چابکی می‌شود.

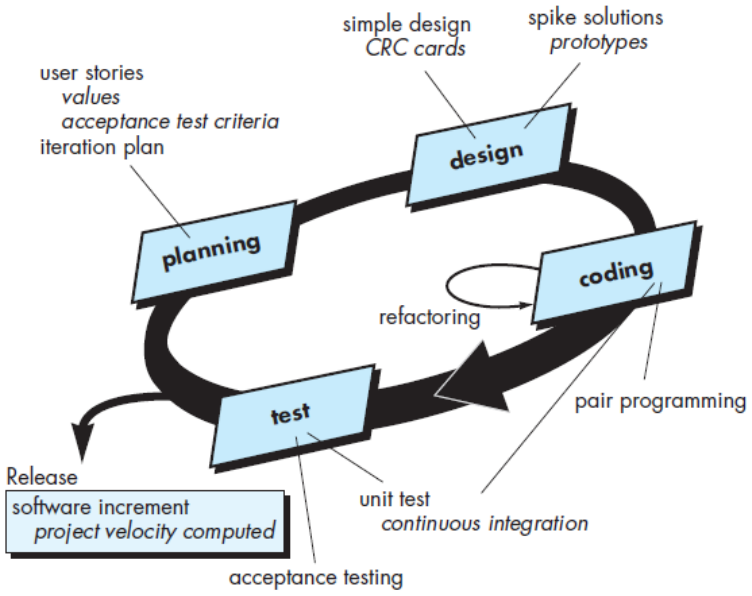
۱۰- ساده‌سازی ضروری است، یعنی هنر به انجام رساندن کارهای انجام ناپذیر.
۱۱- بهترین دستاورد از فعالیت‌های چارچوبی(ارتباطات، برنامه‌ریزی، مدل‌سازی، ساخت و استقرار) از تیم‌های خود سازمانده ظهور می‌کنند. به بیان دیگر بهترین تحلیل‌ها، طراحی‌ها، معماری‌ها و شناسایی نیازها از تیم‌های خود سازمانده منتج می‌شود.
۱۲- تیم در بازه‌های زمانی منظم، بازخوردی از میزان بهبود اثربخشی خود ارائه می‌دهد و سپس رفتار خود را مطابق این بازخورد تنظیم می‌کند.

این دوازده اصل در تمامی فرآیندهای چابک با وزن مساوی به کار برده نمی‌شوند و در برخی مدل‌ها از اهمیت یک یا چند اصل چشم پوشی می‌شود یا دست کم نقش آنها کمرنگ‌تر می‌شود. این اصول تعیین کننده هسته و جوهره چابکی هستند که در مدل‌های فرآیند چابک اغلب حفظ خواهند شد.

توجه: چابکی به معنی نادیده‌گرفتن مهندسی نرم‌افزار نیست. نباید بین چابکی و مهندسی‌نرم‌افزار یکی را انتخاب کرد، بلکه باید یک رویکرد مهندسی نرم‌افزار انتخاب شود که چابک هم باشد! اما روش‌شناسان **سنتی** یک مُشت آدم‌های فاقد خلاقیت هستند که ترجیح می‌دهند مستنداتی بدون نقص تهیه کنند تا اینکه سیستمی کاری ارائه دهند که نیازهای مشتری را برآورده سازد. مکتب چابکی به معنی خروج از مکتب مهندسی نرم‌افزار نیست، بلکه اتفاقا چابکی به معنی ورود و دقت بیشتر بر اصول اساسی و طراحی معماری ایده‌آل مهندسی نرم‌افزار است اما اینبار با سرعت و چابکی بیشتر. با در نظر گرفتن بهترین ایده‌ها از هر دو مکتب، بیشترین بهره عاید خواهد شد و چیزی از تخریب دیگری، به‌دست نخواهد آمد.

متدولوژی XP

متدولوژی XP یا متدولوژی برنامه‌نویسی حدی (Extreme Programming) پرکابردترین رویکرد در توسعه نرم‌افزار به شیوه چابک است. این متدولوژی با نام‌های XP و یا X-programming نیز شناخته می‌شود. علت نامگذاری این متدولوژی به برنامه‌نویسی حدی این است که نسبت به متدولوژی‌های دیگر، مرحله‌ی برنامه‌نویسی را با تاکید بیشتری انجام می‌دهد. متدولوژی XP از روش شیءگرا جهت توسعه برنامه استفاده می‌کند. در این متدولوژی فعالیت‌های چارچوبی (framework activities) شامل چهار فعالیت برنامه‌ریزی، طراحی، برنامه‌نویسی و تست می‌باشد. به تازگی، شکل دیگری از XP موسوم به IXP (Industrial Extreme Programming) به معنی XP صنعتی پیشنهاد شده است. IXP پالایشی از XP است که فرآیند چابک را مشخصا برای استفاده در سازمان‌های بزرگ هدف قرار داده است. شکل زیر گویای روند کارکرد متدولوژی XP است:



برنامه‌ریزی (Planning)

فعالیت برنامه‌ریزی با گوش‌سپردن (listening) به خواسته‌های مشتری توسط سازنده طی جلساتی غیررسمی و شفاهی مابین سازنده و مشتری آغاز می‌شود. این جلسات منجر به بیان خواسته‌ها توسط مشتری و درک و جمع‌آوری خواسته‌ها توسط سازنده می‌شود. خواسته‌های شفاهی مشتری در قالب use caseها قرار می‌گیرند، سپس در ادامه برای هر use case (مورد کاربرد یا نیاز) سناریو یا شرح حال نوشته می‌شود.

سناریو بر دو طبقه اصلی و فرعی می‌باشد:

سناریوی اصلی: بیان روال حرکت قدم به قدم کارها داخل یک use case از نقطه شروع تا رسیدن به یک نتیجه مطلوب (موفق). (حرکت از خود موجود و رسیدن به خود مطلوب).

سناریوی فرعی: بیان روال حرکت قدم به قدم کارها داخل یک use case از نقطه شروع تا رسیدن به یک نتیجه نامطلوب (ناموفق). (حرکت از خود موجود و رسیدن به خود نامطلوب).

توجه: هر use case، فقط و فقط یک سناریوی اصلی دارد و می‌تواند چندین سناریوی فرعی داشته باشد. سناریوها بر روی کارت شاخص (Index Card) نوشته می‌شوند.

در سناریوهای اصلی، خروجی‌ها و ورودی‌های نرم‌افزار، خصوصیات نرم‌افزار و کارکردهای نرم‌افزار مشخص می‌شود. مشتری باید برای هر سناریو اولویتی را مشخص کند. از آنجاکه متدولوژی XP بر پایه رویکرد تکرار و تکامل است، این اولویت‌بندی سناریوها توسط مشتری، منجر به شناخت اولویت پیاده‌سازی سناریوها توسط سازنده در هر تکرار و افزایش می‌شود، ضمن اینکه در هر تکرار می‌توان سناریوهای جدیدی به پروژه اضافه نمود و یا سناریوهایی از پروژه

حذف کرد. پس از درک سناریوهای مشتری توسط سازنده و اولویت‌بندی آنها توسط مشتری، سازنده هزینه و زمان لازم جهت پیاده‌سازی هر سناریو را بر حسب تعداد هفته‌های لازم مشخص می‌کند. اگر تعداد هفته‌های لازم جهت توسعه هر سناریو بیش از سه هفته بود، از مشتری خواسته می‌شود که سناریوی مورد نظر را به سناریوهای کوچکتر تقسیم کند و دوباره آنها را اولویت‌بندی کند. سازنده و مشتری با توافق یکدیگر سناریوهایی که در یک تکرار باید توسعه یابد را اولویت‌بندی و مشخص می‌کنند، اما ترتیب توسعه سناریوها در یک تکرار بر حسب شرایط پروژه بر اساس یکی از سه رویکرد زیر تعیین می‌شود:

۱- همه سناریوهای موجود در یک تکرار پیاده‌سازی شود.
۲- برخی از سناریوهای موجود در یک تکرار پیاده‌سازی شود، به این شکل که ابتدا سناریوهایی با اولویت بالاتر پیاده‌سازی شوند. بنابراین اگر یک تکرار یا افزایش تحت فشار زمان‌بندی قرار گرفت، این مزیت وجود دارد که سناریوهای پراولویت‌تر پیاده‌سازی شده‌اند.
۳- برخی از سناریوهای موجود در یک تکرار پیاده‌سازی شود، به این شکل که ابتدا سناریوهایی که ریسک بیشتری دارند پیاده‌سازی شوند. بنابراین اگر یک تکرار یا افزایش تحت فشار زمان‌بندی قرار گرفت، این مزیت وجود دارد که سناریوهای پرریسک‌تر پیاده‌سازی شده‌اند.

همانطور که گفتیم، متدولوژی XP بر اساس رویکرد تکرار و تکامل پیش می‌رود، بنابراین اندازه‌گیری‌های انجام‌شده بر روی سناریوهای انجام شده در تکرار و افزایش قبلی، مبنایی برای کار برنامه‌ریزی شامل فعالیت‌های «برآورد میزان کار»، «برآورد زمان لازم برای انجام کار»، «برآورد هزینه لازم برای انجام کار»، «مدیریت ریسک» و «زمان‌بندی» در افزایش و تکرار بعدی است. آینده نزدیک، بسیار شبیه به گذشته نزدیک است.

طراحی (Design)

طراحی در متدولوژی XP برپایه‌ی سادگی بنا شده است، این اصل که سادگی را حفظ‌کن (keep it simple). در متدولوژی XP همواره یک طراحی ساده بر یک طراحی پیچیده برتری دارد. در این متدولوژی صرفا طراحی سناریوهایی انجام می‌شود که نیاز وضع موجود است و قرار به پیاده‌سازی قطعی آن است، بنابراین طراحی سناریوهایی که نیاز به آن در آینده محتمل است و قرار به پیاده‌سازی آن قطعی نیست، در این متدولوژی جایگاهی ندارد. متدولوژی XP در جهت حفظ سادگی در فعالیت طراحی خود فقط از کارت‌های CRC استفاده می‌کند. پس از شناسایی موارد کاربرد و سناریونویسی برای هر یک از موارد کاربرد، زمان تعریف کلاس‌ها، صفات، متدها و ارتباطات میان کلاس‌های همکار برای هر یک از موارد کاربرد می‌رسد. برای شناسایی کلاس‌ها، صفات، متدها و ارتباطات میان کلاس‌ها برای هر یک از موارد کاربرد، از تکنیکی موسوم به CRC که سرواژه‌ی عبارت Class – Responsibility Collaborator و به معنی «مدل همکاری مسئولیت‌های کلاس‌ها» می‌باشد، استفاده می‌شود. مدل‌سازی CRC روشی ساده جهت تعیین و

سازماندهی کلاس‌های داخل هر مورد کاربرد است. اگر در بخشی از طراحی یک سناریو، سختی ایجاد شود، متدولوژی XP، ایجاد فوری یک نمونه اولیه عملیاتی (operational prototype) را برای آن بخش از طراحی توصیه می‌کند. استفاده از نمونه اولیه عملیاتی که موسوم به راهکار خیزشی (spike solution) نیز می‌باشد، دو پیامد دارد، اول اینکه، این راهکار منجر به کاهش ریسک فنی در پیاده‌سازی واقعی سناریوی مورد نظر می‌شود و دوم اینکه، این راهکار منجر به اعتبارسنجی برآوردهای اولیه یعنی «برآورد میزان کار»، «برآورد زمان لازم برای انجام کار»، «برآورد هزینه لازم برای انجام کار»، «مدیریت ریسک» و «زمان‌بندی» مربوط به سناریوی مورد نظر می‌شود.

کدنویسی (Coding)

پس از کشف سناریوهای مشتری و انجام‌شدن کارهای طراحی مقدماتی، تیم توسعه به کدنویسی نمی‌پردازد، بلکه یک سری «آزمون واحد» تهیه می‌کند تا هریک از سناریوهایی که قرار است در نسخه جاری یعنی تکرار فعلی گنجانده شود، مورد تست و بررسی قرار گیرد. هنگامی‌که آزمون واحد تهیه شد، سازنده بهتر می‌تواند توجه خود را به آن چیزی معطوف کند که باید پیاده‌سازی شود تا آزمون را با موفقیت پشت سر بگذارد. هنگامی‌که کدها کامل شدند، می‌توان تست واحدی را بلافاصله انجام داد و در نتیجه، بازخوردی فوری را حاصل کرد.

یکی از اصلی‌ترین مفاهیم متدولوژی XP، برنامه‌نویسی زوجی (pair programming) است که در مرحله کدنویسی قرار دارد. متدولوژی XP توصیه می‌کند که برای تولید کد هر سناریو، دو نفر از اعضای تیم توسعه بر روی یک کامپیوتر اقدام به کدنویسی نمایند. زیرا دو فکر غالباً بهتر از یک فکر است، مانند حضور مؤلف، هنگام تایپ یک کتاب، که به کشف زود هنگام خطاها کمک می‌کند. درواقع زمانی‌که برنامه توسط یکی از برنامه‌نویسان نوشته می‌شود، برنامه‌نویس دیگر در همان لحظه کد را با هدف کشف خطاهای آن بازبینی (review) می‌کند. که به آن تضمین کیفیت بی‌درنگ نیز گفته می‌شود.

به موازاتی که برنامه‌نویسان زوجی کار خود را کامل می‌کنند، کدی که می‌نویسند در کنار کار دیگران قرار داده می‌شود. در برخی موارد، این کار به صورت روزانه توسط تیم یکپارچه‌کننده (جامعیت) انجام می‌شود. در موارد دیگر، برنامه‌نویسان زوجی، مسئولیت جامعیت تدریجی (افزایشی) را نیز برعهده دارند. این راهبرد یعنی «جامعیت تدریجی پیوسته» به پرهیز از مشکلات سازگاری و ایجاد واسط کمک می‌کند و یک محیط تست دود (smoke testing) فراهم می‌سازد که به کشف زودهنگام خطاها کمک می‌کند.

همانطور که پیش‌تر گفتیم، طراحی در متدولوژی XP برپایه‌ی سادگی بنا شده است، بنابراین این دغدغه وجود دارد که گاهی طراحی‌های XP، از کیفیت الگوریتمی (غیروظیفه‌مندی یا مزه) مناسبی برخوردار نباشد. متدولوژی XP برای مرتفع نمودن این دغدغه، روش فاکتورگیری مجدد یا بازآرایی (refactoring) را پیشنهاد می‌کند. بنابراین با وجود اینکه روش فاکتورگیری مجدد در مرحله کدنویسی انجام می‌شود اما ماموریت آن ارتقاء کیفیت الگوریتمی و خوشمزه‌سازی طراحی

است. فاکتورگیری مجدد یا بازآرایی، فرآیند تغییر، بهبود و ارتقاء کیفیت کد است، بازآرایی، کد را خواناتر می‌کند، بازآرایی، کد را تغییرپذیرتر می‌کند، بازآرایی، ساختار داخلی کد را بهبود می‌دهد اما بدون آنکه بازآرایی، رفتار بیرونی کد را اصلاح کند، به عبارت دیگر بازآرایی، رفتار بیرونی کد را اصلاح نمی‌کند یعنی ورودی‌ها و خروجی‌های کد را تغییر نمی‌دهد.

بازآرایی، یک روش مناسب جهت اصلاح، ساده‌سازی و بهبود کد است. این کار احتمال وجود خطا در کد را کاهش می‌دهد. واضح است که عمل بازآرایی پس از تولید کد باید انجام شود. بازآرایی منجر به این می‌شود که علاوه بر بهبود ساختار داخلی کد، فعالیت طراحی نیز در طول فعالیت کدنویسی دستخوش تغییر، تحول و بهبود شود. یعنی هم قبل از کدنویسی و هم بعد از کدنویسی طراحی درحال انجام است. شعار طراحی قبل از کدنویسی «سادگی» است، اما شعار طراحی بعد از کدنویسی «کیفیت» است. و این کدنویسی است، که راهنمایی لازم برای چگونگی بهبود بخشیدن به طراحی را فراهم می‌سازد.

تست (Testing)

شروع زودهنگام و غیررسمی آزمون واحد و آزمون جامعیت (یکپارچگی) در مرحله کدنویسی است، اما ادامه آن و شروع رسمی آزمون واحد، آزمون جامعیت (یکپارچگی) و آزمون اعتبارسنجی در مرحله تست است. آزمون واحد **صحت عملکرد یا روند اجرای** یک بخش، یک جزء یا یک مولفه از نرم‌افزار را مستقل از سایر مولفه‌های دیگر برنامه مورد تست و ارزیابی قرار می‌دهد. آزمون جامعیت یا یکپارچه‌سازی، اعتبارسنجی عملکرد یا روند اجرای چندین مولفه در کنار یکدیگر و یا کل مولفه‌های نرم‌افزار ایجاد شده در کنار یکدیگر به همراه **نقاط اتصال** مابین آن‌ها را مورد تست و ارزیابی قرار می‌دهد. در یک نرم‌افزار ممکن است هر یک از مؤلفه‌های برنامه به تنهایی درست عمل کنند اما ترکیب آنها با یکدیگر به درستی عمل نکند. زیرا ممکن است به دلیل عدم تنظیم درست نقاط اتصال و ارتباطی مابین مولفه‌ها، این مولفه‌ها قادر نباشند در کنار یکدیگر، کارکرد مورد انتظار را نشان دهند. بنابراین آزمون جامعیت، جهت تست تدریجی یکپارچه‌شدن مولفه‌های برنامه در کنار یکدیگر مورد استفاده قرار می‌گیرد. منظور از نقاط اتصال مابین مولفه‌های مرتبط با هم، همان کانال یا محل تبادل داده مابین مولفه‌های مرتبط با هم هست، مانند فراخونی با مقدار یا فراخوانی با ارجاع. اگر این نقاط اتصال مابین مولفه‌های مرتبط با هم درست تنظیم نشوند ممکن است داده‌ها در گذر از این نقاط اتصال از بین بروند مانند حالتی که یک متغیر دوبایتی به یک متغیر یک بایتی در روش فراخوانی با مقدار پاس داده می‌شود، در واقع در این حالت بخشی از داده‌ها از دست رفته است. این نقاط اتصال به واسط مولفه نیز موسوم است. واضح است که، برطرف کردن مشکلات کوچک در هر چند ساعت یکبار، نسبت به برطرف‌کردن مشکلات بزرگ درست قبل از پایان مهلت، زمان کمتری می‌برد، بنابراین بهتر است آزمون جامعیت به شکل روزانه و به روش آزمون دود انجام شود. پس از انجام بازآرایی کد (refactoring) نیز می‌بایست آزمون جامعیت به روش آزمون رگرسیون انجام شود. آزمون اعتبارسنجی، اعتبارسنجی عملکرد یا روند اجرای کل مولفه‌های نرم‌افزار ایجاد شده در کنار یکدیگر را در شرایط **سهل و آسان** مورد تست و

ارزیابی قرار می‌دهد. هنگامی‌که یک **نرم‌افزار سفارشی** تنها برای یک مشتری توسعه می‌یابد، «**آزمون پذیرش**[1]» اجرا می‌شود تا مشتری را قادر به اعتبارسنجی کلیه‌ی خواسته‌ها کند. آزمون پذیرش، که به جای مهندس نرم‌افزار، توسط مشتری انجام می‌شود، می‌تواند از یک آزمون غیررسمی تا یک سری آزمون‌های برنامه‌ریزی شده‌ی سیستماتیک را شامل شود. درواقع، آزمون پذیرش را می‌توان در عرض یک هفته یا یک ماه انجام داد و لذا کشف خطاهایی که ممکن است سیستم را با گذشت زمان تنزل دهد، امکان‌پذیر می‌شود. در متدولوژی XP، آزمون اعتبارسنجی توسط آزمون پذیرش انجام می‌شود. مبنای آزمون پذیرش توسط مشتری، کنترل کارکرد همان سناریوهایی است که توسط مشتری در هر تکرار یا افزایش بیان شده‌است.

ارزش‌های متدولوژی XP

در متدولوژی XP پنج ارزش تعریف شده است که مبنایی برای انجام صحیح همه کارهای موجود در متدولوژی XP است. هرکدام از این ارزش‌ها به عنوان محرکه‌ای برای فعالیت‌ها، کنش‌ها و وظایف XP به کار می‌رود، این ارزش‌ها به صورت زیر است:

۱- **ارتباطات (communication)**: برقراری ارتباطات نزدیک به شکل رو در رو و غیررسمی مابین سازنده و مشتری جهت مشخص‌سازی خواسته‌های دقیق و واضح مشتری و پرهیز از برقراری ارتباط با مشتری برپایه تهیه مستندات پرحجم. هدف اصلی این ارتباطات شناخت نیازها و تعیین سناریوهای مشتری (customer stories) است.

۲- **سادگی (Simplicity)**: متدولوژی XP برای دستیابی به سادگی، به سازندگان توصیه می‌کند که فقط نیازهای فعلی بررسی و طراحی شوند و بررسی و طراحی نیازهای آینده به آینده واگذار شود و به حال آورده نشود. طراحی نیز باید ساده باشد و اگر نیاز به بهبود بود، بازآرایی می‌شود.

۳- **بازخورد (Feedback)**: در متدولوژی XP تیم توسعه باید توجه ویژه‌ای به بازخورد محصول، مشتری و حتی خود تیم توسعه داشته باشد. با ایجاد هر کلاس مربوط به یک use case آزمون واحد بر روی هر کلاس مطابق قابلیت‌های مورد انتظار انجام می‌شود و به تدریج کارکرد کلاس‌های همکار درون یک use case در کنار هم تحت بررسی آزمون جامعیت (یکپارچگی) قرار می‌گیرد. در نهایت نیز آزمون اعتبارسنجی توسط آزمون پذیرش بر روی نحوه پیاده‌سازی سناریوهای موجود در use caseهای تحویلی در هر افزایش انجام می‌شود. همچنین در هنگام برنامه‌ریزی افزایش بعدی، بازخورد برنامه‌ریزی افزایش قبلی مورد استفاده قرار می‌گیرد.

۴- **شجاعت (Courage) یا انظباط (discipline)**: پایبندی سفت و سخت به برخی جنبه‌های متدولوژی XP به جسارت و جرات نیاز دارد. یک واژه‌ی بهتر می‌تواند انظباط

[1] Acceptance Test

باشد. اغلب تمایل و فشار برای انجام طراحی نیازهای آینده وجود دارد. اکثر تیم‌های نرم‌افزاری هم سر تسلیم فرود می‌آورند، با این استدلال که طراحی برای آینده در بلندمدت منجر به صرفه‌جویی در زمان و تلاش می‌شود. اما تیم XP چابک باید انظباط و شجاعت لازم برای طراحی در امروز و زمان حال را داشته باشد. و در عین بداند که خواسته‌های آینده ممکن است به طرز چشمگیری تغییر کند که طراحی و پیاده‌سازی آنها در زمان حال منجر به دوباره‌کاری و اتلاف زمان و هزینه‌ها می‌گردد.

۵- **احترام (Respect)**: تیم چابک، با محقق‌کردن هریک از ارزش‌های فوق، احترام متقابل میان سازنده و مشتری را برقرار می‌کند. با تحویل موفق هریک از نسخه‌های نرم‌افزار این احترام و اعتقاد بر موثر بودن متدولوژی چابک XP بیشتر و بیشتر نیز می‌شود.

متدولوژی Scrum

نام متدولوژی Scrum از بازی راگبی گرفته‌شده است. راگبی ورزش گروهی است که در یک زمین چمن مستطیل شکل با دروازه‌های H شکل در دو سوی زمین و با یک توپ بیضی شکل بازی می‌شود. اسکرام در واقع یک شروع دوباره در بازی راگبی است به طوری که وقتی خطایی در این بازی روی می‌دهد یا اینکه توپ از زمین خارج می‌گردد از روش اسکرام برای آغاز دوباره بازی استفاده می‌شود. به این شکل که همه بازیکن‌ها دور هم جمع می‌شوند، سرهایشان را پایین می‌گیرند و شروع مجدد می‌کنند، اسکرام در بازی راگبی بسیار اتفاق می‌افتد.

اسکرام یک روش گروهی برای تولید و توسعه نرم‌افزار است. اینکه در هریک از فعالیت‌های چارچوبی شامل ارتباط، برنامه‌ریزی، تحلیل، طراحی، پیاده‌سازی، تست و استقرار چه وظایفی باید انجام شود، توسط sprintها مشخص می‌شود که در ادامه به آن می‌پردازیم.

نقش‌های اسکرام (scrum roles)

- **اسکرام مستر (scrum master) :** رهبر اسکرام وظیفه دارد تا تمامی اعضای تیم را هدایت و راهنمایی نماید تا هیچ یک از اعضای تیم از چارچوب و قوانین اسکرام خارج نشوند. رهبر اسکرام نقش مدیر را ندارد بلکه تنها وظیفه رهبری تیم را بر عهده دارد تا با رفع مشکلات و موانع پیش رو، در صورتی که اعضای تیم قادر به رفع موانع نباشند، اجرای اسکرام را بهبود بخشد.
- صاحب محصول (product owner) : صاحب محصول که نماینده ذینفعان (Stakeholders) پروژه و business است، با اعلام دقیق نیازمندی‌های خود به تیم تولید، با رهبر اسکرام و تیم تولید همکاری می‌نماید. صاحب محصول باید به سوالات تیم تولید پاسخ داده و همواره در دسترس باشد.
- **تیم تولید و توسعه نرم‌افزار (development team) :** افراد این تیم در چارچوب قوانین اسکرام، به تولید آن چه که صاحب محصول درخواست کرده است، می پردازند. تعداد اعضای تیم تولید نه باید آن قدر کم باشد که همکاری گروهی و کار تیمی بی‌معنا شود و نه

آن قدر زیاد باشد که هماهنگی بین اعضای تیم تبدیل به امری دشوار و وقت‌گیر گردد. تعداد اعضای تیم تولید، بستگی به پروژه دارد اما معمولا ۶ تا ۹ نفر اعضای این تیم را تشکیل می‌دهند.

روند کارکرد اسکرام

هسته اصلی اسکرام را sprintها تشکیل می‌دهند. در متدولوژی‌های تکرارشونده (iterative) دوره‌های زمانی تکراری (iteration) وجود دارد که در این دوره‌ها به تدریج محصول کامل می‌گردد. بدین صورت که در تولید یک محصول، تعدادی تکرار در نظر گرفته می‌شود که در پایان دوره‌ی زمانی هر تکرار، یک محصول قابل ارائه وجود دارد. به این دوره‌های زمانی تکرارشونده در اسکرام sprint می‌گویند. در پایان هر sprint، محصول کامل‌تر شده و در نهایت محصول نهایی تولید می‌گردد. هر sprint دارای تعریفی است که در آن باید مشخص شده باشد که چه چیزی قرار است ساخته شود، نیازمندی‌ها، راهنمای ساخت و محصول خروجی نیز باید مشخص باشند. نتیجه اینکه مثل تمام متدولوژی‌های incremental و iterative در اسکرام نیز دوره‌های زمانی یا iteration داریم که در طی آنها محصول نهایی پروژه به‌تدریج تکمیل می‌شود.

در طی یک sprint که معمولا یک دوره دو تا چهار هفته است یعنی حداکثر ۳۰ روز (طول دوره را تیم مشخص می‌کند) اعضاء، یک محصول قابل ارائه و قابل استفاده را تدریجا تولید می‌کنند.

اصطلاح Product Backlog نامی است که به لیست نیازمندهای وظیفه‌مندی و غیروظیفه‌مندی کل یک پروژه اطلاق می‌شود و در حقیقت مجموعه‌ای اولویت‌بندی شده از نیازمندی‌های مشتری است که در نهایت بایستی تحویل داده شود.

مواردی از Product Backlog که در طی یک sprint بایستی انجام شود، در طول جلسه‌ی طراحی sprint یا Sprint Planning Meeting مشخص می‌شود. در طول این جلسه، صاحب محصول (Product Owner)، تیم تولید و توسعه نرم‌افزار (development team) را درباره‌ی مواردی از Product Backlog آگاه می‌کند. سپس اعضاء تیم تولید مشخص می‌کنند که چه مقدار از موارد مشخص شده توسط Product Owner را می‌توانند در این sprint انجام دهند و چه میزان از آنرا در sprint های بعدی.

مواردی از Product Backlog که قرار است در یک Sprint انجام شود را اصطلاحا Sprint Backlog می‌نامند. مفاد Sprint Backlog در واقع توافقی است بین اعضاء تیم تولید و Product Owner، که بعد از تصویب‌شدن مفاد یک sprint، دیگر هیچ‌کس نمی‌تواند آنرا در طول sprint تغییر دهد. در طول دوره، نیازمندی‌های لحاظ شده در Sprint Backlog از Product Backlog حذف می‌شوند. اما امکان دارد به دلایلی که در ادامه می‌آید این نیازمندی‌ها دوباره به Product Backlog برگردد.

در ساده‌ترین روش معمولا از نرم‌افزارهای صفحه گسترده (Spread Sheet) همچون Microsoft Excel برای ساختن و نگهداری Product Backlog و Sprint Backlog استفاده

می‌شود، اما می‌توان از طیف وسیعی از ابزارهای نرم‌افزاری که برای استفاده در تیم‌های Agile نوشته شده‌اند نیز استفاده کرد.

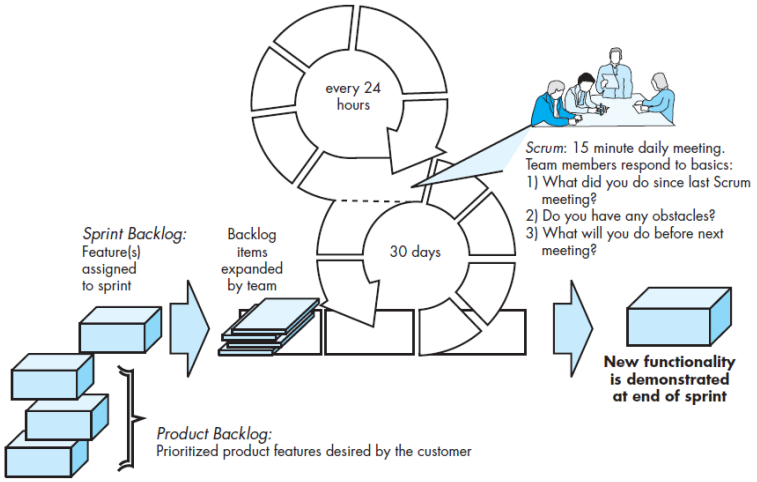
مانند تمام متدولوژی‌های iterative، توسعه‌ی نرم‌افزار در اسکرام نیز Time Boxed است، به این معنی که sprint بایستی دقیقا سر وقت تمام شود و اگر نیازمندی‌های تعیین شده در Sprint Backlog به هر علتی تکمیل نشده باشند آنها را کنار گذاشته و دوباره وارد Product Backlog می‌کنند.

در طول اجرای هر sprint جلساتی روزانه (هر ۲۴ ساعت) و کوتاه حدود ۱۵ دقیقه با هدف کشف زودهنگام خطاها و بررسی پیشرفت پروژه توسط تیم اسکرام و با حضور اعضای تیم (تیم تولید و ذینفعان) برگزار می‌شود که به آن جلسات اسکرام (scrum meeting) گفته می‌شود.

در این جلسات باید به سه پرسش زیر پاسخ داده شود:

- از جلسه قبل تا این جلسه چه کارهایی انجام شده است؟
- از جلسه قبل تا این جلسه با چه موانعی مواجه شده‌اید؟
- از این جلسه تا جلسه بعدی چه کارهایی قرار است انجام شود؟

اگرچه جلسات اسکرام به نسبت کوتاه است، اما تاثیر جلسات بر روند بهبود فرآیند قابل چشم‌پوشی نیست. کشف زودهنگام خطاهای بالقوه از مهم‌ترین مزایای این جلسات است. این موضوع منجر به این می‌شود که تیم پروژه زمانی خطا را مرتفع کند که هزینه و زمان مورد نیاز برای رفع آن هنوز مقداری قابل قبول است. مزیت دیگر این جلسات آن است که دانش عمومی تیم، نسبت به کار یکدیگر و کل کاری که در sprint جاری انجام می‌شود، بالا رود. این موضوع باعث استحکام کار تیمی خصوصا در تیم‌های خودسازمانده (self-organizing) می‌شود. جلسات اسکرام توسط رهبر اسکرام (scrum master) هدایت می‌شود. رهبر اسکرام ضمن هدایت جلسات، مسئولیت ارزیابی پاسخ‌های اعضا در این جلسات را نیز برعهده دارد. شکل زیر گویای روند کارکرد متدولوژی اسکرام است:

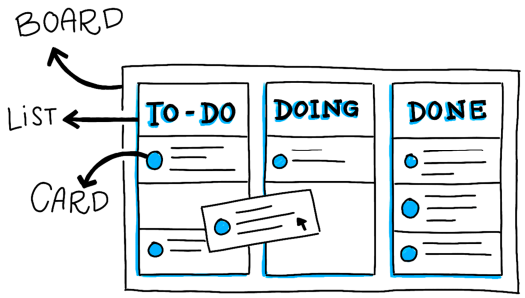


بعد از خاتمه یک sprint، اعضاء تیم طی جلسه‌ای به Product Owner و سایر ذینفعان پروژه نشان می‌دهند که چکار کرده‌اند و چطور از نسخه‌ی جاری نرم‌افزار می‌شود استفاده کرد. در واقع پس از اتمام هر sprint کارکردهای جدید که در طول این sprint ساخته‌شده‌اند، در قالب یک افزایش جدید از نرم‌افزار به صاحب محصول و ذینفعان ارائه می‌شود. همانطور که پیش‌تر نیز گفتیم، کارکردهای جدید هر افزایش، دقیقا برابر تمام کارکردهایی که در ابتدای sprint برنامه‌ریزی شده بود، نیست. در واقع گاهی محدودیت زمانی sprint باعث می‌شود در مواقعی که sprint از زمان‌بندی عقب است، برخی از کارکردها در آن افزایش پیاده‌سازی نشوند. در این شرایط پیاده‌سازی این کارکردها به افزایش‌های بعدی سپرده می‌شود. به مرحله‌ی ارائه‌ی کارکردهای جدید نرم‌افزار در انتهای هر sprint یک demo گفته می‌شود. پس از تحویل هر افزایش یا sprint، افزایش یا sprint بعدی برنامه‌ریزی می‌شود. همچنین در هنگام برنامه‌ریزی sprint بعدی، بازخورد برنامه‌ریزی sprint قبلی مورد استفاده قرار می‌گیرد.

نتیجه اینکه از مجموع جلسات اسکرام یک Sprint بوجود می‌آید و از مجموع این sprintها هم کل فرآیند توسعه‌ی نرم‌افزار ایجاد می‌گردد. متدولوژی اسکرام برای انجام پروژه‌های پراهمیت که تحت فشار زمان‌بندی بوده و همواره درحال تغییر هستند، مناسب و کارآمد است.

متدولوژی Kanban

تابلوی Kanban در متدولوژی چابک Kanban مورد استفاده قرار می‌گیرد. کانبان (Kanban) واژه‌ای ژاپنی به معنای «نشانه بصری» یا «کارت» است که کمپانی تویوتا آن را برای نخستین بار وارد فضای مدیریت کرد. تابلوی Kanban به بهبود مهارت مدیریت زمان کمک می‌کند. در اغلب موارد متدولوژی چابک Scrum با متدولوژی چابک Kanban ادغام می‌شود. شکل زیر گویای روند کارکرد متدولوژی Kanban است:



متدولوژی چابک Kanban چهارچوبی مصور است که برای پیاده‌سازی و اجرای مدیریت پروژه چابک استفاده می‌شود و نشان می‌دهد که چه محصولی، در چه زمانی و به چه مقدار باید تولید گردد. همانطور که گفتیم روش کانبان از سیستم تولید شرکت تویوتا و تولید ناب الهام گرفته شده است. در سال ۱۹۴۰ تویوتا پروسه‌ی مهندسی خود را با مدل‌سازی آن بر اساس چگونگی عملکرد قفسه‌های سوپرمارکت‌ها بهبود بخشید. مهندس تایچی اوهنو (Taiichi Ohno) متوجه این

نکته شد که سوپرمارکت‌ها تنها به اندازه‌ای محصول ذخیره می‌کنند که پاسخگوی تقاضای مشتری باشد، که موجب بهینه‌سازی جریان میان سوپرمارکت و مشتری می‌شود. موجودی نیز تنها زمانی ذخیره می‌شود که فضای خالی روی قفسه‌ها موجود باشد (نشانه بصری). تویوتا همین اصول ساده را به کارخانه خود راه داد. تیم‌های مختلف کارت‌هایی (یا کانبان) می‌ساختند تا اعلام کنند که ظرفیت اضافی دارند و برای دریافت مواد بیشتر آماده‌اند. روش کانبان گاهی سیستم کششی (pull system) نیز نامیده می‌شود چرا که تمام بخش‌های آن از سفارشات دریافت می‌شود.

امروزه همین ایده‌ها و اصول بر روی تیم‌های نرم‌افزاری و پروژه‌های فناوری اطلاعات نیز اعمال می‌شود. در این زمینه، توسعه کار در حال انجام (WIP) جای موجودی را می‌گیرد و کار جدید تنها زمانی اضافه می‌شود که فضای خالی بر روی تخته بصری تیم (team's visual Kanban board) وجود داشته باشد. کانبان میزان کار درحال انجام (WIP) را با ظرفیت تیم تطبیق می‌دهد که موجب افزایش انعطاف، شفاف‌سازی و تولید می‌شود. کانبان یک تکنیک برای مدیریت یک پروسه توسعه نرم‌افزاری توسط روشی با اثربخشی بالاست. زیربنای روش کانبان، سیستم تولید همزمان تویوتا (JIT: Just In Time) است. هرچند که توسعه نرم‌افزار یک فعالیت خلاقانه و مبتکرانه و متفاوت از تولید انبوه اتومبیل است؛ اما همچنان مکانیزم پایه برای مدیریت خط تولید می‌تواند بر روی آن پیاده‌سازی شود. تخته کانبان ابزاری برای پیاده‌سازی متدولوژی کانبان در پروژه هاست. اصولا این ابزار یک تخته فیزیکی است که دارای آهن ربا، چسب‌های پلاستیکی یا برگه‌هایی با قابلیت چسبیدن بر روی تخته کانبان است و یا گاهی به صورت دستی آیتم‌های کاری بر روی تخته نوشته می‌شوند. با این حال در سال‌های اخیر نرم‌افزارهای مدیریت پروژه بسیاری این تخته را به صورت آنلاین ارائه داده‌اند.

یک تخته کانبان، چه به صورت فیزیکی و چه آنلاین، از ستون‌ها و سطرهای متفاوتی تشکیل شده است. ساده‌ترین شکل این تخته‌ها شامل سه ستون است: ستون انجام دادن (To Do)، در حال انجام (Doing) و انجام شده (Done). ستون‌های مربوط به یک پروژه توسعه نرم‌افزاری ممکن است شامل ستون‌های بانک اطلاعاتی (Backlog)، آماده انجام (ready)، کدزنی (Coding)، آزمایش (Testing)، تأیید (approval) و انجام شده (Done) باشد.

کارت‌های کانبان (مانند یادداشت‌های چسبانده) نشان دهنده کارها است و هر کارت بر روی تخته نصب و در ستونی قرار می‌گیرد تا بازگو کننده وضعیت کار مورد نظر باشد. برای مثال کاری که هنوز شروع نشده است و در مرحله پیش از انجام است باید در ستون انجام دادن (To Do) قرار گیرد. این کارت‌ها می‌توانند با یک نگاه وضعیت کارها را به شما بازگو کنند. می‌توان از کارت‌هایی با رنگ‌های مختلف برای نشان دادن جزئیات بیشتر استفاده کرد. برای مثال کارت سبز می‌تواند نشان دهنده یک ویژگی باشد و یا رنگ نارنجی نشان دهنده یک وظیفه یا عملیات.

طبیعت تصویری بودن روش کانبان یک مزیت منحصر به فرد برای چابک‌سازی پروژه می‌دهد. عملکرد تخته کانبان بسیار ساده و سریع بوده و گردش کار را بهبود می‌بخشد و زمان چرخه را کاهش می‌دهد. اسکرام بر پایه ترکیب تکرار دوره‌های ثابت زمانی با برنامه‌ریزی، بهبود فرآیند و انتشار شکل گرفته است. اما در کانبان هیچ چهارچوب زمانی در نظر گرفته نشده است. اسکرام در

مقابل تغییر مقاوم است، در حالی که کانبان به سادگی با هر نوع تغییری مطابقت پیدا می‌کند. در اسکرام زمانی که تیم، داستان‌های کاربری (User Story) را به اسپرینت‌ها (Sprints) متصل می‌کنند، اضافه کردن داستان کاربری جدید به اسپرینت امکان‌پذیر نخواهید بود. اما در کانبان می‌توان داستان‌های کاربری را در هر زمان اضافه و یا ویرایش کرد.

متدولوژی FDD

متدولوژی FDD سرواژه عبارت Feature-Driven Development و به معنی توسعه مبتنی بر ویژگی است. متدولوژی‌های چابک بر پایه شی‌گرایی هستند، چون با شی‌گرایی می‌توان چابک بود.

متدولوژی FDD همانند سایر روش‌های چابک به موارد زیر تاکید دارد:

(۱) بر همکاری میان اعضای تیم FDD تاکید دارد.

(۲) پیچیدگی مساله و پروژه را با استفاده از تجزیه مبتنی بر ویژگی‌ها و سپس منسجم ساختن نسخه‌های نرم‌افزار مدیریت می‌کند.

(۳) ارتباط میان جزئیات فنی را با استفاده از ابزارهای لفظی، تصویری و متنی برقرار می‌سازد. FDD با تشویق راهبرد توسعه‌ی افزایشی، استفاده از وارسی طراحی و کدها، به کارگیری ممیزهای تضمین کیفیت نرم افزار، جمع‌آوری معیارها، و به کارگیری الگوها (برای تحلیل، طراحی و ساخت)، بر فعالیت‌های تضمین کیفیت نرم افزار تاکید دارد.

توجه: در حیطه‌ی FDD، ویژگی (Feature) «یک عملکرد است که نزد متقاضی دارای ارزش بوده و در کمتر از دو هفته قابل پیاده‌سازی است.» تاکید بر تعریف ویژگی‌ها مزایای زیر را به همراه دارد:

- چون ویژگی‌ها قطعات کوچکی از قابلیت‌های قابل تحویل‌اند، کاربران راحت‌تر می‌توانند آنها را توصیف کنند؛ چگونگی ارتباط آنها با یکدیگر را بهتر درک کنند و بهتر آنها را از نظر ابهام، خطا یا موارد جا افتاده بازبینی کنند.
- ویژگی‌ها را می‌توان در قالب گروه‌های سلسله مراتبی و بر اساس ارتباط تجاری میان آنها سازمان‌دهی کرد.
- از آنجا که هر ویژگی یک نسخه‌ی قابل تحویل در FDD به شمار می‌رود، تیم باید عملکردها را هر دو هفته یک‌بار توسعه دهد.
- از آنجا که ویژگی‌ها کوچک هستند، وارسی موثر طراحی و کدهای آنها آسان‌تر است.
- سلسه مراتب ویژگی‌هاست که برنامه‌ریزی، زمان‌بندی و پیگیری پروژه را به پیش می‌برد نه مجموعه‌ای از وظایف مهندسی نرم‌افزار که به دلخواه تعیین شده باشد.

قالب زیر برای تعریف یک ویژگی (توصیف نیازمندی‌های عملکردی) استفاده می‌شود:

<action> the **<result>** **<by | for | of | to>** a(n) **<object>**

که در این قالب‌بندی، < شیء > یک «شخص، مکان یا هر یا هر چیز دیگری (از جمله نقش‌ها، لحظاتی از زمان یا بازه‌های زمانی، یا توصیفی شبه کاتالوگی)» می‌تواند باشد. مثال‌هایی از

ویژگی‌های مربوط به یک نرم‌افزار تجارت الکترونیک در زیر داده شده است:

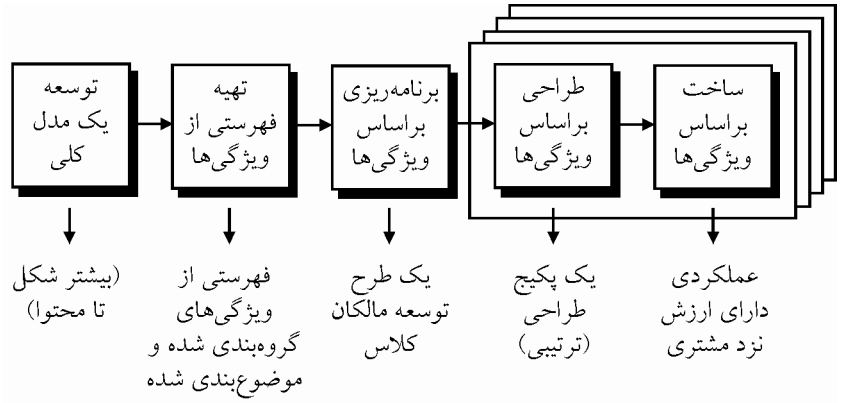
- افزودن محصول به سبد خرید
- نمایش مشخصات فنی محصول
- نگهداری اطلاعات حمل برای مشتری

در مجموعه ویژگی‌ها، ویژگی‌ها در قالب گروه‌هایی با ارتباط تجاری دسته‌بندی می‌شوند، مجموعه ویژگی‌ها به صورت زیر تعریف می‌شود:

<action> <ing> a(n) <object>

برای مثال، به فروش رساندن یک محصول، مجموعه ویژگی‌هایی است که شامل ویژگی‌های ذکر شده در قبل و ویژگی‌های دیگر می‌شود.

در رویکرد FDD پنج فعالیت یا فرآیند چارچوبی «مبتنی بر همکاری» تعریف می‌شود که در شکل زیر نشان داده شده‌اند:



در FDD بیش از هر روش چابک دیگر، بر تکنیک‌ها و دستورالعمل‌های مدیریت پروژه تاکید می‌شود. با رشد اندازه و پیچیدگی پروژه‌ها، مدیریت پروژه به شیوه‌ای موردی غالبا ناکافی به نظر می‌رسد. درک وضعیت پروژه، اینکه چه پیشرفت‌هایی انجام شده است و چه مشکلاتی به بار آمده است، برای سازندگان، مدیران و سایر ذی‌نفع‌ها ضروری است. اگر فشار مهلت زمانی چشمگیر باشد، تعیین اینکه آیا افزایش‌های نرم افزاری (ویژگی‌ها) به خوبی زمان‌بندی شده‌اند، اهمیت بسیار دارد. FDD برای این منظور، شش نقطه‌ی عطف در طول طراحی و پیاده‌سازی یک ویژگی تعیین کرده است: «بررسی در طراحی، طراحی، بازرسی طراحی، کدنویسی، بازرسی کد، ارتقا به ساخت»

تست‌های فصل یازدهم

۱- کدام مورد به عنوان یک متدولوژی شناخته می‌شود؟ (مهندسی IT – دولتی ۸۸)

۱) ORACLE ۲) 4GT ۳) RAD ۴) X-Programming

۲- کدام‌یک از گزینه‌های زیر از اصول رسیدن به چابکی نیست؟ (مهندسی IT – دولتی ۹۳)

۱) شرط چابکی ادغام طراحی و ساخت است.
۲) تولیدکنندگان و مشتریان باید روزانه و پیوسته با یکدیگر همکاری کنند.
۳) بهترین طراحی‌ها، معماری‌ها و نیازها از تیم‌های خود سازمانده منتج می‌شود.
۴) رضایت مشتری از طریق تحویل نسخ محصول بطور پیوسته و سریع بالاترین اولویت را دارد.

۳- کدام عبارت در مورد بازآرایی کد (Refactoring) در XP، هیچگاه درست نیست؟
(مهندسی IT – دولتی ۹۶)

۱) بازآرایی، کد را خواناتر می‌کند.
۲) بازآرایی، کد را تغییرپذیرتر می‌کند.
۳) بازآرایی رفتار بیرونی کد را اصلاح می‌کند.
۴) بازآرایی ساختار داخلی کد را بهبود می‌دهد.

۴- کدام محصول، به طور معمول در روش XP تولید می‌شود؟ (مهندسی IT – دولتی ۹۷)

۱) کارت‌های CRC ۲) مدل پیکربندی ۳) نمودارهای حالت ۴) تابلوی Kanban

---

۵- در کدام مدل فرآیندی فقط از قالب زیر برای توصیف نیازمندی‌های عملکردی استفاده می‌شود؟

<action> the <result> <by | for | of | to> a(n) <object>

(مهندسی IT – دولتی ۹۸)

۱) FDD ۲) RUP ۳) DSDM ۴) Scrum

۶- کدام گزینه در مورد جلسات روزانه متدولوژی اسکرام درست است؟ (مهندسی IT – دولتی ۹۹)

۱) جلسات حداکثر یک ساعته هستند.
۲) جلسات در غیاب استاد اسکرام (Scrum Master) برگزار می‌شوند.
۳) در ابتدای هر جلسه، محصول ساخته شده به مشتری نمایش داده می‌شود.
۴) در جریان هر جلسه، هر یک از اعضای تیم به سه سوال خاص پاسخ می‌دهند.

۷- کدام‌یک از موارد زیر، یکی از اصول چابکی است؟ (مهندسی IT – دولتی ۱۴۰۰)

۱) از تغییر در نیازمندی‌ها استقبال کنید.
۲) بدهی فنی در صورت لزوم قابل قبول است.
۳) مستندسازی در طراحی باید حذف شود.
۴) معماری باید در دیرترین زمان ممکن تعیین شود.

پاسخ تست‌های فصل یازدهم

۱- گزینه (۴) صحیح است.

ORACLE یک DBMS است، 4GL زبان‌های نسل چهارم را گویند و RAD یک مدل توسعه سریع می‌باشد. X-Programming یک متدولوژی چابک است.

۲- گزینه (۱) صحیح است.

طراحی در متدولوژی XP بر پایه‌ی سادگی بنا شده است، این اصل که سادگی را حفظ‌کن (keep it simple). در متدولوژی XP همواره یک طراحی ساده بر یک طراحی پیچیده برتری دارد. اما این سادگی در طراحی به این معنی نیست که شرط چابکی حذف طراحی و یا ادغام طراحی و ساخت است.

۳- گزینه (۳) صحیح است.

صورت سوال به این شکل است:

کدام عبارت در مورد بازآرایی کد (Refactoring) در XP، هیچگاه درست نیست؟

۱) بازآرایی، کد را خواناتر می‌کند.

گزینه اول پاسخ سوال نیست.

۲) بازآرایی، کد را تغییرپذیرتر می‌کند.

گزینه دوم پاسخ سوال نیست.

۳) بازآرایی رفتار بیرونی کد را اصلاح می‌کند.

گزینه سوم پاسخ سوال است، زیرا طراحی در متدولوژی XP برپایه‌ی سادگی بنا شده است، بنابراین این دغدغه وجود دارد که گاهی طراحی‌های XP، از کیفیت الگوریتمی (غیروظیفه‌مندی یا مزه) مناسبی برخوردار نباشد. متدولوژی XP برای مرتفع نمودن این دغدغه، روش فاکتورگیری مجدد یا بازآرایی (refactoring) را پیشنهاد می‌کند. بنابراین با وجود اینکه روش فاکتورگیری مجدد در مرحله کدنویسی انجام می‌شود اما ماموریت آن ارتقاء کیفیت الگوریتمی و خوشمزه‌سازی طراحی است. فاکتورگیری مجدد یا بازآرایی، فرآیند تغییر، بهبود و ارتقاء کیفیت کد است، بازآرایی، کد را خواناتر می‌کند، بازآرایی، کد را تغییرپذیرتر می‌کند، بازآرایی، ساختار داخلی کد را بهبود می‌دهد اما بدون آنکه بازآرایی، رفتار بیرونی کد را اصلاح کند، به عبارت دیگر بازآرایی، رفتار بیرونی کد را اصلاح نمی‌کند یعنی ورودی‌ها و خروجی‌های کد را تغییر نمی‌دهد.

۴) بازآرایی ساختار داخلی کد را بهبود می‌دهد.

گزینه چهارم پاسخ سوال نیست.

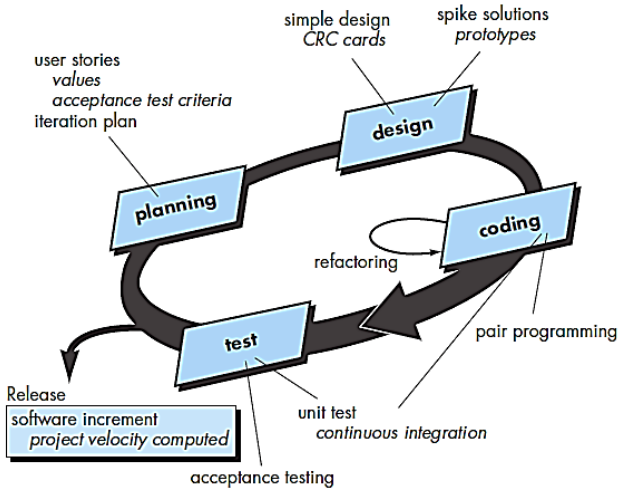
۴- گزینه (۱) صحیح است.

صورت سوال به این شکل است:

کدام محصول، به طور معمول در روش XP تولید می‌شود؟

۱) کارت‌های CRC

گزینه اول پاسخ سوال است، زیرا متدولوژی XP یا متدولوژی برنامه‌نویسی حدی (Extreme Programming) پرکابردترین رویکرد در توسعه نرم‌افزار به شیوه چابک است. این متدولوژی با نام‌های XP و یا X-programming نیز شناخته می‌شود. علت نامگذاری این متدولوژی به برنامه‌نویسی حدی این است که نسبت به متدولوژی‌های دیگر، مرحله‌ی برنامه‌نویسی را با تاکید بیشتری انجام می‌دهد. متدولوژی XP از روش شیءگرا جهت توسعه برنامه استفاده می‌کند. در این متدولوژی فعالیت‌های چارچوبی (framework activities) شامل چهار فعالیت برنامه‌ریزی، طراحی، برنامه‌نویسی و تست می‌باشد. شکل زیر گویای روند کارکرد متدولوژی XP است:



فعالیت طراحی در متدولوژی XP برپایه‌ی سادگی بنا شده است، این اصل که سادگی را حفظ‌کن (keep it simple). در متدولوژی XP همواره یک طراحی ساده بر یک طراحی پیچیده برتری دارد. در این متدولوژی صرفا طراحی سناریوهایی انجام می‌شود که نیاز وضع موجود است و قرار به پیاده‌سازی قطعی آن است، بنابراین طراحی سناریوهایی که نیاز به آن در آینده محتمل است و قرار به پیاده‌سازی آن قطعی نیست، در این متدولوژی جایگاهی ندارد. متدولوژی XP در جهت حفظ سادگی در فعالیت طراحی خود فقط از کارت‌های CRC استفاده می‌کند. پس از شناسایی موارد کاربرد و سناریونویسی برای هر یک از موارد کاربرد، زمان تعریف کلاس‌ها، صفات، متدها و ارتباطات میان کلاس‌های همکار برای هر یک از موارد کاربرد می‌رسد. برای شناسایی کلاس‌ها، صفات، متدها و ارتباطات میان کلاس‌ها برای هر یک از موارد کاربرد، از تکنیکی موسوم به CRC که سرواژه‌ی عبارت Class – Responsibility Collaborator و به معنی «مدل همکاری مسئولیت‌های کلاس‌ها» می‌باشد، استفاده می‌شود. مدل‌سازی CRC روشی ساده جهت تعیین و سازماندهی کلاس‌های داخل هر مورد کاربرد است. اگر در بخشی از طراحی یک سناریو، سختی ایجاد شود، متدولوژی XP، ایجاد فوری یک نمونه اولیه عملیاتی (operational prototype) را برای آن بخش از طراحی توصیه می‌کند. استفاده از نمونه اولیه

عملیاتی که موسوم به راهکار خیزشی (spike solution) نیز می‌باشد، دو پیامد دارد، اول اینکه، این راهکار منجر به کاهش ریسک فنی در پیاده‌سازی واقعی سناریوی مورد نظر می‌شود و دوم اینکه، این راهکار منجر به اعتبارسنجی برآوردهای اولیه یعنی «برآورد میزان کار»، «برآورد زمان لازم برای انجام کار»، «برآورد هزینه لازم برای انجام کار»، «مدیریت ریسک» و «زمان‌بندی» مربوط به سناریوی مورد نظر می‌شود.

۲) مدل پیکربندی

گزینه دوم پاسخ سوال نیست، زیرا مدل پیکربندی اثرات هرگونه تغییرات را در سرتاسر فرآیند تولید نرم‌افزار مدیریت می‌کند. مدیریت پیکربندی نرم‌افزار را می‌توان معادل مدیریت و کنترل تغییرات در نظر گرفت. بنابراین مدیریت پیکربندی نرم‌افزار همانند مدیریت تغییرات، هم روی تغییراتی که پس از تحویل محصول به مشتری رخ می‌دهند، اعمال می‌شود و هم تغییراتی که قبل از تحویل به مشتری رخ داده‌اند را کنترل می‌کند.

۳) نمودارهای حالت

گزینه سوم پاسخ سوال نیست. زیرا Sequence Diagram یا نمودار توالی، Object Diagram یا نمودار شیء جهت مدل‌سازی تعاملات پویای میان اشیاء همکار داخل یک use case مورد استفاده قرار می‌گیرد.

۴) تابلوی Kanban

گزینه چهارم پاسخ سوال نیست. زیرا تابلوی Kanban در متدولوژی چابک Kanban مورد استفاده قرار می‌گیرد

۵- گزینه (۱) صحیح است.

در متدولوژی FDD قالب زیر برای تعریف یک ویژگی (توصیف نیازمندی‌های عملکردی) استفاده می‌شود:

<action> the **<result>** **<by | for | of | to>** a(n) **<object>**

۶- گزینه (۴) صحیح است.

صورت گزینه‌ها به صورت زیر است:

۱) جلسات حداکثر یک ساعته هستند.

گزینه اول پاسخ سوال نیست، زیرا در طول اجرای هر sprint جلساتی روزانه (هر ۲۴ ساعت) و کوتاه حدود ۱۵ دقیقه با هدف کشف زودهنگام خطاها و بررسی پیشرفت پروژه توسط تیم اسکرام و با حضور اعضای تیم (تیم تولید و ذینفعان) برگزار می‌شود که به آن جلسات اسکرام (scrum meeting) گفته می‌شود.

۲) جلسات در غیاب استاد اسکرام (Scrum Master) برگزار می‌شوند.

گزینه دوم پاسخ سوال نیست، زیرا جلسات اسکرام توسط رهبر اسکرام (scrum master) هدایت می‌شود.

۳) در ابتدای هر جلسه، محصول ساخته شده به مشتری نمایش داده می‌شود.

گزینه سوم پاسخ سوال نیست، زیرا بعد از خاتمه یک sprint، اعضاء تیم طی جلسه‌ای به Product Owner و سایر ذینفعان پروژه نشان می‌دهند که چکار کرده‌اند و چطور از نسخه‌ی جاری نرم‌افزار می‌شود استفاده کرد. در واقع پس از اتمام هر sprint کارکردهای جدید که در طول این sprint ساخته‌شده‌اند، در قالب یک افزایش جدید از نرم‌افزار به صاحب محصول و ذینفعان ارائه می‌شود.

۴) در جریان هر جلسه، هر یک از اعضای تیم به سه سوال خاص پاسخ می‌دهند.

گزینه چهارم پاسخ سوال است، زیرا در این جلسات باید به سه پرسش زیر پاسخ داده شود:

- از جلسه قبل تا این جلسه چه کارهایی انجام شده است؟
- از جلسه قبل تا این جلسه با چه موانعی مواجه شده‌اید؟
- از این جلسه تا جلسه بعدی چه کارهایی قرار است انجام شود؟

۷- گزینه (۱) صحیح است.

صورت گزینه‌ها به صورت زیر است:

۱) از تغییر در نیازمندی‌ها استقبال کنید.

گزینه اول پاسخ سوال است، زیرا در پیمان چابک (Agile Alliance) دوازده اصل برای کسانی که می‌خواهند به چابکی دست پیدا کنند، تعریف شده است. و «**پذیرا بودن تغییرات در خواسته‌ها حتی در اواخر فرآیند توسعه. که مزیتی رقابتی مابین شرکت‌های سازنده نرم‌افزار است.**» یکی از اصول دوازده‌گانه پیمان چابک است.

۲) بدهی فنی در صورت لزوم قابل قبول است.

گزینه دوم پاسخ سوال نیست، زیرا «توجه پیوسته به اعتلای فنی و طراحی معماری ایده‌آل (پنهان‌سازی اطلاعات، افزایش انسجام، کاهش اتصال و استقلال عملیاتی) و عدم فراموشی و عدم ایجاد بدهی فنی، باعث افزایش چابکی می‌شود.» یکی از اصول دوازده‌گانه پیمان چابک است. بدهی فنی به معنی انجام ندادن برخی کارهای فنی است که با اصول چابکی منافات دارد. مکتب چابکی به معنی خروج از مکتب مهندسی نرم‌افزار نیست، بلکه اتفاقا چابکی به معنی ورود و دقت بیشتر بر اصول اساسی و طراحی معماری ایده‌آل مهندسی نرم‌افزار است اما اینبار با سرعت و چابکی بیشتر. با در نظر گرفتن بهترین ایده‌ها از هر دو مکتب، بیشترین بهره عاید خواهد شد و چیزی از تخریب دیگری، به‌دست نخواهد آمد.

۳) مستندسازی در طراحی باید حذف شود.

گزینه سوم پاسخ سوال نیست، زیرا حذف مستندسازی در طراحی از اصول چابکی نیست، بلکه مستندسازی نرمال، عامل چابکی است.

۴) معماری باید در دیرترین زمان ممکن تعیین شود.

گزینه چهارم پاسخ سوال نیست، زیرا معماری و سنگ بنای نرم افزار باید در زودترین زمان ممکن تعیین شود یعنی توجه پیوسته به اعتلای فنی و طراحی معماری ایده‌آل (پنهان‌سازی اطلاعات، افزایش انسجام، کاهش اتصال و استقلال عملیاتی)، باعث افزایش چابکی می‌شود.

بهترین منابع

کنکور ارشد فناوری اطلاعات

	عنوان درس	منابع
زبان عمومی و تخصصی	زبان انگلیسی	کلاس آنلاین یا فیلم استاد زارع یا کتاب راهیان ارشد
دروس مشترک	ریاضیات گسسته	کلاس آنلاین یا فیلم استاد گیلک یا کتاب راهیان ارشد
	ساختمان داده‌ها	کلاس آنلاین یا فیلم استاد گیلک یا کتاب راهیان ارشد
	طراحی الگوریتم	کلاس آنلاین یا فیلم استاد گیلک یا کتاب راهیان ارشد
	مهندسی نرم‌افزار	کلاس آنلاین یا فیلم استاد خلیلی‌فر یا کتاب راهیان ارشد
	شبکه‌های کامپیوتری	کلاس آنلاین یا فیلم استاد خلیلی‌فر یا کتاب راهیان ارشد
دروس تخصصی	پایگاه داده‌ها	کلاس آنلاین یا فیلم استاد خلیلی‌فر یا کتاب راهیان ارشد
	هوش مصنوعی	کلاس آنلاین یا فیلم استاد گیلک یا کتاب راهیان ارشد
	سیستم عامل	کلاس آنلاین یا فیلم استاد خلیلی‌فر یا کتاب راهیان ارشد
	اصول و مبانی مدیریت	مبانی مدیریت و مبانی رفتار سازمانی استیفن رابینز یا کتاب راهیان ارشد

گروه بابان
BABAN.IR

بهترین منابع
کنکور ارشد علوم کامپیوتر

	عنوان درس	منابع
زبان عمومی و تخصصی	زبان انگلیسی	کلاس آنلاین یا فیلم استاد زارع یا کتاب راهیان ارشد
دروس پایه	ریاضی عمومی ۱ و ۲	کلاس آنلاین یا فیلم استاد گیلک یا جزوه بابان
	مبانی علوم ریاضی	کتاب لیپن ولیپن ترجمه عمید رسولیان فصل های اول، پنجم و هفتم کلاس آنلاین ریاضیات گسسته یا فیلم استاد گیلک یا کتاب ریاضیات گسسته راهیان ارشد
	مبانی ماتریس ها و جبرخطی	کتاب جبرخطی هافمن یا کتاب جبرخطی اونان یا کتاب جبرخطی پیام نور
	مبانی آنالیز ریاضی	اصول آنالیز ریاضی رودین فصل یک تا شش یا کتاب آنالیز ریاضی یک پیام نور
	مبانی آنالیز عددی	کتاب راهیان ارشد بهزاد خداکرمی یا کتاب آنالیز عددی بابلیان
	مبانی احتمال	کلاس آنلاین یا فیلم استاد گیلک یا جزوه بابان کتاب آمار و احتمال شلدون راس
دروس مشترک	ساختمان داده	کلاس آنلاین یا فیلم استاد گیلک یا کتاب راهیان ارشد
	طراحی الگوریتم	کلاس آنلاین یا فیلم استاد گیلک یا کتاب راهیان ارشد
	مبانی نظریه محاسبه	کلاس آنلاین یا فیلم استاد گیلک یا کتاب علوم رایانه استاد شاپوری
	مبانی منطق و نظریه مجموعه ها	کلاس آنلاین یا فیلم استاد گیلک یا جزوه بابان
	ریاضیات گسسته و مبانی ترکیبیات	کلاس آنلاین یا فیلم استاد گیلک یا کتاب راهیان ارشد

گروه بابان
BABAN.IR

مشاوره تخصصی رشته کامپیوتر و IT

در راستای رسالت **مؤسسهٔ فرهنگی و انتشاراتی بابان** مبنی بر ارتقای سطح علـم و دانش کشور و کمک همه جانبه به دانشجویان و داوطلبان گرامـی، در جهـت قبـولی در کنکور کارشناسی ارشد و دکتری مهندسـی کـامپیوتر و IT دو طـرح زیـر را پایـه‌ریـزی کرده‌ایم:

۱) ارائه مشاوره تخصصی **حضوری** و **غیرحضوری** (تلفنی و آنلاین)

۲) برگزاری کلاس‌های **حضوری** و **غیرحضوری** (فیلم آموزشی و کلاس آنلاین)

برای آشنایی بیشتر با خدمات ارائـه شـده توسـط **موسسـه بابـان** بـه وب سـایت **khalilifar.ir** یا کانال تلگرام **@arastookhalilifar** مراجعه فرمایید.

تلفن دفتر مرکزی موسسه بابان: ۷۷۹۷۲۸۶۸-۰۲۱

تلفن دفتر فروشگاه انتشارات بابان: ۷۷۹۷۳۳۸۶-۰۲۱

پایگاه اطلاع رسانی موسسه بابان: www.baban.ir