دانشکده مهندسی برق و کامپیوتر

جزوه درسی

مهندسی نرم‌افزار پیشرفته

تهیه کننده: دکتر فریدون شمس

فهرست مطالب

۱- مشکلات توسعه نرم‌افزار و بررسی مسئله پیچیدگی در نرم‌افزار

۱-۱- مقدمه

توسعه نرم‌افزار در سال‌های اخیر دچار تحولات گسترده‌ای شده است، بطوریکه امروزه نرم‌افزار نقش دوگانه‌ای را بازی می‌کند. در یک نقش به عنوان محصول نهائی[1] محسوب می‌شود و در نقش دیگر، به عنوان تولید کننده محصول نهائی است. در نقش اول، نرم‌افزار، پتانسیل بالقوهٔ سخت‌افزار را به فعلیت می‌رساند و در این نقش- در کاربردهای گوناگونی که مورد استفاده قرار می‌گیرد از تلفن همراه گرفته تا کامپیوترهای بزرگ[2]- به عنوان تبدیل کننده (تولید، مدیریت، بازیابی، بهنگام سازی و نمایش) اطلاعات عمل می‌نماید. این اطلاعات می‌تواند به سادگی یک بیت و به پیچیدگی یک شبیه سازی چند رسانه‌ای[3] باشد. اما در نقش دوم، نرم‌افزار به عنوان ابزار اساسی کنترل سیستم‌های کامپیوتری (سیستم عامل)، کنترل شبکه‌های کامپیوتری و طراحی و توسعه نرم‌افزارهای دیگر (ابزارها و محیط‌های برنامه‌نویسی) عمل می‌کند. بعقیده صاحبنظران، نرم‌افزار یکی از نیروهای اصلی و محرک قرن بیست و یکم خواهد بود، زیرا مهمترین محصول قرن که همان اطلاعات است را پردازش می‌نماید. امروزه، نرم‌افزار عاملی حیاتی در گردش کار موسسات، کارخانجات، صنعت حمل و نقل، پزشکی، بانکداری، شبیه سازی سیستم‌های علمی و صنعتی و دیگر موارد است. همچنین کاربردهای نرم‌افزار از نمایش بهتر و قابل استفاده‌تر اطلاعات شخصی گرفته تا مدیریت اطلاعات سازمان‌های بزرگ و فراهم کردن یک بستر اطلاعاتی قوی (همچون اینترنت) که بوسیله آن ایده دهکده جهانی تحقق گردیده، گسترش یافته است.

۱-۲- بحران نرم‌افزار

در نیمه دوم قرن بیستم، نقش نرم‌افزار دستخوش تغییرات زیادی شده است. پیشرفت شگرف سخت‌افزار، تغییرات اساسی در معماری سیستم‌های کامپیوتری، افزایش شگفت انگیز ظرفیت حافظه‌های

[1] End-Product
[2] Mainframes
[3] Multimedia Simulation

اصلی و جانبی و ارزان شدن آنها، عواملی بوده اند که باعث افزایش تقاضا برای سیستم‌های کامپیوتری گردیده اند. این عوامل در کنار ضعف روش‌های توسعه نرم‌افزار و ناتوانی این روش‌ها در کنترل پیچیدگی نرم‌افزار باعث بوجود آمدن معضلاتی در تولید آن شد، که به آنها اصطلاح «بحران نرم‌افزار[1]» اطلاق می‌شود.

علایم و نشانه‌های این بحران عبارت بودند از [۱]:

- ناتوانی نرم‌افزار در بهره‌گیری کامل از پیشرفت سریع، قدرت و اطمینان پذیری رو به افزایش سخت‌افزار.
- ناتوانی روش‌های توسعه نرم‌افزار در پاسخگویی به افزایش تقاضای سیستم‌های اطلاعاتی پیچیده.
- هزینه‌های هنگفتی که برای توسعه نرم‌افزار پرداخته می‌شد.
- تأخیری (احیاناً تا سال‌ها) که در توسعه نرم‌افزار رخ می‌داد.
- عدم تامین مشخصات و نیازمندی‌های مورد نظر کاربر.
- کیفیت پایین، نامطمئن و ناکارا بودن نرم‌افزار.
- هزینه‌های هنگفت نگهداری نرم‌افزار، چرا که قدرت ما در نگهداری سیستم‌های موجود منوط به کیفیت طراحی و وجود منابع مناسب است. از آنجاییکه -معمولاً- سطح اغلب طراحی‌ها پایین بوده و منابع مناسب وجود نداشت، نگهداری پر هزینه می‌گردد.

متأسفانه یکی از نتایج نامطلوب این بحران، هدر دادن منابع انسانی است که مهمترین و گرانبهاترین سرمایه به شمار می‌آیند. در واقع، تعداد متخصصان کامپیوتر و برنامه‌نویسان خوب برای پاسخگویی به نیاز کاربران کافی نیست. از طرف دیگر اهمیت صنعت نرم‌افزار و نقش آن در کاربردهای علمی، تجاری، صنعتی و دیگر قلمروها روز به روز برجسته تر می‌شود. با توجه به این نکته آیا سزاوار نیست که پی راه حل مناسب برای این بحران باشیم؟

[1] Software Crisis

۱-۳- متدولوژی و ضرورت توجه به آن

چنانکه بیان نمودیم، یکی از علل اساسی بحران نرم‌افزار عدم وجود روش‌های مناسبی برای توسعه نرم‌افزار است. با توجه به این مقدمه ضرورت روی آوردن به یک متدولوژی مدون که تا حد زیادی مشکلات فوق را برطرف نماید نظر بسیاری از دست اندرکاران تولید سیستم‌های نرم‌افزاری را به خود جلب نموده و در نتیجه متدولوژی‌های گوناگون ارائه گردیده است.

برای روشنتر شدن بحث، ابتدا به بیان فرق بین متدولوژی و متد می‌پردازیم [۱]:

«متد عبارت است از فرآیندی منظم برای تولید مجموعه‌ای از مدل‌ها که هر کدام بخشی از سیستم نرم‌افزاری در حال تولید (یا توسعه) را تشریح نموده و با یک علامت‌گذاری روشن نمایش داده شده‌اند.»

«متدولوژی عبارت از مجموعه‌ای از متدها که در تمام چرخه حیات سیستم نرم‌افزاری اعمال شده و بر یک نوع نگرش کلی درباره جهان نرم‌افزار متکی باشند.» توجه داشته باشید که تعریف ذکر شده از واژه متدولوژی با تعریف آن در علم و فلسفه تفاوت می‌نماید. در واقع اینجا مقصود از متدولوژی، متدولوژی توسعه نرم‌افزار است. یک متدولوژی توسعه نرم‌افزار باید حداقل ۷ ویژگی زیر را داشته باشد [۱]:

۱- **ارئه تعاریفی از مفاهیم اولیه بکار رفته در متدولوژی**: در واقع، دیدگاه‌های اصلی یک متدولوژی درباره روش حل مسئله باید روشن باشد. مثلاً برای ایجاد یک ساختمان باید مولفه‌های بکاررفته در آن مانند آجر و آهن تعریف شده باشد. مثال دیگر، هنگامیکه متدولوژی شی‌گرائی نرم‌افزار را بعنوان مجموعه‌ای از اشیاء مستقل که با یکدیگر به صورت هوشمندانه همکاری می‌نماید تلقی می‌کند، باید بتواند تعریف روشنی از شی ارائه نماید.

۲- **ارائه مدلی برای فرآیند تولید**: مدل فرآیند[1] تولید عبارت از الگوی توسعه نرم‌افزار است. بعبارت بهتر، مدل فرآیند گام‌های لازم برای توسعه نرم‌افزار (همان نحوه و ترتیب استفاده از متدها) و چگونگی انتقال از یک گام به گام دیگر را بیان می‌نماید. انتظار می‌رود بین متدولوژی و فرآیند تولید سنخیت و سازگاری وجود داشته باشد. برای مثال متدولوژی ساخت‌یافته که نرم‌افزار را بصورت مجموعه‌ای از گام‌های متوالی تعریف می‌نماید با فرآیند تولید «آبشاری» که توسعه نرم‌افزار را به تعدادی مرحله متوالی(که هر کدام باید خاتمه یابد تا دیگری شروع نماید) تقسیم نموده یک سازگاری وجود دارد. در حالیکه ذات متدولوژی شی‌گرائی که بر مفهوم «کلاس»

[1] Process Model

متکی بوده با یک فرآیند تولید متکی بر تکرار و توسعه افزایشی بیشتر سازگار است. چراکه فرآیند کشف کلاس‌های یک مسئله خود یک فرآیند تکراری و تدریجی می‌باشد.

۳-داشتن مدل زیر بنائی(مدل معماری) : مشخص نمودن مراحل لازم برای ساختن یک ساختمان امر ضروری است ولی وقتی می‌توان گفت که این عمل موفق است که بدانیم چه نوع ساختمانی می‌خواستیم ایجاد نماییم و آیا این ساختمان ایجاد شد یا نه؟ در نرم‌افزار نیز همینطور است یعنی یک متدولوژی علاوه بر معرفی فرآیند تولید باید بتواند توضیح دهد که نرم‌افزار ساخته شده با این متدولوژی چگونه خواهد بود؟ یعنی ساختار کلی سیستم و نحوه ارتباط مولفه‌های سیستم با یکدیگر و روش‌هایی که این ساختار را قادر به تامین کلیه ویژگی‌های کلیدی سیستم می سازد، که همان معماری نرم‌افزار است.

٤-ارائه یک شیوه علامت گذاری استاندارد[1]: وجود یک شیوه علامت‌گذاری استاندارد که با رعایت آن مدل‌های گوناگون تولید می‌شوند برای یکنواختی درک همه دست اندرکاران تولید از سیستم در حال تولید ضروری می‌باشد.

۵-معرفی تکنیک‌هایی برای پیاده‌سازی متدولوژی: مقصود همان معرفی روش‌های مختلفی است که در مراحل تولید باید اعمال گردند. اهمیت این روش‌ها در چند نکته نهفته است: اولاً روش‌ها یک نوع نظم در فرآیند ایجاد سیستم‌های نرم‌افزاری پیچیده را بوجود می‌آورند. همچنین بوسیله روش‌ها، فرآورده‌هایی تولید شده که بعنوان وسائل ارتباطی بین افراد تیم عمل نمایند و بالآخره بوسیله این روش‌ها می‌توان نقاطی را در طول زمان تولید معرفی نمود که در آن مدیریت روند پیشرفت پروژه را ارزیابی کرده و خطرات بالقوه را کنترل می‌نماید.

٦-ارائه معیارهای برای ارزیابی نتایج حاصل از بکارگیری متدولوژی[2]: یک متدولوژی باید معیارهایی در اختیار افراد تیم قرار دهد که بوسیله آن می‌توان درباره درستی یا نادرستی نحوه بکارگیری متدولوژی در یک پروژه قضاوت نمود. مثلاً از یک متدولوژی شی‌گرائی انتظار می‌رود که معیارهایی برای مقایسه بین دو مدل کلاس یا تعیین میزان قابلیت استفاده مجدد از یک مولفه را

[1] Standard Notation
[2] Software Metrics

معرفی نماید. وجود این معیارها می‌تواند بحث استاندارد پذیری و قابلیت اعتماد و کارایی را گسترش دهد.

۷-وجود ابزار اتوماتیک برای کمک به تولید و اجرای مدل‌های مبتنی بر متدولوژی[1]: این ویژگی جزو متدولوژی نیست ولی برای تضمین بکارگیری کارا و صحیح و تسهیل در امر بهره‌برداری از آن ضروری می‌باشد.

با توجه به آشنائی نسبتاً دیرینه بشر با روش‌های مهندسی و اینکه مهندسی سخت‌افزار با مشکلات مهندسی نرم‌افزار مواجه نشده است، این سؤال مطرح میگردد که آیا استفاده از روش‌های سنتی مهندسی در توسعه نرم‌افزار، مشکلات آنرا برطرف نمی سازد؟

در پاسخ باید گفت که مسلماً استفاده از روش‌هایی کلی[2]-که بشر تا به امروز شناخته- در کنترل پیچیدگی سیستم‌های نرم‌افزاری-چنانکه خواهیم دید-باید کارساز باشد ولی این روش‌ها بسیار کلی هستند. به عبارت دیگر اگر جزئیات بکاربردن روش بخصوصی را نیز مد نظر داشته باشیم، خواهیم دید که در توسعه نرم‌افزار بعنوان یک محصول نهائی امکان استفاده از روش‌هایی است که در ساخت و تولید محصولاتی دیگری همچون سخت‌افزار به کاربرده می‌شود، مشکل خواهد بود اما بکارگیری این روش‌ها بعنوان یک ایده در نظر سازندگان نرم‌افزار هنوز از جلوه خاص برخوردار بوده و زمینه چالش در این مورد را فراهم می‌نماید.

۱-٤- تفاوت روش توسعه نرم‌افزار و سخت‌افزار

«فرآیند توسعه نرم‌افزار یک فرآیند مهندسی[3] است نه یک فرآیند ساختن[4] سنتی.»

برای بیان تفاوت بین این دو فرآیند باید گفت [٤]:

در سخت‌افزار می‌توان با ساخت قطعاتی مانند ICهای استاندارد و مجتمع سازی آنها به محصول نهائی دست یافت که خود حاصل انجام یک فرآیند اتوماتیک است. این همان فرآیند تولید سنتی است. در مقابل، در توسعه نرم‌افزار امکان ساخت فیزیکی محصول بر اساس محصولات موجود کوچکتر از طریق سرهم بندی آنها بسادگی وجود ندارد. بلکه برای دستیابی به محصول نهائی نرم‌افزار لازم است یک

---

[1] CASE Tools

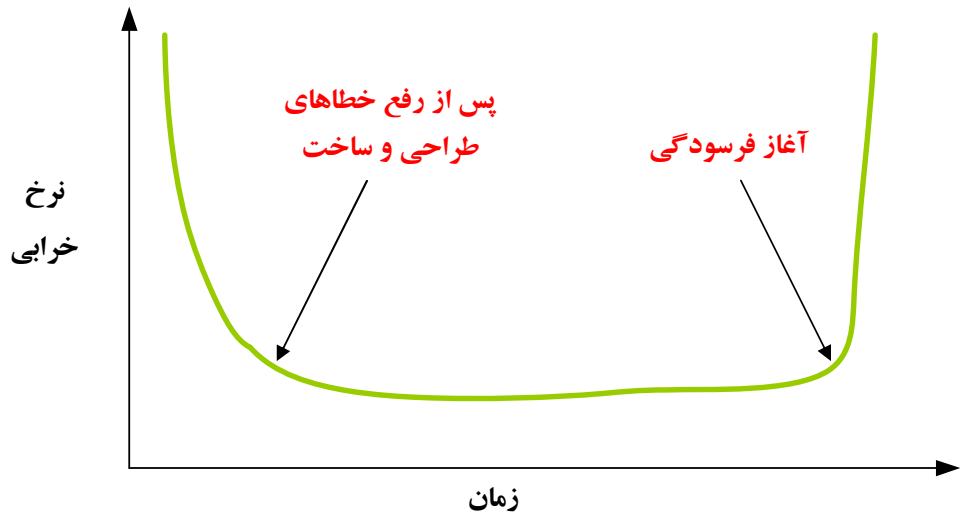
[2] مانند تجزیه (Decomposition) ، تجرید(Abstraction) و سلسله مراتب (Hierarchy). نگاه کنید به فصل بعد

[3] Engineering Process

[4] Manufacturing Process

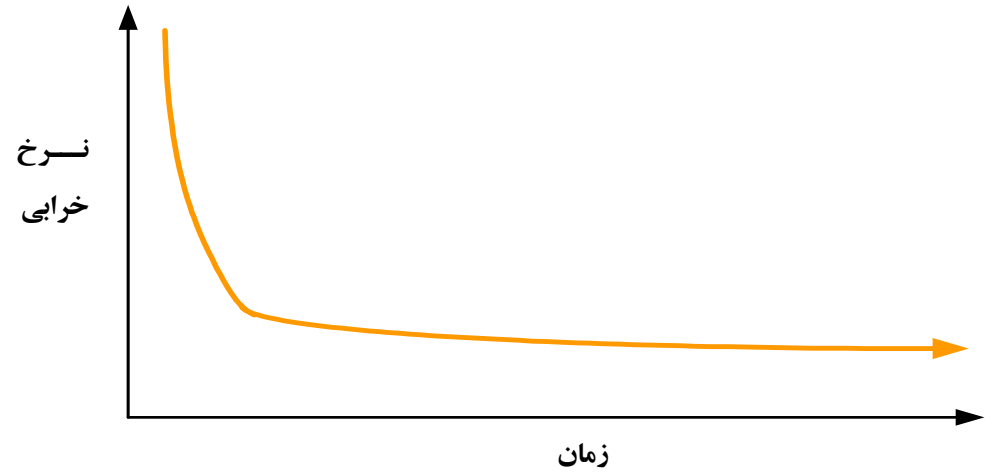
فرآیند مهندسی که برای هر کاربرد جدید منحصر به فرد است را دنبال نمود. این مسأله از متفاوت بودن طبیعت نرم‌افزار بعنوان یک محصول منطقی که بیشتر زاییده فکر انسان بوده در مقابل سخت‌افزار بعنوان یک محصول فیزیکی که مواد اولیه موجود در آن تابع قوانین فیزیکی مشخصی است، ناشی می‌شود.

از سوی دیگر، نرم‌افزار فرسوده نمی‌شود، بلکه فاسد می‌شود. در هنگام تولید یک قطعه سخت‌افزاری ممکن است مشکلات بیشماری بوجود آید، اما پس از تولید یک نمونه از محصول سخت‌افزاری، می‌توان آن را به تولید انبوه رساند. مدت زمانی که طول می‌کشد تا یک قطعه سخت‌افزاری فرسوده شود بسته به جنس قطعات بکار رفته در قطعه است و تولید کننده محصول سخت‌افزاری می‌تواند عمر مفید آن را بصورت تقریبی حدس بزند. پس از پایان عمر مفید یک قطعه سخت‌افزاری احتمال اینکه این قطعه دچار مشکل و عملکرد نادرستی از خود به نمایش بگذارد، افزایش می‌یابد تا جائیکه قطعه کاملاً فرسوده می‌شود، این موضوع در شکل ۱-۱ نشان داده شده است. اما در نرم‌افزار اینگونه نیست، نرم‌افزار می‌تواند مدت‌ها بکار خود ادامه دهد و هیچگاه فرسوده نمی‌شود زیرا به جنس قطعات بکار رفته در آن بستگی ندارد. نرم‌افزار می‌تواند سال‌ها به کاربر خود جواب دهد و نیازهای کاربران را تامین نماید. اما با پیشرفت تکنولوژی و تغییر نیازمندی‌های کاربران نرم‌افزار کهنه می‌شود و دیگر جوابگوی نیاز کاربران نیست هر چند هنوز می‌تواند کار کند. بنابراین نرم‌افزار همانند سخت‌افزار دچار فرسایش نمی‌شود بلکه کهنه یا فاسد می‌شود.



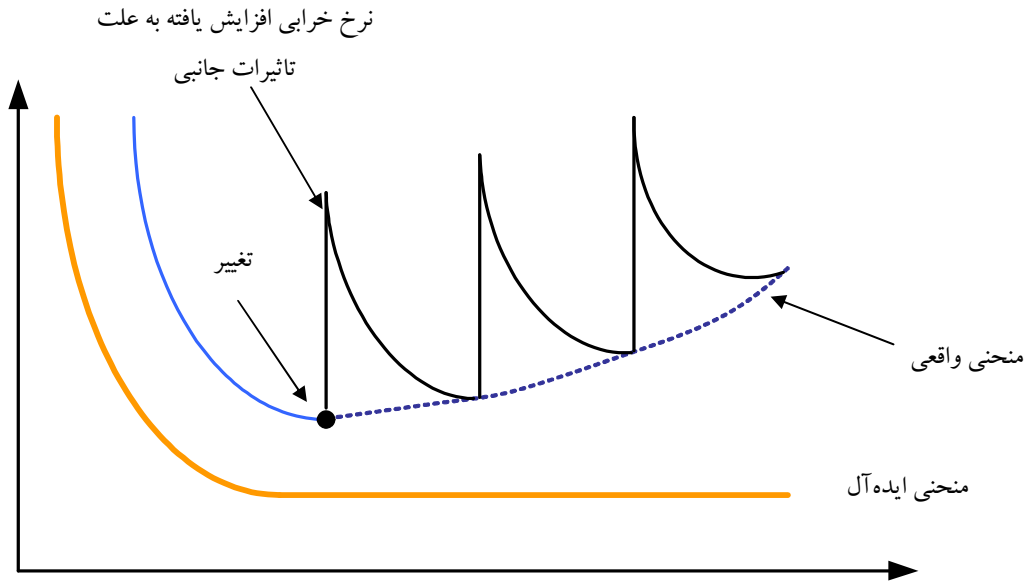
شکل ۱-۱- منحنی نرخ خرابی سخت‌افزار نسبت به زمان

با نکاتی که ذکر گردید، نرم‌افزار برخلاف سخت‌افزار دارای فرسودگی نیست و منحنی خرابی نرم‌افزار ایده‌آل همانند آنچه در شکل ۱-۲ نشان داده شده، است.



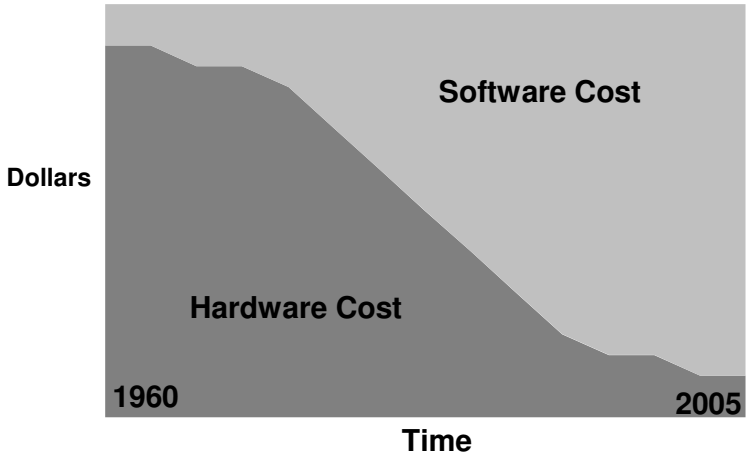
شکل ۱-۲- منحنی نرخ خرابی ایده‌آل نرم‌افزار نسبت به زمان

نکته جالب توجه این است که خرابی نرم‌افزار از منحنی خرابی ایده‌آل تبعیت نمی‌کند، بلکه با توجه به تغییر نیازمندی‌ها و اصلاح نرم‌افزار برای پوشش این تغییرات، خرابی نرم‌افزار از منحنی شکل ۱-۳ تبعیت می‌نماید.



شکل ۱-۳- منحنی نرخ خرابی ایده‌آل نرم‌افزار نسبت به زمان

در سال‌های اخیر با توجه به دستیابی به فناوری‌های نوین در توسعه سخت‌افزار نسبت به سال‌های اولیه ظهور کامپیوتر، نرخ رشد هزینه‌های نرم‌افزار به سخت‌افزار افزایش چشمگیری داشته است. بطوریکه در سال ۲۰۰۵ نسبت هزینه سخت‌افزار به نرم‌افزار یک به ۵ برآورد شده است.



شکل ۱-٤- رشد هزینه نرم‌افزار نسبت به سخت‌افزار

بدلیل این تفاوت ذاتی بین نرم‌افزار و سخت‌افزار پیچیدگی‌های خاصی در ابعاد مختلف از جمله در تعریف نرم‌افزار، طراحی، پیاده‌سازی، تست و نگهداری آن وجود دارد که لازم است از ابزاری و روش‌هایی برای مقابله با این پیچیدگی‌ها استفاده نمود. در این راستا متدولوژی‌های مختلف، هر یک روش‌هایی و ابزار خاصی را برای کنترل و فائق آمدن بر این پیچیدگی مطرح نموده‌اند و از این جمله متدولوژی شی‌گرایی که در ادامه خواهیم دید.

۱-۵- شیوه مقابله با بحران نرم‌افزار

برای حل مشکلات نرم‌افزار نیاز به روش‌های فنی و سیستماتیک[1] برای کنترل پیچیدگی نرم‌افزار (و در نتیجه بحران نرم‌افزار) احساس می‌شود. همچنین نیاز به یک فرآیند تولید مدون که چارچوب گام‌های مورد نیاز برای اعمال این روش‌ها- که نتیجه آن تولید یک نرم‌افزار با کیفیت عالی و قابلیت اعتماد بالا در حالیکه از نظر اقتصادی مقرون بصرفه باشد- وجود دارد. البته جایگاه ابزارهایی[2] که بکاربردن این روش‌ها را در چارچوب یک فرآیند تولید معین آسانتر می‌نمایند، قابل اغماض نیست.

یکی از روش‌های مدعی نوآوری در زمینه کنترل پیچیدگی نرم‌افزار، شی‌گرایی است [۱].

[1] Technical and Systematic Methods
[2] Computer-Aided Software Engineering (CASE) Tools

روش‌های ارائه شده قبل از روش شی‌گرایی (که آنها را روش‌های سنتی می‌نامیم) توانمندی لازم برای طرح یک نگرش عمیق و در عین حال ساده به سیستم‌ها - به میزانی که در روش‌های شی‌گرایی بوده است- را نداشته‌اند.

تکنیک‌های سنتی موجود توان پاسخگویی به مثال‌هایی از سیستم‌های بزرگ و واقعی را نداشته، به عنوان مثال در مورد طراحی ساخت‌یافته[1]، که در عمل بیشتر از بقیه روش‌های طراحی مورد استفاده قرار گرفته است، کاربردهای عملی نشان داده که برنامه‌نویسی ساخت‌یافته هنگامیکه تعداد خطوط برنامه از مرز ۱۰۰،۰۰۰ خط فراتر رود، این روش در کنترل برنامه با مشکل مواجه می‌گردد.

سیستم‌های بزرگ و کاربردهای جدیدی که روز به روز همراه با پیشرفت سخت‌افزارها مطرح شده‌اند نیازمند ساختاری برای مدیریت مدل‌های پیچیده‌تری بوده‌اند که تنها مکانیزم‌های موجود در تکنولوژی شی‌گرایی پتانسیل برخورد با گستردگی و پیچیدگی این سیستم‌ها را دارند. ایده اصلی برای فراهم آوری این پتانسیل تکیه بر مفهوم قابلیت استفاده مجدد[2] و تعریف وساخت مؤلفه‌های استاندارد با قابلیت کاربرد در محیط‌های مختلف است. همانطوری که بیان نمودیم روش‌ها، باید در چارچوب فرآیندها اعمال گردند.

فرآیندی که در تولید سیستم‌های سنتی به کار می‌رود فرآیند آبشاری می‌باشد. به نظر می‌رسد که فرآیند آبشاری برای تولید سیستم‌های جدید و پیچیدهٔ امروزی مناسب نیست، چراکه پایه و اساس این روش فهم کامل صورت مسأله و ثابت بودن نیازمندی‌ها بوده که در غیر از سیستم‌های بسیار ساده این مطلب صادق نیست. علاوه بر این، روش شی‌گرایی یک طبیعت تکراری و تدریجی دارد که با طبیعت ترتیبی این روش سنخیت ندارد. بنابر این ما به یک فرآیند تولید قویتر نیاز داریم.

خلاصه، یکی از علل اساسی بحران نرم‌افزار عدم وجود روش‌های مناسبی برای توسعه نرم‌افزار بوده، و این بمعنی ناتوانی روش‌های موجود در کنترل پیچیدگی نرم‌افزار است. پس بیاید پیچیدگی نرم‌افزار را مورد بررسی قرار دهیم.

۱-٦- پیچیدگی ذاتی نرم‌افزار

در واقع می‌توان سیستم‌های نرم‌افزاری را با توجه به پیچیدگی‌شان به دو دسته کلی طبقه‌بندی کرد [۲]:

[1] Structured Design
[2] Reusability

- سیستم‌های نرم‌افزاری معمولی(غیر پیچیده): این سیستم‌ها معمولاً بوسیله یک نفر طراحی، پیاده‌سازی و نگهداری می‌شوند. چنین سیستم‌هایی وقتی نیازهای مورد نظر کاربران خود را برآورده نمی‌سازند با سیستم‌های جدید جایگزین می‌گردند، در واقع این سیستم‌ها ارزش استفاده مجدد یا اصلاح را ندارند.
- اما در مقابل دسته فوق، دسته دیگری از سیستم‌های نرم‌افزاری وجود دارد که به آنها نرم‌افزارهای با بنیه صنعتی[1] اطلاق می‌شود. این دسته جزو سیستم‌های پیچیده به شمار می‌آیند که بارزترین ویژگی آنها این است که درک همه جزئیات آن از عهده قدرت ذهنی یک نفر خارج است. در واقع اصطلاح بحران نرم‌افزار در رابطه با تولید مشکلات اینگونه سیستم‌ها مطرح شده است. مثال‌هایی از این سیستم‌ها عبارتند از سیستم‌های بلادرنگ[2]، پایگاه داده‌هایی که باید میلیون‌ها رکورد ذخیره نمایند در حالیکه دستیابی صدها کاربر به صورت همزمان به داده‌ها با کارایی بالا را فراهم کند، سیستم‌های اطلاعاتی بزرگ و سیستم‌های هوشمند.

چنانچه قبلاً بیان کردیم یک تفاوت اساسی بین نرم‌افزار و دیگر ساخته‌های دست بشر وجود دارد بدین‌صورت که نرم‌افزار یک محصول منطقی است که بیشتر زاییده فکر انسان بوده لذا بین عناصراولیه آن قوانین بنیادی حاکم نیست. در حالیکه محصولات دیگر ساخت بشر بگونه‌ای هستندکه عناصر اولیه تشکیل دهنده آن از قوانین فیزیکی معینی پیروی می‌نمایند [۵].

با توجه به این مطلب، دو نکته درباره پیچیدگی سیستم‌های نرم‌افزاری قابل ذکر است: یکی اینکه این پیچیدگی با پیچیدگی سیستم‌های طبیعی و محصولات فیزیکی ساخت دست بشر یک تفاوت اساسی دارد: مدل‌های ساده برای سیستم‌های طبیعی وجود دارد در حالیکه این مطلب برای سیستم‌های نرم‌افزاری صادق نیست این همان مفهوم پیچیدگی غیر قانونمند[3]است.

---

[1] Industrial-Strength Software

[2] Real-Time Systems

[3] Arbitrary Complexity با یک مثال پیچیدگی غیرقانونمند را بیشتر توضیح می دهیم: در صنعت عمران و ساختمان‌سازی اگر مشتری از یک شرکت ساختمانی درخواست اضافه یک طبقه زیرزمینی به یک ساختمان صد طبقه را نماید، مسلماً یک نوع شوخی تلقی خواهد شد!، در حالیکه در صنعت نرم‌افزار چنین درخواستی کاملاً طبیعی بنظر می‌رسد! (البته از نظر مشتری). علاوه بر این، مشتریان چنین استدلال می‌کنند که انجام این کار ساده است زیرا بیشتر از نوشتن چند خط برنامه نیست!

نکته دوم این است که این پیچیدگی یک ویژگی ذاتی سیستم‌های نرم‌افزاری بزرگ است به عبارت دیگر نمی‌توان این پیچیدگی را از بین برد بلکه باید آن را کنترل نمود.

۱-۷- عوامل پدید آورنده پیچیدگی ذاتی

اما چرا پیچیدگی یک خاصیت جدائی ناپذیر برای سیستم‌های نرم‌افزاری بزرگ است؟

در واقع، این پیچیدگی از چهار فاکتور ناشی می‌شود [۱]:

۱) پیچیدگی خود مسأله

معمولاً سیستم‌های نرم‌افزاری بزرگ حاوی عناصری است که پیچیدگی آنها اجتناب ناپذیر بوده، این پیچیدگی ناشی از :

- وجود نیازمندی‌های گوناگون و مختلف و گاهی حتی متضاد

چنانکه می‌دانیم نیازمندی‌ها به دو گروه کلی تقسیم می‌شوند: نیازمندی‌های وظیفه‌ای[1] که عبارتست از عملکرد خام سیستم و نیازمندی‌های غیر وظیفه‌ای[2] که معمولاً به صورت ضمنی بیان می‌شوند مانند کارایی[3]، قابلیت اعتماد[4] و ... است. مثلاً یک سیستم هوشمند مانند روبات را در نظر بگیرید: مشخص است که درک عملکرد چنین سیستمی به اندازه کافی سخت است (نیازمندی‌های وظیفه‌ای) حال اگر به این نیازمندی‌ها، نیازمندی‌های غیروظیفه‌ای نیز اضافه گردد همان پیچیدگی غیرقانونمند بوجود خواهد آمد.

- ناتوانی کاربر و مهندس نرم‌افزار در درک صحیح یکدیگر

معمولاً برای کاربران توصیف نیازهای مورد نظر خود به صورتی قابل فهم برای توسعه‌دهنده[5] سیستم خیلی مشکل است. در موارد بسیاری کاربران فقط ایده‌های مبهمی از آنچه می خواهند سیستم نرم‌افزاری حاوی آن باشد، را دارند. البته بدین معنی نیست که کاربر یا توسعه‌دهنده سیستم (یا هر دو) مقصرند، بلکه-به طور کلی- این مشکل ناشی از عدم وجود آشنائی کافی هر دو از زمینه‌های کاری یکدیگر است. در واقع، کاربران و توسعه‌دهندگان سیستم دارای دو دید مختلف نسبت به مسأله و راه حل آن هستند و حتی اگر کاربر شناخت کافی از نیازهای خود داشته باشد، ما

[1] Functional Requirements
[2] Non-Functional Requirements
[3] Efficiency
[4] Reliability
[5] System Developers

هنوز به ابزاری که بتواند این نیازها را به صورت صحیح و دقیق بیان نماید احتیاج داریم (البته در راستای حل مشکل تعیین نیازمندی‌ها به کمک ابزارهایی مانند موارد کاربری و کارت‌های CRC[1] گام‌های موثری برداشته شده است).

- ❖ تغییر نیازها در زمان طراحی سیستم و بعد از تولید آن

این فاکتور در افزایش پیچیدگی مسأله سهم بسزایی دارد. در واقع دیدن نمونه های اولیه سیستم[2] و مستندات طراحی (در زمان طراحی) سپس استفاده از سیستم بعد از نصب آن، باعث پی بردن و درک بهتر و واقعیتر کاربر نسبت به نیازهای خود است.

از طرف دیگر خبرگی توسعه‌دهندگان سیستم نسبت به مسأله و راه‌های حل آن بیشتر می‌شود و می‌توانند سوالهای بهتری درباره جنبه‌های مبهم رفتار سیستم طرح نمایند و بدین صورت دید کاملتری نسبت به سیستم پیداخواهند کرد.

۲) مشکل کنترل فرآیند تولید

سیستم‌های نرم‌افزاری روز به روز پیچیده‌تر می‌شوند. امروزه ما شاهد سیستم‌هایی نرم‌افزاری بزرگ هستیم که حجم آنها با صد هزار خط یا حتی یک ملیون خط اندازه‌گیری می‌شود. چنانکه قبلاً گفتیم درک کامل و دقیق چنین سیستم‌های بزرگی از عهده یک نفر خارج است. حتی اگر سعی کنیم سیستم را به واحدهای[3] با معنی تجزیه نماییم، باز با صدها واحد روبرو خواهیم گشت. پس ما به یک گروه یا یک تیم از متخصصین نیاز داریم.

به صورت ایده‌آل باید سعی کنیم از یک تیم با حداقل افراد استفاده نماییم ولی صرف‌نظر از اندازه تیم، توسعه مبتنی بر تیم‌ها همیشه با مشکلات مهمی روبرو بوده است، زیرا افراد بیشتر، به معنی ارتباطات پیچیده‌تر و در نتیجه هماهنگی بین افراد تیم مشکل‌تر می‌شود بخصوص اگر تیم از نظر جغرافیائی پراکنده باشد. در واقع مشکل کلیدی که توسعه تیمی با آن مواجه است همان مدیریت صحیح افراد تیم به طوری که یگانگی و یکپارچگی تحلیل و طراحی حفظ گردد.

---

[1] Class, Responsibilities, and Collaborators
[2] System Prototypes
[3] Modules

۳) استاندارد نبودن نرم‌افزار

معمولاً یک شرکت ساختمان سازی برای ساختن یک ساختمان از مصالح ساختمانی با مشخصات استاندارد مانند آجر و آهن که بوسیله شرکت‌های دیگر تولید شده است، استفاده می‌نماید یا در صنعت سخت‌افزار مثلاً برای ساختن یک برد از ICهای استاندارد ساخت شرکت‌های دیگر استفاده می‌شود و هرگز خود شرکت سازنده اقدام به ساختن همه قطعات مورد نیاز این بُرد نمی‌کند ولی متاسفانه چنین اتفاقی در صنعت نرم‌افزار به فراوانی رخ می‌دهد!. اینها مثال‌هایی از استفاده مجدد[1] بوده است، چیزی که صنعت نرم‌افزار شدیداً به آن نیازمند است. البته در این راستا گام‌های خوبی برداشته شده است (در قسمت مربوط به مؤلفه‌ها[2] به تفصیل به این مطلب خواهیم پرداخت).

به علاوه، در صنعت ساختمان سازی استانداردهایی برای تضمین کیفیت مواد اولیه مورد نیاز وجود دارد در حالیکه صنعت نرم‌افزار از این نظر خیلی عقب مانده است!. لذا تولید سیستم‌های نرم‌افزاری یک کار طاقت فرسا به شمار می‌آید.

٤) مشکل توصیف رفتار سیستم‌های پیچیده

از نظر رفتار می‌توان سیستم‌ها را –به طور کلی– به دو گروه تقسیم نمود:

- **سیستم‌های پیوسته**[3] : رفتار این سیستم‌ها بوسیله یک تابع پیوسته توصیف می‌گردد، که توسط آن می‌توان رفتار سیستم و عکس‌العمل آن در مقابل رویدادهای گوناگون را پیش بینی کرد.
- **سیستم‌های گسسته**[4] : این سیستم‌ها از تعداد متناهی حالت تشکیل شده است که این تعداد در سیستم‌های گسسته پیچیده معمولاً عدد بزرگی است. ویژگی بارز این سیستم‌ها این است که نمی‌توان انتقال بین حالت‌های مختلف سیستم بوسیله توابع پیوسته را مدل سازی نمود.

لذا این امکان بالقوه وجود دارد که به ازای یک رویداد غیر منتظره خارجی، سیستم از حالت فعلی به حالت جدید(و غیر مطلوب) منتقل گردد که این انتقال می‌تواند غیر قطعی[5] باشد و در بدترین شرایط این رویداد می‌تواند حالت سیستم را خراب نماید. با توجه به اینکه کامپیوترهای دیجیتال سیستم‌های گسسته هستند و اینکه نرم‌افزار روی این دستگاه‌ها اجرا می‌گردد، پس ما با یک سیستم گسسته سروکار داریم.

---

[1] Reuse
[2] Components
[3] Continuous Systems
[4] Discrete Systems
[5] Non-Deterministic

حال به این مثال توجه نمایید: فرض کنید یک هواپیما بوسیله یک کامپیوتر کنترل می‌شود، چون با یک سیستم گسسته مواجه هستیم احتمال بروز حالتی مانند اینکه وقتی یک مسافر لامپ بالای سر خود را روشن کند. آنگاه –مثلاً– هواپیما به طور ناگهانی به سمت پایین حرکت نماید، وجود دارد!

در یک سیستم پیوسته ما انتظار بروز چنین رویدادی را نداریم ولی متأسفانه در یک سیستم گسسته به علت اینکه طراحان سیستم فعل و انفعال‌هایی که بین تعدادی از رویدادهای ویژه رخ می‌دهد را در نظر نگرفته اند احتمال بروز چنین حالت‌هایی وجود دارد.

با توجه به مثال فوق و اینکه نرم‌افزار یک سیستم گسسته است، می‌بینیم که این خاصیت یکی از عوامل افزایش پیچیدگی سیستم‌های نرم‌افزاری است. پس برای رسیدن به نرم‌افزار با کیفیت و قابلیت اطمینان بالا چه باید کرد؟ این هدف جز با انجام آزمایش‌های گوناگون و جامع تحقق نمی‌یابد.

لذا فاز تست در چرخه توسعه نرم‌افزار از اهمیت بسزائی برخوردار است. بنابر آنچه بیان شد، مشکل اصلی نرم‌افزار پیچیدگی ذاتی خود است. برای پیدا نمودن راه مقابله با آن بیایید خصوصیات کلی سیستم‌های پیچیده(اعم از کامپیوتری و غیر کامپیوتری) را بررسی نماییم.

۱-۸- ساختار سیستم‌های پیچیده

با ذکر چند نمونه از سیستم‌های پیچیده، ساختار و خصوصیات کلی آنها را بررسی می‌نماییم:

❖ کامپیوتر شخصی

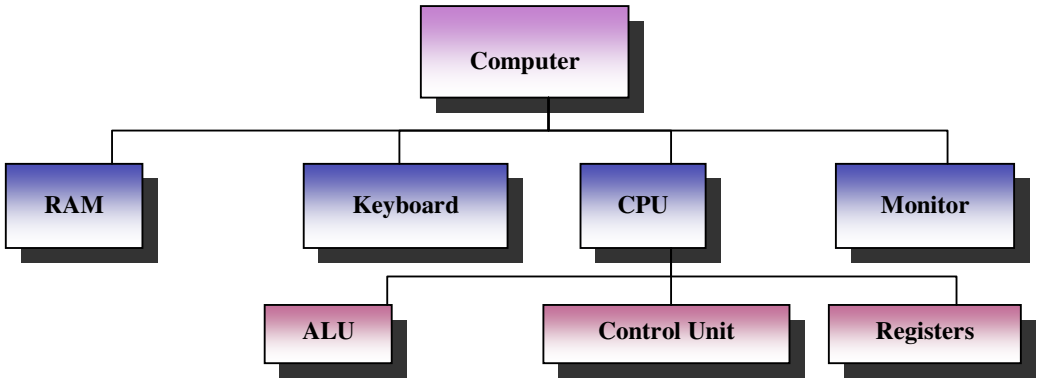
کامپیوتر شخصی دستگاهیست که پیچیدگی آن از نوع متوسط است. چنانکه می‌دانیم کامپیوتر شخصی از مجموعه‌ای از مؤلفه‌های اصلی مانند واحد پردازش مرکزی، نمایشگر، صفحه کلید و حافظه جانبی مانند دیسک سخت و ... تشکیل می‌شود. هر کدام از این مؤلفه‌ها به نوبه خود از مؤلفه‌های کوچکتر تشکیل می‌گردد مثلاً CPU از واحد کنترل، ALU، ثبات‌ها و ... تشکیل می‌شود، همچنین ALU از مجموعه‌ای از ثباتها تشکیل شده است.

اینجاست که طبیعت سلسله مراتبی سیستم‌های پیچیده را می‌توان مشاهده نمود. در واقع علت درستی عملکرد کامپیوتر شخصی عبارت از فعالیت گروهی و مشترک مؤلفه‌های اصلی آن است. این مؤلفه‌های جداگانه با هم یک واحد منطقی تشکیل می‌دهند.

مسلماً قدرت ما در استدلال بر نحوه عملکرد کامپیوتر شخصی از طریق قدرت ما بر تجزیه این سیستم به اجزای کوچکتر که می‌توان هر کدام را به صورت جداگانه و مستقل از بقیه اجزاء بررسی نمود، حاصل شده است. لذا می‌توانیم –مثلا– عملکرد نمایشگر را جداگانه از عملکرد دیسک سخت بررسی نماییم.

توجه داشته باشید که سطوح ساختار سلسله مراتبی در سیستم‌های پیچیده، سطوح مختلف تجرید[1] را نمایش می‌دهند که هر سطحی روی سطح پایین تر از آن بنا شده است. به هر سطحی از سطوح تجرید که نگاه کنیم می‌بینیم که اولا این سطح بخودی خود قابل فهم است و ثانیاً این سطح سرویس هایی به سطوح بالاتر را ارائه می‌دهد. ما –معمولاً– یکی از این سطوح که مناسبتر است برای حل مسأله مورد نظر را انتخاب می کنیم.

مثلا اگر با مشکلی در زمانبندی حافظه اصلی مواجه شویم باید این مشکل را در سطح گیت ها بررسی نماییم. اما استفاده از این سطح برای پیدا کردن علت اشکالات در یک برنامه کاربردی مناسب نخواهد بود.



شکل ۱-۵- ساختار سلسله مراتبی در کامپیوتر

❖ گیاهان

درباره ساختار گیاه به عنوان یک سیستم پیچیده نکات زیر قابل ذکرند:

- یک گیاه از سه قسمت اصلی تشکیل می‌شود: ریشه‌ها، ساقه‌ها و برگ‌ها که هر کدام نیز از قسمت‌های کوچکتر دیگری تشکیل می‌شوند. به علاوه این قسمت‌های کوچکتر به قسمت‌های ریزتر تقسیم می‌شوند و این روند تقسیم پذیری ادامه می یابد تا به سطح سلول‌ها برسیم که سلول‌ها نیز از ساختار پیچیده ای برخودارند. این همان ساختار سلسله مراتبی است

[1] Abstraction Levels

که از همکاری اجزای مختلف گیاه بوجود می‌آید و هر سطح آن دارای پیچیدگی خاص خود است.

- همه اجزائی که در یک سطح واحدی از سطوح تجرید واقعند با هم دیگر بوسیله راه‌هایی که بخوبی تعریف شده است، ارتباط پیدا می‌کنند. برای مثال، بالاترین سطح تجرید را در نظر بگیرید: در این سطح -که از سه مؤلفه اساسی ذکر شده تشکیل می‌شود- ریشه‌ها وظیفه جذب کردن آب و نمک های معدنی از خاک را دارند، ریشه‌ها با ساقه‌ها ارتباط برقرار می‌کنند که بوسیله ساقه‌ها مواد خام جذب شده به برگها منتقل می‌گردد. برگها به نوبه خود این مواد خام را بوسیله عمل فتوسنتز به غذا تبدیل می‌کنند.
- نکته دیگری که مشاهده می‌شود، وجود یک مرز مشخص بین درون و برون یک سطح تجریدی است، لذا می‌توان گفت که آثاری که از یک برگ بروز می‌نماید-که نتیجه فعالیت‌های گروهی اجزای تشکیل دهنده برگ بوده-ارتباطی مستقیم با عناصر اصلی تشکیل دهنده ریشه را ندارد (یا اگر هم وجود داشته باشد خیلی ارتباط ضعیفی است) به عبارت ساده تر یک مرز روشن بین اجزاء سطوح مختلف تجرید وجود دارد.
- همچنین وجود عناصر مشترک در ساختارهای مختلف یک گیاه ملاحظه می‌گردد. برای مثال سلول گیاهی را در نظر بگیرید: در واقع، سلول گیاهی به عنوان بلوک اصلی تشکیل دهنده همه ساختارها در گیاه عمل می‌کند ولی با وجود این، انواع مختلفی از سلول‌های گیاهی وجود دارد مانند سلول‌های حاوی کلوروپلاست، سلول‌های بدون کلوروپلاست وحتی سلول‌های مرده و سلول‌های زنده. در واقع، استفاده از یک ساختار مشترک یک کار اقتصادی تر و مقرون به صرفه تر بوده، و اصطلاحاً به آن اقتصاد در بیان[1] گفته می‌شود. ساختمان جوامع بشری و ساختمان ماده نمونه های دیگری از سیستم‌های پیچیده است که در مورد آنها نکات مشابهی می‌توان مطرح نمود.

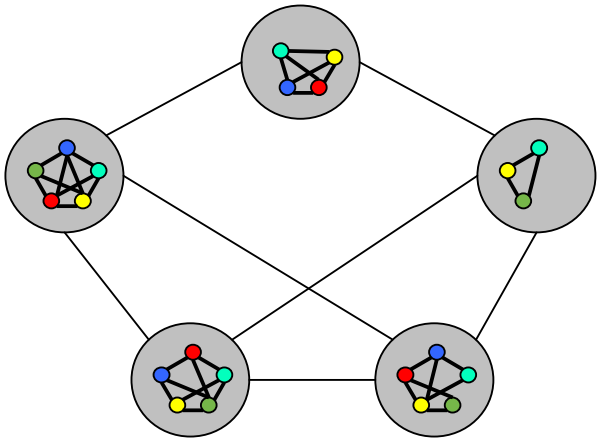
[1] Economy in Expression

۱-۹- ویژگی‌های سیستم‌های پیچیده

با توجه به نمونه های ذکر شده پنج ویژگی مشترک بین تمامی سیستم‌های پیچیده را مشاهده می‌نماییم، که عبارتند از [۱]:

۱. در اغلب سیستم‌های پیچیده، پیچیدگی به صورت سلسله مراتب ظاهر می‌شود. به عبارت دیگر، یک سیستم پیچیده از چند زیرسیستم مرتبط به هم تشکیل شده که هر کدام به نوبه خود از چند زیر سیستم کوچکتر تشکیل می‌شود. این روند تجزیه ادامه می یابد تا به پایینترین سطح از مؤلفه‌های اولیه برسیم.

این حقیقت که اغلب سیستم‌های پیچیده دارای ساختار تقریبا تجزیه پذیر[1] و سلسله مراتبی[2] هستند، یک عامل اساسی است که درک، توصیف و حتی دیدن این گونه سیستم‌ها و اجزای آن را برای ما آسان می‌نماید. در واقع می‌توان گفت که ذهن انسان سیستم‌هایی را می‌تواند بهتر درک و تحلیل کند که دارای ساختار سلسه مراتبی باشند.



شکل ۱-٦- سلسله مراتب پیچیدگی

نکته دیگر این است که معماری یک سیستم پیچیده تابعی است از مؤلفه‌های سیستم باضافه روابط سلسله مراتبی که بر این مؤلفه‌ها حاکمند. اما دربارهٔ طبیعت این مؤلفه‌های اولیه می‌توان گفت:

۲. انتخاب اینکه کدام یک از مؤلفه‌ها در سیستم اولی هستند، امری است نسبتا دلخواه و تا حدود زیادی بستگی به دید طراح سیستم دارد.

[1] Nearly Decomposable
[2] Hierarchic

لذا مؤلفه‌هایی که از دید یک طراح اولی هستند می‌توانند از دید طراح دیگر در سطح بالایی از تجرید باشند.

سیستم‌های سلسله مراتبی را تجزیه پذیر[1] گویند زیرا قابل تجزیه به اجزائی مشخص هستند. به بیان دقیقتر این سیستم‌ها، تقریباً تجزیه پذیرند. زیرا اجزای آنها به صورت کامل از یکدیگر مستقل نیستند واین نکته ما را به خاصیت سوم راهنما می‌نماید.

۳. در سیستمی که از چند زیرسیستم تشکیل می‌شود، ارتباط بین اجزای هر کدام از این زیر سیستم‌ها (ارتباط درون مؤلفه‌ای[2]) قویتر از ارتباط بین خود زیر سیستم‌ها (ارتباط برون مؤلفه‌ای[3]) است.

این حقیقت در جدا کردن خواصی که دارای نرخ تغییر زیاد بوده[4] (ساختار داخلی مؤلفه‌ها) از خواصی که دارای نرخ تغییر پایین[5] (ارتباط بین مؤلفه‌ها) است، کمک می‌نماید.

این تفاوت بین ارتباطات درون مؤلفه‌ای و ارتباطات برون مؤلفه ای یک مرز مشخص و روشن بین ساختار داخلی یک مؤلفه و ارتباط آن با بیرون معرفی می‌نماید که به ما کمک می‌کند که هر مؤلفه را به صورت تقریبا جداگانه از بقیه سیستم بررسی و تحلیل نماییم.

همانطوری که در نمونه‌ها ملاحظه کردیم، بسیاری از سیستم‌های پیچیده بوسیله اقتصاد در بیان پیاده‌سازی می‌شوند. با توجه به این مطلب خاصیت چهارم را بیان می‌کنیم:

٤. سیستم‌های سلسله مراتبی معمولا از تعداد کمی از زیر سیستم‌های مشخص و متفاوت تشکیل می‌شوند که این زیرسیستم‌ها به صورتهای گوناگون و ترتیب های مختلف ظاهر می‌شوند.

٥. معمولاً سیستم‌های پیچیده که به صورت محکم و استوار عمل می‌کنند حاصل تکامل سیستم‌های ساده ای هستند که به درستی عمل می‌کردند. سیستم‌های پیچیده که از ابتدا به صورت پیچیده طراحی می‌شوند، هرگز کار نخواهند کرد.

اشیایی که که در یک دوره تکامل سیستم، پیچیده به شمار می‌آیند در دوره بعدی به عنوان اشیاء اولیه مورد استفاده قرار خواهند گرفت که بوسیله آنها سیستم‌های پیچیده تری بنا خواهند شد.

---

[1] Decomposable
[2] Intracomponent Linkage
[3] Intercomponent Linkage
[4] High-Frequency Dynamics
[5] Low-Frequency Dynamics

۱-۱۰-پیچیدگی سازمان‌یافته و سازمان نایافته[1]

شکل اصلی سیستم‌های پیچیده[2]

در سیستم‌های پیچیده دو نوع سلسله مراتب قابل مشاهده است [۱]:

- ساختار کلاس[3]: این همان سلسله مراتب IS-A است که در بحث سلسله مراتب توضیح داده خواهد شد.
- ساختار شی[4]: این همان سلسله مراتب PART-OF است که در بحث سلسله مراتب توضیح داده خواهد شد.

اگر این دو نوع سلسله مراتب را به خواص پنجگانه سیستم‌های پیچیده ضمیمه نماییم، شکل اصلی سیستم‌های پیچیده بدست خواهد آمد. مقصود این است که همه سیستم‌های پیچیده این شکل را بخود می‌گیرند. تجربه نشان داده است که موفق‌ترین سیستم‌های نرم‌افزاری پیچیده آنهایی بودند که در طراحی شان ساختار کلاس و شی و خواص پنجگانه سیستم‌های پیچیده مراعات شده است.

حال که ما نحوه طراحی سیستم‌های پیچیده را می‌دانیم، پس چرا توسعه نرم‌افزار هنوز با مشکلات جدی روبرو است؟

در واقع ایده پیچیدگی سازمان یافته(که استفاده از شی‌گرائی و راهنماهای مدل شی[5] باعث سازمان یافتگی می‌شود)، یک ایده نسبتاً نو است و هنوز برای به کاربردن صحیح و کارای آن نیاز به تحقیقات و بررسی های بیشتر وجود دارد [۸]. ولی یک عامل مهمتری وجود دارد و که همان قدرت محدود ذهن انسان در پردازش پیچیدگی است. به عبارت دیگر ذهن انسان قادرست مقدار محدودی از اطلاعات همزمان را پردازش یا درک نماید. زمانیکه تجزیه و تحلیل یک سیستم نرم‌افزاری پیچیده را شروع می‌کنیم، با اجزای خیلی زیاد و روابط پیچیده ای که بر آنها حاکم است روبرو می شویم، که مشترکات کمی دارند. این مثالی از پیچیدگی سازمان نایافته است.

ما سعی می‌کنیم از راه فرآیند تحلیل و طراحی این پیچیدگی را سازمان‌دهی نماییم، ولی باید برای اینکار به چیزهای زیادی همزمان فکر کنیم زیرا–چنانچه قبلاً توضیح دادیم– سیستم‌های کامپیوتری،

---

[1] Organized and Disorganized Complexity
[2] Canonical Form of a Complex Systems
[3] Class Structure
[4] Object Structure
[5] Object Model

سیستم‌های گسسته هستند بنابراین با فضای حالت بسیار بزرگ و پیچیده روبرو خواهیم شد که متاسفانه درک آن برای یک نفر کار غیر ممکنی است. تجارب روانشناسی نشان می‌دهد که حد اکثر تعداد قطعه‌های اطلاعاتی که مغز انسان همزمان می‌تواند درک نماید ۲±۷ است [۹].

با توجه به این محدودیت ما در مقابل یک معضل قرار داریم:

از طرفی پیچیدگی سیستم‌های نرم‌افزاری روز به روز در حال افزایش است، و از طرفی دیگر قدرت پردازش همزمان مغز انسان محدود است. چگونه می‌توان این معضل را حل نمود؟

نقش تجزیه

در واقع تکنیک کنترل و تسلط بر پیچیدگی همان اصل بسیار معروف تفرقه بیانداز و حکومت کن[1]. برای استفاده از این اصل سیستم را به اجزای کوچکتر و کوچکتر تقسیم می‌کنیم سپس هر کدام از آنها را به تنهایی حل می‌کنیم. بدین صورت ذهن برای درک هر سطح از سیستم مجبور نخواهد بود همه اجزاء را در نظر بگیرد بلکه با تعداد محدودی از اجزاء سروکارخواهد داشت. بنابر این محدودیت ذهن انسان بر طرف خواهد شد.

اما به چه صورت باید این اصل را اعمال کنیم؟

برای پاسخ به این سوال مدل شی و اصول اساسی آن را در فصل بعدی مطرح خواهیم نمود.

[1] Divide and Rule

۲- معرفی اصول شی‌گرائی برای مقابله با پیچیدگی

۲-۱- پیشینه تاریخی

با توجه به پیچیدگی روز افزون سیستم‌های نرم‌افزاری، روش‌های مقابله با آن نیز به نوبه خود تکامل یافته‌اند. در روزهای اولیه عصر کامپیوتر نقش نرم‌افزار در یک سیستم کامپیوتری نقش ثانویه تلقی شده و هزینه اساسی طراحی یک سیستم کامپیوتری برای سخت‌افزار پرداخت می‌شد. چون در آن روزها قابلیت سخت‌افزار بسیار محدود بوده، برنامه‌ها ساده و کوچک بوده و زبان رایج همان زبان ماشین بود. سپس برای تسهیل در نوشتن برنامه‌های بزرگتر زبان اسمبلی ابداع شد. با اینکه در آن زمان سخت‌افزار همه منظوره وجود داشت اما نرم‌افزارها تک منظوره بودند بدین معنی که برای کاربرد بخصوصی طراحی شده‌اند[1]. بیشتر نرم‌افزارها بوسیله یک نفر طراحی، پیاده‌سازی، تست، نگهداری و حتی اجرا می‌شد لذا ظاهرا نیازی به مستند سازی نبوده است، و با توجه به طبیعت شخصی این محیط، فرآیند طراحی به صورت ضمنی در ذهن برنامه‌نویس صورت می‌گرفت بدون اینکه طرح خود را روی وسیله ای در خارج از ذهن خود مانند کاغذ نمایش دهد.

ولی در اوایل دهه ۷۰ میلادی این وضعیت شروع به دگرگونی نمود. سخت‌افزار سریعتر، قابل اطمینان‌تر و ارزانتر شده است و این پیشرفت شگفت‌انگیز سخت‌افزار باعث اقتصادی شدن فرآیند خودکارسازی بسیاری از کاربردهای صنعتی و تجاری و این بمعنی افزایش تقاضا برای سیستم‌های نرم‌افزاری پیچیده‌تر است. در مقابل این فشار زبان‌های سطح بالا[2] بعنوان ابزارهای مهم تولید سیستم‌های نرم‌افزاری وارد صحنه شده‌اند. در واقع این سیستم‌ها بازدهی برنامه‌نویسان منفرد و تیم‌های نرم‌افزاری را افزایش داده بودند.

در دهه ٦۰ و ۷۰ میلادی توجه پیشگامان رشته نرم‌افزار به «طراحی» بعنوان ابزار مهم مقابله با پیچیدگی سیستم‌های نرم‌افزاری معطوف گشته است. لذا در این دو دهه و بعد از آن روش‌های زیادی ابداع شده است که مهمترین آنها طراحی ساخت‌یافته است. با توجه به تقسیمی که آقای Sommerville پیشنهاد کرده است، سه طبقه‌بندی کلی برای روش‌های طراحی نرم‌افزار وجود دارد [۵].

[1] Custom-Build Software
[2] High Level Languages

❖ طراحی ساخت‌یافته (یا بالا به پایین)

در این روش هر واحد در سیستم یک گام از یک فرآیند کلی را نمایش می‌دهد. به عبارت دیگر سیستم نرم‌افزاری به صورت مجموعه‌ای متوالی از توابع دیده می‌شود. تاکید بیشتر این روش روی فرآیندهای سیستم و نه روی داده‌های آن است. به علاوه ارتباط میان این دو (داده و فرآیند) ارتباط ضعیفی تصور شده است.

❖ طراحی مبتنی بر دادها[1]

در این روش ساختار سیستم با نگاشت ورودی‌ها به خروجی‌ها تعیین می‌شود. این روش مانند طراحی ساخت‌یافته در بعضی از کاربردهای پیچیده موفقیت خود را ثابت نموده است بخصوص در زمینه سیستم‌های اطلاعاتی که در آن یک رابطه مستقیم بین ورودی‌های سیستم و خروجی‌های آن وجود دارد.

❖ طراحی شی‌گرایی

نگرش شی‌گرایی یک روشی برای تفکر در مورد یک مسأله با استفاده از مفاهیم موجود در دنیای واقعی به جای مفاهیم موجود در دنیای کامپیوتر است. زیر ساخت اساسی برای این نوع تفکر همان مفهوم شی است که ضمن مدل سازی اشیاء واقعی(با توجه به میزان تجرید مورد نظر) قابلیت تلفیق ساختمان داده‌ها و رفتار و اعمال روی آن را در یک واحد به نام شی داراست.

در این نگرش دنیای نرم‌افزار را به عنوان سیستم‌هایی می‌توان در نظر گرفت که شامل مجموعه‌ای از اشیاء مستقل از همدیگر بوده ولی بین آنها روابطی حاکم است که بر اساس مدلسازی این روابط و اثرات متقابل آنها، ارائه راه حلی برای یک مسأله از طریق نرم‌افزار در دنیای واقعی میسر می‌شود.

در مدل شی تعریف یک سیستم و عناصر اصلی آن بر چهار اصل زیر استوار است:

- تجرید یا چکیده سازی[2]
- پنهان سازی جزئیات یا محصورسازی[3]
- واحدبندی[4]
- سلسله مراتب[5]

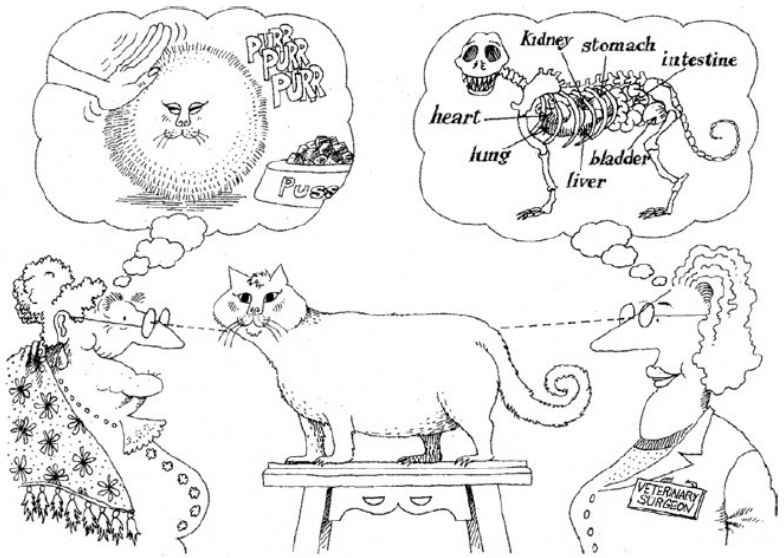
---

[1] Data-Driven Design
[2] Abstraction
[3] Encapsulation
[4] Modularity
[5] Hierarchy

در ادامه به ارائه توضیحی درباره هرکدام از این مفاهیم می‌پردازیم.

۲-۲- تجرید (چکیده‌سازی، انتزاع)

تجرید عبارتست از فرآیند متمرکزشدن و تاکید کردن روی ویژگی‌ها و رفتارهای اصلی و ذاتی یا مهم یک شی- نسبت به یک دید مشخص- و نادیده گرفتن ویژگی‌ها موقت یا غیر مهم آن شی- البته نسبت به همان دید- است. این فرآیند به ساختن یک مدل یا یک دید از آن شی منتهی می‌گردد که به این مدل نیز تجرید (یا به عبارت دقیقتر مجرد) گفته می‌شود.



شکل ۲-۱- تمرکز بر ویژگی‌ها در تجرید

❖ نقش تجرید در کنترل پیچیدگی

اصل تجرید به مدلساز یا طراح سیستم کمک می‌کند که به عوض در نظر گرفتن همه جزئیات مربوط به یک سیستم تنها ابعاد اساسی آن را —از زاویه دید خود- مد نظر قرار دهد. در واقع جزئیات بی‌شماری در مورد یک پدیده و یا یک سیستم می‌تواند مطرح باشد که توجه به همه آنها در آن واحد کاری غیر ضروری و نامطلوب است و بدین ترتیب می‌توان اشیاء موجود در یک سیستم را به طور مستقل و فارغ از سایر جزئیات غیر مهم (از دید طراح) بررسی نمود. از این رو تجرید ابزاری مهم برای کنترل و غلبه بر پیچیدگی به شمار می‌آید.

نکات زیر درباره تجرید مطرح است:

- چون همیشه تجرید با توجه به یک دید معین و برای یک منظور مشخص ساخته می‌شود، بنابراین برای یک شی انواع گوناگونی از تجرید وجود دارد. بعنوان مثال وقتیکه می‌خواهیم آناتومی بدن انسان را مورد مطالعه قرار دهیم، می‌توان این کار را با توجه به چند دید انجام داد: مثلا برای متخصص ارتوپدی مشاهده اسکلت بندی انسان حائز اهمیت بوده در صورتیکه برای متخصص اعصاب مشاهده نقشه سیستم عصبی انسان و برای متخصص خون مشاهده سیستم گردش خون در انسان مورد توجه است. اینها همه انواع گوناگونی از تجرید برای شی واحدی که همان انسان است. در واقع هر متخصص از زاویه تخصص خود به بدن انسان می‌نگرد لذا هر کدام، سایر جزئیات دستگاه بدن بعنوان یک مکانیزم همه جانبه را نادیده می گیرد، و فقط روی جنبه مورد نظر خود متمرکز می‌شود.
- تجرید همواره با نمود خارجی[1] از یک شی سر وکار دارد. به عبارت دیگر تجرید روی اینکه یک شی چیست و چه کار میکند تاکید مینماید و جزئیات مربوط به نحوه پیاده‌سازی آن شی را نادیده می گیرد.
- تجرید سطوحی دارد، هرچه به سمت بالای آن سطوح حرکت کنیم به معنای تاکید کردن و متمرکز شدن روی اطلاعات مهمتر است. همچنین حجم اطلاعاتی که باید بدانیم و تعداد قطعه‌های اطلاعاتی کمتر می‌شود. برعکس هرچه به سمت پایین حرکت نماییم با جزئیات بیشتر، تعداد قطعات بیشتر و حجم اطلاعات بزرگتر روبرو می‌شویم. بنابراین با انتخاب سطح تجرید مناسب، می‌توان به میزانی از جزئیات که دلخواه و مورد نظر طراح باشد پرداخت و از تزاحم سایر جزئیات جلوگیری نمود.
- همه تجریدها دارای ویژگی‌های ساکن[2] و پویا[3] هستند. بعنوان مثال شی فایل را در نظر بگیرید، یک فایل دارای اسم، اندازه و محتویات است. خود این ویژگی‌های ثابتند ولی مقادیر آنها متغیرند زیرا اسم، اندازه و محتویات در طول حیات فایل تغییر می یابد.

❖ انواع تجرید

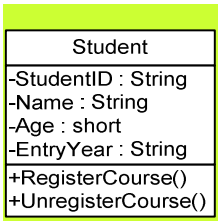
تجرید دارای انواع مختلفی است که بشرح زیر می‌باشد [۲]:

[1] Outside View
[2] Static
[3] Dynamic

۱. تجرید موجودیت[1]: یک مدل مفید از یک موجودیت واقعی(شی)-که در قلمرو مسأله مطرح است- را نمایش می‌دهد.

Real Object: Student

Abstraction: Student



شکل ۲-۲- تجرید موجودیت

۲. تجرید رفتار[2]: یک مجموعه عمومی از اعمال[3] که عملکرد یکسانی دارند را نمایش می‌دهد. مانند عمل اضافه کردن به لیست.

۳. تجرید مجازی[4]: عبارتست از مجموعه‌ای از اعمال متکی بر یک سطح پایینتر از خود و یا این که خود به عنوان یک سطح پایینتر برای استفاده یک سطح بالاتر مورد استفاده قرار می‌گیرند. مثال آن معماری لایه ای پروتکل TCP/IP است.

مهمترین نوع تجرید همان تجرید موجودیت است و باید سعی و تلاش ما بر این باشد که فقط از این نوع تجرید در مدل خود استفاده نماییم زیرا فقط این نوع با موجودیت های واقعی تطبیق می‌نماید.

مثال‌هایی از تجرید عبارتند از:

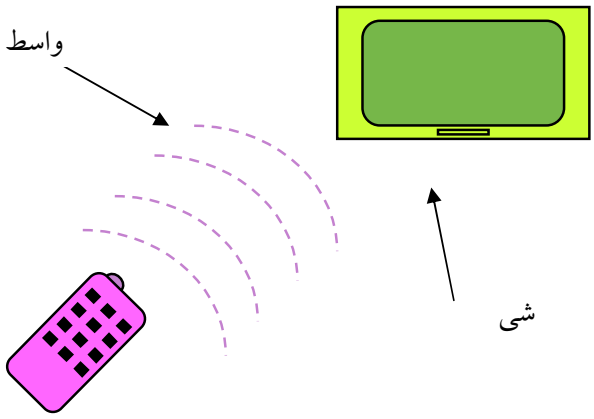
- بیان روابط میان اجزاء یک سیستم مکانیکی توسط یک معادله ریاضی.
- استفاده از یک نماد برای نمایش حضور یک موجودیت در یک صحنه خاص(مثلاً شکل زیر برای نمایش انسان)

- نمایش گرافیکی از رفتار یک سیستم.

[1] Entity Abstraction
[2] Action Abstraction or Procedural Abstraction
[3] Operations
[4] Virtual Abstraction

۲-۳- پنهان‌سازی جزئیات[1] (محصورسازی)

پنهان‌سازی جزئیات عبارت از عدم پذیرش تاثیرات ناخواسته و یا کنترل‌نشده و محدود کردن راه‌های دسترسی و استفاده از یک شی است. این فرآیند با جداسازی ویژگی‌های اصلی یک شی -که برای اشیاء دیگر قابل دسترسی است- از جزئیات پیاده‌سازی این شی -که می‌بایستی از بقیه اشیاء مخفی شود- انجام می‌پذیرد. بعبارت بهتر پنهان‌سازی جزئیات همان فرآیند مخفی‌سازی جزئیات پیاده‌سازی یک شی است.



شکل ۲-۳- محصورسازی

برای درک بهتر این مطلب از مفاهیم برنامه‌نویسی شی گرا کمک می گیریم:

می‌دانیم که مفهوم تجرید با ساختار Class در اغلب زبان‌های شی گرا پیاده‌سازی می‌شود که در این زبان‌ها هر کلاس باید دو قسمت داشته باشد:

- واسط[2]: در واقع همان تجرید یک شی بوده زیرا در این قسمت توصیف رفتار مشترک بین همه نمونه‌های کلاس بیان می‌شود. سرویس‌های یک کلاس تنها بوسیله واسط آن دسترسی پذیرند.
- پیاده‌سازی: در این قسمت رفتار مورد نظر کلاس پیاده‌سازی می‌شود.

❖ نقش پنهان سازی جزئیات در کنترل پیچیدگی

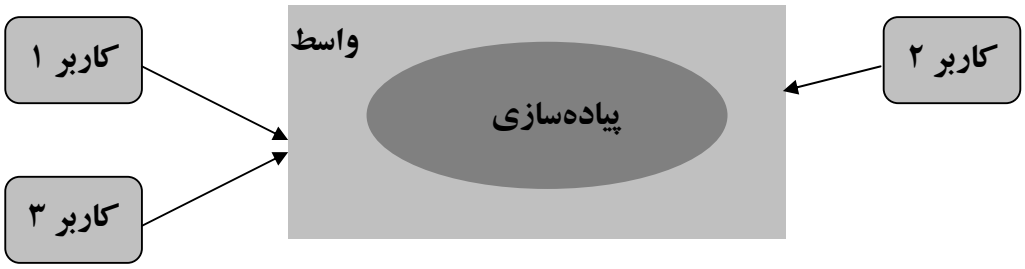
در مورد دسترسی به اشیاء و استفاده از آنها نیز، در مواردی بسیار برای جلوگیری از خرابکاریهای احتمالی و خطرات متقابل آنها نسبت به یکدیگر لازم است که اشیاء بشکل کاملا حمایت شده و تعریف شده ای اعمال مرتبط با یکدیگر را انجام دهند و بدین صورت گستره خطاها در خود شی محدود می‌شود

[1] Encapsulation
[2] Interface

و نگهداری برنامه آسانتر می‌شود. همچنین با استفاده از پنهان سازی جزئیات، تغییرات در پیاده‌سازی یک شی-تا وقتی که واسط آنرا تغییر نکرده باشد-می‌تواند به آسانی و باقابلیت اعتماد بالا انجام شود و بدون اثرگذاری روی اشیاء استفاده کننده از سرویس‌های این شی است.

نکات زیر درباره پنهان سازی مطرح است:

- قاعده محصورسازی ایجاب می‌کند که ارتباط بین اشیاء تنها از راه واسط‌ها باشد. به عبارت دیگر هیچ قسمتی از یک سیستم نرم‌افزاری نباید به جزئیات داخلی یک قسمت دیگر وابسته باشد. شکل ۲-٤ نمونه‌ای محصورسازی واسط را نشان می‌دهد.



شکل ۲-٤- نمونه‌ای از محصورسازی

- گاهی بین دو مفهوم تجرید و محصورسازی خلط می‌شود، برای بیان رابطه این دو با یکدیگر باید گفت که می‌توان از تجرید به عنوان مکانیزم تعیین جزئیاتی که باید پنهان شود، استفاده نمود اما خود عمل پنهان سازی جزئیات (که شامل طراحی واسط و پنهان‌سازی پیاده‌سازی) را محصورسازی گویند.
- محصورسازی یک مفهوم نسبی است: آن چیزی که در یک سطح از تجرید مخفی است می‌تواند نمود خارجی یک سطح دیگری باشد.

۲-۴- واحدبندی

قبل از بیان مفهوم واحدبندی مفاهیم واحد، انسجام و وابستگی را توضیح می‌دهیم.

- **واحدها**

واحدها عبارت از واحد تشکیل‌دهنده ساختار فیزیکی سیستم نرم‌افزاری است. به بیان دیگر در یک سیستم نرم‌افزاری طراحی شده به روش OO، کلاس‌ها و شی‌ها ساختار منطقی سیستم را تشکیل می‌دهند و با گروه‌بندی تجریدهای منطقا مرتبط در یک واحد، ساختار فیزیکی سیستم مشخص می‌گردد.

برای درک بهتر این مفهوم طراحی یک بورد کامپیوتری را در نظر بگیرید، در این فرآیند ابتدا با استفاده از گیت‌های منطقی(NAND, AND, NOR،..) رفتار مورد نظر را پیاده‌سازی می‌نماییم (سطح طراحی منطقی). اما برای پیاده‌سازی این طرح باید از ICهای موجود که عبارت از بسته‌های استاندارد شده این گیت‌ها هستند، استفاده شود. در این مثال بسته‌های IC با واحدهای نرم‌افزاری متناظرند.

- **انسجام**[1]

انسجام عبارتست از خاصیتی متعلق به درجه ارتباط عملکردی[2] عناصر داخلی یک واحد نسبت به هم است. واحد A را در نظر بگیرید، اگر همه عناصر A برای رسیدن به یک هدف منسجم و واحد با هم همکاری می‌کنند پس A یک واحد کاملاً منسجم[3] است. به هر اندازه که عناصر واحد A وظایف گوناگونی را انجام می‌دهند و به هر اندازه که ارتباط این وظایف با همدیگر ضعیف باشد، به همان اندازه درجه انسجام ضعیفتر خواهد بود.

- **وابستگی**[4]

وابستگی عبارتست از درجه ارتباط واحدهای گوناگون بهم دیگر است. دو واحد A و B را در نظر بگیرید. اگر درک عملکرد واحد A مستلزم درک نسبتاً کامل عملکرد واحد B باشد پس درجه وابستگی بین A و B بالا[5] خواهد بود. اما اگر این رابطه ضعیف است انگاه درجه وابستگی بین A و B ضعیف[6] خواهد بود.

حال می‌توان واحدبندی را تعریف نمود:

«سیستمی را واحدبندی شده گویند که به مجموعه‌ای از واحدهای منسجم و معنی‌دار که وابستگی بین آنها حداقل است، تجزیه شده باشد.»

❖ نقش واحدبندی در کنترل پیچیدگی

یکی از روش‌های مقابله با پیچیدگی سیستم‌ها شکستن یک مسأله به اجزایی کوچکتر است که میزان هزینه و تلاشی را که باید صرف حل چنین اجزای کوچکتری بنماییم در مجموع کمتر از زمانی است که بخواهیم کل مسأله را یکباره حل کنیم.

---

[1] Cohesion
[2] Function Relatedness
[3] Completely Cohesive
[4] Coupling
[5] A and B are Highly Coupled
[6] A and B are Loosley Coupled

بعبارت دیگر فرض کنید مسئله P را به زیر مسئله‌های P1، P2 و P3 قابل شکستن می‌باشد. آنگاه رابطه زیر خواهیم داشت:

Complexity (P) > Complexity (P1) + Complexity (P2) + Complexity (P3)

زیرا وقتی P شکسته شود، وابستگی بین P1، P2 و P3 در نظر گرفته نمی‌شود.

بنابراین می‌توان نوشت:

E(P) > E(P1) + E(P2) + E(P3), E: تلاش

نکات زیر درباره واحدبندی مطرح است:

- اگر شرایط بیان شده در تعریف واحدبندی رعایت گردد آنگاه واحدهای بدست آمده قابلیت استفاده مجدد خواهد داشت.
- تعداد اجزا : اگر مسئله به n قسمت شکسته گردد آنگاه n(n-1)/2 رابطه بین قسمت ها خواهیم داشت. حال اگر n بالا رود آنگاه با واحدهای بسیار کوچک روبرو خواهیم بود که خود یک مشکل پدید می‌آورد. یکی از وظایف طراح اینست که متوسط n را بدست آورد (n بر اساس سرویس‌های لازم در سیستم تعیین می‌گردد).
- معیار تجزیه: تعییین معیاری مناسب برای شکستن یک سیستم به واحدهای کوچکتر مهمترین عامل موفقیت در استفاده از خاصیت واحدبندی بوده و در عین حال به هیچ وجه کار آسانی نیست (در واقع سختی این کار متناظر با سختی انتخاب مجموعه کلاس‌های مناسب یک مسأله است). لذا در انتخاب واحدها باید دقت کافی به خرج داد چه بسا انتخاب ناآگاهانه گاهی بدتر از عدم استفاده از خاصیت واحدبندی است. اگر این کار بدرستی انجام شود و همچنین واحدهای بدست آمده به خوبی مستند شوند آنگاه مزیت سوم واحدبندی روشن می‌شود زیرا تقسیم برنامه به واحدهای خوش‌تعریف[1] که به خوبی مستند شده باشند در آسان کردن درک برنامه و نگهداری آن نقش مهمی ایفا می‌نماید.

۲-۵- سلسله مراتب

تا به حال سه مفهوم از مفاهیم اساسی مدل شی مطرح نموده و نقش هر کدام در کنترل پیچیدگی نرم‌افزار را بیان نمودیم. برای رعایت پیوستگی مطالب و روشن شدن نقش سلسله مراتب، تسلسل منطقی

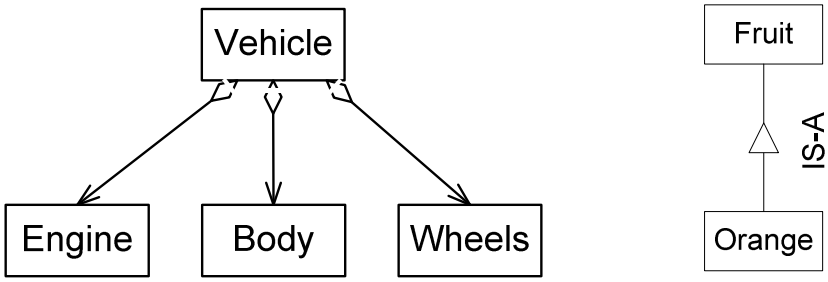
[1] Well-Defined Modules

مکانیزم‌های مدل شی برای مقابله با پیچیدگی سیستم‌های نرم‌افزاری را بیان می‌کنیم: گام اول در این راه متمرکز شدن روی ویژگی‌های اساسی سیستم و نادیده گرفتن جزئیات غیر مهم آن نسبت به یک دید معین که همان مفهوم تجرید است. در بیشتر سیستم‌های واقعی با وجود استفاده از این خاصیت معمولاً با کلاس‌های(تجریدهای) زیادی که درک همزمان آنها برای ما بسیار مشکل است، روبرو خواهیم گشت. گام دوم پنهان نمودن نمای داخلی سیستم و محدود کردن روش دسترسی به اشیاء بوسیله استفاده از واسط‌ها است. همچنین واحدبندی از طریق ارائه راهی بمنظور دسته بندی تجریدهای منطقا مرتبط کمک شایانی می‌کند. اما این به تنهایی کافی نیست و اغلب مجموعه‌ای از تجریدها خود تشکیل دهنده یک سلسله مراتب هستند. با تشخیص این سلسله مراتب ها و در نظرگرفتن آنها در طراحی، درک ما نسبت به مسأله به صورت قابل توجهی افزایش می‌یابد.

سلسله مراتب به صورت زیر تعریف می‌شود:

«سلسله مراتب عبارت از مرتب ساختن تجریدها در سطوح مختلف است.»

سلسله مراتب دارای چند نوع بوده که مهمترین آنها سلسله مراتب ساختار کلاس(IS-A)[1] و سلسله مراتب ساختار شی (PART-OF)[2] می‌باشد (شکل ۲-۵).



شکل ۲-۵- رابطه IS-A و رابطه Part-of

وراثت مهمترین شکل سلسله مراتب IS-A بوده و یک عنصر اساسی در سیستم‌های شی‌گرا است. وراثت عبارت است از «رابطه بین چند کلاس که در آن یک کلاس در ساختار، رفتار یا هر دو با یک کلاس(وراثت یگانه[3]) یا چند کلاس(وراثت چندگانه[4]) دیگر شرکت دارد.» به عبارت بهتر، وراثت

[1] also Kind-Of or Generalization/Specialization Realtionship
[2] also Aggregation or Whole/Part Relationship
[3] Single Inheritance
[4] Multiple Inheitance

عبارتست از سلسله مراتبی از تجریدها که در آن کلاس فرزند خصوصیات کلاس پدر را به ارث می برد. در واقع کلاس فرزند یک تخصیصی از کلاس عمومی‌تر(کلاس پدر) را نمایش می‌دهد.

بدین صورت که کلاس فرزند علاوه بر فیلدها (ساختار) و عملیات (رفتار) اختصاصی خود شامل ساختار و رفتارکلاس پدر است. با توجه به این مطالب به خوبی روشن است که با استفاده از وراثت می‌توان سیستم‌هایی را ساخت که دارای ویژگی اقتصاد در بیان باشند.

سلسله مراتب دوم همان سلسله مراتب PART-OF است که در آن یک کلاس از یک یا چند کلاس دیگر تشکیل می یابد. مثلا یک کامپیوتر از یک پردازشگر، مانیتور و صفحه کلید تشکیل می‌شود.

❖ نقش سلسله مراتب در کنترل پیچیدگی

با سازمان دهی قطعه‌های اطلاعاتی منفصل(تجریدها) در سلسله مراتب‌های IS-A و PART-OF درک ما نسبت به سیستم بهبود می یابد. اساسا ذهن انسان به گونه‌ای شکل گرفته که ترجیح می‌دهد برای مدیریت بهتر یک مسأله آنرا در لایه‌های متفاوتی از تجریدها مورد بررسی قرار دهد.

اهمیت سلسله مراتب PART-OF در این است که روابطی که بین اشیاء مختلف در یک سیستم موجود بوده و نحوه همکاری این اشیاء را که به صورت الگوهایی[1] از فعل و انفعالاتی[2] که بین آنها رخ می‌دهد بیان شده‌اند، را نمایش می‌دهد.

اهمیت سلسله مراتب IS-A در این است که افزونگی[3] موجود در سیستم را مدیریت نموده و بوسیله آن سیستم‌هایی با خاصیت اقتصاد در بیان را می‌توان پیاده‌سازی کرد. یک نکته قابل توجه است که استفاده از وراثت با پنهان‌سازی تام، تعارض دارد زیرا مستلزم دسترسی مستقیم کلاس فرزند به بعضی از عملیات و داده‌های اختصاصی[4] کلاس پدر است.

۲-٦- مزایای مدل شی و کاربردهای آن

- هدف نهایی تکنولوژی شی‌گرا انجام فرآیند توسعه نرم‌افزار شبیه به روشی که در تولید سخت‌افزار استفاده می‌گردد، یعنی از طریق گروه‌بندی اشیا در لایه‌های مختلفی از تجرید است.

[1] Patterns
[2] Interactions
[3] Redudancy
[4] Private Members

- تکنیک‌های سنتی موجود توان پاسخگویی به پیاده‌سازی گونه‌های مختلف از سیستم‌های پیچیده امروزی را ندارند. این سیستم‌ها نیازمند ساختاری برای مدیریت مدل‌های پیچیده‌اند. مکانیزم‌های موجود تکنولوژی شی‌گرایی پتانسیل برخورد با گستردگی و پیچیدگی سیستم‌های تجارتی امروز را دارند.
- از طریق کاربرد اشیاء به بعنوان واحد مجتمع پذیر تفکیک ناشدنی، تاثیرات یک تغییر را محدود می نماییم که این خود، تشخیص تاثیرات تغییر پیشنهادی را آسانتر نموده و هزینه و زمان لازم برای تغییر را کاهش می‌دهد. از سوی دیگر این اشیاء وابستگی به اطلاعات دربرگرفته شده را محدودتر کرده و این خود نیز اثر تغییر را کاهش داده و نهایتا زمان توسعه نرم‌افزار را نیز کاهش خواهد یافت.
- محصورسازی و جداسازی لایه‌های معماری نرم‌افزار باعث ایجاد مقیاس پذیری و قابلیت توسعه تدریجی یک سیستم می‌شود.
- قابلیت انعطاف برای اجرای اشیاء بصورت توزیع شده در زمانیکه یک سیستم بزرگ از مجموعه‌ای از اشیاء تعریف شده است بوجود خواهد آمد. اشیاء نیازمند به قدرت پردازش بالا در سایت‌های قویتر قرار می‌گیرند و بر اساس نیازمندی‌های جغرافیایی این توزیع انجام می‌گیرد.
- قابلیت استفاده مجدد: امروزه تمایل زیادی به سمت ساخت عناصر نرم‌افزاری استاندارد که می‌توان از آنها-بدون تغییر- در توسعه نرم‌افزارهای مختلف استفاده نمود (معادل نرم‌افزاری IC). ساخت مؤلفه‌ها[1] که بر دیدگاه شی‌گرایی مبتنی است می‌تواند زمینه ساز چنین تحولی باشد.

[1] Components

۳- آشنائی با مفاهیم اولیه شی‌گرائی

۳-۱- مفاهیم اساسی

▪ شی[1]

کلمه Object در فارسی تحت عنوان مفهوم، ایده عینیت، هدف و شی ترجمه شده است که متاسفانه هیچیک از این عناوین بیانگر معنی دقیقی از کلمه Object نیستند. اما شاید نزدیکترین کلمه معادل فارسی همان «شی» یا «چیز» باشد و البته مقصود ما از شی یک مفهوم کلی است بگونه‌ای که دارای هویت بوده و قادر به بروز رفتار و ثبت حالات (وضعیت) خود باشد. مثلاً یک ماشین نمونه کاملی از یک شی است. انسان، درخت، موجودات زنده و حتی عناصر بی‌جان نیز گونه‌ای از اشیاء هستند که در دنیای واقعی دارای هویتی مستقل بوده و از خود رفتار نشان داده (چه رفتار فعال [عمل] و چه رفتار غیر فعالانه[عکس‌العمل]) و نیز حالاتی در آنها وجود دارد که گویا یکی از وضعیت‌های کلی است که آن شی می‌تواند در آن بسر ببرد. بسته به فضا و محیطی که به آن می‌اندیشیم و سطح تجرید مورد علاقه می‌توان اشیاء را تمییز داده و بگونه‌ای کاملاً روشن تبیین نمود.

از نظر تکنیک شی‌گرائی، یک شی دارای سه مشخصه ذاتی زیر می‌باشد [۱]:

۱. هویت[2]: آن ویژگی از یک شی است که آن را از سایر اشیاء متمایز می سازد. هویت نهفته در ذات شی است لذا دو شی که از همه جهات مشابه یکدیگر باشند، همچنان دو شی به شمار می‌آیند نه یک شی.

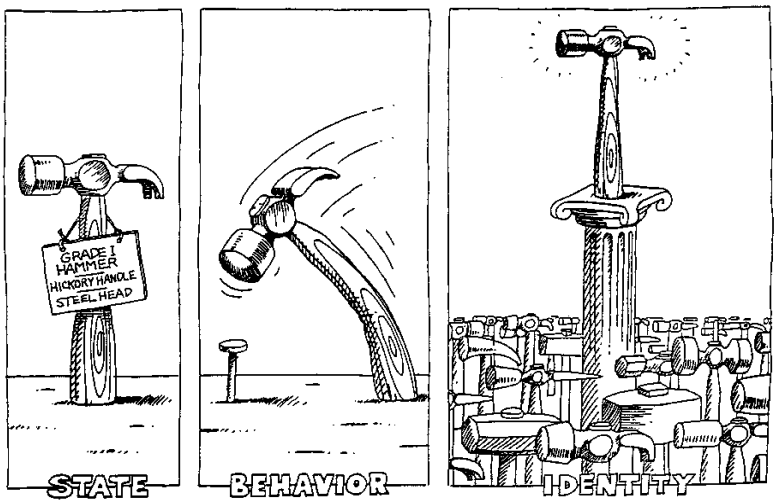
در زبان‌های شی‌گرا هویت یک شی با یک اسم منحصر بفرد(یا یک Handle) نمایش داده می‌شود.

۲. حالت[3]: حالت یا وضعیت یک شی در بردارنده تمام خواص آن شی(معمولا ایستا) بعلاوه مقادیر جاری(معمولا پویا) برای هر یک از این خواص است.

۳. رفتار[1]: رفتار؛ چگونگی عمل و عکس العمل یک شی در قالب تغییر حالت در مقابل دریافت و یا ارسال پیام است، را نشان می‌دهد.

---

[1] Object
[2] Identity
[3] State



حالت **رفتار** **هویت**

شکل ۳-۱- شی و مشخصات آن

مثال‌هایی از اشیاء عبارتند از:

- موجودیت‌های خارجی[۲]
- اسباب[۳]
- نقش‌ها[٤]: مدیر، کارمند، معمار نرم‌افزار
- واحدهای سازمانی[٥]
- مکان‌های فیزیکی
- ساختارها

در زبان‌های شی‌گرا یک شی به صورت زیر نمایش داده می‌شود:

Object Name
Attributes
Operations

شکل ۳-۲- نمایش شی در زبان‌های شی‌گرا

---

[1] Behaviour
[2] External Entities
[3] Things
[4] Roles
[5] Organization Units

مثال: کتاب

- هویت: کتاب
- صفات(حالت):
  - اطلاعات فهرست نویسی
  - مکان نگهداری فیزیکی
  - وضعیت فعلی(امانت/رزرو/آزاد)
- رفتار:
  - ثبت اطلاعات کتاب
  - جستجو
  - سفارش دادن

بنابراین می‌توان گفت که:

Object = Data Structure (S) + Algorithm(s)

اگر به جهان بنگریم اشیاء زیادی پیدا می‌کنیم که عبارتند از نمونه هائی از یک مفهوم کلی تر که آنـرا کلاس می‌نامیم. برای مثال ماشین پیکان که رنگ زرد دارد و متعلق به آقای فلان یک شی است. این شـی نمونه‌ای از یک مفهوم کلی‌تر که همان ماشین می‌باشد.

- **کلاس**[1]

مجموعه‌ای از اشیاء که دارای ساختار و رفتار مشترک باشند را یک کلاس می‌نامیم. مزیت گروه‌بندی اشیاء در مفهوم کلاس مدیریت بهتر و قابلیت استفاده مجدد است. در واقع یک الگوی کلی داریم کـه بـر حسب نیاز اشیائی از آن برداشت می‌نماییم. چنانکه می‌دانیم نوع داده مجرد[2] امکانی را فراهم می‌آورد کـه باعث شناسائی(معرفی) اشیاء از طریق بیان ساختار و رفتار آنها بدون نیاز به پیاده‌سازی آن ساختار یـا رفتـار می‌شود.

در زبان‌های شی‌گرا کلاس از دو قسمت تشکیل شده:

Class Decleartion → ADT
Class Body → Behaviour Implementation

[1] Class
[2] Abstract Data Type

مثالی از یک کلاس (به زبان C++)

```
Class Student {
    Private:
        long student_id;
        char name[30];
        char birth_date[12];
        int entry_year;
        enum State{HAS_WORK = 1, HAS_NOT_WORK = 0};
        State cur_state;
    Public:
        Student ( );     //Constructor
        ~Student ( );   //Destructor
        void ChangeState( State new_state );
        void Display( );
};
```

توجه داشته باشید که در برنامه‌نویسی شی‌گرا، هر شی کلاس خود را می‌شناسد.

- **نمونه**[1]

یک نمونه به یک مورد مشخص از یک کلاس اشاره می‌نماید. عمل تعریف یک شی در برنامه‌نویسی شی‌گرا را Instantation گویند. برای مثال: Studenet s1, s2;

- **ارتباط بین اشیاء**

مکانیزم ارتباط بین اشیاء و بهرمندی از سرویس‌ها (عملیات) آنها از طریق تبادل پیام[2] صورت می‌گیرد. در شی‌گرائی اصل Encapsulation از طریق محدود کردن راه استفاده اشیاء از یکدیگر در مکانیزم تبادل پیام اِعمال می‌گردد. برای تصور بهتر تبادل پیام به مفهوم Client/Server بعنوان مدلی مناسب برای نشان دادن ارتباط توجه کنید.

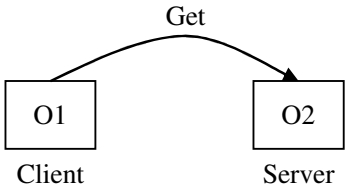
«شی ارسال کننده پیام تقاضادهنده[3] حساب شده و شی دریافت‌کننده پیام سرویس‌دهنده محسوب می‌گردد[4].»

مثال: شی O1 پیام Get به شی O2 می‌فرستد (شکل ۳-۳)

[1] Instance
[2] Message Passing
[3] Client
[4] Server

در C++ این پیام به صورت درخواست اجرای Method به نام ()Get از O2 نمایش داده می‌شود:

O2.Get(param_list);



شکل ۳-۳- رابطه Client/Server

- **نقش‌ها:** برای تعیین رفتار یک شی می‌توان از مسئولیت (نقش) آن شی استفاده نمود.
- **واسط‌ها**[1]: واسط‌ها نحوه استفاده از یک کلاس بدون نیاز به شناسائی جزئیات پیاده‌سازی آن را به ما نشان می‌دهند. در C++ خود تعریف کلاس (که معمولاً در Header File قرار می‌گیرد) واسط حساب می‌شود.

سه نوع واسط وجود دارد:

۱. عمومی[2]: برای همه استفاده‌کنندگان قابل دسترسی است.

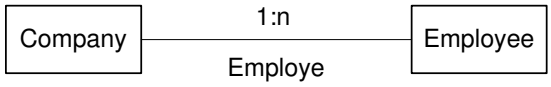
۲. اختصاصی[3]: تنها برای همان کلاس و دوستان آن کلاس قابل دسترسی است.

۳. حفاظت شده[4]: تنها برای خود کلاس، دوستان کلاس و زیرکلاس‌ها[5] قابل دسترسی است.

۳-۲- رابطه بین کلاس‌ها

به صورت کلی ۳ نوع ارتباط اصلی بین کلاس‌ها وجود دارد:

- **رابطه انجمنی**[6]: نوعی وابستگی معنایی[7] بین کلاس‌های متفاوت که با حذف وابستگی عملاً هیچ ارتباط بین دو کلاس وجود نخواهد داشت. مثال:



شکل ۳-٤- رابطه انجمنی

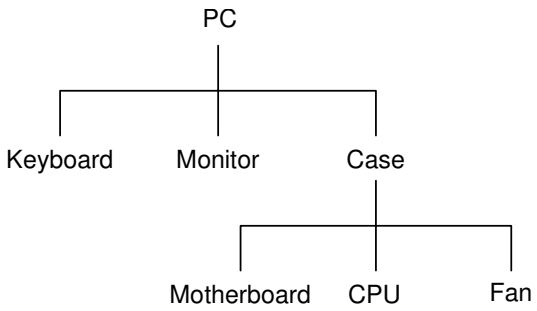
[1] Interfaces
[2] Public
[3] Private
[4] Protected
[5] Subclasses
[6] Association
[7] Semantic Relationship

در این رابطه دو مسئله اهمیت دارد:

۱) نوع وابستگی

۲) درجه وابستگی

- **رابطه تجمعی**[1]: زمانی که یک شی از تعدادی اشیاء دیگر تشکیل می‌گردد، این رابطه را تجمعی(جزئی-از) گویند (شکل ۳-۵).



شکل ۳-۵- رابطه تجمعی

- **رابطه وراثت**[2]**:** وراثت عبارت از رابطه بین چند کلاس که در آن یک کلاس در ساختار و رفتار یا هر دو با یک کلاس(وراثت یگانه) یا چند کلاس (وراثت چندگانه) دیگر شرکت دارد.

بعبارت دیگر وراثت عبارتست از سلسله مراتبی از تجریدها[3] که در آن کلاس فرزند[4] خصوصیات کلاس پدر[5] را به ارث می برد.

در واقع کلاس فرزند یک تخصیص[6] از کلاس پدر را نمایش داده و همزمان کلاس پدر یک تعمیم[7] از کلاس فرزند به حساب می‌آید.

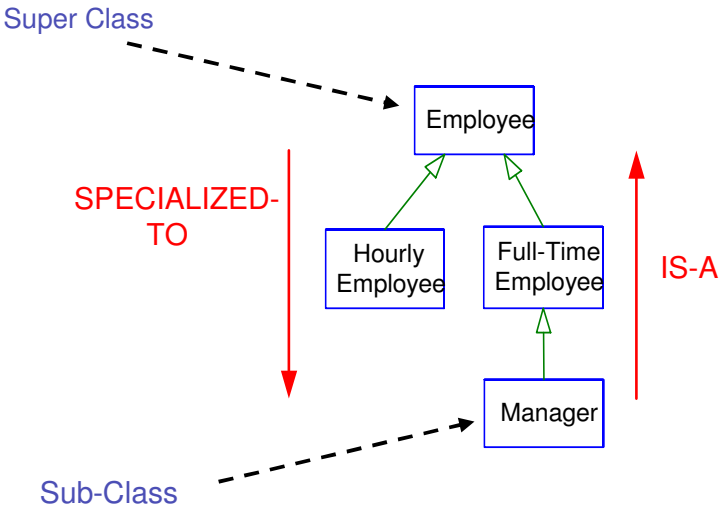
مثال:

A Rose *IS-A* Flower
A Flower *IS-A* Plant

---

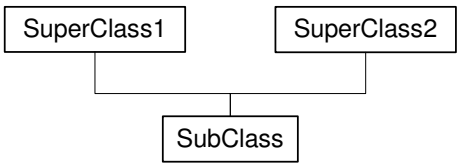
[1] Aggregation
[2] Inheritance or Generalization/Specialization or IS-A Relationship
[3] Abstract Hierarchy
[4] Subclass
[5] Superclass
[6] Specialization
[7] Generalization

شکل ۳-۶ مثالی از وراثت یگانه را نمایش می‌دهد.



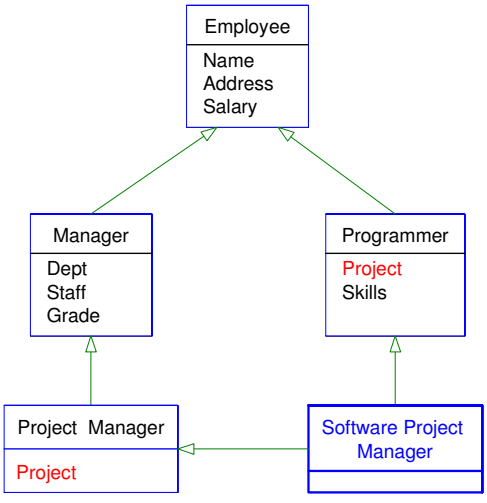
شکل ۳-۶- وراثت یگانه

شکل ۳-۷ مثالی از وراثت چندگانه را نشان می‌دهد.



شکل ۳-۷- وراثت چندگانه

به عنوان نمونه می‌توان انواع کارمند را به‌صورت شکل ۳-۸ نشان داد.



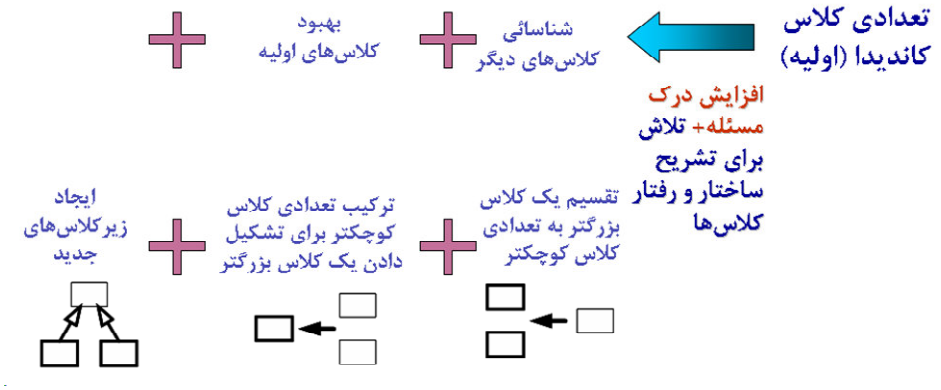
شکل ۳-۸-نمونه‌ای از وراثت – انواع کارمند

٤- شناسایی کلاس‌ها

٤-١- طبقه‌بندی

شناسایی کلاس‌ها و یافتن آنها در هر مسئله نیاز به درک شفافی از کلاس‌ها و طبقه‌بندی آنها دارد. طبقه‌بندی[1] کلاس‌ها ابزاری است که بوسیله آن، ما انسان‌ها دانش خود را مرتب می‌سازیم. از سوی دیگر، با استفاده از طبقه‌بندی توانایی ارتباط با سایر انسان‌ها نیز فراهم می‌شود، چرا که افراد یک جامعه دارای مفاهیم و معیارهای طبقه‌بندی یکسان هستند که هنجارها، قوانین، سنن و رفتارهای جامعه را شکل می‌دهند. مطمئناً یک طبقه‌بندی کامل وجود ندارد و طبقه‌بندی‌های متفاوت با توجه به معیارهای متفاوت و گوناگون وجود دارند.

در متدولوژی‌های شی گرا طبقه‌بندی کمک می کند تا کلاس‌هایی انتخاب شوند که با توجه به معیارهاو محدودیت‌های موجود سازگار باشند. فرآیند طبقه‌بندی متدولوژی‌های شی گرا، فرآیند تدریجی و افزایشی است، چرا که با هر بار طبقه‌بندی به دسته‌بندی بهتری خواهیم رسید. نکته مهمی که باید در نظر داشت، دستیابی به راه‌حل مناسب است و نه راه‌حل طلایی. به عبارت بهتر، شناسایی کلاس‌ها باید تا آنجا ادامه یابد که کلاس‌های مناسبی برای مسئله شناسایی و یافته شود و این امر می‌تواند با در نظر گرفتن قانون ۲۰/۸۰ انجام شود. در شکل ۴-۱ فرآیند طبقه‌بندی و کمک آن در شناسایی و بهبود کلاس‌هایی شناسایی شده به صورت مختصر نشان داده شده است.



شکل ۴-۱- استفاده از طبقه‌بندی در شناسایی و بهبود کلاس‌ها

[1] Classification

۴-۲- منابع تشخیص کلاس‌ها

بطور کلی دو منبع اصلی برای شناسایی کلاس‌ها وجود دارد:

- فضای مسئله: با استفاده از مدل تحلیل می‌توان برخی از کلاس‌ها را شناسایی نمود.
- فضای راه‌حل: با استفاده از مدل توصیفی حاصل از راه‌حل اولیه ایجاد شده، می‌توان کلاس‌های بهبودیافته را ایجاد نمود.

در واقع می‌توان گفت که فرآیند شناسایی کلاس‌ها شامل دو فعالیت کشف کلاس از فضای مسئله و ابداع کلاس‌های جدید از فضای راه‌حل است.

۴-۳- رهیافت‌های شناسایی کلاس‌ها

برای شناسایی کلاس‌ها دو رهیافت وجود دارد [۲]:

❖ مبتنی بر داده[1]

در این روش‌ها مبنای شناسائی کلاس‌های مناسب سیستم، شناسائی ساختار داده‌های مورد نیاز هر کلاس است. فرآیند تعیین کلاس‌ها با پرسیدن دو سوال صورت می‌گیرد:

۱- ساختار هر کلاس چیست؟

۲- چه عملیاتی بوسیله هر کلاس انجام می‌گیرد؟

برای پاسخ به این سوالات دو گام انجام می‌شود. در گام اول داده‌های مورد نیازی که باید نگهداری شوند، یافت می‌شوند و در گام دوم، عملیات مورد نیاز بر اساس ساختار داده‌های پیشنهادی را تعیین می‌شوند. بدین ترتیب مجموعه‌ای از کلاس‌ها یافت می‌شوند که باید نگهداری شوند. اما نکته اصلی در این روش وابستگی سرویس‌های کلاس به ساختار داخلی کلاس است که بنوبه خود سبب وابستگی سرویس‌گیرندگان به ساختار داخلی کلاس خواهد بود. این وابستگی سبب نقض اصل محصورسازی و استفاده از واسط می‌باشد و با هر تغییری در ساختار داخلی کلاس، سرویس‌گیرندگان دچار تغییر می‌شوند. به عبارت بهتر، این روش سبب طراحی مبتنی بر داده خواهد شد و وابستگی داده‌ای را در نظر می‌گیرد. تنها مزیت این روش، سادگی استفاده و بکارگیری آن است.

[1] Data-Driven

❖ مبتنی بر وظیفه[1]

در این روش‌ها مبنای شناسائی کلاس‌های مناسب سیستم، شناسائی مسئولیت‌های[2] مورد نیاز هر کلاس است. فرآیند تعیین کلاس‌ها با پرسیدن دو سوال صورت می‌گیرد:

۱- هر کلاس چه مسئولیتی دارد؟ (چه عملیاتی بوسیله این کلاس انجام می‌گیرد؟)

۲- این شی، چه اطلاعاتی با بقیه اشیاء به اشتراک می‌گذارد؟

برای پاسخ به این سوالات دو گام انجام می‌پذیرد. در گام اول، عملیات مورد نیاز تعیین می‌شوند و در گام دوم ساختار دادهای مورد نیاز تعیین می‌شوند. بدین ترتیب مجموعه‌ای از کلاس‌های تعیین می‌شوند که سرویس‌های آن‌ها بستگی به ساختار داخلی کلاس ندارند. بنابراین، سرویس‌گیرندگان نیز مستقل از ساختار داخلی خواهند شد. نگاه اصلی در این روش به هر کلاس، عبارت از موجودیتی است که در هر آن می‌تواند نقش سرویس‌دهنده یا نقش سرویس‌گیرنده را ایفا نماید.

هر کلاس در نقش سرویس‌دهنده می‌تواند فراهم‌کننده خدمات برای ۳ نوع سرویس‌گیرنده باشد:

۱. سرویس‌گیرندگان خارجی[3]

۲. سرویس‌گیرندگان مشتق شده[4]

۳. خود کلاس[5]

بزرگترین مزیت این روش به حداکثر رساندن محصورسازی در سطح طراحی است که باعث افزایش قابلیت نگهداری و انعطاف‌پذیری سیستم نسبت به تغییرات آتی خواهد گردید.

در واقع، تفاوت اصلی این دو روش در نقطه شروع شناسائی کلاس‌ها می‌باشد. در روش اول مبنای کار، شناسائی کلاس‌ها بر اساس ساختمان داده‌های مورد نیاز مسئله می‌باشد. در حالیکه در روش دوم کلاس‌ها بر اساس وظیفه‌ها شناسائی می‌شوند.

۴-۴- فرآیند شناسایی کلاس‌های اولیه

فرآیند شناسایی کلاس‌های اولیه حاوی گام‌ها و مراحل زیر است:

۱. استفاده از طبقه‌بندی‌های پیشنهاد شده بوسیله متدولوژی‌های شی‌گرا

---

[1] Responsibility-Driven
[2] Responsibilities
[3] External Clients
[4] Derived Clients
[5] Self Client

۲. تحلیل دامنه[1]

۳. تحلیل موارد کاربری[2]

٤. تحلیل لغوی صورت مساله[3]

٥. استفاده از الگوها[4]

٦. کارت‌های CRC

در ادامه به بررسی هر یک از این مراحل می‌پردازیم.

❖ طبقه‌بندی‌های پیشنهاد شده

منابع بالقوه زیر برای شناسائی کلاس‌های اولیه به صورت ذیل پیشنهاد می‌شوند:

۱. دستگاه‌ها[5]: دستگاه‌هایی که برنامه با آنها تعامل دارد

۲. نقش‌ها: نقش‌های گوناگون که کاربر در تعامل با سیستم ایفا می‌نماید

۳. محل‌های فیزیکی (مانند دفاتر، سایت‌ها،...) که برای سیستم مهم هستند

٤. سازمان‌ها: مجموعه‌های سازماندهی شده (مردم، منابع، ...) که دارای ماموریت‌های مشخصند

٥. مفاهیم منطقی: اصول و ایده‌های منطقی که در منطق کاری سازمان بکار گرفته می‌شوند

٦. ساختار: همان روابط IS-A و PART-OF

۷. دیگر سیستم‌ها: سیستم‌های خارجی که برنامه با آنها تعامل دارد

❖ تحلیل دامنه

عبارتست از شناسائی کلاس‌ها و اشیاء مشترک در همه برنامه‌های کاربردی متعلق به یک دامنه مشخص (مانند کامپایلرها، سیستم‌های اطلاعاتی جغرافیایی، ...). در این روش، با دیدن سیستم‌های مرتبط و اسناد آنها و با صحبت با کارشناسان خبره در زمینه سیستم مورد نظر می‌توان کلاس‌های کلیدی یک سیستم را حدس زد.

---

[1] Domain Analysis
[2] Use-Case Analysis
[3] Problem Statement Analysis
[4] Patterns
[5] Devices

❖ تحلیل موارد کاربری

هر مورد کاربرید «دنباله‌ای از عملیات است که یک سیستم انجام می‌دهد تا یک نتیجه قابل مشاهده و ارزشمند برای کاربر فراهم نماید». با استفاده از تحلیل موارد کاربری که در فصول بعد آنها را بیان می‌نمائیم می‌توان کلاس‌های کلیدی را یافت نمود.

❖ تحلیل لغوی صورت مساله

با تحلیل صورت مکتوب مساله می‌توان کلاس‌های اولیه را بدست آورد. برای اینکار نیاز است تا دو گام زیر انجام شود. در گام اول نام‌ها و فعل‌های موجود پیدا می‌شود و در گام دوم نام‌ها و فعل‌های غیرضروری باید حذف می‌شود. به این ترتیب تعدادی از کلاس‌ها و عملیات کاندیدا یافت می‌شوند.

❖ استفاده از الگو

یک الگو، یک مساله طراحی که در یک زمینه مشخص مرتباً تکرار می‌گردد را توصیف کرده و سپس یک راه‌حل کلی و تکرارپذیر برای آن ارائه می‌کند. هر الگو در حقیقت برای پاسخ به یک مشکل ایجاد شده است و بنابراین می‌توان در سیستم‌هایی که خواص مفهومی مشترک داشته و همان مسئله را دارند مورد استفاده قرار بگیرد. استفاده از الگوهای معتبر می‌تواند باعث توسعه قابلیت استفاده مجدد در سطح طراحی گردد و قابلیت اطمینان نیز را بالا ببرد.

❖ استفاده از روش کارت‌های CRC

کارت‌های CRC روشی غیررسمی[1] برای شناسائی و توصیف کلاس‌ها، رفتار و مسئولیت‌های آنها و همکارانی (کلاس‌های دیگر) که به کمک آنها وظایف خود را انجام می‌دهند. در ادامه به بررسی روش CRC خواهیم پرداخت.

۴-۵- روش CRC

در این روش هر کلاس در مسئله متناظر با یک کارت۲×۳ اینچ بوده که دارای ۳ فیلد نام کلاس، وظایف کلاس[2] و همکاران کلاس[3] در انجام وظایف خود می‌باشد.

[1] Informal
[2] Class Responsibities
[3] Class Collaborators

Class Name	
Responsibilities	Collaborators

شکل ۴-۲- نمونه ای از کارت CRC

همانطوریکه گفتیم این روش مبتنی بر وظیفه می‌باشد. یعنی مبنای شناسائی کلاس‌ها وظایف و مسئولیت‌های آنها می‌باشد. ولی چطور وظیفه کلاس را قبل از خود کلاس بشناسیم؟

در حقیقت، در بیشتر مسئله ها یک سری کلاس‌های مشهود و واضح وجود داشته که شناسائی آنها می‌تواند مبنای شناسائی کلاس‌های دیگر باشد.

بعبارت روشنتر اگر بازای هر کلاس مشهود وظایف سپس همکاران آنرا بنویسیم می‌توان بقیه کلاس‌ها را شناخت. برای شناخت کلاس‌های جدید به ازای هر وظیفه یا مسئولیت یک کلاس تلاش می‌کنیم جوابی برای این سوال که «اگر این وظیفه بخواهد انجام شود چه اتفاقی می افتد؟» پیدا نماییم بدین صورت همکاران کلاس شناسائی خواهند شد.

از قابلیت‌های روش CRC می‌توان به موارد ذیل اشاره نمود:

- سادگی روش: بر اساس یک بازی ساده با کارت‌ها
- طبیعی بودن روند کار و نمایش سناریوهای واقعی
- فرآیندگرایی براساس کار گروهی

برای مدلسازی با استفاده از کارت‌های CRC نیاز است تا گام‌های ذیل انجام شود:

۱. موارد کاربری کلیدی سیستم را مرور کنید.
۲. در صورت نیاز یکی (یا ترکیبی) از تکنیک‌های یافتن کلاس‌های اولیه را برای شناسائی مجموعه‌ای از کلاس‌های کاندیدا بکار ببرید.
۳. به ازای هر مورد کاربری گام‌های زیر را انجام دهید.

الف) از کلاس‌های موجود، کلاس‌هایی که مناسب این مورد کاربری است را مشخص نمایید.

ب) اگر کلاس مناسبی وجود نداشته باشد پس کلاس جدیدی را ایجاد نمایید.

ج) مسئولیت‌های کلاس را تشخیص دهید:

- ❖ این کلاس باید چه وظیفه‌ای را انجام دهد؟
- ❖ اگر وظیفه‌ای را در اختیار دارید، این وظیفه متعلق به کدام کلاس است؟
- ❖ بعضی از مسئولیت‌ها بوسیله همکاری کلاس با دیگر کلاس‌ها انجام می‌پذیرند، بنابراین عجله نکنید.

د) همکاران کلاس را تشخیص دهید:

- ❖ سناریوی «What if …?» را اجرا کنید
- ❖ همکاری هنگامی رخ می‌دهد که یک کلاس نیازمند اطلاعاتی باشد که در اختیار ندارد
- ❖ همکاری هنگامی رخ می‌دهد که یک کلاس نیازمند به روزرسانی اطلاعاتی باشد که در اختیار ندارد
- ❖ در هر همکاری، حداقل یک کلاس آغازکننده باید وجود داشته باشد

هـ) کارت‌های CRC را دور میز چرخش دهید:

- ❖ کارت‌های کلاس‌هایی که با یکدیگر همکاری دارند نزدیک هم قرار دهید
- ❖ هر چه همکاری قویتر باشد نزدیکی دو کلاس به یکدیگر می‌بایست بیشتر باشد
- ❖ کارت‌های پر(شلوغ) را در وسط میز قرار دهید
- ❖ کارت‌ها را دور میز بچرخانید
- ❖ از کسانی که در جلسه شرکت دارند بخواهید که به کارت‌های در حال چرخش توجه نمایند
- ❖ حاضرین در جلسه ارتباطات جدیدی بین کلاس‌ها تشخیص خواهند داد
- ❖ سناریوی «چه می‌شود اگر...؟» را اجرا نمایید

به‌عنوان نمونه، مسئله معروف تولیدکننده[1] و مصرف‌کننده را در نظر بگیرید: در این مسئله تولید کننده یک Item تولید می‌نماید که مصرف کننده آنرا مصرف می‌کند. برای هماهنگی این دو کلاس به یک بافر میانی نیاز داریم (که خود یک کلاس است). کارت‌های این مسئله به صورت شکل ٤-٣ هستند.

[1] Produce & Consumer Problem

Consumer	
Responsibilities	Collaborators
GetItem)(ConsumeItem)(	Buffer

Producer	
Responsibilities	Collaborators
ProduceItem)(PutItem)(	Buffer

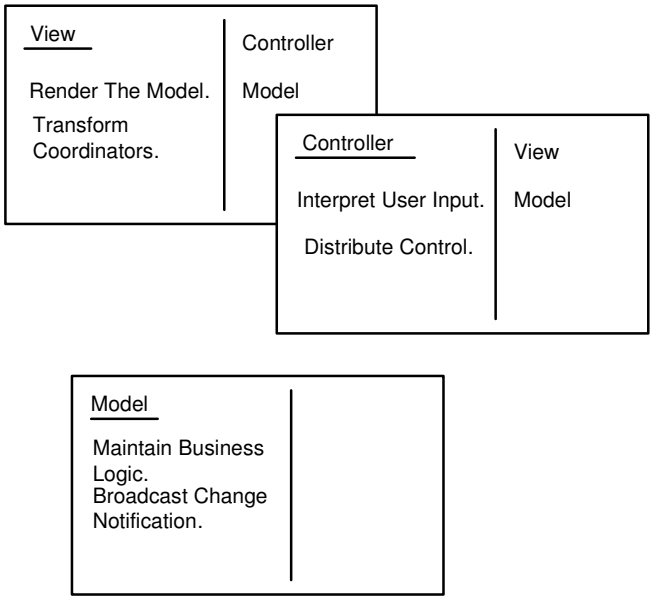
Buffer	
Responsibilities	Collaborators
SeizeBuffer)(ReleaseBuffer)(	

شکل ٤-٣- مسئله تولید کننده/مصرف کننده

کلاس تولید کننده را در نظر بگیرید: مسئولیت این کلاس اینست که یک عنصر تولید نماید و در بافر قرار دهد که برای تولید آن احتیاج به همکاری ندارد. در هنگام قرار دادن Item در بافر نیاز به همکار دارد که خود بافر است. بافر طوری است که یک عنصر بیشتر در آن جا می‌گیرد و غیر فعال[1] نیز هست.

به عنوان نمونه دیگر می‌توان الگوی MVC را در نظر گرفت. هر الگوی معماری در حقیقت برای پاسخ به یک مشکل ایجاد شده است و بنابراین می‌توان در سیستم‌هایی که خواص مفهومی مشترک داشته و همان مسئله را دارند مورد استفاده قرار بگیرد. استفاده از الگوهای معماری معتبر می‌تواند باعث توسعه قابلیت استفاده مجدد در سطح طراحی گردد و قابلیت اطمینان نیز را بالا ببرد. الگوی MVC از سه کلاس پایه استفاده می‌کند. کلاس Model برای نگهداری اطلاعات پایه مورد استفاده می‌گیرد، کلاس View برای نمایش نمودارهای مختلف و اصلاح این نمودارها مطابق با تغییر داده‌ها مورد استفاده قرار می‌گیرد، کلاس Conttoller که جزئیات انتخاب کاربر را بیان می‌کند. همکاری بین این سه کلاس سبب نمایش تغییرات روی نمودار و کنترل نمودارهای مختلف می‌شود. کلاس‌های و وظایف این سه کلاس در شکل شکل ٤-٤ ارائه شده‌اند.

[1] می‌توان اشیاء را به دو قسمت تقسیم نمود: فعال (Active) و غیر فعال (Passive). شئ فعال روی اشیاء دیگر تاثیر می‌گذارد (با فرستادن پیام) و شئ غیر فعال تاثیر می‌پذیرد. اگر شئ فعال و غیر فعال باشد آنگاه Agent نامیده می‌گردد.



شکل ٤-٤- کلاس‌ها و وظایف مربوط به هر کلاس با استفاده از الگوی CRC

از جمله مزایابی روش CRC می‌توان به موارد ذیل اشاره نمود:

- ❖ نقطه مناسبی برای شروع تحلیل (جرقه ذهنی)
- ❖ پوشش جنبه‌های اصلی یک سیستم (Encapsulation, Instantiation, Communication)
- ❖ قابلیت شبیه‌سازی رفتار سیستم (مرور سناریو)
- ❖ قابلیت انتقال حرکت در اطراف و تجمع
- ❖ با نمودار کلاس‌ها سازگار است
- ❖ امکان کارگروهی
- ❖ بیان معماری یک سیستم
  - در اختیار گرفتن جوهر[1] نرم‌افزار
  - استفاده از ایده Client/Server
  - نداشتن مشکل سیم‌بندی[2] (یکی از معایب شی‌گرایی سطح پائین بودن آن است که سبب می‌شود که فرد درگیر با ارسال پیام شود و اینکه هر پیام به کدام کلاس ارسال می‌شود، این مسئله معروف به سیم‌بندی است)

[1] Essence
[2] Wire Syndrome

- مقوله‌بندی[1]

❖ رضایت کاربران

❖ تحلیل بوسیله خبرگان

اما کارت‌های CRC در اجرا دارای مشکلاتی نیز هستند

- مشکل برقراری ارتباط با کاربران
- کارت‌های CRC تنها بخشی از نیازمندی‌های یک سیستم شی‌گرا را تشکیل می‌دهند

۴-۶- طبقه‌بندی[2]

در یک سیستم نرم‌افزاری انواع گوناگونی از اشیاء معنی دار وجود دارد. این اشیاء در سطوح متفاوتی از انتزاع[3] (بستگی به میزان پرداختن به جزئیات) قرار دارند. مثلاً در یک سطح می‌توان کامپیوتر را یک شی فرض نمود ولی در یک سطح پایینتر خود کامپیوتر از تعدادی اشیاء تشکیل شده است. مقوله‌بندی و لایه‌بندی کمک می‌کنند تا دسته‌بندی مناسبی برای اشیاء ارائه دهیم. هرکدام از ذینفعان به سیستم از یک زاویه دید معین می‌نگرند و دسته‌بندی خاصی را مد نظر قرار می‌دهند، به همین جهت دسته‌بندی و استفاده از این دیدگاه‌ها می‌تواند کمک موثری در شناخت سیستم نماید.

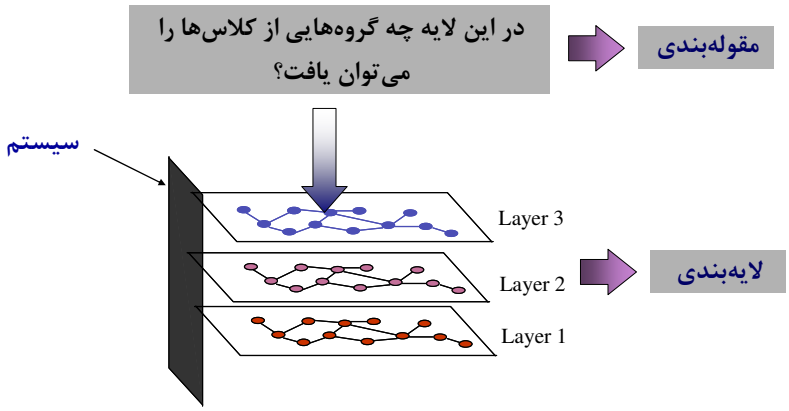
برای دسته‌بندی و توصیف دیدگاه‌های مختلف دو اصل وجود دارد:

۱. از دید چه کسی این سیستم توصیف می‌گردد؟

۲. میزان پرداختن به جزئیات چه قدر است؟

جواب به سوال اول در واقع توصیف لایه‌بندی[4] و جواب به سوال دوم توصیف مقوله‌بندی است. شکل ٤-٥ این تفاوت را نشان می‌دهد.

[1] Stereotyping
[2] Classification
[3] Abstraction
[4] Layering



شکل ٤-٥- تفاوت لایه‌بندی و مقوله‌بندی

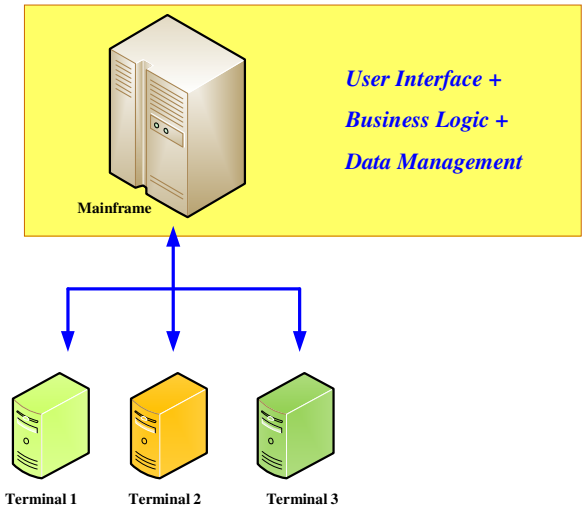
٤-٧- لایه‌بندی

در لایه‌بندی یک سیستم نرم‌افزاری به صورت تعدادی از لایه‌ها تقسیم‌بندی می‌گردد. هر لایه از تعدادی مولفه تشکیل شده که همکاری گروهی این مولفه‌ها بوجود آورنده رفتار لایه می‌باشد. استفاده از لایه‌بندی وابستگی‌ها را کاهش می‌دهد بطوریکه لایه‌های پایین تر از جزئیات و واسط‌های لایه‌های بالاتر اطلاعی ندارند.

همچنین این روش می‌تواند به شناسائی بخش‌های قابل استفاده مجدد و تصمیم‌گیری در مورد مولفه‌های قابل خریداری و مولفه‌های قابل ساخت کمک نماید. می‌توان به معماری‌های متمرکز، معماری Client/Server و معماری 3-Tier بعنوان نمونه‌های از لایه‌بندی در دنیای نرم‌افزار اشاره نمود. یک برنامه کاربردی از نظر منطقی به سه قسمت کلی (واسط کاربر[1]، منطق حرفه[2] یا منطق برنامه و سرویس‌های داده‌ای[3]) تقسیم می‌شود. تفاوت این معماری‌ها در نحوه ارتباط این قسمت‌ها با یکدیگر می‌باشند.

در معماری متمرکز این سه قسمت با یکدیگر شدیداً آمیخته شده که باعث ساختن سیستم‌های غیر انعطاف‌پذیر و غیر قابل نگهداری می‌شود، بنابراین در صورت نیاز به قدرت محاسباتی بیشتر پدیده جزایر برنامه‌های کاربردی[4] به وجود می‌آید. عمده‌ترین مزیت این روش سادگی آن در طراحی و پیاده‌سازی است که در سال‌های اولیه توسعه نرم‌افزار مورد استفاده قرار می‌گرفت. شکل ٤-٦ این روش را نشان می‌دهد.

[1] User Interface
[2] Business Logic
[3] Data Services
[4] Applications Islands



شکل ٤-٦- معماری متمرکز

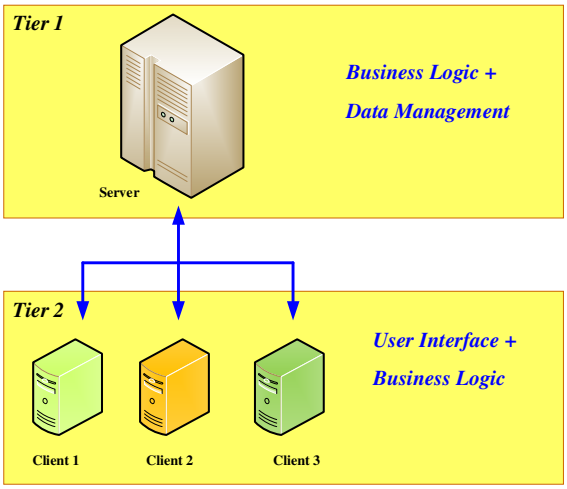
در معماری دوم یعنی Client/Server دو لایه داریم: لایه Client شامل واسط کاربر و منطق حرفه بوده و لایه Server سرویس‌های داده‌ای را بعهده گرفته است. این روش که به‌عنوان نسل بعدی معماری متمرکز ارائه شده است، دارای مزایایی بسیاری نسبت معماری متمرکز است که از جمله می‌توان به موارد ذیل اشاره نمود:

- واسط کاربر قابل استفاده مجدد است
- استفاده از مدل برنامه‌نویسی مبتنی بر رویداد
- وجود محیط‌های برنامه‌سازی یکپارچه و قوی مانند Delphi، Visual Studio .NET، و ... که امکان دسترسی به داده‌های قدیمی و جدید سازمان به صورت یکپارچه را فراهم می‌کنند
- امکان استفاده مجدد جزئی از منطق حرفه در قسمت سرویس‌دهنده

اما این معماری مشکلات مربوط به خود را نیز به‌همراه دارد که از آن جمله می‌توان به موارد ذیل اشاره نمود:

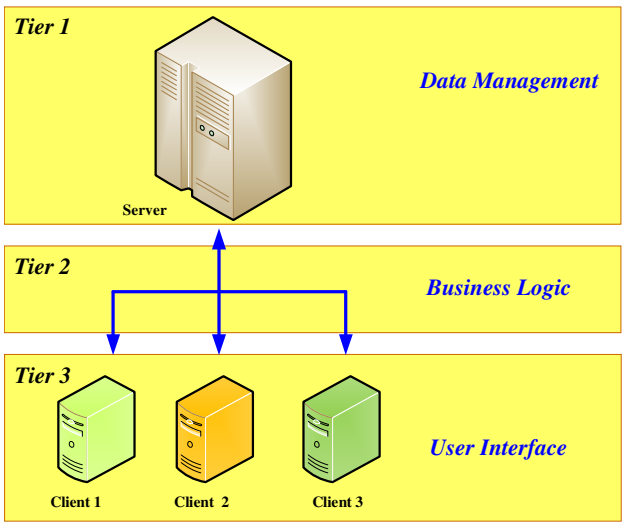
- محل منطق حرفه مشخص نیست
- تنها بخشی از منطق حرفه قابل استفاده مجدد است
- منطق حرفه قابل استفاده مجدد، معمولاً، به صورت روال‌های ذخیره شده که متعلق به یک پایگاه داده معینی است، وجود دارد. به علت این وابستگی منطق حرفه قابلیت استفاده مجدد را در سطح تمام سازمان نخواهد داشت.

شکل ٤-٧ معماری Client/Server را نمایش می‌دهد. مشکلات این معماری سبب استفاده از معماری 3-Tier شده است.



شکل ٤-٧- معماری Client/Server

در معماری سوم یعنی 3-Tier این لایه‌ها از یکدیگر کاملاً جدا هستند که این ویژگی قابلیت استفاده مجدد در این سیستم‌ها را بالا می‌برد. در واقع، منطق حرفه بر روی سرویس‌دهنده مستقلی که به سرویس‌دهنده کاربردی[1] معروف است، قرار می‌گیرد. شکل ٤-٨ این معماری را نمایش می‌دهد.



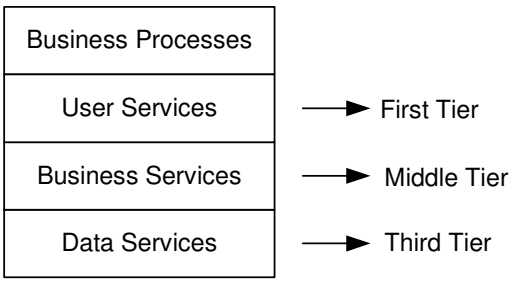
شکل ٤-٨- معماری 3-Tier

[1] Application Server

بعنوان یک مثال اخیر می‌توان از متدولوژی شی‌گرای Select Perspective نام برد. این متدولوژی از معماری 3-Tier بهره برده و بر ایده سرویس مبتنی بوده که عبارتست از مجموعه‌ای منطقاً مرتبط از وظایف که بوسیله یک واسط قابل دسترسی می‌باشند.

در معماری Perspective (شکل ٤-٩) چهار لایه وجود دارد به شرح زیر می‌باشد:

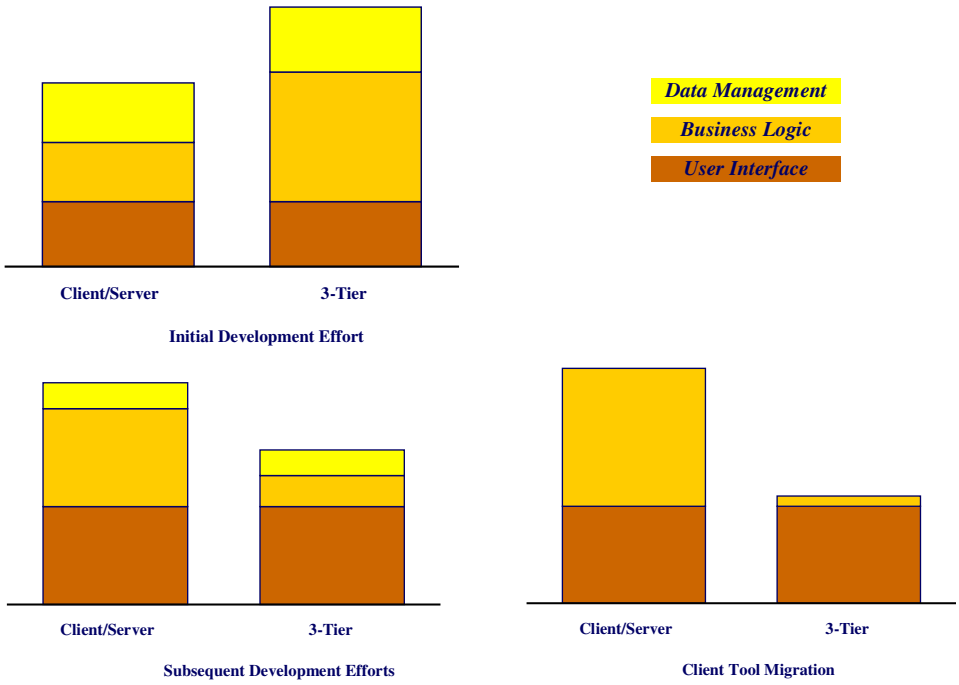
١. فرآیندهای حرفه[1]: فرآیند حرفه عبارتست از مجموعه از فعالیت‌ها که یک یا چند ورودی دریافت کرده و خروجی ارزشمندی برای کاربر ایجاد می‌کند.
٢. سرویس‌های کاربر[2]: وظیفه این لایه فراهم نمودن سرویس‌های ارتباط با کاربر(واسط کاربر)
٣. سرویس‌های حرفه[3]: سرویس‌های عمومی که منطق حرفه سیستم را تشکیل می‌دهند. این سرویس‌ها داده‌ها را از سرویس‌های کاربر و سرویس‌های داده ای را گرفته و مورد پردازش قرارداده و اطلاعات تولید می‌نماید.
٤. سرویس‌های داده‌ای: این سرویس داده‌های مورد نیاز فرآیندهای حرفه متفاوت را فراهم می‌نمایند. سرویس‌های داده‌ای عمل پردازش داده‌ها به صورت مستقل از نحوه ذخیره‌سازی آنها انجام می‌دهند.



شکل ٤-٩- معماری Perspective

در مقایسه‌ای که بین دو معماری 3-Tier و Client/Server براساس هزینه «میزان تلاش در توسعه اولیه»، «میزان تلاش‌های توسعه بعدی» و «ابزار انتقال» انجام شده است و در شکل ٤-١٠ نشان داده شده است، مشخص شده است که معماری 3-Tier تنها در ابتدا هزینه زیادی دارد، اما در ادامه توسعه و با توجه به قابلیت استفاده مجدد آن، هزینه کمتری دارد.

[1] Business Process
[2] User Services
[3] Business Services



شکل ٤-١٠- مقایسه بین معماری Client/Server و 3-Tier

٤-٨- مقوله‌بندی

مقوله‌بندی اشیاء کمک می‌کند که سطوح انتزاعی را شناخته شده و بدین صورت تمام افراد تیم می‌توانند در یک سطح واحدی از انتزاع فعالیت نمایند. بعبارت بهتر، مقوله‌بندی روشی برای میزان کردن سطح انتزاع[1] بین ذینفعان است.

در متدولوژی‌های مختلف مسئله مقوله‌بندی به شکل‌های گوناگون مورد توجه قرار گرفته است. به عنوان نمونه در متدولوژی USDP کلاس‌ها به سه گروه تقسیم می‌شوند:

- ❖ کلاس‌های مرزی[2]
  - مثال: واسط کاربر، پروتکل‌های ارتباطی، واسط چاپگر، حسگرها و ...
- ❖ کلاس‌های کنترلی[3]
- ❖ کلاس‌های موجودیتی[4]
  - کلاس‌های موجودیتی مفاهیم کلیدی در سیستم در حال توسعه را نمایش می‌دهند.

[1] Abstract Level Tuner
[2] Boundary Classes
[3] Control Classes
[4] Entity Classes

در متدولوژی Perspective سه رده کلاس وجود دارند:

۱. اشیاء واسط/کاربری[1]: منظور اشیائی هستند که وظیفه نمایش اطلاعات برای کاربران، دریافت اطلاعات از آنها و بطور کلی برقراری ارتباط کاربر با سیستم را به عهده دارند . این لایه متناظر با لایه اول مدل 3-Tier می‌باشد.

- مثال: در سیستم هتل داری: اپراتوری که با سیستم کار می کند با فرم‌هایی (همان اشیاء کاربری) ارتباط دارد مانند: فرم رزور اتاق، فرم حساب شخص، گزارشی از تعداد افرادی که اتاقها را رزرو کردند و ...

۲. اشیاء حرفه: سرویس‌های مورد احتیاج نیازمندی‌های حرفه را فراهم می‌نمایند. بعبارت ساده‌تر منطق حرفه (منطق برنامه) از همکاری این اشیاء با یکدیگر بوجود می‌آید. این اشیاء با بکاربردن قواعد حرفه و استفاده از سرویس‌های داده‌ای، داده‌ها را به اطلاعات تبدیل می‌نمایند.

- مثال: در سیستم هتل داری، اشیائی مثل رزرو کردن و سفارش مشتری مشاهده می‌شوند که سرویس‌های مورد انتظار از سیستم را فراهم می‌نمایند.
- اگر شی فقط در همان برنامه کاربردی استفاده شود شی محلی[2] و اگر خارج از آن قابل استفاده باشد شی عمومی[3] گویند.

۳. اشیاء داده‌ای: برای فراهم نمودن سرویس‌های داده ای این اشیاء با یکدیگر همکاری کرده و با استفاده از سرویس‌های سیستم پایگاه داده‌ها (ذخیره/ بازیابی/بهنگام سازی)، داده‌های مورد نیاز لایه‌های بالاتر را تامین می کنند. این لایه داده‌ها را به صورت اشیاء پایا[4] ذخیره و بازیابی می‌نماید اگر DBMS مورد استفاده از نوع شی گرا باشد مشکلی پدید نمی‌آید اما اگر از مدل رابطه‌ای پیروی می کند آنگاه این لایه عمل نگاشت اشیاء پایا به مدل رابطه ای را انجام می‌دهند. در واقع هدف از وجود این لایه حفاظت لایه سرویس‌های حرفه از تغییرات در تکنولوژی DBMS ها می‌باشد.

[1] User/Interface Objects
[2] Local Object
[3] Corporate Object
[4] Persistent Objects

- مثال: در سیستم هتل‌داری: اشیائی مانند مدیر داده‌های حساب، تبدیل‌کننده داده‌های حساب و دستیابی به داده‌های حساب مشاهده می‌شوند که سرویس‌های داده‌ای مورد انتظار اشیاء دیگر را فراهم می‌نمایند

یک طبقه‌بندی دیگر که خانم Wirfs-Brock در روش RDD[1] که یک روش وظیفه‌گرا بوده، ارائه نموده بشرح ذیل می‌باشد:

۱. اشیاء کنترل‌کننده[2]: وظیفه اساسی این اشیاء فراخوانی و کنترل اشیاء دیگر.
۲. اشیاء هماهنگ‌کننده[3]: باعث آغاز یک فعالیت شده و یا باعث هماهنگی بین اشیاء Client و اشیاء Server می‌شوند.
۳. اشیاء واسط[4]: این اشیاء ارتباط بین سیستم و محیط خارج از خود را برقرار می‌نمایند.
٤. اشیاء فراهم کننده سرویس[5]: اشیائی هستند که سرویس‌های بخصوصی فراهم می‌نمایند مانند Buffer.
۵. اشیاء نگهدارنده اطلاعات[6]: شبیه اشیاء داده ای در تقسم قبلی می باشند.
٦. اشیاء ساختار[7]: برای نگهداری ارتباط بین اشیاء استفاده می‌شود.

٤-۹- کارت‌های CRC و مقوله بندی

می‌توان از کارتهای CRC برای تشخیص مقوله بندی اشیاء استفاده نمود.

به‌عنوان مثال در یک سیستم بانکی مشتری می خواهد مبلغی از حساب خود برداشت نماید. می‌توان این مبلغ را شی تصور نمود. در واقع عمل برداشت مبلغ تراکنش[8] حساب می‌شود. در کارت‌های CRC می‌توان از مفهوم وراثت بهره برد. برای مثال یک تراکنش مالی[9] بعنوان Superclass و یک Withdraw Transaction بعنوان Subclass تعریف کرده که مسئولیت‌های تراکنش مالی به ارث برده و علاوه بر آن مسئولیت های دیگری اضافه می‌نماید. در شکل ٤-۱۱ کارت CRC این مثال نمایش شده است.

[1] Respnsibility-Driven Design
[2] Controller Objects
[3] Coordinators Objects
[4] Interface Objects
[5] Services Provider Objects
[6] Information Holder Objects
[7] Structure Objects
[8] Transaction
[9] Financial Transaction

Withdrawal Transaction	
Supercalss:Financial Transaction	
Responsibilities	Collaborators
Knows account Knows amount Perfoms Withdraw Logs Transaction Initiates Dispensing cash	Cash Dispensy

شکل ٤-١١- کارت CRC

در آغاز تراکنش Amount را از Account کم می‌کند سپس در فایل Log می‌نویسد که چه کسی در چه زمانی چه مبلغی از چه حسابی برداشته است و بالاخره به دستگاه پیغام (Initiates Dispensing Cash) می‌دهد که پول را تحویل دهید.

در پشت کارت CRC اطلاعات طبقه‌بندی را یاداشت می نماییم (شکل ٤-١٢)

Withdrawal Transaction	
Purpose	Withdraws cash from an account and dispense it
Stereotypes	Service Provider ,Coordinator , Business Object

شکل ٤-١٢- پشت کارت CRC، Withdraw Transaction که در آن طبقه‌بندی کلاس دیده می‌شود

این اطلاعات شامل هدف (وظیفه اصلی) و طبقه‌بندی شی می‌باشد.

۵- فرآیند توسعه نرم‌افزار در متدولوژی USDP

۵-۱- فرآیند توسعه نرم‌افزار

مجموعه‌ای از فعالیت‌های نیمه‌مرتبی که برای توسعه نرم‌افزار بکار گرفته می‌شوند را می‌توان تعریف ساده‌ای از فرآیند توسعه نرم‌افزار دانست. یک فرآیند توسعه نرم‌افزار چهار نقش اساسی دارد:

- مشخص نمودن ترتیب فعالیت‌هایی که باید صورت گیرد تا نیازمندی‌های کاربران به یک محصول واقعی تبدیل شوند
- بیان اینکه چه فرآورده‌هایی باید تولید شوند و در چه زمانی
- تعیین روش اداره وظایف توسعه‌دهندگان منفرد و تیمی، نقش‌های مورد نیاز در پروژه و انتساب این نقش‌ها به اعضای تیم
- فراهم نمودن معیارهایی[1] برای اندازه‌گیری کیفیت محصولات پروژه و روند پیشرفت فعالیت‌های آن

در واقع در فرآیند تولیدی که بخوبی مستند شده باشد نرم‌افزارهای مورد نیاز به صورت منظم و قابل پیش‌بینی قابل تولیدند، اما در فرآیندی که بخوبی مستند نشده باشد، موفقیت بستگی به تلاش طاقت فرسای اعضای تیم دارد.

۵-۲- مشخصات فرآیندهای توسعه موفق

یکی از مشخصات بارز یک فرآیند توسعه خوب استفاده از تجربیات موفق[2] بدست آمده از اجرای پروژه‌های نرم‌افزاری موفق است که این تجربیات موفق عبارتند از:

- استفاده از روش تکرار و توسعه تدریجی
- مدیریت نیازمندی‌ها
- استفاده از معماری مبتنی بر مولفه‌ها
- راهبری بر مبنای موارد کاربری

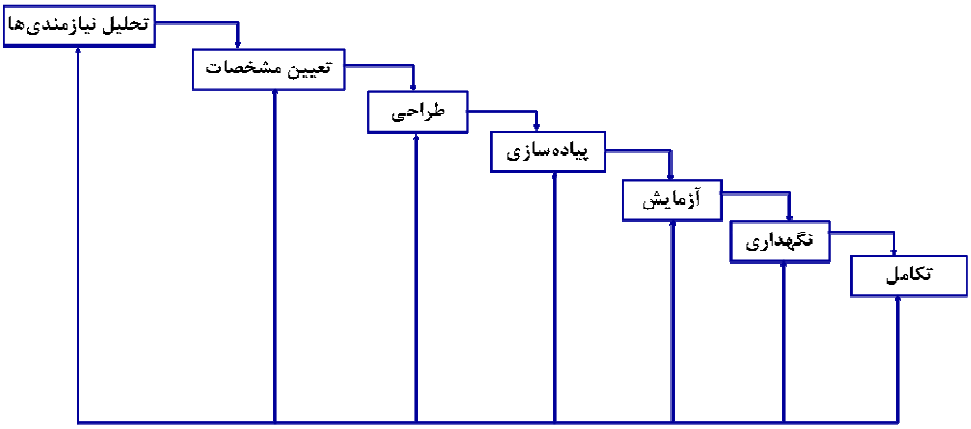
[1] Software Metrics
[2] Best Practice

- مدلسازی تصویری نرم‌افزار
- کنترل تغییرات
- بررسی کیفیت نرم‌افزار

۵-۲-۱- تکرار و توسعه تدریجی

فرآیند تولید سنتی(آبشاری[1]) شامل مراحل تحلیل نیازمندی‌ها، تعیین مشخصات، طراحی، پیاده‌سازی، آزمایش، نگهداری و تکامل[2] است(شکل ۵-۱). ویژگی اساسی این روش، طبیعت ترتیبی آن بوده زیرا برای اینکه مرحله بعدی شروع شود باید مرحله قبلی کاملا خاتمه یافته باشد. در واقع ریشه مشکل اصلی این روش همان شیوه نگرش است زیرا با توجه به این مطلب که معمولا در مرحله آزمایش خطاها کشف می‌شوند؛ برای کشف یک خطا در پیاده‌سازی که ریشه آن اشتباهی در طراحی یا گردآوری نیازمندی‌ها بوده، باید همه مستندات طراحی بررسی شوند (که حجم آن در پروژه‌های بزرگ بسیار زیاد است) لذا این کار هزینه زیادی می طلبد. حال اگر به این هزینه، هزینه اصلاح خود خطا هم اضافه شود به راز شکست بسیاری از پروژه‌هایی که در آن خطاهای طراحی دیر کشف می‌شود، پی می‌بریم!

بطور خلاصه می‌توان گفت که در این روش با گذشت زمان هزینه ریسک بالاتر می‌رود.



شکل ۵-۱- مراحل توسعه در روش آبشاری [٤]

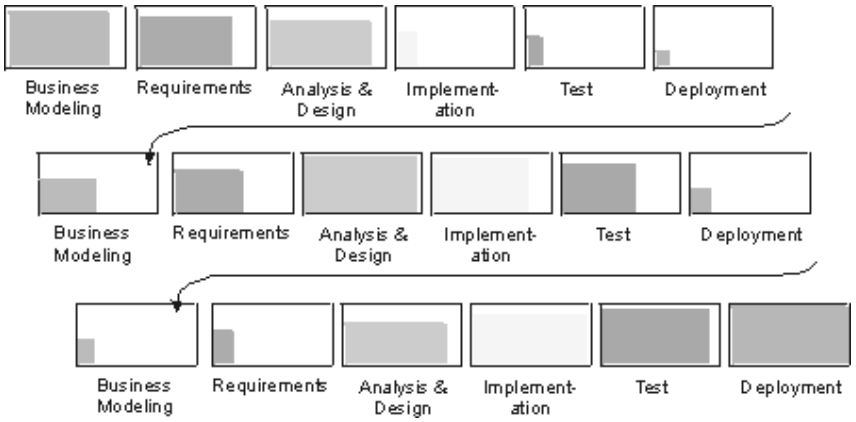
در واقع روش آبشاری برای پروژه‌های کوچک (کوتاه مدت) یا پروژه‌هایی که بیشتر جزئیات طراحی آن به خوبی شناخته شده مناسب است اما برای پروژه‌های بزرگ یا پروژه‌های جدید و غیر سنتی (مانند

[1] Waterfall Approach
[2] Integration

سیستم‌های هوشمند و سیستم‌های اطلاعاتی بزرگ) این روش جوابگو نیست. علاوه بر این ریسک این روش نیز بسیار بالاست.

شکل ۵-۲- هزینه ریسک در روش آبشاری

پس بیایید یک پروژه بزرگ را به چند زیر پروژه کوچک و متوالی تقسیم نموده واز روش آبشاری که در آن مدیریت ریسک نیز منظور شده است برای تولید هر کدام از این زیر پروژه‌ها استفاده نماییم. بدین صورت ما مقدار کمی از نیازمندی‌ها را مشخص نموده سپس در مورد همین مقدار، مراحل مدیریت ریسک، تحلیل، طراحی، پیاده‌سازی و آزمایش را انجام می‌دهیم. در تکرار بعدی یک مقدار دیگر از نیازمندی‌ها تشخیص نموده و این عملیات را در مورد آنها تکرار می نماییم. این همان روش تکرار و توسعه تدریجی است(شکل ۵-۳). در واقع روش توسعه تدریجی با یک بعدی بودن فرآیند توسعه نرم‌افزار(مثلاً روش آبشاری) سازگاری نداشته و یک روش دو بعدی می‌طلبد.



شکل ۵-۳- تکرار و توسعه تدریجی

چنانکه می‌بینیم در این روش ریسک‌ها به صورت زود هنگام تشخیص داده می‌شوند و هرگاه که ممکن است تلاش می‌شود تا با آنها مقابله شود.

ویژگی‌های روش تکرار و توسعه تدریجی [٤] را می‌توان به صورت ذیل عنوان نمود:

١. تشخیص زود هنگام خطاهای که در درک مسأله، تحلیل یا طراحی رخ می‌دهند.

٢. تشخیص زود هنگام ناسازگاری های موجود بین تحلیل نیازمندی‌ها، طراحی و پیاده‌سازی.

٣. با توجه به شیوه عمل تکراری کاربر می‌تواند همیشه بر روند پیشرفت پروژه نظارت داشته باشد. همچنین می‌تواند نظر خود را ارائه نماید که در تشخیص بهتر نیازمندی‌های مسأله کمک می‌نماید.

٤. بوسیله این روش تیم توسعه سیستم می‌تواند روی قسمت‌های مهمتر پروژه متمرکز شود و از پرداختن به قسمت‌های کم اهمیت تر پرهیز نماید.

٥. آزمایش تکراری و مستمر امکان تشخیص بهتر روند پیشرفت پروژه را به ما می‌دهد.

٦. بارکاری[1] تیم ها-بخصوص آزمایش کنندگان-روی چرخه تولید پروژه به صورت متوازن توزیع می‌شود.

در فرآیندهای توسعه نرم‌افزار بر مبنای روش شی‌گرا غیر از بکاربردن مفهوم توسعه تدریجی از روش تحلیل موارد کاربری و معماری نرم‌افزار استفاده می‌گردد.

٥-٢-٢- مدیریت نیازمندی‌ها

نیازمندی عبارتست از شرطی یا قابلیتی که سیستم باید دارای آن باشد. در طول توسعه سیستم، اغلب نیازمندی‌های در حال تغییر می‌باشند و مدیریت نیازمندی‌ها با کنترل و سازماندهی تغییرات مواجه می‌شود. بطور کلی می‌توان وظایف مدیریت تغییرات را به صورت ذیل دانست:

١) دریافت، سازماندهی و مستندسازی عملکرد مطلوب سیستم و محدودیت‌های موجود

٢) اعمال تغییرات مطلوب روی نیازمندی‌های جمع آوری شده

٣) ردیابی و مستندکردن اثرات بوجود آمده و تصمیم‌های اتخاذ شده

[1] Workload

برای مدیریت نیازمندی‌ها به یک روش منظم و سیستماتیک نیاز است تا بتوان نیازمندی‌ها را اولویت‌بندی، فیلتربندی یا ردیابی نمود. با استفاده از مدیریت نیازمندی‌ها امکان تشخیص واقعی و منصفانه عملکرد و کارایی سیستم بوجود می‌آیند و ناسازگاری‌ها به آسانی قابل کشفند.

۵-۲-۳- استفاده از معماری مبتنی بر مؤلفه‌ها

معماری سیستم عبارتست از تعیین ساختار کلی سیستم و روش‌هایی که این ساختار را قادر به تامین کلیه ویژگیهای کلیدی سیستم[1] می‌سازد. معماری سیستم شامل تصمیم‌گیری‌هایی در سطح کلان در موارد ذیل است:

- نحوه سازماندهی سیستم نرم‌افزاری
- انتخاب عناصر ساختاری و واسط‌های آنها و مشخص نمودن رفتار این عناصر
- سازماندهی این عناصر در گروه‌های بزرگتر (زیرسیستم‌ها)
- سبک معماری مورد استفاده

علاوه بر ساختار و رفتار سیستم، معماری با مواردی از قبیل کارایی، انعطاف‌پذیری، استفاده مجدد و محدودیت‌های تکنولوژی و اقتصادی نیز سروکار دارد. یکی از شیوه‌های مهم معماری نرم‌افزار، توسعه مبتنی بر مؤلفه‌ها[2] است زیرا این روش امکان استفاده مجدد از آنها را به ما می‌دهد.

از عمده‌ترین ویژگی‌های استفاده از معماری می‌توان به موارد ذیل اشاره نمود

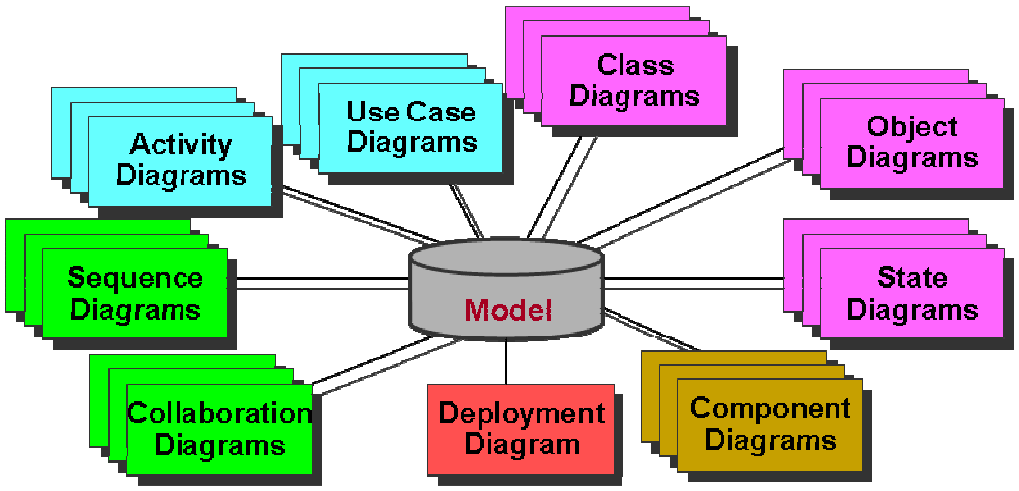
- به داشتن یک معماری کشسان کمک می‌کند.
- قابلیت استفاده مجدد از مؤلفه‌ها و تکنولوژی‌های موجود افزایش می‌یابد.
- مؤلفه یک پایه خوب و مناسب برای مدیریت پیکربندی است.

۵-۲-۴- مدلسازی تصویری نرم‌افزار

مدل عبارت از یک توصیف ساده شده، با توجه به یک نگرش معین، از سیستم است. انواع مدل‌ها می‌توانند در توصیف نرم‌افزار بکار گرفته شوند. شکل ۵-٤ برخی از مدل‌های مورد استفاده در توصیف نرم‌افزار را نشان می‌دهد. نکته با اهمیت در مدل‌سازی یکپارچگی و استفاده از مخزن مشترک برای ذخیره

[1] Cross-cutting Concerns
[2] Component-based Development

عناصر مدل‌های مختلف است. ارتباط بین عناصر مدل‌های مختلف می‌تواند کمک شایانی در درک نرم‌افزار به تیم توسعه و سایر ذینفعان نماید.



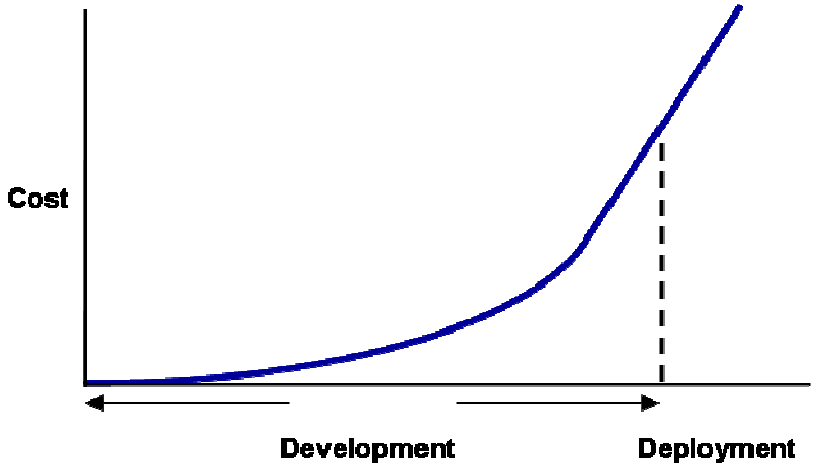
شکل ٥-٤- برخی از مدل‌های مورد استفاده برای توصیف نرم‌افزار

مدل‌سازی دارای ویژگی‌هایی است که از آن جمله می‌توان به موارد ذیل اشاره نمود:

- امکان توصیف سیستم با میزان دلخواهی از جزئیات فراهم می‌آید.
- بوسیله مدل‌ها می‌توان طراحی سیستم را به صورت روشن و صریح بیان نمود.
- امکان تشخیص معماری‌های غیر قابل انعطاف و واحدبندی نشده فراهم می‌آید.
- نرم‌افزار خوب حاصل مدل‌های با کیفیت بالا است.

٥-٢-٥- بررسی کیفیت نرم‌افزار

برای رسیدن به یک نرم‌افزار با کیفیت عالی باید فرآیند تشخیص کیفیت به صورت مستمر و پیوسته از همان مراحل اولیه توسعه نرم‌افزار به اجرا درآید. در واقع، هزینه اصلاح خطاهای نرم‌افزار به صورت نمایی رشد می‌کند و این بدان معنی است که شناسایی زودتر خطاها می‌تواند خسارات کمتری را متوجه تیم نماید. پس از استقرار سیستم، هزینه خطاها بمراتب بیشتر از هزینه خطا قبل از استقرار است. شکل ٥-٥ هزینه خطاها در مراحل مختلف نرم‌افزار را نشان می‌دهد.



شکل ۵-۵- هزینه کشف خطا در مراحل مختلف توسعه نرم‌افزار

بررسی کیفیت نرم‌افزار ویژگی‌های دارد که از آن جمله می‌توان به موارد ذیل اشاره نمود:

- فرآیند تشخیص روند پیشرفت پروژه مبتنی بر واقعیت‌ها، نه بر حدس‌ها و محاسبات کاغذی خواهد بود
- این فرآیند ناسازگاری‌های موجود بین نیازمندی‌ها، طراحی و پیاده‌سازی را آشکار می‌سازد
- امکان کشف زود هنگام خطاها را به ما می‌دهد و بدین صورت هزینه اصلاح آنها بشدت کاهش می‌یابد

۵-۲-٦- مدیریت پیکربندی

مدیریت پیکربندی، هنر تشخیص، سازماندهی و کنترل تغییراتی است که برای نرم‌افزار در مدت کارکرد خود (از ابتدای تولید تا خارج شدن از عمل) رخ می‌دهند. در واقع، یکی از مشکلات اساسی توسعه نرم‌افزار مدیریت پیکربندی است. این مشکل بویژه در پروژه‌های بزرگ که در آن تیم‌های متعددی با یکدیگر بر روی تکرارها، نشرها، محصولات و سکوهای متفاوت کار می‌کنند، قابل مشاهده است.

از عمده‌ترین ویژگی‌های مدیریت پیکربندی می‌توان به موارد ذیل اشاره نمود:

- یک روش سیستماتیک و قابل‌تکرار برای کنترل تغییرات نرم‌افزار می‌باشد
- کنترل انتشار تغییرات را می‌توان مدیریت نمود
- کاهش تداخل بین کار توسعه‌دهندگان که به صورت موازی با هم کار می‌کنند

- نرخ تغییر معیار مناسبی برای تشخیص وضعیت فعلی پروژه است

۵-۳- متدولوژی‌های توسعه نرم‌افزار

بطور کلی می‌توان متدولوژی‌های توسعه نرم‌افزار را به دو دسته کلی تقسیم نمود:

- **متدولوژی‌های سنگین وزن**[1]

این نوع متدولوژی‌ها معمولاً مستندات، محصولات و فرآورده‌های بیشتری تولید می‌نمایند

- **متدولوژی‌های سبک وزن**[2]

تاکید بر تولید محصول نهایی و نه فرآورده‌ها و مستندات بسیار است

این دو دسته متدولوژی تفاوت‌های بسیاری در نوع اجرا، میزان مستنداتی که تولید می‌کنند و ... دارند که آنها را برای پروژه‌های خاصی مناسب می‌کند. در ادامه به بررسی متدولوژی‌های سبک وزن (سریع‌الانتقال) و سنگین وزن از نقطه نظرات متفاوت خواهیم پرداخت.

- روش اجرا
  - روش‌های سریع‌الانتقال بصورت سازگار[3] عمل می‌کنند و با شرایط منطبق می‌شوند
  - روش‌های سنگین وزن بصورت پیشگو[4] عمل می‌کنند و سعی می‌کنند در آغاز همه چیز را پیش‌بینی کنند
- معیار موفقیت
  - معیار موفقیت در روش‌های سریع‌الانتقال دستیابی به ارزش کاری[5] است
  - در روش‌های سنگین وزن معیار موفقیت پیش رفتن در راستای طرح اولیه است
- اندازه پروژه
  - اندازه پروژه در روش‌های سریع‌الانتقال کوچک است
  - اندازه پروژه در روش‌های سنگین وزن می‌تواند بسیار بزرگ باشد
- سبک مدیریت
  - مدیریت در روش‌های سریع‌الانتقال بصورت غیرمتمرکز و آزاد است

[1] Heavyweight
[2] Lightweight
[3] Adaptive
[4] Predictive
[5] Business Value

- در روش‌های سنگین وزن مدیریت بصورت مطلق و استبدادی است

- مستندسازی
  - مستندسازی در روش‌های سریع‌الانتقال بصورت بسیار محدود انجام می‌شود
  - در روش‌های سنگین وزن مستندسازی بصورت کامل و جامع انجام می‌شود
- تعداد چرخه‌ها[1]
  - تعداد چرخه‌ها در روش‌های سریع‌الانتقال بسیار زیاد است اما زمان آنها کوتاه است
  - در روش‌های سنگین وزن تعداد چرخه‌ها کم است ولی زمان آنها بسیار زیاد است
- اندازه تیم
  - در روش‌های سریع‌الانتقال اندازه تیم کوچک است (بین ۲۰ تا ۳۰ نفر)
  - در روش‌های سنگین وزن اندازه تیم توسعه بزرگ است
- برگشت سرمایه
  - در روش‌های سریع‌الانتقال سرمایه خیلی زود در طول پروژه بر می‌گردد
  - در روش‌های سنگین وزن برای برگشت سرمایه باید تا انتهای پروژه صبر کرد

۵-٤- معرفی Unified Software Development Process

USDP فرآیند تولید مهندسی نرم‌افزار است که یک روش سیستماتیک و منظم برای ترتیب انجام فعالیت‌ها در یک پروژه نرم‌افزاری را پیشنهاد می‌نماید [۲]. این فرآیند دارای یک چارچوب فرآیند[2] است که همه عناصر لازم برای تولید محصولات نرم‌افزاری از سیستم‌های سنتی و معمولی گرفته تا سیستم‌های هوشمند و سیستم‌های اطلاعاتی بزرگ را دربردارد. هدف این فرآیند، توسعه نرم‌افزارهایی با کیفیت عالی که علاوه بر برآوردن نیازهای کاربران خود، در زمان و با هزینه پیش بینی شده تولید شوند.

USDP از مدل شی‌گرایی حمایت نموده و از روش‌های مدرن توسعه نرم‌افزار مانند:

- استفاده از روش تکرار و توسعه تدریجی
- معماری مبتنی بر مولفه‌ها
- راهبری بر مبنای موارد کاربری

[1] Cycles
[2] Process Framework

- مدلسازی بصری نرم‌افزار
- کنترل تغییرات
- بررسی کیفیت نرم‌افزار

پشتیبانی می‌نماید.

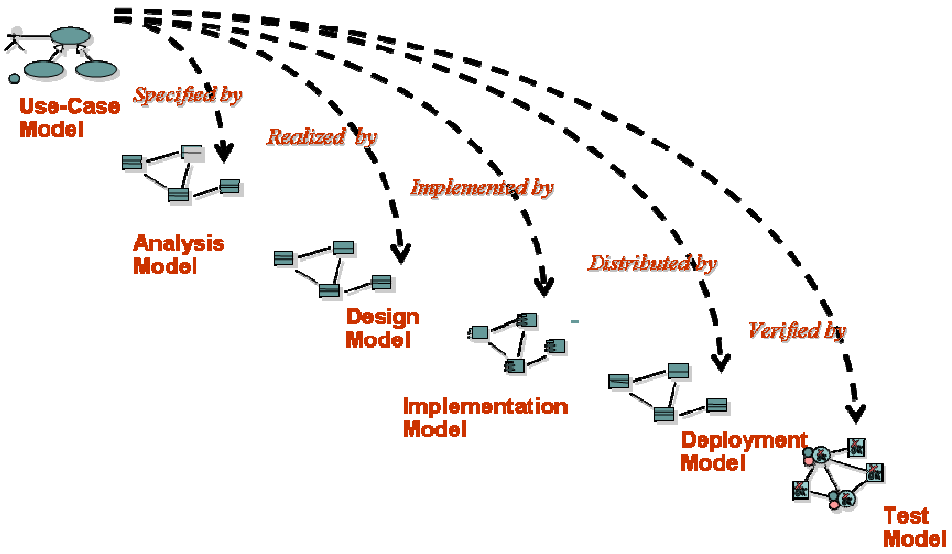
۵-۴-۱- محورهای USDP

USDP دارای سه محور اصلی است که آن را از سایر متدولوژی‌های توسعه نرم‌افزار مجزا می‌کند. این سه محور عبارتند از:

- **راهبری بر مبنای موارد کاربری**[1]

مورد کاربری عبارت از دنباله‌ای از عملیات است که یک سیستم انجام می‌دهد تا یک نتیجه قابل مشاهده و ارزشمند برای کاربر فراهم نماید. در نگرش سنتی که عملکردهای سیستم مورد توجه قرار می‌گیرند بدنبال شناسایی عملکردهای سیستم هستیم. در نگرش مبتنی بر موارد کاربری بدنبال این هستیم که سیستم به ازای هر کاربر، باید چه عملکردهایی از خود نشان دهد؟

در این رویکرد، موارد کاربری به‌عنوان پیشران برای مدل‌های دیگر توسعه مورد استفاده قرار می‌گیرد (شکل ۵-۶)



شکل ۵-۶- راهبری بر مبنای موارد کاربری

[1] Use-Case Driven

- **محوریت قرار دادن معماری**[1]

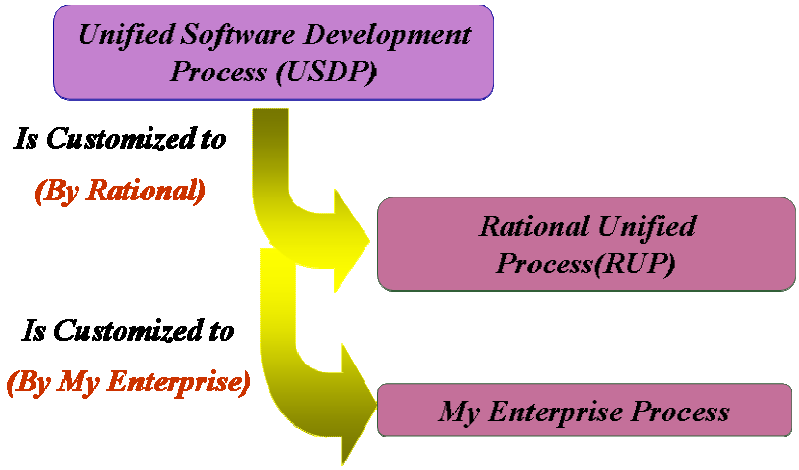
محوریت قرار دادن معماری در USDP به‌عنوان یکی از بهترین تجربیات توسعه نرم‌افزار، آن را از سایر متدولوژی‌های قبلی مجزا نموده است.

- **استفاده از روش تکرار و توسعه تدریجی**[2]

اساس فرآیند USDP توسعه تکراری و تدریجی است که این موضوع سبب دو بعدی بودن این فرآیند توسعه شده است.

۵-٤-۲- توسعه USDP

USDP به عنوان ابزاری برای توسعه فرآیندهای توسعه جدید همچون RUP مورد استفاده قرار گرفته است. در واقع شرکت Rational با سفارشی نمودن فرآیند USDP و افزودن ابزارها، راهنماها و ... محصولی به نام RUP را ارائه نموده است. اصولاً شرکت‌های نرم‌افزار متناسب با شرایط خود، فرآیند USDP را سفارشی می‌کنند تا بتوانند از آن در توسعه نرم‌افزارهای خود استفاده نمایند.



شکل ۵-۷- استفاده از USDP برای توسعه فرآیندهای دیگر

در فصل بعد به بررسی فرآیند RUP خواهیم پرداخت.

[1] Architecture Centric
[2] Iterative and Incremental Development

٦- بررسی فرآیند توسعه RUP

در حال حاضر، RUP به عنوان یکی از فرآیندهای توسعه بسیار مورد توجه قرار گرفته است. این فرآیند توسعه که اساس آن USDP است هم می‌تواند به‌عنوان محصول و هم به‌عنوان فرآیند مورد توجه قرار می‌گیرد.

- **RUP به عنوان فرآیند**

فرآیند RUP یک روش نظام‌مند برای تخصیص کارها و مسئولیت‌ها در یک تیم توسعه نرم‌افزار می‌باشد و هدف آن توسعه نرم‌افزار با کیفیت بالاست که نیازهای کاربران نهایی را توسط یک برنامه و با بودجه قابل پیش‌بینی تأمین می‌نماید. RUP قابلیت شکل‌دهی و سفارشی‌شدن دارد، این موضوع بدین سبب است که هیچ فرآیند واحدی برای همه نرم‌افزارها مناسب نمی‌باشد. این فرآیند بهره‌وری تیم را با فراهم نمودن دسترسی تمام افراد تیم به پایگاه دانش سهل‌الوصول، راهنماها، الگوها و ابزارهای کمکی برای همه فعالیت‌های توسعه، افزایش می‌دهد. با تأمین دسترسی همه اعضای تیم به یک پایگاه دانش، افراد هر قسمت، از یک زبان فرآیند و دید مشترک برای توسعه نرم‌افزار برخوردار خواهند شد. (شکل ۶-۱)

شکل ٦-١- RUP به عنوان فرآیند

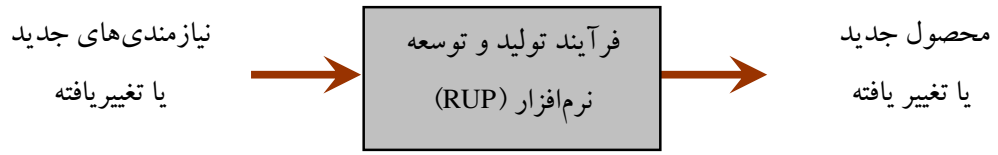
- **RUP به عنوان محصول**

چون مستندسازی RUP بوسیله اسناد کاغذی میسر نیست از اسناد HTML برای مستندسازی آن استفاده شده است. به عنوان محصول، RUP شامل مجموعه‌ای کامل از مستندات و ابزارها است که استفاده‌کنندگان از آن را هدایت می‌کند. از جمله مهمترین مستندات و ابزارهای موجود در RUP می‌توان به موارد ذیل اشاره نمود:

- راهنماهای لازم برای بکارگیری RUP به عنوان یک فرآیند تولید در مراحل مختلف توسعه نرم‌افزار
- راهنماهای ابزار[1]
- الگوها[2]
- یک Development Kit که چگونگی تغییر، گسترش و تنظیم ویژه RUP را نشان می‌دهد

۶-۱- ابعاد فرآیند RUP

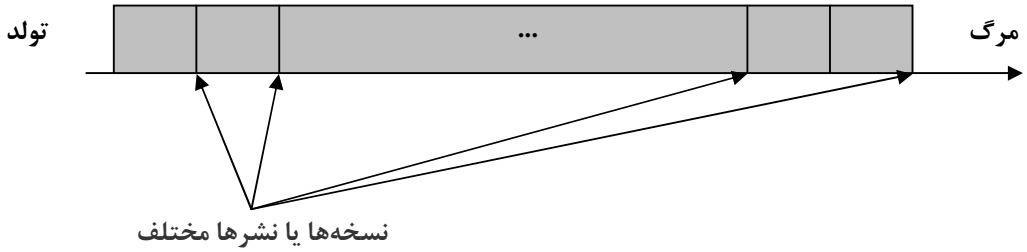
بطور ساده، فرآیند RUP را می‌توان از سه زاویه مورد مطالعه قرار داد. در نگاه اول، فرآیند RUP مجموعه‌ای از نیازمندی‌های جدید و یا تغییریافته را دریافت داشته و محصول جدید یا تغییریافته را تحویل می‌دهد (شکل ٦-٢). در این نگاه، فرآیند توسعه نرم‌افزار به‌عنوان یک جعبه سیاه مطرح است که ورودی‌هایی را دریافت و خروجی‌های مناسب را نتیجه می‌دهد.



شکل ٦-٢- فرآیند RUP به عنوان جعبه سیاه

در نگاه دیگر می‌توان از زمان شروع توسعه نرم‌افزار تا پایان عمر نرم‌افزار را چرخه حیات نرم‌افزار نامید. در این مدت مجموعه‌ای از نشرها[3] و نسخه‌ها[1] برای نرم‌افزار تولید می‌شود که آن را کامل می‌کند.

[1] Tools Mentors
[2] Templates
[3] Releases



شکل ٦-٣- چرخه حیات نرم‌افزار

در واقع، نرم‌افزار یک محصول فیزیکی که یک بار تولید و مستهلک می‌شود نیست، بلکه مانند یک موجود زنده‌ای که در سازمان تولد و رشد کرده و در دوران حیات خود باید با تغییر نیازهای سازمان و اهداف و ماموریت‌های آن خود را تطبیق دهد. چرخه حیات نرم‌افزار را می‌توان با زندگی انسان تشبیه نمود:

- تولد و مرگ دارد
- دارای دوره‌های گوناگون: کودکی، نوجوانی، جوانی، و پیری
- هر دوره دارای ویژگیها و نیازهای مختص خود است
- در میان دوره‌ها نیازهای مشترکی زیادی وجود دارد
- ولی شیوه برآوردن آنها از یکدیگر متفاوت است
- همچنین میزان تاکید روی هرکدام متفاوت است

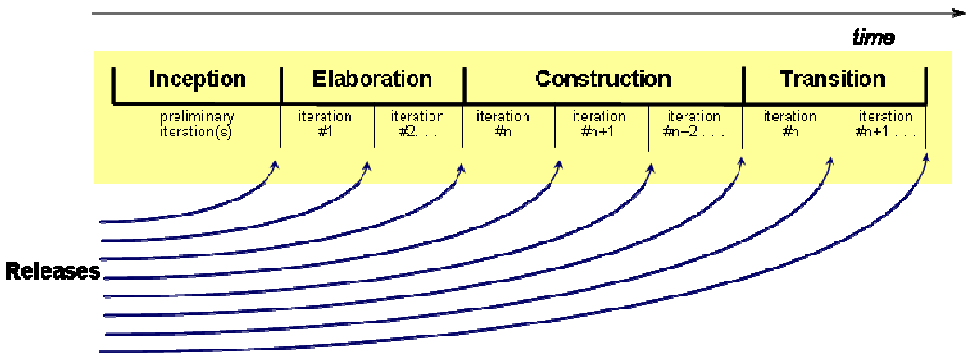
در مورد نرم‌افزار نیز چنین مواردی صدق می‌کند

- دوران حیات یک سیستم نرم‌افزاری شامل دوره‌های گوناگونی است
- میان آنها فعالیت‌های مشترکی وجود دارد
- در هر دوره به اندازه معینی روی هر کدام از این فعالیت‌ها تاکید می‌شود
- هرکدام از این دوره‌ها می‌توانند فعالیت‌های ویژه خود را داشته باشند

در RUP دوران حیات یک نرم‌افزار به چهار مرحله آغازین، تشریح، ساخت و انتقال تقسیم می‌شود که در بخش‌های بعدی آنها را مورد مطالعه قرار می‌دهیم (شکل ٦-٤). سه مرحله اول شامل فعالیت‌های

[1] Versions

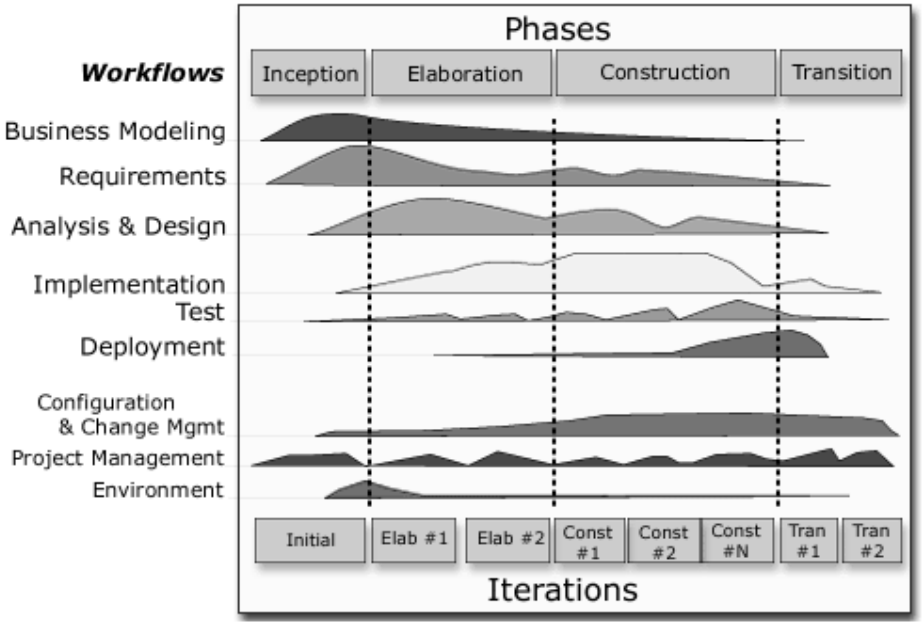
تولید یا توسعه نرم‌افزار بوده و مرحله چهارم دربردارنده انتقال نرم‌افزار بـه محیط واقعی و نگهـداری آن است.



شکل ٦-٤- چرخه حیات نرم‌افزار در RUP

در نگاه سوم، RUP برخلاف فرآیندهای توسعه سنتی که یک بعدی هستند، فرآیندی دو بعدی است. همانطوری که گفتیم RUP یک فرآیند تولید دو بعدی است (شکل ٦-٥):

- محور عمودی: این محور گردش کارهای اصلی را نشان می‌دهد.
- محور افقی(زمان): این محور ساختار چرخه توسعه نرم‌افزار در RUP در بستر زمان را نشان می‌دهد.



شکل ٦-٥- دو بعد مختلف فرآیند RUP

بعد اول، جنبه ایستای سیستم را نمایش می‌دهد که شامل فعالیت‌ها، گردش کارها، فرآورده‌ها و افراد است. این فرآیندها متناظر با مراحل فرآیند تولید سنتی هستند که شامل جمع آوری نیازمندی‌ها، تحلیل و طراحی، پیاده‌سازی و آزمایش است. به عبارت دیگر این بعد فرآیندهای خرد[1] که بوسیله آنها روش تکرار و توسعه تدریجی پیاده‌سازی می‌شود را نشان می‌دهد.

بعد دوم، جنبه‌های پویای سیستم را نمایش می‌دهد. این جنبه‌ها به صورت چرخه‌ها[2]، فازها[3]، تکرارها[4] و فرسنگ شمارها[5] بیان می‌شوند. به عبارت دیگر این بعد فرآیندهای کلان[6] را نشان می‌دهد.

٦-٢- ساختار ایستا

RUP نیز همانند سایر فرآیندهای توسعه نرم‌افزار یک فرآیند خشک[7] که گام‌های آن کاملاً مشخص و از پیش‌تعریف شده باشد نیست (اصطلاحاً مانند کتاب آشپزی[8] نیست!) بلکه یک فرآیند تدریجی و تکراری که در آن فضا برای ابداع و خلاقیت وجود دارد. در واقع، از یک فرآیند توسعه نرم‌افزار انتظار می‌رود که معین می‌کند چه کسی[9]، چه چیزی را باید انجام دهد[10]، به چه صورت[11]، و در چه زمانی[12].

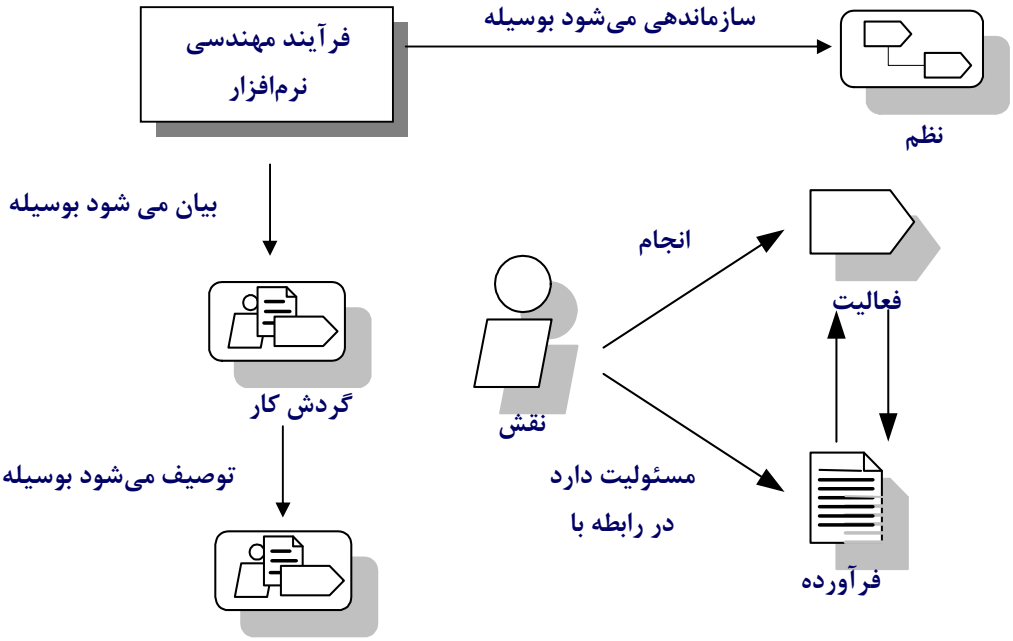
RUP برای این چهار سؤال، چهار عنصر مدل‌سازی زیر را ارائه می‌نماید [۳]:

۱. نقش: چه کسی باید انجام دهد
۲. فعالیت[13]: چه چیزی را باید انجام دهد
۳. فرآورده[14]: به چه صورت انجام دهد
٤. نظم[15]: در چه زمانی انجام دهد

---

[1] Micro Processes
[2] Cycles
[3] Phases
[4] Iterations
[5] Milestones
[6] Macro Processes
[7] Rigid Process
[8] Cookbook
[9] Who
[10] What
[11] How
[12] When
[13] Activity
[14] Artifact
[15] Discipline

شکل ٦-٦ ارتباط بین عناصر اصلی بعد ایستای RUP و فرآیند مهندسی نرم‌افزار را نشان می‌دهد.

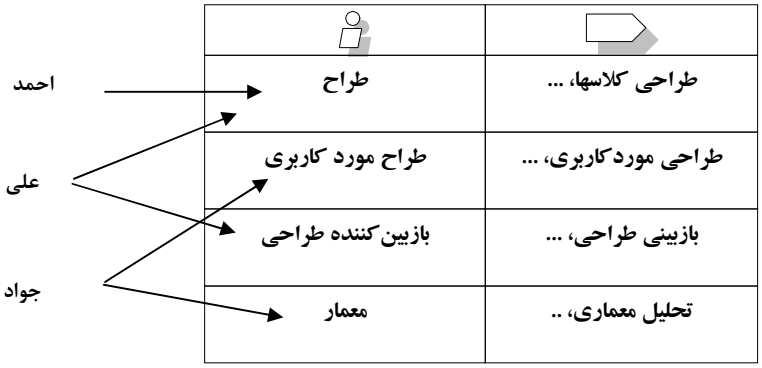


شکل ٦-٦- ارتباط بین مؤلفه‌های اساسی بعد ایستای RUP و فرآیند مهندسی نرم‌افزار

در ادامه به توضیح این عناصر چهارگانه خواهیم پرداخت.

٦-٢-١- نقش

این اصطلاح رفتار و مسئولیت‌هایی که یک نفر (یا یک تیم) در پروژه بعهده دارد را مشخص می‌نماید. این رفتار به صورت فعالیت‌هایی که باید این نفر انجام دهد، بیان می‌شود. همچنین مسئولیت‌ها این نفر در رابطه با تولید/بروز رسانی/استفاده از فرآورده‌ها بیان می‌شوند. هر فرد می‌تواند چند نقش ایفا کند و چند نفر می‌توانند یک نقش را ایفا نمایند (شکل ٦-٧).



شکل ٦-٧- نگاشت افراد به نقش‌ها

تحلیلگر سیستم، طراح، معمار، مشتری، کاربر نهایی و ... نمونه‌های از نقش‌ها هستند.

توجه داشته باشید که برای هر نقش دانستن مجموعه‌ای از مهارت‌ها لازم است که بوسیله کسی که این نقش را بعهده دارد باید تامین شوند. در واقع، هر نقش نیازمندی مجموعه‌ای از مهارت‌ها است که فرد یا افراد گیرندهٔ نقش باید دارا باشند.

٦-٢-٢- فعالیت

کارهای که یک نقش باید انجام دهد به صورت فعالیت بیان می‌شوند. پس یک فعالیت عبارت از واحد انجام کار است. هر فعالیت دارای هدف مشخصی بوده و معمولاً به صورت تولید فرآورده‌های معینی بیان می‌شود. همچنین هر فعالیت به یک نقش انتساب داده می‌شود.

در اصطلاح شی‌گرایی نقش یک شی فعال[1] بوده و فعالیت‌هایی که بوسیله این نقش انجام می‌شوند همان عملیات آن شی خواهند بود.

مثال‌هایی از فعالیت‌ها شامل:

- یافتن موارد کاربری و عامل‌ها بوسیله «تحلیل‌گر سیستم» انجام می‌شود
- بازبینی طراحی که بوسیله «بازبینی کننده طراحی[2]» انجام می‌شود

هر فعالیت از چند گام زیر تشکیل می‌شود:

۱) گام‌های اندیشیدن[3]: در این گام‌ها نقش کار مورد نظر را درک کرده و فرآورده‌های ورودی را آماده می‌کند.

۲) گام‌های اجرا[4]: نقش یک فرآورده را تولید یا بهنگام سازی می‌نماید.

۳) گام‌های بازبینی[5]: نقش نتایج را ارزیابی می‌نماید.

برای درک بهتر این گام‌ها، فعالیت «پیدا نمودن موارد کاربری و عوامل مربوطه» را برای نمونه مورد بررسی قرار می‌دهیم. این فعالیت از گام‌های زیر تشکیل شده است:

---

[1] Active Object
[2] Design Reviewer
[3] Thinking Steps
[4] Performing Ste
[5] Reviewing Steps

۱) عوامل را پیدا کنید.

۲) موارد کاربری را پیدا کنید.

۳) نحوه ارتباط عوامل با موارد کاربری را توضیح و توصیف نمایید.

٤) موارد کاربری و عوامل را با هم بسته‌بندی نمایید.

٥) مدل موارد کاربری را به صورت نمودار موارد کاربری را نمایش دهید.

٦) مدل موارد کاربری را مستند سازید.

۷) نتایج را ارزیابی نمایید.

گام‌های ۳-۱ مربوط به اندیشیدن بوده، گام‌های ٦-٤ مربوط به اجرا بوده و گام ۷ مربوط به بازبینی است.

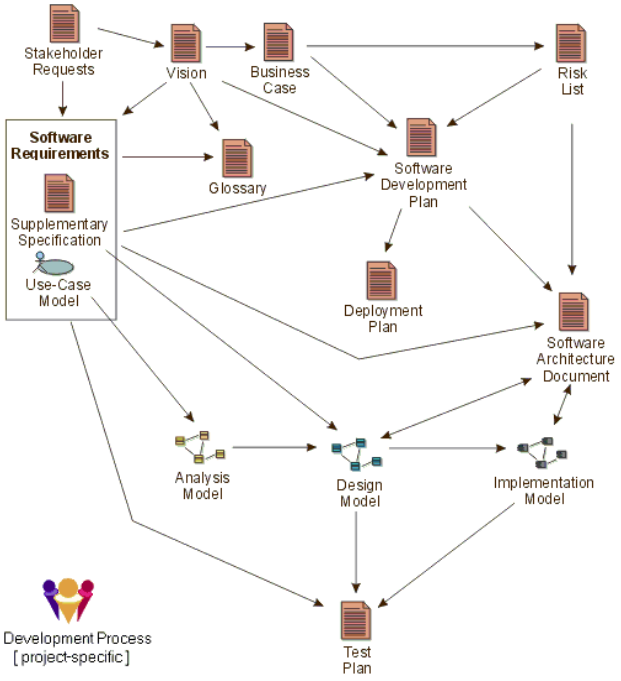
٦-۲-۳- فرآورده

فرآورده‌ها عبارتند از قطعه‌های اطلاعاتی(یا محصولاتی) که در طی فرآیند توسعه نرم‌افزار، تولید، استفاده یا بهنگام‌سازی می‌شوند (شکل ٦-۸). فرآورده‌ها به عنوان ورودی و خروجی(نتیجه) فعالیت‌های عمل می‌کنند. از دید شی‌گرایی، همانطوری که فعالیت‌ها به عنوان عملیاتی که یک شی انجام می‌نماید، عمل می‌کنند، فرآورده‌ها-نیز- به عنوان پارمترهای این فعالیت‌ها به کار می‌روند. نمونه‌هایی از فرآورده‌ها عبارتند از:

- مدل ها، مانند مدل موارد کاربری، مدل طراحی و ...
- عناصر مدل‌ها مانند کلاس‌ها، موارد کاربری، زیر سیستم‌ها
- مستندات، کد منبع[1] و فایل‌های اجرایی

توجه داشته باشید که فرآورده‌ها یک مستند کاغذی نیستند. بسیاری از فرآیندهای توسعه نرم‌افزار روی عمل مستند سازی و بالاخص مستندسازی کاغذی تاکید می ورزند ولی RUP چنین دیدگاهی ندارد بلکه استفاده از ابزارهای نرم‌افزاری مناسب برای تولید و ذخیره نمودن فرآورده‌ها را تشویق می‌نماید. هرگاه نیاز به مستند کاغذی وجود داشته باشد می‌توان از همان ابزارها برای تولید نسخه کاغذی فرآورده‌های مورد نظر را استفاده نمود.

[1] Source Code



شکل ٦-٨- فرآورده‌های اصلی در RUP

در RUP فرآورده‌ها در ۹ مجموعه طبقه‌بندی می‌شوند [٦]:

۱) مجموعه مدلسازی حرفه

- مدل موارد کاربری حرفه[۱]
- مستند معماری حرفه[۲]
- مستند دورنمای حرفه[۳]

۲) مجموعه نیازمندی‌ها

- مستند دورنما
- نیازمندی‌ها: نیازهای ذینفعان، مدل موارد کاربری و مشخصات تکمیلی[٤]
- برنامه مدیریت نیازمندی‌ها

۳) مجموعه تحلیل و طراحی

- مدل تحلیل[۱]

[1] Business Use-Case Model
[2] Business Architecture Document
[3] Business Vision Document
[4] Supplementary Specification

- مستند معماری نرم‌افزار[۲]
- مدل طراحی
- مدل استقرار

٤) مجموعه پیاده‌سازی

- کد منبع و فایل‌های اجرائی
- فایل‌های داده‌ای مورد نیاز

٥) مجموعه آزمایش

- برنامه آزمایش[۳]
- روال آزمایش[٤]
- مدل آزمایش
- داده‌های آزمایش

٦) مجموعه استقرار

- برنامه استقرار
- محصول نهائی
- مستندات کاربر
- مواد آموزشی[٥]

۷) مجموعه مدیریت پیکربندی

- برنامه مدیریت پیکربندی[٦]
- مستند درخواست تغییر[۷]

۸) مجموعه مدیریت پروژه

---

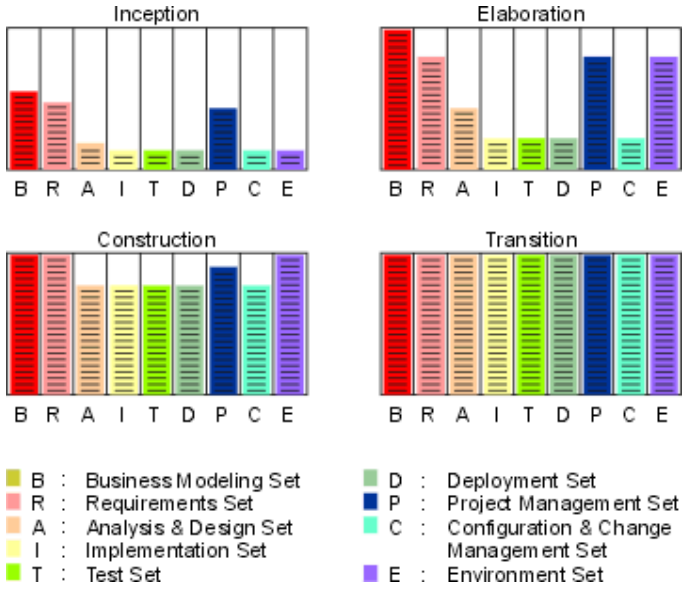
[1] Analysis Model
[2] Software Architecture Document
[3] Test Plan
[4] Test Procedure
[5] Training Materials
[6] Configuration Management Plan
[7] Change Request Document

- فرآورده‌های برنامه‌ریزی: برنامه توسعه نرم‌افزار، نقشه تکرار، فهرست ریسک‌ها و مورد حرفه[1]
- فرآورده‌های عملکردی مانند توصیف نشرها[2]، تشخیص وضعیت پروژه، ...

۹) مجموعه محیط[3]

- مورد توسعه[4]
- راهنمائی‌های مدلسازی حرفه
- راهنمائی‌های طراحی

در روش توسعه افزایشی این ۹ مجموعه در طی چرخه توسعه نرم‌افزار تکامل می‌یابند (شکل ٦-۹).



شکل ٦-۹- رشد مجموعه‌های فرآورده‌ها در طی چهار فاز تولید

٦-۲-٤- نظم[5]

نظم‌ها مجموعه‌ای از فعالیت‌های مرتبط با یکدیگر هستند که به یکی از نواحی مهم[1] پروژه وابسته باشند. نواحی مهم اشاره‌ای به مراحل کلاسیک فرآیند تولید آبشاری است. نظم‌ها طبیعت نیمه مرتبی دارند و به دو گروه تقسیم می‌شوند: نظم فرآیندی[2] و نظم پشتیبانی[3].

[1] Business Case
[2] Release Description
[3] Environment Set
[4] Development Case
[5] Discipline

نظم‌های فرآیندی عبارتند از:

- مدل‌سازی گردش کار حرفه[4]
- جمع‌آوری نیازمندی‌ها
- تحلیل و طراحی
- پیاده‌سازی
- آزمایش
- استقرار

نظم‌های پشتیبانی[5] عبارتند از:

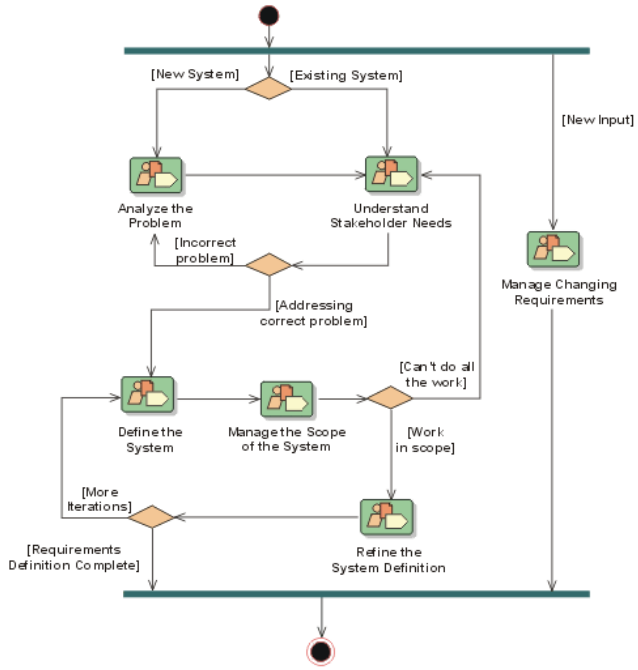
- مدیریت پروژه
- مدیریت پیکربندی
- محیط

هر نظم با یک گردش‌کار[6] نمایش داده می‌شود. هر گردش کار عبارت است از توالی مجموعه‌ای از فعالیت‌ها که نتیجه با ارزشی در پی دارند. در UML می‌توان گردش کارها را به صورت نمودار ترتیبی، همکاری یا فعالیت نمایش داد. در RUP معمولاً از نمودار فعالیت برای نمایش گردش‌کار استفاده می‌شود (شکل ٦-١٠).

می‌توان مجموعه همه فعالیت‌ها را در گردش‌کارهای متعددی سازمان‌دهی و طبقه‌بندی نمود. این مجموعه از فعالیت‌ها که ارتباط شدیدی با یکدیگر دارند، بهمراه مجموعه‌ای از فرآورده‌های مرتبط با آنها جزئیات گردش‌کار را ایجاد می‌کنند. جزئیات گردش کار، جریان اطلاعات-یعنی فرآورده‌های ورودی و خروجی-و نحوه ارتباط فعالیت ها بوسیله فرآورده‌های متفاوت را نمایش می‌دهد

---

[1] Area of Concerns
[2] Process Discipline
[3] Support Discipline
[4] Business Modeling Workflow
[5] Support Disciplines
[6] Workflow



شکل ٦-١٠- نمونه‌ای از یک نمودار فعالیت که گردش کار جمع‌آوری نیازمندی‌ها را نمایش می‌دهد

٦-٢-٥- عناصر ثانوی RUP

نقش‌ها، فعالیت‌ها (سازماندهی شده در گردش کار) و فرآورده‌ها، ستون فقرات ساختار ایستای RUP را تشکیل می‌دهند. اما مؤلفه‌های دیگری وجود دارند که با استفاده از آنها درک و به کارگیری RUP آسانتر می‌شود:

- **راهنما**[1]: فعالیت‌ها و گام‌ها معمولا به صورت مختصر بیان می‌شوند تا بتوان از آنها بعنوان مرجع استفاده نمود. همراه فعالیت‌ها، گام‌ها و فرآورده‌ها، راهنماها و جود دارند که عبارتند از قواعد پیشنهادی و اکتشافی[2] که نحوه انجام این فعالیت‌ها و گام‌ها را بیان می‌نمایند.
- **الگو**[3]: الگوها عبارتند از مدل‌هایی یا نمونه‌هایی از فرآورده‌ها که بوسیله آنها می‌توان فرآورده‌های مورد نظر را ساخت. RUP حاوی الگوهای زیر است:

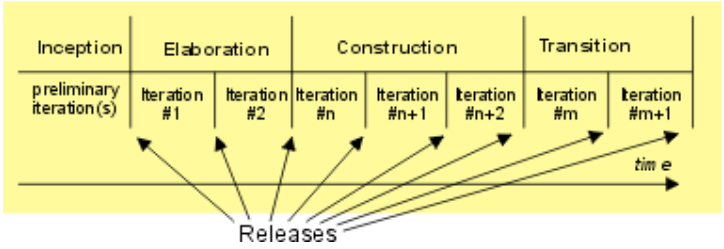
[1] Guideline
[2] Heuristics
[3] Template

- الگوهای Microsoft Word 97/2000 برای مستندسازی
- الگوهای Microsoft Frontpage برای بهنگام سازی RUP
- الگوهای Microsoft Project برای طرح و برنامه‌ریزی

- **راهنمای ابزار**[1]: راهنمایی‌های ابزار عبارتند از روش‌های انجام گام‌های یک فعالیت بوسیله یک نرم‌افزار معین. در RUP راهنمایی‌های ابزار، روش انجام فعالیت‌های گوناگون بوسیله نرم‌افزارهای شرکت Rational مانند: Rose، RequisitePro، Clear Case و Test Studio توضیح می‌دهند.
- **مفاهیم**: بعضی از مفاهیم پایه مانند: ریسک، تکرار، فرسنگ شمار و ... در قسمت‌های مختلف RUP توضیح داده شده‌اند.
- **چارچوب فرآیند**[2]: مؤلفه‌های ذکر شده چارچوب انعطاف‌پذیر RUP را بوجود می‌آورند زیرا این مؤلفه‌ها قابل اضافه، بهبود یا جایگزینی با توجه به نیازهای سازمان هستند.

۶-۳- ساختار پویا

این بعد چرخه توسعه نرم‌افزار را از دیدگاه مدیریتی بررسی می‌کند. در دیدگاه مدیریتی چرخه حیات نرم‌افزار به تعدادی دوره[3] تقسیم می‌شود که هر دوره با یک نسخه از محصول[4] که به مشتریان تحویل داده می‌شود، خاتمه می‌یابد. هر دوره شامل چهار فاز آغازین[5]، تشریح[6]، ساختن[7] و انتقال[8] است. هر فاز به نوبه خود به تعدادی تکرار[9] تقسیم می‌شود.(شکل ٦-١١)



شکل ٦-١١- رابطه فازها، تکرارها و نشرها

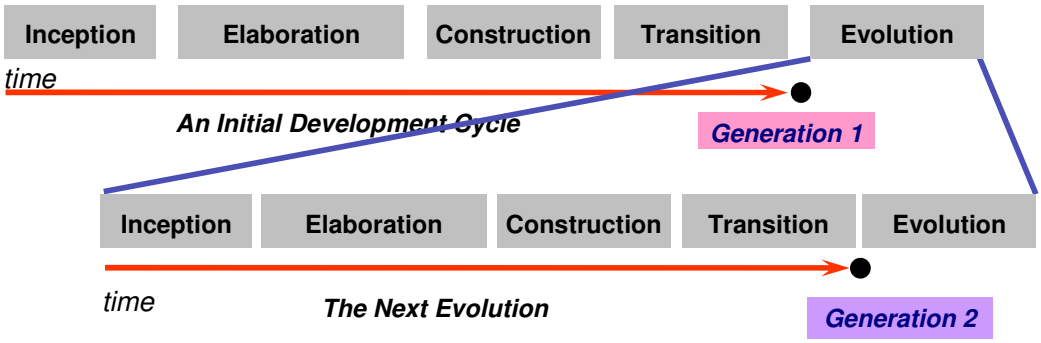
[1] Tool Mentor
[2] Procces Framework
[3] Cycle
[4] Software Generation
[5] Inception
[6] Elaboration
[7] Construction
[8] Transition
[9] Iterations

در پایان هر تکرار یک نشر تولید می‌شود که عبارت است از زیر مجموعه‌ای قابل اجرا (نه ضرورتاً کد قابل اجرا بلکه می‌تواند–نیز– یک نمونه باشد) از محصول نهائی که می‌تواند داخلی یا خارجی باشد. نشر داخلی بوسیله تیم توسعه‌دهندگان برای ارزیابی موفقیت یک تکرار و برای اهداف نمایشی استفاده می‌شود. در مقابل، نشر خارجی (مانند Beta Release) برای آزمایش بوسیله کاربران و مشتریان در اختیار آنها قرار می‌گیرد. سپس بر اساس نظرات آنها توسعه‌دهندگان اشکالات سیستم را برطرف می‌سازند.

توسعه‌دهندگان بوسیله مدل‌های متعدد آنچه در این فازها رخ می‌دهد را مجسم می‌کنند. از دیدگاه مدیریتی ما به روشی نیازمندیم که بتوان بوسیله آن روند پیشرفت پروژه را مشخص نمود به طوریکه مطمئن باشیم که گذشت تکرارهای متوالی ما را به سمت مقصد– همان محصول نهائی که نیازهای کاربران را برآورده سازد–می‌رساند.

همچنین ما نیاز داریم نقاطی را در زمان معین نماییم که در این نقاط بیاییم و بر اساس ملاک‌های دقیق مشخص نماییم: آیا ادامه دهیم، ندهیم یا تغییر روش نماییم. این نقاط شبیه فرسنگ شمارهای بزرگراه‌ها هستند که بوسیله آن مشخص می‌شود چه فاصله‌ای تا رسیدن به مقصد مانده است. از اینجا به این نقاط فرسنگ شمار[1] گویند. دو نوع فرسنگ شمارها داریم: اصلی[2] و فرعی[3]. از فرسنگ شمار اولی برای تعیین پایان یافتن یک فاز و از دومی برای تعیین پایان یافتن یک تکرار استفاده می‌شود.

هر تکرار، عبارتست از یک گذر کامل از همه نظم‌ها که همانطور که بیان شده، منتهی به یک نشر داخلی یا خارجی می‌گردد (شکل ٦-١١). پس از اتمام آخرین فاز و برای توسعه نسل بعدی نرم‌افزار، دوره بعدی شروع شده و تمام فازها دوباره انجام می‌شوند (شکل ٦-١٢).



[1] Milestones
[2] Major Milestone
[3] Minor Milestone

شکل ۶-۱۲- توسعه نسل بعدی از نرم‌افزار

۶-۳-۱- فاز آغازین

هدف اصلی این فاز بررسی امکان انجام پروژه از نقطه نظر اقتصادی و اطمینان از توافق همه ذینفعان روی صورت مسأله (پروژه) و اهداف آن است. این فاز به دلیل تلاش در توسعه نرم‌افزار جدید دارای اهمیت فراوانی است و در آن ریسک‌های نیازسنجی و تجاری مهمی وجود دارد که باید پیش از اینکه اجرای پروژه مورد توجه قرار گیرند. برای پروژه‌هایی که بر توسعه سیستم موجود متمرکزند، فاز آغازین کوتاهتر است، با اینحال این فاز برای حصول اطمینان از اینکه پروژه ارزش انجام دادن دارد و امکان پذیر نیز هست، انجام می‌شود. اهدافی که در این فاز دنبال می‌شوند، عبارتند از:

- تعیین محدوده سیستم نرم‌افزاری، شرایط مرزی و شرایط ارزیابی تکرارهای این فاز
- شناخت موارد کاربری مهم و حیاتی سیستم
- بدست آوردن یک معماری اولیه
- تشخیص زود هنگام خطرات احتمالی
- برآورد تقریبی هزینه، زمان و سودآوری پروژه
- برنامه‌ریزی برای فاز بعدی

از مهمترین فعالیت‌هایی که در این فاز انجام می‌شوند می‌توان به موارد ذیل اشاره نمود:

- تشخیص محدوده پروژه که با تعیین مهمترین نیازمندی‌های سیستم و شناخت محدودیت‌های موجود انجام می‌پذیرد. تعدادی از معیارهای ارزیابی محصول نهائی در اینجا مشخص می‌شوند.
- تهیه و آماده کردن مستند مورد کاری و ارزیابی جایگزین‌های موجود برای مدیریت ریسک، استخدام نیروی انسانی و برنامه‌ریزی پروژه
- موازنه بین فاکتورهای گوناگون مؤثر در تولید سیستم مانند هزینه و زمان مورد نیاز پروژه و سودآوری سیستم
- ارزیابی معماری پیشنهادی و جایگزین‌های موجود برای طراحی است، که تصمیم‌گیری در مورد خرید/تولید/ استفاده مجدد از منابع موجود را نیز دربردارد. با این کار زمان و منابع مورد نیاز به صورت واقعی‌تر پیش‌بینی می‌شوند

مجموعه‌ای از فرآورده‌های در فاز آغازین توسعه داده می‌شوند که از آن جمله می‌توان به موارد ذیل اشاره نمود:

- مستند دورنما
  - یک دید اولیه و کلی، با نگرشی فنی، درباره نیازمندی‌های اصلی، ویژگی‌های کلیدی و محدودیت‌های اساسی سیستم را به توسعه‌دهندگان می‌دهد
- مدل موارد کاربری
  - دربردارنده همه موارد کاربری و عواملی که در این فاز قابل تشخیصند
- واژه‌نامه
  - شامل اصطلاحات مهمی که در پروژه استفاده می‌شود همراه تعریف دقیق آنهاست
- مورد کاری ابتدائی
  - شرح محیط شغلی
  - فاکتورهای موفقیت (برآورد درآمد، شناخت بازار و ...)
  - برآورد هزینه‌های مالی
  - پیش‌بینی ابتدائی ریسک‌های احتمالی
  - برنامه‌ریزی برای پروژه و زمانبندی آن(زمان شروع فازهای چهارگانه و تکرارهای آنها)
- تخمین منابع مورد نیاز در این فاز یا کل پروژه است
- مدل دامنه[1] که از فهرست انعطاف‌پذیری بیشتری دارد
- مدل حرفه[2] که فرآیندها حرفه سیستم را بیان می‌نماید
- نمونه‌هایی[3] از سیستم مورد نظر

فاز آغازین دارای چند فرسنگ شمار با اهمیت است که عدم عبور از آنها به‌منزله عدم موفقیت در توسعه خواهد بود. این فرسنگ‌شمارها عبارتند از:

❖ توافق ذینفعان روی تعیین محدوده سیستم و برآوردهای انجام شده روی زمان و هزینه مورد نیاز

[1] Domain Model
[2] Business Model
[3] Prototypes

❖ پایداری موارد کاربری تشخیص شده در این فاز است. پایداری معیاری برای کیفیت درک نیازمندی‌های سیستم تلقی می‌گردد.

❖ منطقی بودن برآورد هزینه‌ها، زمان مورد نیاز و ریسک‌ها. در واقع پاسخ به این سوال که آیا برنامه فازها حاوی جزئیات کافی هستند؟

۶-۳-۲- فاز تشریح

هدف اصلی این فاز تحلیل دامنه مسأله، بدست آوردن معماری مناسب و مستحکم برای سیستم، توسعه نقشه پروژه و جلوگیری از ریسک‌های حیاتی سیستم است. استحکام و پایداری معماری از طریق یک یا چند نمونه‌ی اولیه ساختاری ارزیابی می‌شود. اهداف این فاز عبارتند از:

- بدست آوردن یک معماری بنیادی[1] (پایا) مناسب و پایا بوده به طوریکه زمینه اصلی و نقطه شروع توسعه سیستم در فازهای بعدی باشد
- بدست آوردن یک دورنمای مناسب که بعنوان دورنمای بنیادی عمل می‌نماید
- بدست آوردن یک برنامه پایا (بنیادی) برای توسعه فاز ساخت
- نشان دادن این که معماری بنیادی قدرت پشتیبانی از دورنمای بدست آمده با هزینه و زمان مناسب داراست

از مهمترین فعالیت‌هایی که در این فاز انجام می‌شوند می‌توان به موارد ذیل اشاره نمود:

- توسعه و بدست آوردن جزئیات دورنما
- مشخص نمودن محیط‌های توسعه مورد نیاز و جایگاه ابزارهای CASE در خودکارسازی فرآیند تولید
- توسعه معماری و انتخاب مؤلفه‌های لازم است. مؤلفه‌های در دسترس ارزیابی می‌شوند و درباره ساختن/خریدن /استفاده مجدد از مؤلفه‌های مورد نیاز، تصمیم‌گیری‌های لازم اتخاذ می‌شوند و بدین صورت می‌توان هزینه و زمان مورد نیاز فاز بعدی (ساخت) پیش‌بینی و برای آن برنامه‌ریزی مناسبی نمود

[1] Architecture Baseline

مجموعه‌ای از فرآورده‌های در فاز تشریح توسعه داده می‌شوند که از آن جمله می‌توان به موارد ذیل اشاره نمود:

- مدل موارد کاربری (حداقل باید ۸۰٪ آن کامل باشد) که در آن بیشتر موارد کاربری سیستم و عوامل آن، شناسائی و مستند شده باشند
- نیازمندی‌های تکمیلی[1] که شامل نیازهای غیر وظیفه‌مندی و نیازمندی‌هایی که به یک مورد کاربری معینی انتساب داده نشده‌اند
- توصیف معماری سیستم
- نمونه (آزمایشگاهی) از یک معماری قابل اجرا
- فهرست ریسک‌های و موارد کاری بازبینی شده
- برنامه تفصیلی توسعه کل پروژه

فاز تشریح دارای چند فرسنگ شمار با اهمیت است که عدم عبور از آنها به‌منزله عدم موفقیت در توسعه خواهد بود. این فرسنگ‌شمارها عبارتند از:

- ❖ آیا به یک دورنمای پایا[2] رسیده‌ایم؟
- ❖ آیا معماری بدست آمده پایدار است؟
- ❖ آیا نمونه‌های اجرائی ساخته شده نشان می‌دهند که ریسک‌های اصلی به خوبی شناخته و راه مقابله با آن مشخص شده است؟
- ❖ آیا برنامه فاز ساخت حاوی جزئیات کافی است؟
- ❖ آیا همه ذینفعان بر توانایی دستیابی به دورنمای مورد نظر بوسیله اجرای دقیق نقشه فعلی و با توجه به معماری فعلی اتفاق نظر دارند؟
- ❖ آیا مصرف حقیقی منابع، تا به حال، با مصرف پیش‌بینی شده سازگار است؟

[1] Supplementary Requirement
[2] Stable Vision

۶-۳-۳- فاز ساخت

از یک نگاه، عبارتست از فرآیند تولید صنعتی[1] که در آن روی مدیریت منابع، کنترل عملیات، بهینه‌سازی هزینه‌ها، زمانبندی و کیفیت تاکید می‌شود. در واقع، در این فاز تولیدات ذهنی ایجاد شده و مدل‌های تبدیل به واقعیت می‌شوند. اهداف این فاز عبارتند از:

- به‌حداقل رساندن هزینه‌های تولید بوسیله بهینه‌سازی استفاده از منابع و نادیده گرفتن بعضی از کارهای تکراری و غیر مهم
- بدست آوردن یک کیفیت عالی در سریعترین زمان عملی ممکن
- رسیدن به نسخه‌های قابل استفاده عملی کاربران (آلفا، بتا) در سریعترین زمان ممکن

از مهمترین فعالیت‌هایی که در این فاز انجام می‌شوند می‌توان به موارد ذیل اشاره نمود:

- مدیریت منابع و کنترل آن و همچنین بهینه‌سازی فرآیند تولید
- تکمیل توسعه مؤلفه‌ها و انجام آزمایش‌های گوناگون با توجه شرایط ارزیابی
- ارزیابی نشرها در مقایسه با دورنمای مطلوب (همان شرایط ارزیابی)

مجموعه‌ای از فرآورده‌های در فاز ساخت توسعه داده می‌شوند که از آن جمله می‌توان به موارد ذیل اشاره نمود:

- محصول نهائی نرم‌افزار
- دفترچه راهنمای کاربران
- توصیف نشرهای فعلی

فاز ساخت دارای چند فرسنگ شمار با اهمیت است که عبارتند از:

- ❖ آیا نشر محصول به اندازه کافی محکم و پایدار است که برای استفاده کاربران آماده باشد؟
- ❖ آیا هزینه واقعی منابع با هزینه پیش‌بینی شده هنوز سازگار است؟

با توجه به اینکه پروژه به حالت ثابتی رسیده است، عدم عبور از این فرسنگ‌شمارها به معنی ایجاد تکرار بیشتر و اصلاح زمان‌بندی پروژه است و منجر به عدم موفقیت پروژه نخواهد شد. در صورتی پروژه به حالت عدم موفقیت خواهد رفت که تکرارهای این فاز خارج از حد بودجه و یا زمانبندی باشد و مدیریت تصمیم به توقف پروژه بگیرد.

[1] Manufacturing

۶-۳-۴- فاز انتقال

هدف اصلی این فاز عملیاتی کردن نرم‌افزار یا انتقال آن به جامعه کاربران است. تمرکز این فاز بر این است که تضمین نماید نرم‌افزار برای کاربران نهایی آماده می‌باشد و می‌تواند در محیط کاربران نصب و راه‌اندازی شود. اهداف این فاز عبارتند از:

- انتقال نرم‌افزار به محیط کاربران و گرفتن نظرات آنها در مورد نحوه عملکرد سیستم جدید
- بدست آوردن توافق همه ذینفعان درباره کامل بودن Deployment Baseline و سازگار بودن آن با شرایط ارزیابی دورنما
- بدست آوردن Product Baseline نهائی در سریعترین زمان و با کمترین هزینه ممکن

از مهمترین فعالیت‌هایی که در این فاز انجام می‌شوند می‌توان به موارد ذیل اشاره نمود:

- انجام جنبه‌های مهندسی مربوط به استقرار شامل بسته‌بندی و نصب محصول
- انجام فعالیت‌های بهینه‌سازی مانند اصلاح خطاها و سرعت بخشیدن به اجرای برنامه
- انجام آزمایش بتا برای آزمایش سیستم و ارزیابی نتایج این آزمایش با توجه به عملکرد مورد انتظار کاربران
- آماده‌سازی مستندات، آموزش کاربران و آمادگی برای پاسخگویی و پشتیبانی از آنها
- اجرای هر دو سیستم، قدیمی و جدید با هم به صورت موازی، برای مدتی از زمان، برای مقایسه عملکرد این دو سیستم

مجموعه‌ای از فرآورده‌های در فاز انتقال توسعه داده می‌شوند که از آن جمله می‌توان به موارد ذیل اشاره نمود:

- تکمیل دفترچه راهنمای کاربران
- تکمیل دفترچه نصب و نگهداری
- مستند Notes Release که حاوی اطلاعات مربوط به اشکالات برنامه، شماره نسخه فعلی و ... است

فاز انتقال دارای چند فرسنگ شمار با اهمیت است که عبارتند از:

- ❖ آیا کاربر راضی است؟
- ❖ هزینه‌های پیش‌بینی‌شده با هزینه‌های واقعی چه تفاوتی دارند؟

۷- نظم‌های RUP

فرآیند RUP دارای ۹ نظم است که هر نظم توسط تعدادی فعالیت انجام می‌شود تا به یک مجموعه فرآورده‌های خاص دست یافته شود. هر نظم را می‌توان در قالب نمودار فعالیت UML نمایش داد. نظم‌های RUP به ترتیب عبارتند از:

۱. نظم مدلسازی حرفه[۱]
۲. نظم نیازمندی‌ها[۲]
۳. نظم تحلیل و طراحی[۳]
۴. نظم پیاده‌سازی[۴]
۵. نظم آزمایش[۵]
۶. نظم استقرار[۶]
۷. نظم مدیریت پروژه[۷]
۸. نظم مدیریت پیکربندی[۸]
۹. نظم محیط[۹]

در ادامه به بررسی هر یک از نظم‌های RUP می‌پردازیم.

۷-۱- نظم مدلسازی حرفه

نظم مدلسازی حرفه زمانی مورد استفاده قرار می‌گیرد که پروژه دارای کاربران زیاد است و/یا حجم زیادی از داده‌ها/فرآیندها باید پردازش/تحلیل گردند. در غیر این دو صورت استفاده از نظم مدلسازی حرفه توصیه نمی‌شود. در این نظم، سعی به استفاده از مفاهیم مدلسازی حرفه می‌شود که قابل استفاده در

[1] Business Modeling Discipline
[2] Requirement Discipline
[3] Analysis & Design Discipline
[4] Implementation Discipline
[5] Test Discipline
[6] Deployment
[7] Project Management
[8] Configuration Management
[9] Environment

مهندسی نرم‌افزار نیز هستند. استفاده از روش‌های مشترک در مهندسی نرم‌افزار و مدلسازی حرفه دو مزیت عمده را به همراه خواهد داشت که عبارتند از:

- تسهیل درک ارتباط بین این دو فعالیت که معمولاً به صورت مجزا انجام می‌شوند
- نگاشت فرآورده‌های مدلسازی حرفه به فرآورده‌های مدلسازی نرم‌افزار مشکل نخواهد بود

هدف از نظم مدلسازی حرفه، درک رفتار سازمان و نحوه عملکرد آن است. با استفاده از مدلسازی انجام شده اطمینان حاصل می‌شود که مشتریان، کاربران نهائی و توسعه دهندگان دارای یک دیدگاه واحد و مشترک از سازمان هستند. به این ترتیب می‌توان نیازمندی‌های سیستم مورد نظر را با دقت و صحت بیشتری تعیین نمود. در نظم مدلسازی حرفه از مدل‌های حرفه استفاده می‌شود که شامل مدل مورد کاربری حرفه و مدل شی حرفه هستند.

۷-۱-۱- مفاهیم نظم مدلسازی حرفه

در نظم مدلسازی حرفه مفاهیم خاصی مورد استفاده قرار می‌گیرند، که در ادامه به بررسی آنها می‌پردازیم:

- حرفه (کسب و کار)[1]
  - سیستمی که نرم‌افزار مورد نظر بخشی از آن به‌شمار می‌رود
- مورد کاربری حرفه[2]
  - ترتیبی از کنش‌هایی[3] که سازمان انجام می‌دهد و و منجر به «نتیجه ارزشمند» و قابل مشاهده برای عامل حرفه خاص می‌شود
  - مورد کاربری حرفه دربردارنده تمام جریان‌های اصلی و فرعی است که منجر به ایجاد نتیجه ارزشمند و قابل مشاهده می‌شود
- عامل حرفه[4]
  - شخص یا سیستمی بیرون از حرفه که با آن تعامل دارد
- کارگر حرفه[1]

[1] Business
[2] Business Use-Case
[3] Actions
[4] Business Actor

- تجریدی از نیروی انسانی یا سیستم نرم‌افزاری است که نقشی در داخل حرفه ایفا می‌کند
- هر کارگر حرفه با کارگران دیگر تعامل داشته و یک یا بیشتر موجودیت حرفه را پردازش می‌نماید

- موجودیت حرفه[2]
  - حاوی قسمت مهم و پایای اطلاعات که هنگام اجرای مورد کاربری حرفه بوسیله کارگران/عاملان حرفه استفاده یا پردازش می‌شود
  - اسناد کاغذی و فرم‌های در گردش نمونه‌هایی از موجودیت‌های حرفه هستند

۷-۱-۲- مهمترین نقش‌های نظم مدلسازی حرفه

در نظم مدلسازی حرفه چهار نقش عمده وجود دارند که اغلب فرآورده‌ها و محصولات این نظم را تولید می‌کنند. در ادامه به بررسی این نقش‌ها می‌پردازیم.

- تحلیلگر فرآیند حرفه[3]
  - وظیفه این نقش، تعریف معماری حرفه، موارد کاربری و عوامل حرفه و تعیین چگونگی تعامل آنها است. با مشخص شدن این موارد، تحلیل‌گر فرآیند حرفه، مدل مورد کاربری حرفه و مدل تحلیل حرفه را تهیه می‌نماید. تحلیل‌گر با روش‌های متداول همانند مصاحبه، پرسش‌نامه و ... اطلاعات مورد نیاز خود را فراهم می‌کند.
- طراح حرفه[4]
  - وظیفه این نقش، تشریح جریان‌های موارد کاربری حرفه، تعیین کارگرها و موجودیت‌های حرفه است. طراح با استفاده اطلاعاتی که تحلیل‌گر در اختیار او قرار می‌دهد و با تجربه‌ای که دارد، موارد کاربری حرفه را تشریح می‌کند.
- سهامداران (ذینفعان)

---

[1] Business Worker
[2] Business Entity
[3] Business Process Analyst
[4] Business Designer

- منظور از سهامداران در اینجا کاربران نهایی و مشتری است. وظیفه این نقش، فراهم نمودن اطلاعات ورودی و بازنگری طرح‌های توسعه‌دهندگان است. ذینفعان اطلاعات مورد نیاز برای تحلیل‌گر را فراهم نموده و پس از ایجاد موارد کاربری آنها را مورد بازنگری قرار می‌دهند تا صحت آنها را مورد تایید قرار دهند.

- بازبین فنی[1]

- وظیفه این نقش، بازنگری فرآورده‌های ایجاد شده در این نظم است. بازنگری محصولات به منظور اطمینان از صحت ارتباط بین مدل مورد کاربری و عملکرد کاربران، صحت موجودیت‌ها و کارگرهای حرفه و محصولات تولید شده در این نظم است.

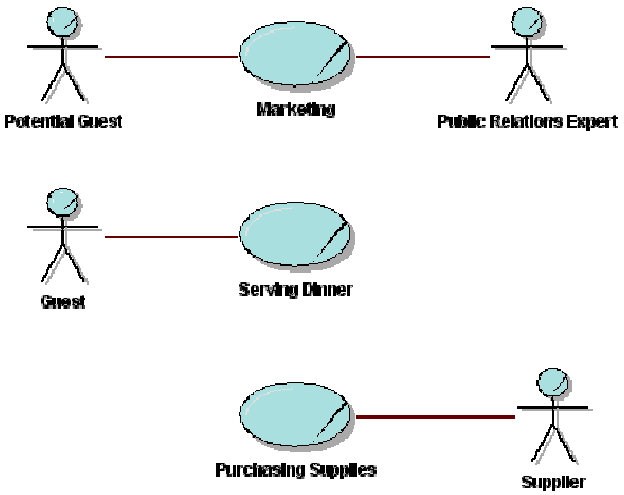
۷-۱-۳- مهمترین فرآورده‌های نظم مدلسازی حرفه

در نظم مدلسازی حرفه مستندات و فرآورده‌های بسیاری در فازهای مختلف مورد استفاده قرار گرفته و تولید یا اصلاح می‌شوند. در ادامه به بررسی مهمترین فرآورده‌هایی این نظم پرداخته می‌شود.

- مدل مورد کاربری حرفه[2]

- این مدل توسط تحلیل‌گر فرآیند حرفه ایجاد می‌شود و هدف آن، نمایش عملکرد سازمان و تعیین حیطه مسئله است. شکل ۷-۱ نمونه‌ای از مدل مورد کاربری حرفه را نشان می‌دهد.

[1] Technical Reviewer
[2] Business Use-Case Model



شکل ۷-۱- نمونه‌ای از مدل مورد کاربری حرفه در یک رستوران

- مدل تحلیل حرفه[1]
  - این مدل عینیت بخشیدن به موارد کاربری حرفه[2] را به همراه تعامل آن با کارگرها و موجودیت‌های حرفه نشان می‌دهد. در واقع، این مدل تجریدی از نحوهٔ ارتباط بین کارگرها و موجودیت‌های حرفه و بیان اینکه چگونه ارتباط بین این دو سبب اجرای مورد کاربری حرفه می‌شود. در واقع این مدل، یک مدل شی حرفه[3] است که توسط تحلیل‌گر فرآیند حرفه ایجاد می‌شود.
- مشخصات تکمیلی حرفه[4]
  - شامل همه تعاریف مهم حرفه که در دو مدل فوق ذکر نشده‌اند، یا اجبارها و محدودیت‌هایی است که باید به دو مدل فوق اعمال شوند. این مشخصات تکمیلی توسط تحلیل‌گر فرآیند حرفه ایجاد می‌شود.
- دورنمای حرفه[5]
  - اهداف و مقاصد مدلسازی حرفه را مشخص می‌کند. این کار توسط تحلیل‌گر فرآیند حرفه انجام می‌شود.

---

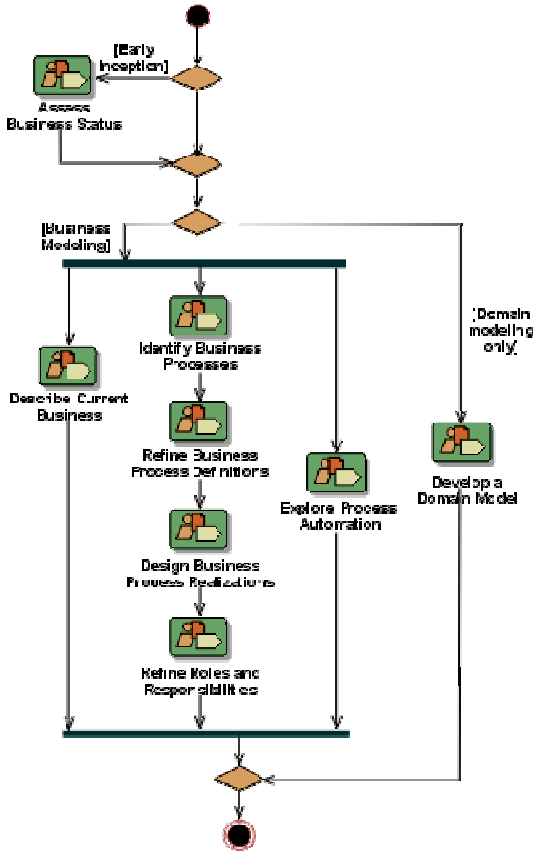
[1] Business Analysis Model
[2] Use-Case Realization
[3] Business Object Model
[4] Supplementary Business Specifications
[5] Business Vision

- مستند معماری حرفه[1]
  - حاوی نگاهی جامع بر جنبه‌های مهم حرفه از نظر معماری با توجه به دیدگاه‌های مختلف است. مستند معماری حرفه ابزاری برای ارتباط بین ذینفعان مختلف و تیم پروژه است، چرا که چیستی و چرایی حرفه را بیان می‌کند. این مستند توسط تحلیل‌گر فرآیند حرفه ایجاد می‌شود.
- فرهنگ لغات حرفه[2]
  - شامل اصطلاحات مهم و تعاریف آنها که در نظم مدلسازی حرفه مورد نیاز هستند. این مستند توسط تحلیل‌گر فرآیند حرفه ایجاد می‌شود.

۷-۱-۴- گردش‌کار نظم مدلسازی حرفه

گردش‌کار نظم مدلسازی حرفه در شکل ۷-۲ نشان داده شده است. این گردش کار هم شامل مدلسازی دامنه و هم شامل مدلسازی حرفه است. در این نظم، با شناخت حرفه و مدلسازی آن، مدل‌های تولید شده تبدیل به مدل‌های قابل درک توسط تیم توسعه می‌شوند.

[1] Business Architecture Document
[2] Business Glossary



شکل ۷-۲- گردش‌کار نظم مدلسازی کسب و کار

بطور کلی در نظم مدلسازی حرفه دو کار عمده انجام می‌شود که عبارتند از:

۱. تحلیلگر فرآیندهای حرفه، فرآیندهای اصلی حرفه و ذینفعان مشترک را شناسایی و به صورت مدل موارد کاربری حرفه توصیف می‌نماید
   - تحلیل‌گر حرفه، مفاهیم مهم را شناسائی و تعریف می‌کند
   - وظیفه طراح حرفه تشریح تفصیلی موارد کاربری شناسائی شده، شناسائی نقش‌ها و مسئولیت‌های مورد نیاز در حرفه و تعیین خروجی فرآیندهای حرفه است
   - ذینفعان در نقش بازبین فنی، جامعیت و درستی مدل حرفه را تعیین می‌کنند
۲. نگاشت مدل حرفه به مدل‌های نرم‌افزاری
   - کارکنان حرفه همان عوامل سیستم هستند
   - رفتار کارکنان حرفه نشان می‌دهد که خودکارسازی باید شامل چه فرآیندهایی باشد
     - این ویژگی در یافتن موارد کاربری سیستم نیز کمک می‌کند

- موجودیت‌های حرفه می‌توانند اشیایی از سیستم باشند که باید نگهداری شوند
  - مدل حرفه در شناسایی کلاس‌های موجودیت[1] (در مدل تحلیل) کمک می‌کند

۷-۱-۵- پشتیبانی ابزار از نظم مدلسازی حرفه

ابزارهایی که می‌توان در نظم مدلسازی حرفه از آنها استفاده نمود، عبارتند از:

- ابزار Rose برای مدلسازی تصویری با استفاده از نمادهای UML
- ابزار Requisite Pro برای نگهداری بخش‌های متنی مدل‌های تولید شده بوسیله Rose
- ابزار SoDA برای خودکارسازی فرآیند تولید مستندات

۷-۲- نظم نیازمندی‌ها[2]

این نظم در مورد مدیریت خواسته‌ها، نیازها و نیازمندی‌های کاربران و مالکان سیستم است. کاربران در ابتدا درخواست‌های خود را به صورت غیرشفاف بیان می‌نمایند ولی با پیشرفت پروژه، شناخت بیشتری از نیازمندی‌ها پیدا نموده و این سبب تغییراتی در نیازمندی‌های اولیه پروژه خواهد شد تا جائیکه این تغییرات گاهی منجر به تغییرات عمده‌ای در روند پروژه خواهد شد.

نیازمندی‌های نرم‌افزار به دو دسته وظیفه‌مندی و غیروظیفه‌مندی تقسیم می‌شوند. نیازمندی‌های وظیفه‌مندی درخواست‌ها و خصوصیاتی هستند که باید در نرم‌افزار وجود داشته باشند و نیازمندی‌های غیر وظیفه‌مندی، درخواست‌ها و خصوصیاتی هستند که بودن آنها سبب بهبود نرم‌افزار خواهد شد. از جمله دسته‌بندی‌هایی که برای نیازمندی‌های غیروظیفه‌مندی انجام شده است می‌توان به دسته‌بندی FURPS اشاره نمود که عبارتند از:

- عملکرد[3] که شامل ویژگیها، قابلیت‌های نرم‌افزار و امنیت است
- قابلیت استفاده[4] که شامل زیبایی واسط کاربر، سهولت آموزش، سهولت استفاده، یکنواختی واسط کاربر و ... است.
- قابلیت اعتماد[1] که شامل فرکانس بروز خطا، برگشت‌پذیری[2]، صحت، قابلیت پیش‌بینی و زمان میانگین بین خطاها است.

[1] Entity Classes
[2] Requirement Discipline
[3] Functionality
[4] Usability

- کارایی[3] که شامل سرعت، قابلیت دسترسی، زمان پاسخ، حافظه مورد نیاز و ... است
- قابلیت پشتیبانی[4] که شامل قابلیت آزمایش، نگهداری، انعطاف‌پذیری در مقابل تغییرات، قابلیت توسعه[5]، قابلیت پیکربندی[6] و ... است.

از دیدگاه عملی تقسیم‌بندی قبل آنچنان حائز اهمیت نیست و ارزش طبقه‌بندی قبل، فراهم کردن یک الگو برای شناسایی نیازمندی‌ها و اطمینان از جامع بودن آن است. با داشتن درک درست نیازمندی‌ها که با دانستن «چراها» و «چگونه‌ها» حاصل می‌شود، می‌توان به درک درست و عمیقی از سیستم دست یافت. با توجه به توضیحات ذکر شده، می‌توان مفهوم ذینفعان را در این نظم یک مفهوم اساسی است.

نظم نیازمندی‌ها اهداف خاصی را دنبال می‌کند که عبارتند از:

- برقراری و نگهداری موارد توافق با مشتریان و سایر ذینفعان در مورد کارهایی که سیستم باید انجام دهد
- ارائه شناخت بهتر از نیازمندی‌های سیستم برای استفاده توسعه‌دهندگان سیستم
- تعریف محدوده سیستم
- تهیه پایه‌ای جهت تخمین هزینه و زمان توسعه سیستم
- تعریف یک واسط کاربری برای سیستم با تمرکز بر روی نیازها و اهداف کاربران

۷-۲-۱- فرآیند شناسایی و استفاده از نیازمندی‌ها

فرآیند شناسایی نیازمندی‌ها و استفاده از نیازمندی‌ها فرآیندی است که از ابتدای توسعه نرم‌افزار تا پایان آن همواره مورد توجه قرار می‌گیرد. این فرآیند، با نیازها و درخواست‌های ذینفعان شروع می‌شود. درخواست‌های ذینفعان منجر به توسعه سند دورنما می‌شود و در این سند به صورت کلان بیان می‌شود. نیازمندی‌های ذینفعان به صورت تفصیلی در مدل موارد کاربری (حرفه) و مشخصات تکمیلی (حرفه) بیان می‌شود. این دو سند پایه‌ای برای توسعه نرم‌افزار و پوشش نیازمندی‌های کاربران هستند. عینیت بخشیدن

---

[1] Reliability
[2] Recoverability
[3] Performance
[4] Supportability
[5] Extensibility
[6] Configurability

به نیازمندی‌های کاربران در مدل طراحی و مستندات کاربر نهایی انجام می‌شود. در واقع، مدل‌های طراحی نشان می‌دهند که چقدر از خواسته‌های کاربران در غالب نیازمندی‌ها برآورده می‌شود.

روش‌های بسیاری برای جمع‌آوری و دریافت خواسته‌ها و نیازهای کاربران وجود دارد که از جمله معروفترین آنها می‌توان به مشاهده محیط کاربران، پرسشنامه، مصاحبه و نمونه‌سازی اشاره نمود. روش نمونه‌سازی[1] در موردی استفاده می‌شود که خواسته‌های کاربران بصورت دقیق مشخص نیست و ایده آن نمایش نمونه‌ای از برنامه به کاربران برای درک بهتر محیط نرم‌افزار توسط کاربران و تعیین بهتر نیازمندی‌ها پس از آشنایی با محیط نرم‌افزار است.

هر نمونه، نمایش اجرایی از امکانات و توانائی‌های سیستم است و می‌تواند سبب کاهش مخاطرات و ابهامات درباره موضوعات زیر شود:

- امکان موفقیت محصول از نظر تجاری و اقتصادی
- کارایی و پایداری تکنولوژی‌های کلیدی که در محصول استفاده خواهند شد
- قابلیت استفاده محصول
- درک نیازمندی‌ها

نمونه‌ها از نظر هدفی که دنبال می‌کنند به دو دسته رفتاری و ساختاری تقسیم می‌شوند. نمونه‌های رفتاری بدنبال نمایش رفتار خاصی از سیستم هستند در حالیکه نمونه‌های ساختاری بدنبال نمایش علاقه‌مندی‌ها و دغدغه‌های معمارانه یا فناوری هستند. از نظر نتیجه نیز نمونه‌ها به دو دسته نمونه‌های تکاملی[2] و دورانداختنی[3] تقسیم می‌شوند. نمونه‌های دورانداختنی پس از انجام کار و یافتن نیازمندی‌های مورد نظر، دورانداخته می‌شود، اما نمونه‌های تکاملی سرانجام منجر به نرم‌افزار نهایی می‌شوند.

فرآیند RUP از تولید و توسعه نمونه‌های ساختاری در فاز تشریح با ضمیمه چند نمونه رفتاری حمایت می‌نماید.

[1] Prototyping
[2] Evolutionary Prototypes
[3] Exploratory Prototypes

۷-۲-۲- مهمترین نقش‌های نظم نیازمندی‌ها

در نظم نیازمندی‌ها چهار نقش عمده وجود دارند که اغلب فرآورده‌ها و محصولات این نظم را تولید می‌کنند، این نقش‌ها عبارتند از:

- تحلیل‌گر سیستم
  - وظیفه اصلی تحلیل‌گر سیستم، راهبری فرآیند جمع‌آوری نیازمندی‌ها و تهیه مدل موارد کاربری است.
- تعیین‌کنندهٔ نیازمندی‌ها[1]
  - وظیفه اصلی تعیین‌کننده نیازمندی‌ها، تشریح تفصیلی وظایف سیستم توسط تشریح نیازمندی‌ها است
- معمار نرم‌افزار[2]
  - معمار نرم‌افزار، مسئول معماری نرم‌افزار و اجبارها و شرایط فنی تاثیرگذار بر طراحی و پیاده‌سازی سیستم است
- بازبین فنی
  - مسئولیت اصلی بازبین فنی، مرور و بازبینی محصولات و فرآورده‌ها تولید شده در این نظم و ارائه بازخورد مناسب است.

۷-۲-۳- مهمترین فرآورده‌های نظم نیازمندی‌ها

فرآورده‌های نظم نیازمندی‌ها را می‌توان به دو سطح کلان و تفصیلی تقسیم نمود. فرآورده‌های سطح کلان حاوی جزییات کمی هستند و عبارتند از:

- درخواست‌های ذینفعان
  - حاوی هر نوع درخواستی از طرف ذینفعان (مشتری، کاربر نهایی، فروشنده و ...) در هنگام توسعه نرم‌افزار است. علاوه بر این، درخواست‌های ذینفعان حاوی هر نوع منبع خارجی مورد استفاد در پروژه است. این مستند توسط تحلیل‌گر سیستم و در فازهای آغازین و تشریح ایجاد و مورد استفاده قرار می‌گیرد.

[1] Requirements Specifier
[2] Software Architect

- مستند دورنما[1]
  - دید ذینفعان پروژه از محصولی که تولید می‌شود، را نشان می‌دهد و حاوی نیازها و قابلیت‌های کلیدی محصول است. مستند دیدگاه در فاز آغازین و با مشارکت تحلیل‌گر سیستم ایجاد می‌شود.
- طرح مدیریت نیازمندی‌ها[2]
  - فرآورده‌های نیازمندی‌ها، انواع نیازمندی و خصوصیات نیازمندی‌های مرتبط را تشریح می‌نماید. همچنین اطلاعاتی که می‌بایست جمع‌آوری شده، مکانیزم‌های کنترلی برای برای سنجش، گزارش‌گیری و کنترل تغییرات نیازمندی‌های محصول را مشخص می‌کند.

فرآورده‌های تفصیلی دارای جزئیات کامل‌تری هستند و عبارتند از:

- مدل موارد کاربری
- مشخصات تکمیلی
- پرده‌داستان مورد کاربری[3]
- فرهنگ لغات
- مخزن خصوصیات نیازمندی‌ها

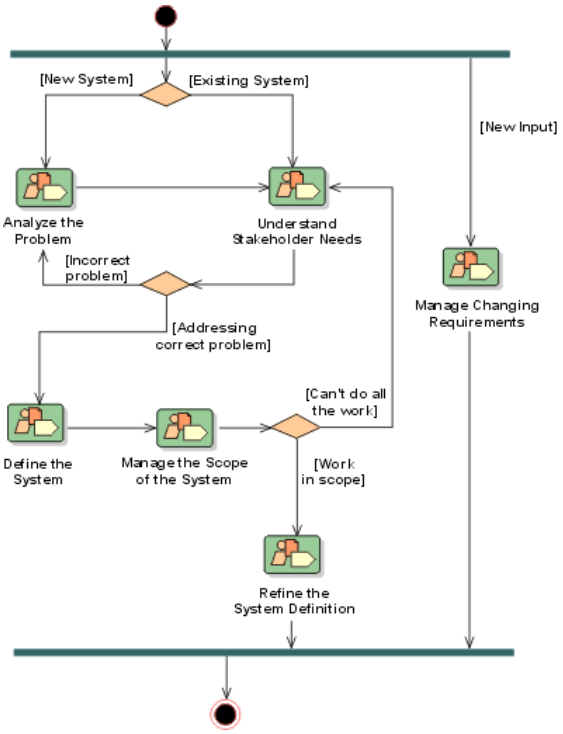
۷-۲-۴- گردش‌کار نظم نیازمندی‌ها

گردش‌کار نظم نیازمندی‌ها در شکل ۷-۳ ارائه شده است. این گردش‌کار هم شامل توسعه سیستم جدید و هم شامل بهبود سیستم موجود است. در این نظم، تحلیل‌گر سیستم به کمک ذینفعان سعی به مشخص نمودن محدوده سیستم می‌نماید. به‌عبارت بهتر مشخص می‌شود که سیستم می‌بایست چه کارهایی را انجام دهد و چه کارهایی را انجام ندهد. همچنین تحلیل‌گر نیازهای غیروظیفه‌مندی را شناسایی نموده تا در مستند دورنما مورد استفاده قرار گیرند.

[1] Vision
[2] Requirements Management Plan
[3] Storyboard Use-Case

پس از شناسایی محدوده پروژه و تعیین موارد کاربری توسط تحلیل‌گر، به تعیین‌کنندهٔ نیازمندی‌ها[1] مجموعه‌ای از موارد کاربری و مشخصات تکمیلی که باید جزئیات آن بیان گردد و سازگاری آن با سایر فرآورده‌ها احراز شود، انتساب پیدا می‌کند. همچنین طراح واسط کاربر[2] به صورت موازی با تعیین‌کنندهٔ نیازمندی‌ها شروع به کار می‌نماید تا واسط‌های اولیه کاربر را تولید نماید. در اغلب پروژه‌ها بین این دو نقش، هماهنگی نزدیکی وجود دارد. بدین معنی که تعیین‌کننده نیازمندی‌ها مجموعه‌ای از نیازمندی‌ها را مشخص می‌نماید و طراح واسط کاربر شکلی از واسط کاربر تهیه می‌کند تا تعیین‌کننده نیازمندی‌ها با بازبینی آنها به‌همراه ذینفعان بتوانند نیازمندی‌های خود را به‌صورت بهتر بیان کنند.

نقش دیگری که در نظم نیازمندی‌ها فعالیت می‌کند، معمار سیستم است. بیشتر کار معمار سیستم در تکرارهای اولیه است که با همکاری تحلیلگر سیستم و تعیین‌کنندهٔ نیازمندی‌ها یکپارچگی و درستی موارد کاربری مهم از دیدگاه معماری مورد بررسی قرار می‌گیرند. نقش بازبین فنی توسط همه کسانی (ذینفعان) که وظیفه آنها بررسی جامعیت و صحت نیازمندی‌ها و اطمینان از درک درست آنها است، ایفا می‌شود.



شکل ۷-۳- گردش‌کار نظم نیازمندی‌ها

[1] Requirement Specifier
[2] User-Interface Designer

۷-۲-۵- پشتیبانی ابزار از نظم نیازمندی‌ها

ابزارهایی که می‌توان در نظم نیازمندی‌ها از آنها استفاده نمود، عبارتند از:

- ابزار Rose برای مدلسازی تصویری با استفاده از نمادهای UML
- RequisitePro در استخراج، ثبت و کنترل تغییرات نیازمندی‌ها کمک نموده و آنها را در یک پایگاه داده نگهداری می‌نماید
- ابزار SoDA برای خودکارسازی فرآیند تولید مستندات

۷-۳- نظم تحلیل و طراحی

هدف عمده این نظم تبدیل نیازمندی‌ها به خصوصیات طراحی است که چگونگی پیاده‌سازی سیستم مورد نظر را شرح می‌دهد. برای این کار نیاز است که نیازمندی‌های سیستم درک شده و سپس با انتخاب بهترین استراتژی پیاده‌سازی به طراحی سیستم تبدیل شود. در واقع، تبدیل نیازمندی‌ها به طراحی با استفاده از موارد کاربری و نیازمندی‌های غیروظیفه‌مندی تعیین شده، انجام می‌شود و این دو طراحی نرم‌افزار را راهبری می‌کنند.

مدل تحلیل یک مدل ایده‌آل از سیستم بوده که در آن نیازمندی‌های غیروظیفه‌مندی و محدودیت‌های پیاده‌سازی نادیده گرفته می‌شود. به‌عبارت بهتر در مدل تحلیل سیستم، مواردی که تحلیل سیستم را دچار محدودیت می‌کنند و مانع از تحلیل مناسب سیستم می‌شوند، نادیده گرفته می‌شوند از جمله این محدودیت‌ها می‌توان به مشکلات اجرایی انجام عملیات به‌صورت دستی اشاره نمود که در صورت تاثیر در مدل تحلیل، سبب پیچیدگی غیرضروری نرم‌افزار می‌شود.

در توسعه مدل تحلیل تا حد ممکن از مقوله‌بندی استفاده می‌شود بدین ترتیب که تا آنجا که ممکن است کلاس‌های دسته‌بندی می‌شوند. این دسته‌بندی کمک می‌کند تا درک بهتری از کلاس‌های تحلیل به‌وجود آید. همانطور که در فصل ۴ در مورد مقوله‌بندی اشاره شد، مقوله‌بندی می‌تواند در متدولوژی‌های مختلف متفاوت باشد. در RUP که یکی از انواعو متدولوژی‌های USDP است، کلاس‌ها به سه دسته مرزی، کنترلی و موجودیت مقوله‌بندی می‌شوند. استفاده از این دسته‌بندی در تحلیل و همچنین طراحی کمک می‌کند که دیدی یکپارچه‌ای از مدل تحلیل بوجود آید.

پس از توسعه مدل تحلیل در این نظم، مدل طراحی نیز ایجاد می‌شود. در مدل طراحی سعی می‌شود تا مدل ایده‌آلی که در تحلیل مد نظر قرار داشته به مدلی واقعی از سیستم تبدیل شود. این مدل واقعی که مدل طراحی نام دارد حاوی محدودیت‌های پیاده‌سازی، نیازمندی‌های غیروظیفه‌مندی و تجربیات ذینفعان پروژه، بخصوص طراح (طراحان) است. مدل تحلیل و طراحی به‌عنوان مرجعی برای برنامه‌نویسان مورد استفاده قرار می‌گیرند و کمک می‌کنند تا برنامه‌نویسان درک یکسانی از نرم‌افزاری که می‌بایست توسعه یابد را بدست آورند.

۷-۳-۱- مهمترین نقش‌های نظم تحلیل و طراحی

در نظم تحلیل و طراحی پنج نقش عمده وجود دارند که اغلب فرآورده‌ها و محصولات این نظم را تولید می‌کنند، این نقش‌ها عبارتند از:

- معمار نرم‌افزار
  - هماهنگی فعالیت‌های فنی و تولید فرآورده‌ها در طول پروژه بعلاوه بدست آوردن ساختار کلی هر دید معماری[1] از عمده‌ترین وظایف این نقش است.
- طراح
  - تشخیص مسئولیت‌ها، اعمال، صفات و روابط حاکم بین کلاس‌ها بعلاوه انجام تغییرات لازم برای اینکه کلاس‌ها مناسب پیاده‌سازی شوند از عمده‌ترین وظایف این نقش است.
- طراح پایگاه‌داده[2]
  - طراحی پایگاه‌های داده‌ای مورد نیاز وظیفه طراح پایگاه‌داده‌هاست.
- طراح واسط کاربر[3]
  - طراحی واسط کاربر مورد نیاز وظیفه طراح واسط کاربر است.
- طراح آزمایش[4]
  - تعریف روش آزمایش مورد نیاز برای آزمایش نرم‌افزار وظیفه طراح آزمایش است.

[1] Architectural View
[2] Database Designer
[3] User-Interface Designer
[4] Test Designer

۷-۳-۲- مهمترین فرآورده‌های نظم تحلیل و طراحی

مهمترین فرآورده‌های نظم تحلیل و طراحی عبارتند از:

- مدل تحلیل
  - هدف از مدل تحلیل، عینیت بخشیدن به موارد کاربری به صورت تجریدی از کلاس‌های طراحی است.
- مدل طراحی
  - هدف از مدل طراحی ایجاد طرح کلی از سیستم[1] است که در آن کلاس‌های کلیدی سیستم و روابط بین این کلاس‌ها مشخص شده‌اند. این مدل همانند سایر مدل‌ها در هر تکرار بهبود می‌یابد و حاوی جزئیات دقیق‌تری از کلاس‌ها می‌شود.
- مدل داده
  - هدف از مدل داده‌ای، توصیف ساختار منطقی و فیزیکی داده‌های پایا[2] سیستم است.
- مدل استقرار
  - هدف از مدل استقرار نمایش نگاشت اشیاء و مولفه‌ها به سخت‌افزار مورد استفاده از جمله گره‌های شبکه، پردازنده‌های مورد استفاده در سیستم‌های چند پردازنده است.
- مستند معماری نرم‌افزار[3]
  - هدف از مستند معماری نرم‌افزار بیان دیدهای گوناگون معماری که از جمله مهترین دیدها می‌توان به Use-case View, Logical View, Implementation View, Deployment View, Process View, Data View اشاره نمود.

۷-۳-۳- گردش‌کار نظم تحلیل و طراحی

گردش‌کار نظم تحلیل و طراحی در شکل ۷-۴ ارائه شده است. نظم تحلیل و طراحی دارای دو گردش‌کار است که یکی سطح بالا و برای ایجاد معماری و دیگری سطح پایین برای طراحی جزئیات است.

[1] System Blueprint
[2] Persistent
[3] Software Architecture Document

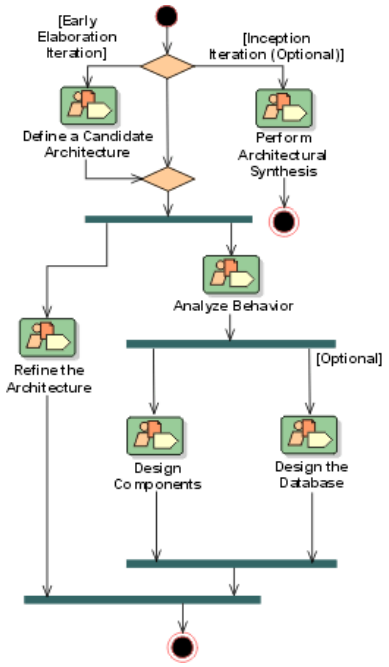
۱) ایجاد معماری سیستم (طراحی سطح بالا)

- ایجاد معماری شامل دو بخش تحلیل معماری و طراحی معماری است
- در تحلیل معماری، چگونگی سازماندهی معماری مشخص می‌شود.
- معمار در بالاترین سطح، الگوهای اساسی معماری، مکانیزم‌های کلیدی و راهنمائی‌های مدلسازی سیستم (لایه‌های سیستم، روش سازماندهی زیرسیستم‌ها و استراتژی استفاده مجدد) را مشخص می‌نماید
- پس از انجام تحلیل موارد کاربری به اندازه کافی، کلاس‌های حاصل برای شروع طراحی معماری بکار گرفته می‌شوند
- در طراحی معماری، کلاس‌های اصلی شناسائی شده و در بسته‌ها و زیرسیستم‌ها گروه‌بندی شده و بالاخره زیرسیستم‌ها در لایه‌ها سازمان‌دهی می‌شوند
- با استفاده از دیدهای دیگر معماری (البته در صورت لزوم) طراحی معماری ادامه می‌یابد

۲) طراحی جزئیات

- با استفاده از نتایج تحلیل موارد کاربری، که در آن عینیت بخشیدن به موارد کاربری انجام شده است و با استفاده از معماری، طراح نرم‌افزار مشخصات کلاس‌ها و روابط موجود بین آنها را شناسائی می‌نماید و بدین صورت کلاس‌ها کامل‌تر خواهند شد
- سپس طراح سعی به تشخیص و پالایش واسط‌های زیرسیستم‌ها که بوسیله آنها رفتار زیرسیستم‌ها نمایان می‌گردد، می‌نماید که سبب می‌شود موارد کاربری به‌صورتی کامل طراحی شوند
- هر عینیت بخشیدن موارد کاربری بوسیله تعیین اعمال روی کلاس‌ها یا زیرسیستم‌ها به صورت کامل مشخص می‌شود
- زمانیکه سیستم مورد نظر به پایگاه داده‌ها نیاز داشته باشد وظیفه طراح پایگاه، نگاشت کلاس‌های پایا[1] به جداول پایگاه‌داده‌ها است

[1] Persistant Classes



شکل ۷-۴- گردش‌کار نظم تحلیل و طراحی

۷-۳-۴- پشتیبانی ابزار از نظم تحلیل و طراحی

ابزارهایی که می‌توان در نظم تحلیل و طراحی از آنها استفاده نمود، عبارتند از:

- Rose برای تحلیل و طراحی با استفاده از UML
  - Rose از Round-Trip Engineering پشتیبانی می‌کند. (Round-Trip Engineering به معنی حفظ هماهنگی بین کد تولید شده و مدل نمایش دهنده این کد. به‌عبارت بهتر با تغییر کد، مدل به‌روزرسانی می‌گردد و بالعکس)
- RUP ابزارهای راهنما برای توضیح استفاده از UML و Rose در فازهای مختلف پروژه را ارائه می‌نماید

۷-۴- نظم پیاده‌سازی

این نظم در مورد نمونه‌ها و چگونگی افزایش تدریجی در مجتمع‌سازی سیستم صحبت می‌کند. دو مفهومی که در این نظم اغلب مورد استفاده قرار می‌گیرد، ساخته[1] و یکپارچه‌سازی[1] هستند. ساخته،

[1] Build

نسخه‌ای کامل یا جزئی از سیستم که قابل استفاده و اجرا بوده (کد اجرائی) و زیرمجموعه‌ای از توانایی‌های محصول نهائی را به معرض نمایش می‌گذارد. یکپارچه‌سازی فعالیتی است که در آن مؤلفه‌های جداگانهٔ نرم‌افزار با هم ترکیب می‌شوند و کل سیستم به عنوان یک واحد را بوجود می‌آورند. مجموعه‌ای از ساخته‌ها با یکدیگر یکپارچه شده و یک ساخته کامل از نرم‌افزار را ایجاد می‌نمایند.

نظم پیاده‌سازی چهار هدف عمده را دنبال می‌کند که عبارتند از:

- تعیین ساختار کد به صورت زیرسیستم‌هایی که در تعدادی از لایه‌ها سازمان‌دهی شده‌اند
- پیاده‌سازی کلاس‌ها و اشیاء به صورت مؤلفه‌ها
- آزمایش مؤلفه‌های تولید شده
- یکپارچه‌سازی مؤلفه‌ها و کد تولید شده

۷-۴-۱- مهمترین نقش‌های نظم پیاده‌سازی

در نظم پیاده‌سازی سه نقش عمده وجود دارند که اغلب فرآورده‌ها و محصولات این نظم را تولید می‌کنند، این نقش‌ها عبارتند از:

- پیاده‌ساز[۲]
  - تولید، توسعه و آزمایش مؤلفه‌ها وظیفه نقش پیاده‌ساز است
- یکپارچه‌ساز[۳]
  - برنامه‌ریزی و انجام یکپارچه‌سازی برای ایجاد ساخته‌ها وظیفه نقش یکپارچه‌ساز است
- معمار نرم‌افزار
  - تبیین معماری نرم‌افزار که شامل تعیین و مستندسازی تصمیمات فنی کلیدی در مورد نیازمندی‌ها، طراحی، پیاده‌سازی و استقرار سیستم با استفاده از دیدهای مختلف وظیفه اصلی معمار نرم‌افزار در این نظم است

۷-٤-۲- مهمترین فرآورده‌های نظم پیاده‌سازی

مهمترین فرآورده‌های نظم پیاده‌سازی عبارتند از:

---

[1] Integration
[2] Implementer
[3] Integrator

- مدل پیاده‌سازی
  - مجموعه‌ای از مؤلفه‌ها و زیر سیستم‌های حاوی آنهاست
- زیرسیستم‌های پیاده‌سازی
  - مجموعه‌ای از مؤلفه‌ها و زیرسیستم‌ها که در سازمان‌دهی مدل پیاده‌سازی با تقسیم آن به قطعات کوچکتر، مورد استفاده قرار می‌گیرند
- مؤلفه‌ها
  - قطعه‌ای از کد نرم‌افزاری (دودویی یا اجرائی) که دسترسی به سرویس‌های آن تنها از طریق یک واسط خوش‌تعریف انجام می‌شود
- طرح یکپارچه‌سازی[1]
  - ترتیب پیاده‌سازی مؤلفه‌ها و زیرسیستم‌ها را مشخص می‌نماید

۷-۴-۳- پشتیبانی ابزار از نظم پیاده‌سازی

ابزارهایی که می‌توان در نظم پیاده‌سازی از آنها استفاده نمود، عبارتند از:

- Rose برای تولید کد به زبان‌های متفاوت از جمله: C++، Visual Basic، Java و SQL
- Purify برای کشف خطاهای زمان اجرا[2]
- Quantify برای تشخیص کارائی و سرعت بخش‌های متفاوت برنامه
- ClearQuest برای مدیریت تغییرات در نرم‌افزار

۷-۵- نظم آزمایش

محور اصلی این نظم بررسی کیفیت نرم‌افزار از دو دیدگاه کیفیت محصول و کیفیت فرآیند تولید است. کیفیت محصول در مورد تولید نرم‌افزاری با کیفیت مطلوب است و کیفیت فرآیند تولید در مورد اجرای درست فرآیند توسعه نرم‌افزار است. در RUP کیفیت وظیفه همه افراد توسعه‌دهنده است و هر فرد با اجرای درست نقش خود می‌تواند هم در کیفیت محصول و هم در کیفیت فرآیند تولید نقش موثر داشته باشد.

[1] Integration Build Plan
[2] Run-Time Errors

اما در کنار کیفیت که به جنبه‌هایی چون قابلیت اعتماد، عملکرد، کارایی برنامه، کارایی سیستم می‌پردازد، محورهای دیگری چون مراحل آزمایش (نظیر آزمایش واحدها، آزمایش یکپارچگی، آزمایش سیستم و آزمایش پذیرش سیستم) و نوع آزمایش (آزمایش پیکربندی، آزمایش نصب) نیز در این نظم وجود دارند که در روند انجام این نظم نقش اساسی ایفا می‌کنند.

۷-۵-۱- مهمترین نقش‌های نظم آزمایش

در نظم آزمایش دو نقش عمده وجود دارند که عبارتند از:

- طراح آزمایش[1]
  - برنامه‌ریزی، طراحی، پیاده‌سازی و ارزیابی آزمایش‌ها از عمده‌ترین وظایف طراح آزمایش است
- آزمایش‌کننده سیستم[2]
  - این نقش، مسئولیت اجرای آزمایش‌های سیستم را به عهده دارد

۷-۵-۲- مهمترین فرآورده‌های نظم آزمایش

مهمترین فرآورده‌های نظم آزمایش عبارتند از:

- برنامه آزمایش[3]
  - این برنامه شامل اطلاعاتی درباره اهداف آزمایش، استراتژی آزمایش کردن و منابع مورد نیاز برای پیاده‌سازی و اجرای آزمایش‌ها می‌باشد
- مدل آزمایش
  - این مدل شامل موارد آزمایش، روال‌های آزمایش و دستورالعمل‌های آزمایش[4] است

۷-۵-۳- گردش‌کار نظم آزمایش

گردش‌کار نظم آزمایش در شکل ۷-۵ ارائه شده است. فعالیت‌های انجام شده در نظم آزمایش را می‌توان به پنج مرحله تقسیم نمود که در ادامه به بررسی هر یک خواهیم پرداخت.

---

[1] Test Designer
[2] System Tester
[3] Test Plan
[4] Test Scripts

۱) برنامه‌ریزی

- طراح آزمایش نیازمندی‌ها، منابع مورد نیاز و استراتژی‌های یک آزمایش را شناسائی نموده و در مستند برنامه‌ریزی، آنها را مستند می‌نماید

۲) طراحی

- طراح آزمایش هدف آزمایش را تحلیل نموده و مدل آزمایش را ایجاد می‌نماید

۳) پیاده‌سازی

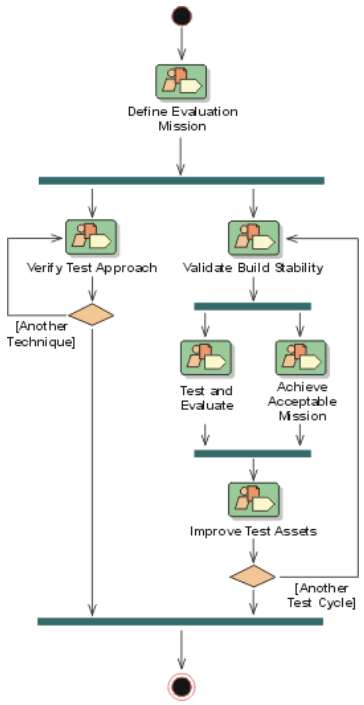
- طراح آزمایش روال‌های آزمایش را پیاده‌سازی می‌نماید. این فعالیت منجر به ایجاد دستورالعمل‌های آزمایش است

۴) اجرا

- آزمایش کننده، آزمایش‌ها را روی هدف اجرا می‌نماید و با بازبینی نتایج از صحت اجرای آزمایش‌ها مطمئن می‌گردد و هرگونه کاستی را مستند می‌سازد

۵) ارزیابی

- طراح آزمایش نتایج آزمایش‌ها را ارزیابی نموده و کیفیت کد تولید شده را بررسی می‌نماید



شکل ۷-۵- گردش کار نظم آزمایش

۷-۵-۴- پشتیبانی ابزار از نظم آزمایش

ابزارهایی که می‌توان در نظم آزمایش از آنها استفاده نمود، عبارتند از:

- Test Manager امکان مدیریت و کنترل تمام فعالیت‌های آزمایش را می‌دهد
- Rational TestFactory امکان ایجاد دستورالعمل‌های آزمایش را می‌دهد
- Rational Robot امکان ایجاد، اصلاح و اجرای آزمایش‌های تابعی خودکار[1]

۷-۶- نظم استقرار

این نظم دربردارنده فعالیت‌های لازم برای عملیاتی و آماده نمودن نرم‌افزار برای کار در محیط کاربران نهایی است. این نظم حاوی سه حالت متفاوت برای استقرار محصول در محیط کاربران است:

- نصب سفارشی[2]
  - نرم‌افزار به‌صورت سفارشی برای کاربر تهیه شده است و امکان نصب به‌صورت سفارشی را به کاربر می‌دهد
- پیشنهاد استفاده از محصول Shrink Wrap
  - نرم‌افزار به‌صورت بسته نرم‌افزاری ارائه می‌شود و کاربران با خرید نرم‌افزار از فروشگاه آن را نصب می‌نمایند
- دستیابی به نرم‌افزار از طریق اینترنت
  - کاربر نرم‌افزار را از طریق اینترنت دونلود می‌کند و نصب را انجام می‌دهد

۷-۶-۱- مهمترین نقش‌های نظم استقرار

در نظم استقرار پنج نقش عمده وجود دارند که عملیات مربوط به این نظم را انجام می‌دهند:

- مدیر استقرار[3]
  - وظیفه مدیر استقرار برنامه‌ریزی انتقال محصول به محیط کاربران و تضمین اینکه برنامه بدرستی انجام می‌شود، است

---

[1] automated functional tests
[2] Custom Install
[3] Deployment Manager

- پیاده‌ساز
  - وظیفه پیاده‌ساز در این نظم، توسعه و آزمایش مولفه‌ها با استفاده از استانداردهای پذیرفته شده در پروژه است
- توسعه‌دهنده دروس[1]
  - وظیفه توسعه‌دهنده دروس، توسعه مواد آموزشی مورد نیاز کاربران است
- مدیر پیکربندی
  - فراهم نمودن زیرساخت و محیط مدیریت پیکربندی برای تیم توسعه وظیفه اصلی مدیر پیکربندی است
- نویسنده فنی
  - نویسنده فنی در این نظم، مسئول تولید مواد مورد نیاز همانند راهنمای کاربران برای کاربران نهایی است

۷-٦-۲- مهمترین فرآورده‌های نظم استقرار

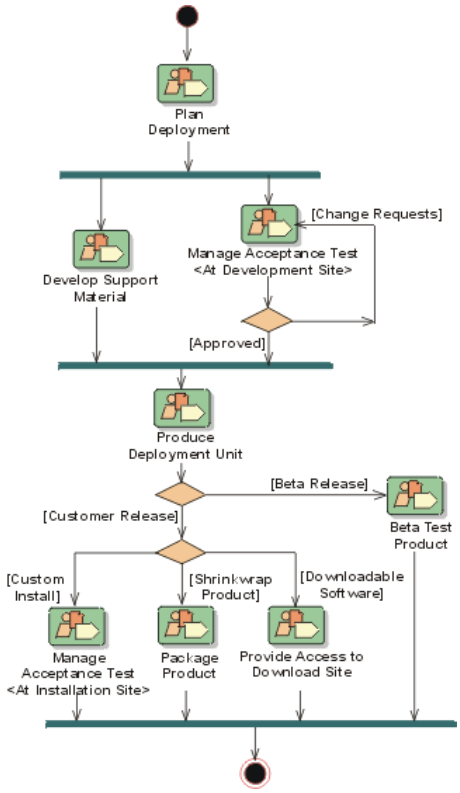
مهمترین فرآورده‌های نظم استقرار عبارتند از:

- طرح استقرار[2]
  - هدف از طرح استقرار، توصیف مجموعه فعالیت‌های لازم برای نصب و آزمایش محصول تا براحتی به محیط کاربران منتقل شود
- محصول
  - محصول قابل عرضه به بازار از اصلی‌ترین فرآورده‌های این نظم و در واقع پروژه است
- مواد آموزشی[3]
  - هدف از تهیه این مواد کمک به کاربران برای یادگیری بهتر و سریعتر محصول است

[1] Course Developer
[2] Deployment Plan
[3] Training Materials

۷-۶-۳- گردش کار نظم استقرار

گردش کار نظم استقرار در شکل ۷-۶ ارائه شده است. پس از توسعه نرم‌افزار و انجام آزمایش پذیرش، نرم‌افزار به یکی از سه طریق مطرح شده می‌تواند نصب شود که باید برای هر یک از روش‌ها، مستندات و مواد آموزشی مورد نیاز فراهم شود.



شکل ۷-۶- گردش کار نظم استقرار

۷-۷- نظم مدیریت پروژه

مدیریت پروژه نرم‌افزاری، هنر متوازن ساختن اهداف رقابتی، مدیریت خطرات و غلبه بر محدودیت‌ها برای تحویل موفقیت‌آمیز محصولی است که هم نیازهای مشتریان (کسانی که برای سیستم پول می‌پردازند) و هم نیازهای کاربران را برآورده کند. این حقیقت که پروژه‌های بسیار کمی هستند که واقعاً موفقیت آمیزند برای توضیح سخت بودن این کار کافی است. اهداف نظم مدیریت پروژه عبارتند از:

- فراهم کردن چارچوبی برای مدیریت پروژه‌های صرفاً نرم‌افزاری
- فراهم کردن رهنمودهای عملی برای طرح‌ریزی، تعیین نیروی انسانی، اجرا و نظارت بر پروژه‌ها

- فراهم کردن چارچوبی برای مدیریت ریسک

نظم مدیریت پروژه در RUP تلاش نمی‌کند همه جنبه‌های مدیریت پروژه را پوشش دهد. در واقع این نظم جنبه‌های زیر را پوشش نمی‌دهد:

- جنبه‌های کاریابی، آموزش و... در مدیریت نیروی انسانی
- مدیریت بودجه
- مدیریت قراردادها با مشتریان و توسعه‌دهندگان

نظم مدیریت پروژه تنها جنبه‌های ذیل را پوشش می‌دهد:

- مدیریت خطرات
- برنامه‌ریزی «پروژه تکراری»[1]
- مراقبت از روند پیشرفت «پروژه تکراری» و تعیین معیارهای لازم برای این کار

۷-۷-۱- مهمترین نقش‌های نظم مدیریت پروژه

در نظم مدیریت پروژه دو نقش اصلی وجود دارند که عملیات مربوط به این نظم را انجام می‌دهند:

- مدیر پروژه
  - وظیفه مدیر پروژه، برنامه‌ریزی، مدیریت و تخصیص منابع؛ شکل‌دهی به اولویت‌ها؛ هماهنگی تعاملات بین مشتریان و کاربران و در جریان بودن از روند پروژه است
- بازبینی‌کننده مدیریتی[2]
  - بازبینی‌کننده مدیریتی مسئول ارزیابی پروژه و فرآورده‌های مدیریت پروژه است

۷-۷-۲- مهمترین فرآورده‌های نظم مدیریت پروژه

مهمترین فرآورده‌های نظم مدیریت پروژه عبارتند از:

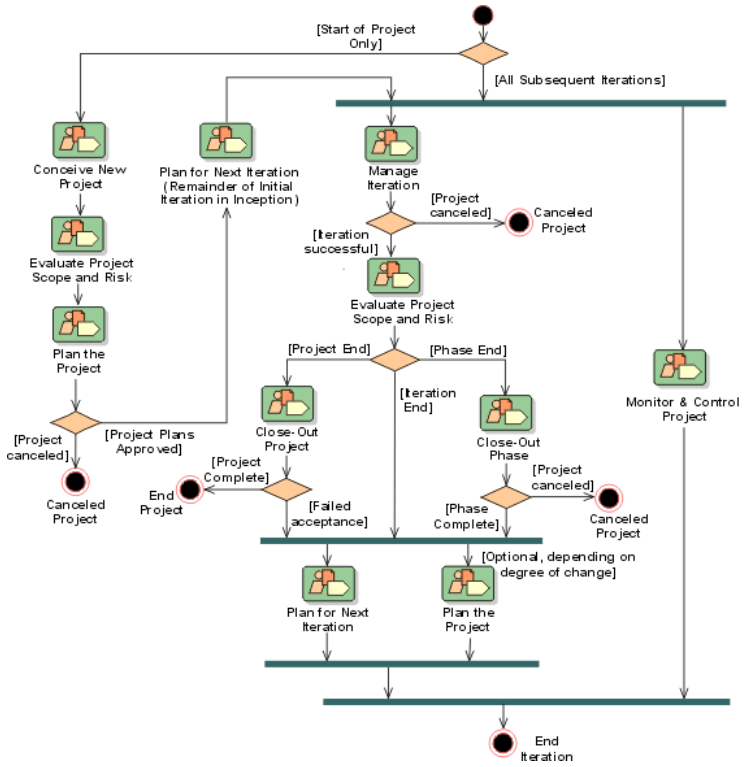
- طرح مدیریت خطر[3]
  - هدف از طرح مدیریت خطر، بیان چگونگی مدیریت خطرات مربوط به پروژه در زمان اجرای پروژه است

[1] Iterative Project
[2] Management Reviewer
[3] Risk Management Plan

- طرح پذیرش محصول[1]
  - هدف از طرح پذیرش محصول، بیان این است که چگونه مشتری محصولات پروژه را ارزیابی می‌کند؛ چه ملاک‌ها و معیارهایی را در نظر می‌گیرد
- طرح توسعه نرم‌افزار[2]
  - این طرح به بیان مجموعه‌ای یکپارچه و کامل از تمام اطلاعات مربوط به مدیریت پروژه می‌پردازد.

۷-۷-۳- گردش‌کار نظم مدیریت پروژه

گردش‌کار نظم مدیریت پروژه در شکل ۷-۷ ارائه شده است. این نظم از نظر تعداد عملیات بیشترین عملیات را در میان نظم‌ها به خود اختصاص داده است. در اغلب تکرارها فعالیت‌های این نظم به‌صورت منظم اجرا می‌شود تا خطرات پروژه در زمان مناسب مدیریت شوند.



شکل ۷-۷- گردش‌کار نظم مدیریت پروژه

[1] Product Acceptance Plan
[2] Software Development Plan

۷-۸- نظم مدیریت پیکربندی

نظم مدیریت پیکربندی بر کنترل تغییرات و حفظ یکپارچگی و سازگاری بین فرآورده‌های یک پروژه نرم‌افزاری تمرکز دارد. این نظم می‌تواند با استفاده از یک سیستم مدیریت پیکربندی انجام پذیرد. یک سیستم مدیریت پیکربندی[1]، به مجموعه فرآیندها، روش‌ها و ابزارهایی که برای مدیریت پیکربندی و مدیریت درخواست تغییر[2] استفاده می‌شوند، گفته می‌شود. سیستم مدیریت پیکربندی باید توانایی مقابله با سه مشکلات زیر را داشته باشد:

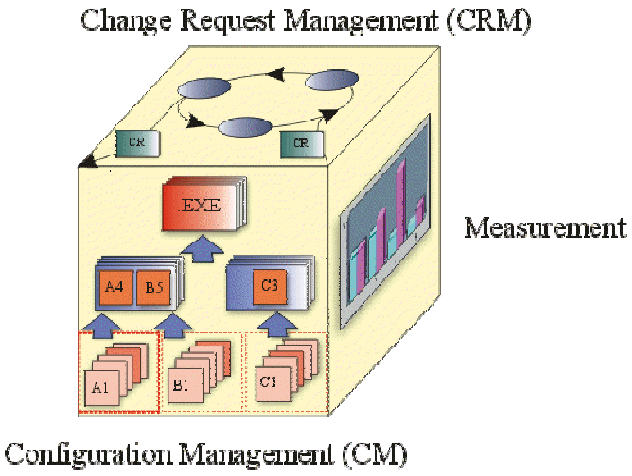
- به‌روزرسانی همزمان[3]
- اطلاع‌رسانی محدود[4]
- نسخه‌های متعدد[5]

این سه مشکل، چالش‌های عمده هر سیستم مدیریت پیکربندی هستند که سیستم با به‌گونه‌ای با آنها مقابله می‌کند. نکته بسیار با اهمیت در این مشکلات، تاثیر آنها بر یکدیگر است. به‌عنوان نمونه به‌روزرسانی همزمان بر داشتن نسخه‌های متعدد تاثیر مثبت دارد بدین معنی که با افزایش به‌روزرسانی همزمان نیاز بیشتری به نسخه‌های متعدد وجود خواهد داشت.

ارتباط بین مدیریت درخواست‌های تغییرات، حسابداری وضعیت پیکربندی[6] و مدیریت پیکربندی در شکل ۷-۸ نشان داده شده است. مدیریت درخواست‌های تغییرات اشاره به ساختار سازمانی دارد که هزینه، زمانبندی و تاثیر تغییر درخواست شده را بر محصول ارزیابی می‌کند. حسابداری وضعیت پیکربندی برای توصیف «حالت» محصول براساس نوع، تعداد، نرخ خطاهای یافته شده و اصلاح شده در زمان توسعه نرم‌افزار مورد استفاده قرار می‌گیرد. معیارهایی که در با استفاده از مقادیر این حسابداری حاصل می‌شود می‌توانند در ارزیابی پروژه مورد استفاده قرار گیرد. در هر سیستم مدیریت پیکربندی می بایست هر درخواست تغییر و نتایج حاصل از تغییر اعم از تغییرات انجام شده در نرم‌افزار یا در درخواست

[1] Configuration Management (CM) System
[2] Change Request
[3] Simultaneous Update
[4] Limited Notification
[5] Multiple Versions
[6] Configuration Status Accounting (Measurement)

نگهداری شود. برخی اوقات نیاز است که به نسخه‌ای قبلی از محصول مراجعه شود، به همین دلیل نیاز است تا تمام مشخصات نسخه‌های مختلف نرم‌افزار نگهداری شود.



شکل ۷-۸- ارتباط مدیریت پیکربندی با مدیریت درخواست تغییرات و معیارها

۷-۸-۱- مهمترین نقش‌های نظم مدیریت پیکربندی

در نظم مدیریت پیکربندی دو نقش اصلی وجود دارند که عملیات مربوط به این نظم را انجام می‌دهند:

- مدیر پیکربندی
  - مسئولیت تهیه زیرساخت و محیط مدیریت پیکربندی برای تیم توسعه محصول برعهده مدیر پیکربندی است
- مدیر کنترل تغییرات[1]
  - مسئولیت کنترل فرآیند تغییرات برعهده مدیر کنترل تغییرات است

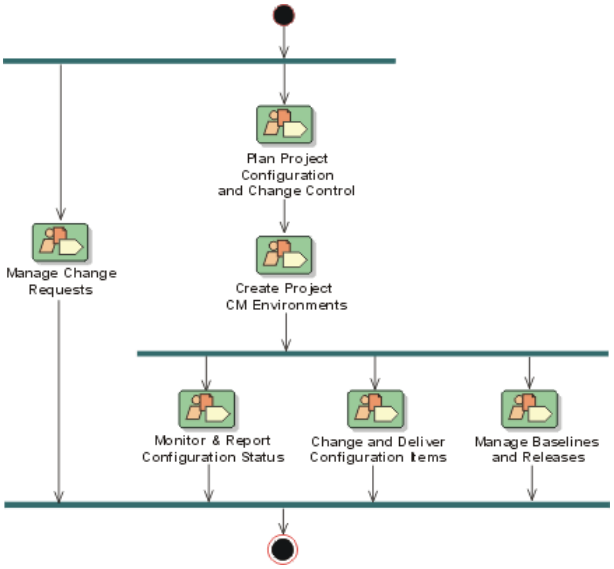
۷-۸-۲- مهمترین فرآورده نظم مدیریت پیکربندی

مهمترین فرآورده نظم مدیریت پیکربندی طرح مدیریت تغییرات[2] است. هدف این طرح، توصیف تمام فعالیت‌های مدیریت کنترل تغییرات و پیکربندی که در طول توسعه محصول انجام می‌شود، است.

[1] Change Control Manager
[2] Configuration Management Plan

۷-۸-۳- گردش کار نظم مدیریت پیکربندی

گردش کار نظم مدیریت پیکربندی در شکل ۷-۹ ارائه شده است.



شکل ۷-۹- گردش کار نظم مدیریت پیکربندی

۷-۸-۴- پشتیبانی ابزار از نظم مدیریت پیکربندی

ابزارهایی که می‌توان در نظم مدیریت پیکربندی از آنها استفاده نمود، عبارتند از:

- ClearQuest برای مدیریت پیکربندی در نرم‌افزار
- ClearCase LT برای مدیریت پیکربندی در نرم‌افزار برای تیم‌های کوچک

۷-۹- نظم محیط

این نظم دربردارنده فعالیت‌هایی است که برای پیکربندی (اصلاح یا تنظیم ویژه) RUP برای یک پروژه ضروری هستند. از جمله فعالیت‌های اصلی که در این نظم انجام می‌شود:

- تنظیم ویژه RUP
- تهیه راهنمائی‌های لازم برای تولید بیشتر فرآورده‌های پروژه (استانداردهای سازمان)

پیکربندی RUP اهمیت ویژه‌ای دارد. چه‌بسا پروژه‌هایی بعلت تولید محصولات غیرضروری یا محصولات نادرست (عمل پیکربندی RUP به صورت صحیحی انجام نشده است) موفقیت‌آمیز نبوده یا

نتیجهٔ مورد انتظار از آن بدست نیامده است. در هر پروژه نیاز است تا متدولوژی RUP از نظر محصولاتی که باید تولید شود و فرآیند اجرا سفارشی شود.

۷-۹-۱- مهمترین نقش‌های نظم محیط

در نظم محیط دو نقش اصلی وجود دارند که عملیات مربوط به این نظم را انجام می‌دهند:

- مهندس فرآیند
  - وظیفه مهندس فرآیند، تجهیز تیم پروژه با استفاده از یک فرآیند کارآمد؛ تضمین اینکه افراد می‌توانند براحتی کار خود را انجام دهند، است
- متخصص ابزار[1]
  - وظیفه متخصص ابزار حمایت از ابزارهای مورد استفاده در پروژه است

۷-۹-۲- مهمترین فرآورده نظم محیط

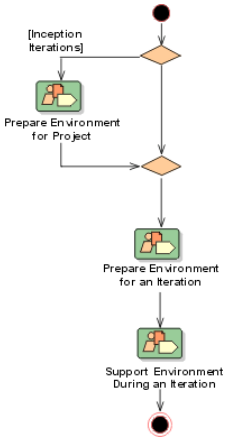
مهمترین فرآورده‌های نظم محیط عبارتند از:

- فرآیند توسعه
  - هدف از فرآیند توسعه، پیکربندی RUP برای حمایت از نیازهای پروژه است
- راهنمای خاص پروژه[2]
  - هدف از راهنمای خاص پروژه، راهنمایی برای انجام مجموعه فعالیت‌های پروژه است

۷-۹-۳- گردش‌کار نظم محیط

گردش‌کار نظم محیط در شکل ۷-۱۰ ارائه شده است.

[1] Tool Specialist
[2] Project Specific Guidelines



شکل ۷-۱۰-گردش کار نظم محیط

۷-۹-۴- پشتیبانی ابزار از نظم محیط

ابزار Process Workbench می‌تواند به‌عنوان تسهیل کنندهٔ کار مهندس فرآیند[1] در پیکربندی RUP مورد استفاده قرار گیرد.

[1] Process Engineer

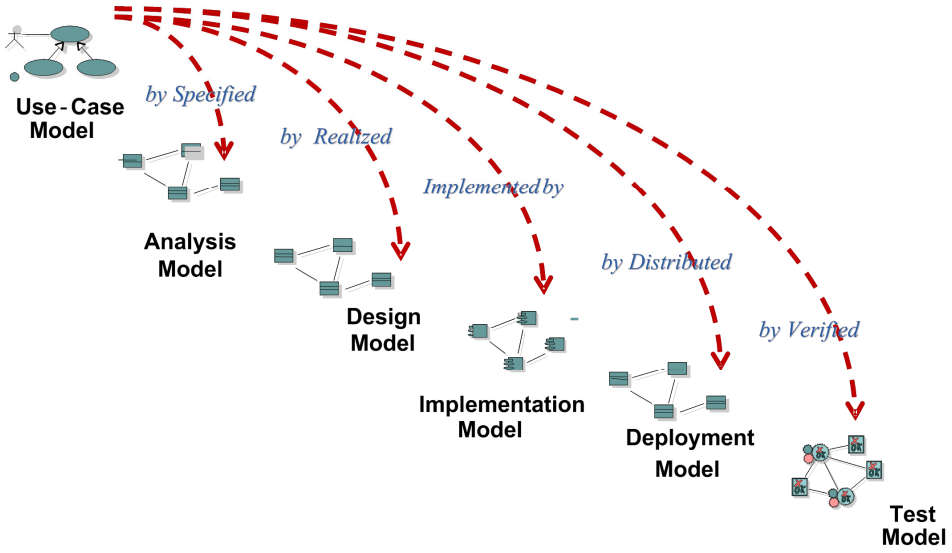
۸- مدلسازی موارد کاربری[1]

۸-۱- مقدمه

مدتها در فرآیندهای شی‌گرا و غیر شی‌گرا از سناریوها برای کمک به فهم نیازمندی‌های سیستم استفاده می‌شد. اما این سناریوها با اینکه همیشه تولید می‌شدند بندرت مستندسازی و نگهداری می‌شدند، تا زمانیکه Ivar Jacobson این وضعیت را با ارائه متدولوژی Objectory خود دگرگون نمود. او در متدولوژی خود نام این سناریوها را مورد کاربری[2] گذاشت و آنرا محور اصلی کار خود قرار داد. بتدریج جامعه شی‌گرا به استفاده از موارد کاربری پرداخت و به اهمیت آن پی برد و آنرا بعنوان ابزار قوی معرفی نمود.

ویژگی بارز این روش این است که مسئله جمع‌آوری نیازمندی‌ها[3] را از دیدگاه غیرتکنیکی بررسی می‌کرد. در واقع دو نگاه به یک سیستم وجود دارند: یکی نگاه به سیستم از بیرون آن (غیرتکنیکی) و دیگری نگاه فردی که می‌خواهد سیستم را بسازد (تکنیکی). این دو دیدگاه کاملاً با یکدیگر متفاوتند. موارد کاربری نقشی اساسی در توسعه سایر مدل‌های موجود در توسعه نرم‌افزار دارد. به‌عنوان نمونه شکل ۸-۱ ارتباط مدل موارد کاربری را با سایر مدل‌های مورد استفاده در فرآیند توسعه نرم‌افزار RUP را نشان می‌دهد. مدل موارد کاربری پایه‌ای برای ایجاد مدل‌های تحلیل ایجاد می‌نماید. همچنین مدل‌های طراحی با استفاده از شناختی که در مدل‌های تحلیل کسب شده و با استفاده از مدل موارد کاربری نمایش داده می‌شوند، تهیه می‌شوند. سایر مدل‌ها همچمون مدل آزمایش، مدل استقرار و مدل پیاده‌سازی نیز مستقیماً از مدل موارد کاربری تاثیر می‌پذیرند و می‌توان گفت استفاده از موارد کاربری مناسب در مدل موارد کاربری تاثیر مستقیمی در نتایج پروژه نرم‌افزاری می‌گذارد.

[1] Use-Case
[2] Use-Case(UC)
[3] Requirements Gathering



شکل ۸-۱- راهبری بر مبنای موارد کاربری

۸-۲- مفاهیم اساسی مدلسازی موارد کاربری

با درک اهمیت موارد کاربری می‌توان تعریف مورد کاربری را به صورت زیر ارائه نمود:

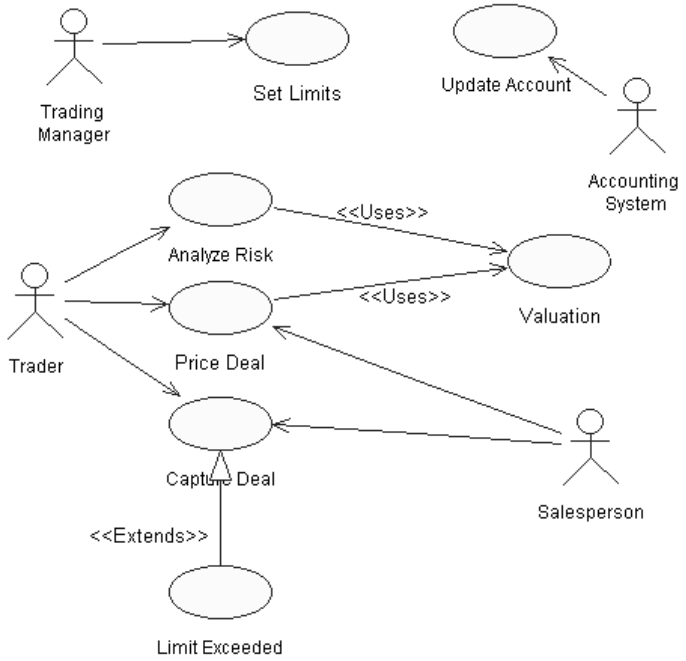
«دنباله‌ای از عملیات است که یک سیستم انجام می‌دهد تا یک نتیجه قابل مشاهده و ارزشمند برای فرد استفاده کننده از سیستم فراهم نماید.»

مثال: در یک سیستم واژه پرداز "تغییر خط یک متن انتخابی" یا "صفحه بندی خودکار یک متن" می‌تواند هر کدام (با توجه به تعریف فوق) یک مورد کاربری باشند. در زبان مدلسازی UML مورد کاربری را با یک بیضی نمایش می‌دهند که نام مورد کاربری در داخل بیضی(یا زیر آن)نوشته می‌شود. مورد کاربری از یک سناریو (جریان) اصلی تشکیل شده است و می‌تواند حاوی جریان‌های فرعی نیز باشد که نشان‌دهندهٔ سایر روندهایی است که در انجام مورد کاربری اتفاق می‌افتند. کامل بودن موارد کاربری نسبی است و هر چه مورد کاربری دقیق‌تر بیان شود سبب افزایش درک از خواسته‌ها می‌شود. البته باید در نظر داشت که بیان مورد کاربری که از هر نظر کامل باشد، مد نظر نیست و در روند تکراری و افزایشی پروژه به‌تدریج موارد کاربری کامل می‌شوند.

مفهوم دیگری که مدل موارد کاربری مورد نیاز است مفهوم عامل[1] می‌باشد. عامل در حقیقت شی خارج از حیطه سیستم است که مستقیماً با آن در ارتباط است. کاربران و کلیه سیستم‌های که با سیستم مورد نظر در ارتباط هستند عامل‌های آن هستند.

مثال: در شکل ۸-۲ چهار عامل دیده می‌شوند: Trading Manager، Salesperson، Trader و Accounting System. در این سیستم احتمالاً افراد زیادی با عنوان Trader وجود دارند اما چون همه یک نقش واحد دارند به ازاء همه یک عامل در نظر گرفته شده است. همچنین هر شخص مطرح در این سیستم می‌تواند نقش‌های متعددی داشته باشد. به عنوان مثال شخصی که Trader است می‌تواند Trading Manager هم باشد یا یک Trader می‌تواند Salesperson هم باشد. بنابراین در بدست آوردن عامل‌های یک سیستم بهترست به نقش‌های افراد بجای تیترهای شغلی آنها توجه شود.

رابطه موارد کاربری و عوامل m:n می‌باشد یعنی هر عامل می‌تواند اجرا کننده چندین مورد کاربری باشد و هر مورد کاربری هم می‌تواند بوسیله چند عامل اجرا گردد.



شکل ۸-۲- نمونه موارد کاربری

[1] Actor

با استفاده از مورد کاربری و عامل‌ها که اصلی‌ترین عناصر مدل موارد کاربری هستند می‌توان رفتار سیستم را از دیدگاه کاربران بیان نمود. نمایش این رفتار در مدل موارد کاربری همانطور که در فصل‌های بعدی خواهیم دید، به‌صورت دقیق‌تر در نمودارهای چون ترتیبی و فعالیت بیان خواهد شد.

در توسعه سیستم با استفاده از مدل موارد کاربری بسته به اندازه سیستم، مجموعه‌ای از مدل‌های موارد کاربری مختلف مورد استفاده قرار می‌گیرند. در صورتیکه سیستم کوچک باشد یک مدل مورد کاربری و در سیستم‌های با اندازه بزرگتر بیش از یک مدل مورد استفاده قرار می‌گیرند. این مجموعه از موارد کاربری، دید موارد کاربری را تشکیل می‌دهند. این دید، یکی از مهمترین دیدهای معماری است که اغلب برای معماران نرم‌افزار مهم و حیاتی است.

از جمله مدل‌های موارد کاربری با اهمیت که در سیستم‌های بزرگ اغلب مورد استفاده قرار می‌گیرند، مدل مورد کاربری حرفه است. مدل موارد کاربری حرفه از مورد کاربری حرفه و عامل حرفه (عامل کاری) تشکیل شده است. مورد کاربری حرفه ترتیبی از کنش‌هایی است که سازمان انجام می‌دهد و یک نتیجه ارزشمند برای یک عامل کاری در پی دارد و عامل کاری شخص یا سیستمی بیرون از سازمان است که با سازمان تعامل دارد. شکل مورد کاربری حرفه و عامل کاری همانند مورد کاربری و عامل است با این تفاوت که خطی در گوشه آن وجود دارد. به‌عبارت بهتر، مورد کاربری برای نمایش کنش‌های سیستم و مورد کاربری حرفه برای نمایش کنش‌های سازمان است.

۸-۳- سازماندهی موارد کاربری

در UML چهار روش ذیل برای سازماندهی موارد کاربری پیشنهاد می‌شوند:

۱) بسته‌بندی[1]

بسته‌بندی مکانیزمی عمومی است که برای گروه‌بندی عناصر منطقاً مرتبط در گروه‌های بزرگتر پیشنهاد می‌شود. برای نمایش بسته از شکل Package موجود در UML استفاده می‌شود. با این روش مدل‌های موارد کاربری مرتبط در یک بسته قرار می‌گیرند.

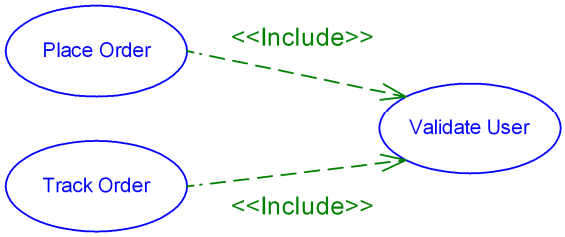
۲) رابطه عام/خاص[2]

[1] Packaging
[2] Generalization/Specialization

رابطه عام/خاص که رابطه وراثت نیز اطلاق می‌شود، برای بیان ارتباط دو یا چند مورد کاربری یا عامل با یک مورد کاربری یا عامل عمومی‌تر مورد استفاده قرار می‌گیرد. به‌عنوان نمونه مورد کاربری «تائید اعتبار کاربر» می‌تواند با موارد کاربری خاص‌تری نظیر «چک‌کردن کلمه عبور» یا «بررسی اثر انگشت» انجام شود. برای نمایش رابطه عام/خاص از شکل ⟶▷ استفاده می‌شود. عکس این رابطه نیز معنادار است، بدین معنی که یک مورد کاربری یا عامل عمومی‌تر توسط یک یا چند مورد کاربری یا عامل خاص‌تر ارتباط دارد.

۳) رابطه «دربرداشتن»[1]

رابطه در برداشتن وقتی مورد استفاده قرار می‌گیرد که یک مورد کاربری برای انجام وظایف خود از مورد یا موارد کاربری دیگری استفاده می‌کند. شکل ۸-۳ نمونه‌ای از استفاده از رابطه در برداشتن را نشان می‌دهد. موارد کاربری Place Order و Track Order برای انجام عملیات خود نیازمند استفاده از مورد کاربری Validate User هستند.

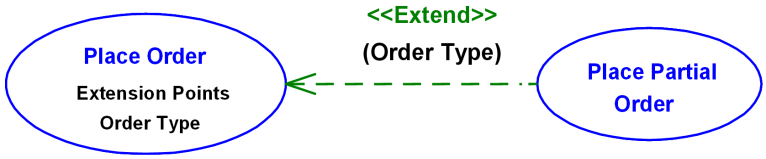


شکل ۸-۳- نمونه استفاده از رابطه در برداشتن

۴) رابطه «گسترش دادن»[2]

رابطه گسترش دادن وقتی مورد استفاده قرار می‌گیرد که یک مورد کاربری احتمال دارد از یک مورد کاربری دیگری استفاده نماید. شکل ۸-۴ نمونه‌ای از استفاده از رابطه گسترش دادن را بین دو مورد کاربری نمایش می‌دهد. در این رابطه Place Order در برخی موارد به مورد کاربری Palce Partial Order گسترش می‌یابد. به‌عبارت بهتر سفارش‌ها می‌توانند Partial Order باشند که در این صورت از مورد کاربری Place Partial Order استفاده می‌شود.

[1] Include Relationship
[2] Extend Relationship



شکل ۸-۴- نمونه استفاده از رابطه گسترش دادن

کاربردهای رابطه گسترش دادن عبارتند از:

- برای مدلسازی بخش انتخابی مورد کاربری
- برای مدلسازی زیرجریانی از مورد کاربری پایه که تنها در شرایط خاصی به اجرا در می‌آید
- برای مدلسازی جریان فرعی پیچیده
- هنگامیکه در یک جریان فرعی از مورد کاربری، لازم باشد که جریان فرعی دیگر داشته باشیم

۸-٤- ایجاد مدل موارد کاربری

یکی از روش‌های ایجاد مدل موارد کاربری، روش جعبه سیاه[۱] بوده که در آن سیستم مورد نظر بصورت یک جعبه سیاه که به رویدادهای گوناگون که آغاز کننده آنها عوامل بوده عکس العمل نشان می‌دهد، دیده می‌شود. موارد کاربری راه های مختلف استفاده از جعبه سیاه بازای رویدادهای مختلف را نمایش می‌دهد. مراحل به کارگیری این روش بشرح ذیل می‌باشد:

۱) صورت مسئله یا هدف سیستم را مشخص نمایید

با استفاده از صورت مسئله، اسناد موجود و فرآورده‌های تولید شده (مانند دورنما، فهرست اصطلاحات، درخواست ذینفعان، مدل مورد کاربری حرفه و مدل شی حرفه) هدف سیستم را دوباره در ذهن خود مرور نمایید. در واقع هدف سیستم با جواب دادن به این سوال که "چرا می‌خواهیم سیستم را بسازیم؟" مشخص می گردد[۲].

۲) شناسائی عوامل و سازماندهی آنها

از تعریف مسئله شروع کرده و سوالاتی درباره اینکه چه کسانی(یا سیستم‌هایی) متصدی انجام سرویس‌هایی که سیستم مورد نظر باید فراهم کند، را می‌پرسیم. پاسخ این سوال‌ها منجر به تشخیص

---

[۱] Black-Box Approach

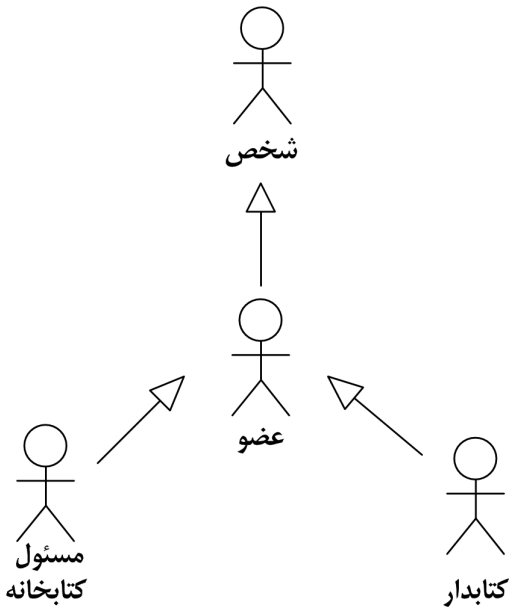
[۲] اصطلاحاً به آن Statement of Purpose گویند.

عوامل اولیه سیستم می‌گردد. معمولاً اولین سوال که باید پرسیده گردد"آیا مشتری به صورت مستقیم با سیستم در تماس است یا خیر؟" جواب این سوال می‌تواند منجر به یافتن اولین عامل سیستم گردد. با ادامه زنجیر سوال‌ها و متمرکز شدن روی نحوه برآوردن هدف های سیستم بقیه عوامل پیدا می‌شوند. سپس کاربران شناسائی شده با توجه به نقشی که در تعامل با سیستم ایفا می‌کنند، گروه‌بندی می‌شوند.

جواب به سوالات ذیل می‌تواند در یافتن عوامل سیستم مفید باشند

- چه نقش‌هایی از عملکرد اصلی سیستم استفاده خواهند کرد؟
- چه نقش‌هایی به پشتیبانی سیستم برای انجام کار خود نیاز دارند؟
- برای چه نقش‌هایی خروجی سیستم جالب یا ارزشمند است؟
- سیستم مورد نظر با چه سخت‌افزارهایی تعامل دارد؟
- سیستم با چه سیستم‌های خارجی (مکانیزه یا غیرمکانیزه) تعامل دارد؟

برای سازماندهی عوامل از رابطه عام/خاص استفاده می‌شود. شکل ۸-۵ نمونه‌ای از استفاده از رابطه عام/خاص را در گروه‌بندی عامل‌ها نشان می‌دهد.



شکل ۸-۵- نمونه استفاده از رابطه عام/خاص در گروه‌بندی عامل‌ها

۳) شناسائی موارد کاربری و دسته‌بندی آنها

برای شناسائی موارد کاربری باید یک نگرش کاربرگرا[1] به سیستم داشته باشیم و از خودمان بپرسیم نحوه ارتباط کاربر با سیستم چگونه خواهد بود و چه سرویس‌هایی مورد انتظار کاربر می‌باشد؟ در شناسایی موارد کاربری باید توجه داشت که مورد کاربری معادل روال یا تابع نیست و باید به نتیجه ارزشمندی از دید کاربر منجر شود و سیستم را از حالت شناخته شده به حالت شناخته شده دیگر منتقل نماید. برای شناسایی موارد کاربری از روی عامل‌های شناسایی شده، به ازای هر عامل سوالات ذیل را بپرسید:

- چه کارهایی باید سیستم انجام دهد تا نیاز این عامل برطرف شود؟
- آیا نیاز است که این عامل از رخدادن پاره‌ای از رویدادها در سیستم اطلاع پیدا نماید؟
- آیا نیاز است که این عامل سیستم را از رخدادن تغییرات ناگهانی یا خارجی خبر کند؟
- آیا سیستم رفتار درستی که مورد انتظار سازمان است را فراهم می‌نماید؟
- آیا همه ویژگی‌های وظیفه‌مندی مطلوب بوسیله موارد کاربری شناسائی شده، برآورده می‌گردد؟
- چه موارد کاربری وظیفه پشتیبانی و نگهداری سیستم را بعهده دارد؟
- چه اطلاعاتی در سیستم باید تولید، استفاده یا به‌روزرسانی شود؟

سوالات مطرح شده منجر به شناسایی اغلب موارد کاربری سیستم خواهند شد، اما برخی از موارد کاربری به علت شرایط خاص گاهاً شناسایی نمی‌شوند. این موارد کاربری عبارتند از:

- ❖ موارد کاربری مربوط به آغاز و خاتمه سیستم
- ❖ موارد کاربری مربوط به نگهداری از سیستم: مانند افزودن کاربران جدید و پیکربندی پروفایل کاربران
- ❖ موارد کاربری مربوط به نگهداری داده‌های سیستم
- ❖ موارد کاربری مربوط به عملکرد مورد نیاز برای اصلاح یا تغییر رفتار سیستم

پس از شناسایی موارد کاربری، برای سازماندهی آنها می‌توانید از روابط «عام/خاص»، «گسترش دادن» و «شامل بودن» استفاده نمایید. در استفاده از هر یک از این روابط باید دقت کافی و مورد نیاز صورت

[1] User-Oriented

پذیرد تا نتیجه دلخواه بدست آید و استفاده اشتباه از این روابط ممکن است منجر به ایجاد سیستمی شود که برخی از عملیات را بدرستی انجام ندهد.

٤) تعیین ارتباط میان عوامل و موارد کاربری

هنگام شناسائی موارد کاربری، بهتر است عواملی که با آنها تعامل دارند را نیز شناسائی نمایید تا مدل به‌صورت درست ایجاد شود. در هر رابطه، جهت رابطه بوسیله آغاز کننده تعامل معین می‌گردد.

٥) بسته‌بندی عوامل و موارد کاربری مرتبط

با تقسیم مدل موارد کاربری به تعدادی بسته که هرکدام شامل تعدادی عامل و مورد کاربری مربوطه بوده، درک و نگهداری مدل آسانتر می‌شود

٦) ترسیم نمودار موارد کاربری

پس از شناسایی و دسته‌بندی موارد کاربری و عامل‌ها، نمودار موارد کاربری ترسیم می‌گردد. این نمودار کلیه موارد کاربری و عوامل را نشان می‌دهد.

٧) تشریح موارد کاربری

هر مورد کاربری می‌بایست به‌گونه‌ای تشریح گردد تا توسعه‌دهندگان بتوانند آن را پیاده‌سازی نمایند. عمده‌ترین مواردی که در تشریح موارد کاربری ذکر می‌گردند عبارتند از:

- نام مورد کاربری: یک نام منحصر به فرد است.
- توصیف مختصر: یک توصیف فشرده از هدف و نقش مورد کاربری است
- جریان رخدادها: توصیفی از آنچه سیستم برای اجرای مورد کاربری باید انجام دهد
- نیازمندی‌های ویژه: این بخش به نیازهای غیروظیفه‌مندی که مختص این مورد کاربری بوده اشاره می‌کند
- پیش‌شرط‌ها[1]: حالتی که سیستم باید دارای آن باشد تا بتوان این مورد کاربری را اجرا نمود
- پس‌شرط‌ها[2]: فهرستی از حالت‌هایی که سیستم پس از اجرای این مورد کاربری دارای یکی از آنها خواهد بود
- نقاط گسترش: فهرستی از محل‌هایی در جریان مورد کاربری که در آنها رفتار اصلی این مورد کاربری بوسیله یک رفتار اضافی گسترش خواهد یافت

[1] Pre-Conditions
[2] Post-Conditions

۸) ارزیابی مدل

پس از تشریح موارد کاربری و ایجاد مدل موارد کاربری باید مدل مورد آزمایش قرار گیرد تا مشخص شود همه مشخصات مورد نظر را در بردارد. ارزیابی مدل موارد کاربری می‌تواند به‌صورت انتخابی یا با استفاده از مجموعه‌ای از مواردی که مورد کاربری می‌بایست حمایت کند، انجام می‌پذیرد.

۸-۵- مثال تعمیرگاه

با یک مثال تفصیلی نحوه اعمال گام‌های ذکر شده قبلی را بیان خواهد شد.

- صورت مسئله

"هدف سیستم فراهم نمودن مدیریت کارا برای همه جنبه‌های چرخه سرویس‌هی و تعمیر از تعریف کارهای[1] مورد نیاز مشتریان گرفته تا خاتمه کارها می‌باشد" سیستم باید تسهیلات زیر را ارائه نماید:

- رزرو کارها(شامل سرویس و تعمیر)
- شناسائی قطعات یدکی مورد نیاز و درخواست آنها
- زمانبندی کارها
- ثبت جزئیات کارهای انجام شده
- مسائل مربوط به اتمام یک کار: مانند تحویل ماشین و محاسبه هزینه کار

اینجا کارها بر دو نوعند: معمولی و اولویت‌دار

استثناها: سرویس‌دهی به ماشین‌های صنعتی یا بسیار سنگین."

- شناسائی عوامل : اینجا مفهوم مشتری داریم که از سیستم انتظار سرویس دارد پس اولین عامل همان مشتری خواهد بود(عامل خارجی) حال این سوال را می پرسیم" ارتباط مشتری با سیستم چگونه است؟ مستقیم یا غیر مستقیم؟" هنگامیکه وارد جزئیات عمل سیستم می‌شویم در می‌یابیم که مشتری به "مسئول پذیرش مشتریان" کار مورد نظر خود را بیان کرده که این مسئول با استفاده از امکاناتی که سیستم در اختیار او گذاشته شده مانند(فرم‌های ثبت کارهای مطلوب،...) درخواست مشتری را یاداشت می‌نماید. بنابراین می‌توان گفت که عامل دوم همان مسئول پذیرش مشتریان می‌باشد. در ادامه این سناریو می‌بینیم که کنترل‌کننده قطعات درخواست مشتری را بررسی نموده و مشخص

[1] اینجا مقصود از کارها همان Customer Jobs که شامل سرویس‌دهی و تعمیر می‌باشد.

می‌نماید آیا به قطعات یدکی نیاز است یا خیر؟ در صورت نیاز و عدم وجود قطعات در تعمیرگاه درخواست خرید را صادر نموده و به تولیدکننده می فرستد. بعد از تهیه قطعات مورد نیاز باتوجه به زمانبندی کارها در تعمیرگاه روزی برای تعمیر ماشین معین می گردد(بوسیله مکانیک) بالاخره ماشین در روز مشخص تعمیر شده و پس از ثبت جزئیات کارهای انجام شده، اطمینان از رضایت مشتری و دریافت مزد به مشتری تحویل می گردد.در این سناریو عوامل زیر قابل شناسائی می‌باشند:

کنترل کننده قطعات، مکانیک ◄..... عوامل داخلی

تولید کننده ◄..... عامل خارجی

در جدول ۸-۱ عواملی که با سیستم در ارتباط مستقیم هستند لیست شده‌اند.

جدول ۸-۱- فهرست عامل‌های تعمیرگاه

عامل	شرح
مسئول پذیرش مشتریان	مسئول ارتباط با مشتریان و شناسائی نیازهای آنها
کنترل کننده قطعات	مسئول نگهداری وتهیه قطعات یدکی مورد نیاز و پیش بینی نیازهای مشتریان
مکانیک	مسئول زمانبندی کارها و اطمینان از درستی انجام آنها

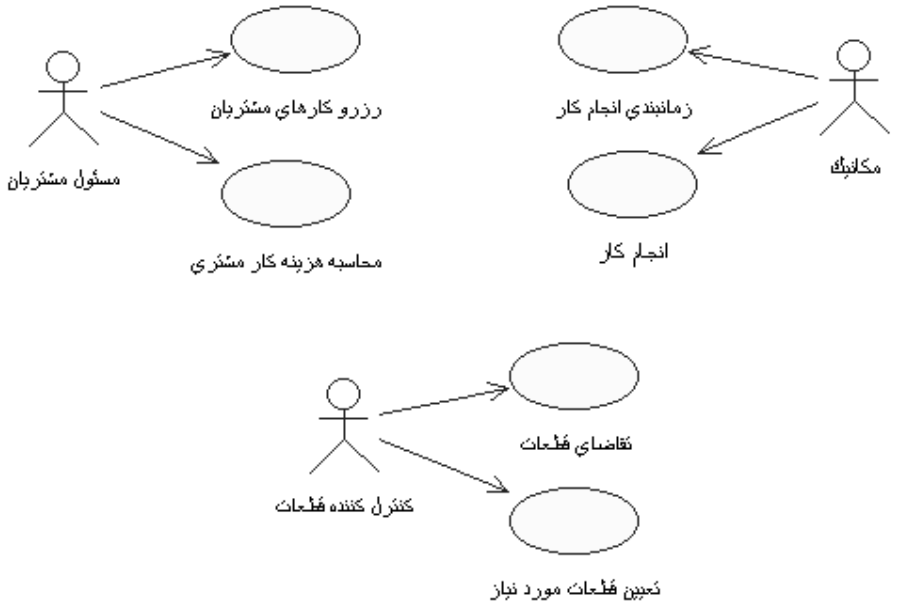
- شناسائی موارد کاربری: با توجه به سناریوی ذکر شده می‌توان موارد کاربری زیر را نام برد.

جدول ۸-۲- فهرست موارد کاربری تعمیرگاه

مورد کاربری	رویداد محرک	عامل
ثبت کار مورد نیاز مشتری	درخواست مشتری	مسئول پذیرش مشتریان
تعیین قطعات مورد نیاز	فرارسیدن زمان بررسی نیازهای مشتریان	کنترل کننده قطعات
درخواست قطعات	عدم وجود قطعات لازم در تعمیرگاه	کنترل کننده قطعات
زمانبندی کارها	فرارسیدن زمانبندی کارها	سرمکانیک
مدیریت کار از ابتدا تا خاتمه، اطمینان از درستی انجام آن و ثبت جزئیات کار انجام شده	فرارسیدن زمان انجام کار	سرمکانیک
اطمینان از رضایت مشتری، دریافت مزد کار و تحویل	رسیدن مشتری برای تحویل	مسئول پذیرش مشتریان

مورد کاربری	رویداد محرک	عامل
ماشین به مشتری	ماشین	

- ایجاد نمودار موارد کاربری: با توجه به آنچه بیان شد، نمودار در شکل ۸-٦ رسم شده است.



شکل ۸-٦- نمودار موارد کاربری مثال تعمیرگاه

- شرح موارد کاربری: بعنوان مثال مورد کاربری "رزرو کارهای مشتریان" را شرح می نماییم:

نام مورد کاربری	رزرو کارهای مشتریان
هدف مورد کاربری	تعیین کار مناسب که نیازهای مشتریان را به نحو احسن برآوردکند
گام‌های مورد کاربری	۱- بدست آوردن جزئیات خودرو و مشتری: ۱-۱ برای مشتریان فعلی جزئیات مربوط به آنها استخراج کنید ۱-۲ برای مشتریان جدید مشخصات مورد نیاز(مانند: نام، آدرس، شماره خودرو، مدل آن، و سال ساخت) را ثبت نمایید. ۲- اگر کار مورد نیاز سرویس باشد، سرویس‌های مناسب مدل خودرو را پیدا کنید. ۳- اگر کار مورد نیاز تعمیر باشد، هزینه آنرا پیش بینی نمایید. ٤- بر زمان و ساعت کار با مشتری به توافق برسید. ٥- مشخصات ذکر شده –بعد از تایید مشتری–راثبت نمایید.

۸-٦- مشکلات مدل‌سازی موارد کاربری

در استفاده از مدل‌سازی موارد کاربری سه مورد مهم را باید در نظر داشت که عبارتند از:

- ❖ تنها با مدل‌سازی از طریق موارد کاربری نمی‌توان همه جنبه‌های سیستم را مدل کرد. به عنوان نمونه نمی‌توان الگوریتم‌ها را از طریق موارد کاربری نمایش داد. در این مورد باید این‌گونه موارد را در ادامه مدل‌سازی و با ابزارهای مراحل تحلیل و طراحی نمایش داد.
- ❖ روی هم افتادگی در موارد کاربری بوجود می‌آید. در این مورد از پیش‌شرط‌ها و شرایط بعد از وقوع به منظور دقیق‌تر نمودن محدوده استفاده کنید.
- ❖ تشخیص اینکه چه موقع کار پایان یافته است. برای تشخیص این موضوع از قاعده ۸۰-۲۰ استفاده کنید. بدین‌معنی که به‌دنبال یافتن موارد کاربری و عامل‌ها به صورت کامل و جامع نباشید.

۸-۷- تفاوت مدل‌سازی مورد کاربری و تحلیل سیستم

مدل‌سازی موارد کاربری به تحلیل سیستم کمک می‌کند و سبب می‌شود تا تحلیل سیستم جامع‌تر و دقیق‌تر انجام شود. هر دو این فعالیت‌ها به منظور شناخت انجام می‌پذیرد و البته تفاوت‌هایی نیز با یکدیگر دارند که عبارتند از:

- ▪ از موارد کاربری برای تعیین نیازها استفاده می‌شود (در مرحله نیازمندی‌ها قبل از مرحله تحلیل)
- ▪ موارد کاربری به زبان قابل فهم برای مشتریان تهیه می‌شود ولی تحلیل برای تولید کننده سیستم نوشته می‌شود.
- ▪ در موارد کاربری، سیستم را از دید خدمات بیرونی نگاه می‌کنیم ولی در تحلیل از دید امکانات داخلی به سیستم نظر می‌کنیم.

۹- مدلسازی کلاس‌ها[1]

در فصل ۳ و ٤، مفاهیم مرتبط به کلاس بعنوان جزء اصلی همه متدولوژی‌ها شی‌گرا بیان گردید. در این فصل به نحوه نمایش کلاس و گام‌های لازم برای رسم نمودار کلاس پرداخته خواهد شد. در واقع، مدلسازی کلاس‌ها در ادامه کار مدلسازی موارد کاربری و برای تکمیل مدل‌های تحلیل و طراحی مورد استفاده قرار می‌گیرد.

۹-۱- منابع اصلی تشخیص کلاس‌ها

قبل از اینکه مدلسازی کلاس‌ها بیان شود، بهتر است نکاتی در مورد شناسایی کلاس‌ها و اینکه کلاس‌ها از کجا و در چه مرحله‌ای شناسایی می‌شوند، مطرح شود. شناسایی کلاس‌ها از اولین مراحل پروژه شروع می‌شود از زمانی که مسئله بیان می‌شود. در واقع، در زمانی که مسئله‌ای برای توسعه نرم‌افزار مطرح می‌شود، کلاس‌های اولیه با روش‌هایی که قبلاً بیان شدند، شناسایی می‌شوند و در ادامه کار این کلاس‌ها بهبود می‌یابند. بطور کلی می‌توان از منبع اصلی کلاس‌ها را تشخیص داد: فضای مسئله و فضای راه‌حل. این دو منبع دو نوع متفاوت کلاس را در اختیار قرار می‌دهند که به‌ترتیب کلاس‌های تحلیل و کلاس‌های طراحی هستند.

با توصیفاتی که از فضای مسئله وجود دارد می‌توان کلاس‌های تحلیل یا مدل تحلیل کلاس‌ها را ارائه نمود. این کلاس‌ها در ادامه بهینه شده و در مدل‌سازی طراحی کلاس‌ها مورد استفاده قرار می‌گیرد. فضای راه‌حلی که برای توسعه نرم‌افزار ارائه می‌شود سبب می‌شود که مدل طراحی کلاس‌ها ایجاد شود. ناگفته نماید که فضای راه‌حل از فضای مسئله تاثیر می‌پذیرد، بنابراین کلاس‌های طراحی از کلاس‌های تحلیل تاثیر می‌پذیرند.

کلاس‌های تحلیل و طراحی با توجه به تکرارهایی که در پروژه وجود دارند، کامل می‌شوند و همانطور که برای مدلسازی موارد کاربری ذکر گردید، بدنبال ایجاد کلاس‌های تحلیل یا طراحی جامع و کامل نباید بود و بهتر است با قاعده ۸۰-۲۰ کلاس‌های تحلیل و طراحی مناسب پروژه انتخاب شوند و کار توسعه نرم‌افزار ادامه پیدا کند.

[1] Class Modeling

۹-۲- ایجاد نمودار کلاس

برای رسم نمودار کلاس‌ها، مراحل زیر باید طی شوند.

۱) **شناسائی کلاس‌ها:** برای شناسایی کلاس‌ها می‌توان از روش‌های زیر استفاده نمود:

- ❖ استفاده از کارت‌های CRC
- ❖ صورت مسئله و نیازهای وظیفه مندی[۱] تحلیل می‌کنیم: نام‌ها، کلاس‌ها یا صفات کاندید و فعل‌ها، اعمال یا ارتباط‌ها کاندید به حساب می‌آیند. این روش باید بدقت-وبا توجه به مفهوم کلاس و عدم اکتفا به تحلیل صرف واژه‌ها- بکاربرد و الا لیستی طولانی از کلاس‌های بی‌معنی خواهیم داشت.
- ❖ مراجعه به شرح موارد کاربری و تطبیق روش قبلی.

۲) **ترسیم برداشت اولیه از نمودار کلاس**: در اینجا کلاس‌های کلیدی و روابط اساسی را مشخص نموده و از بقیه جزئیات صرف نظر می‌نماییم.

۳) **تکمیل جزئیات کلاس‌ها**: در این مرحله جزئیات مورد نیاز برای هر کلاس از جمله صفات و اعمال هر کلاس به هر کلاس افزوده می‌شود.

٤) **تکمیل و توسعه نمودار کلاس‌ها**: نمودار کلاس با استفاده از جزئیاتی که به هر کلاس افزوده شده است تکمیل می‌شود.

دو نکته مهم در ایجاد نمودار کلاس می‌بایست مد نظر قرار گیرند:

- ❖ استفاده از الگوهای معروف برای آسانی طراحی و بهره برداری از استفاده مجدد.
- ❖ معمولاً، تا این مرحله نمودار پیچیده‌ای پدید خواهد آمد که با اعمال مفهوم روابط تجمعی، وراثت، وابستگی می‌توان از پیچیدگی آن کاهش نمود.

در قسمت‌های بعدی (نحوهٔ ایجاد مدل تحلیل) مدلسازی کلاس بیشتر تشریح خواهد شد.

---

[۱] Functional Requirement

۹-۳- مدلسازی کلاس‌ها در RUP

مدلسازی کلاس‌ها در RUP با مدلسازی موارد کاربری شروع می‌شود. مدل موارد کاربری همانند صورت مسئله کلاس‌های تحلیل را مشخص می‌کنند. با استفاده از توصیفاتی که برای موارد کاربری وجود دارد از جمله Use-case Description و Supplementary Specifications کلاس‌های اولیه تحلیل شناسایی می‌شوند. در مرحله طراحی با استفاده از مستند معماری و مدل‌های تحلیل، کلاس‌های طراحی ایجاد می‌شوند.

در تحلیل سیستم برای هر کلاس موارد زیر تعیین می‌شود:

- رفتار
- ساختار سیستم
- نیازهای وظیفه‌مندی
- یک مدل کوچک

در طراحی سیستم برای هر کلاس موارد زیر تعیین می‌شود:

- عملیات و صفات کلاس
- کارایی
- سطح جزئیات نزدیک به کد واقعی
- نیازهای غیر وظیفه‌مندی
- یک مدل بزرگ

۹-۴- مدل تحلیل

یکی از مشکلات اصلی تولید سیستم‌های نرم‌افزاری انتقال از مرحله جمع‌آوری نیازمندی‌ها به مرحله طراحی است. در واقع پاسخ به این سوال که «چگونه می‌توان موارد کاربری را بعنوان نمایش‌دهنده نیازهای وظیفه‌مندی به کلاس‌ها بعنوان پیاده‌کننده رفتار این موارد کاربری، تبدیل کرد؟» می‌تواند با مدل تحلیل کلاس‌ها انجام پذیرد. این مدل ابزاری برای ارتباط مدل تحلیل و طراحی است.

برای یکپارچگی مدل‌سازی کلاس‌ها در تحلیل از سه نوع کلاس در RUP استفاده می‌شود که عبارتند از:

- ❖ کلاس‌های مرزی
- ❖ کلاس‌های کنترلی
- ❖ کلاس‌های موجودیتی

برای استفاده از مدل موارد کاربری برای مدل تحلیل کلاس‌ها می‌بایست ابتدا موارد کاربری را عینیت بخشید. پس از عینیت بخشیدن به موارد کاربری مجموعه‌ای از کلاس‌های تحلیل اولیه ایجاد می‌شوند که می‌توانند در گام‌های بعدی مورد استفاده قرار بگیرند. به‌طور خلاصه می‌توان نحوهٔ ایجاد مدل تحلیل را بدین صورت بیان نمود:

۱) ایجاد فراورده عینیت بخشیدن به موارد کاربری

۲) تکمیل تشریح موارد کاربری

- ❖ به ازای هر مورد کاربری ایجاد شده، گام‌های ذیل را انجام دهید

۳) کلاس‌های تحلیلی را شناسائی نمایید

۴) رفتار مورد کاربری را روی کلاس‌های تحلیلی بدست آمده توزیع کنید

- ❖ به ازای هر کلاس تحلیلی شناسائی شده، گام‌های ذیل را انجام دهید

۵) وظایف یک کلاس تحلیلی را تشریح نمایید

۶) صفات یک کلاس و روابط آن با بقیه کلاس‌ها را شناسائی و تشریح نمایید

۹-۵- مدل طراحی

مدل طراحی در راستای ایجاد مدل کلاس‌ها ایجاد می‌شود. مدل طراحی از روی مدل تحلیل بوسیله اعمال موارد زیر بدست می‌آید:

- ❖ ویژگیهای لازم برای پیاده‌سازی رفتار مطلوب سیستم
  - مانند ماندگاری اشیاء، همزمانی، رسیدگی به خطاها، مدیریت تراکنش‌ها
- ❖ محدودیت‌های پیاده‌سازی
  - استفاده از یک زبان برنامه‌نویسی مشخص
- ❖ نیازهای غیر وظیفه‌مندی
  - مانند قابلیت اعتماد، زمان پاسخ، کارایی، سرعت سیستم

برای ایجاد مدل طراحی مجموعه‌ای از گام‌ها انجام شوند:

۱) کلاس‌های طراحی اولیه را ایجاد کنید
   - طراحی کلاس‌های مرزی
   - طراحی کلاس‌های موجودیتی
   - طراحی کلاس‌های کنترلی
۲) کلاس‌های ماندگار را شناسائی نمایید
۳) مکانیزم دستیابی به هر کلاس را تعیین نمایید
۴) عملیات هر کلاس را مشخص نمایید
۵) حالت کلاس را مشخص نمایید
۶) وابستگی‌ها را مشخص نمایید
۷) روابط انجمنی و تجمعی را مشخص نمایید
۸) رابطه عام/خاص را بدست آورید
۹) تداخل‌های احتمالی بین موارد کاربری را برطرف کنید
۱۰) نیازهای غیر وظیفه‌مندی را اعمال کنید
۱۱) نتایج بدست آمده را ارزیابی نمایید

همانطور که در گام‌ها نیز مشخص است و قبلاً نیز بیان گردد مدلسازی کلاس و استفاده از مفهوم کلاس به‌عنوان یکی از اصلی‌ترین مفاهیم مدلسازی در RUP است. کلاس‌های طراحی پس از کلاس‌های تحلیل (که از عینیت بخشیدن به موارد کاربری ایجاد می‌شوند) حاصل می‌شوند.

۹-۶- مدل داده‌ای

پس از ایجاد مدل طراحی نیاز است تا ساختار داده‌ها به صورت منطقی و فیزیکی مشخص شود. مدل داده‌ای این امکان را می‌دهد که ساختار داده‌هایی که نرم‌افزار می‌بایست ذخیره نماید، مشخص شود. به‌عبارت بهتر، هدف مدل داده‌ای توصیف ساختار منطقی و فیزیکی داده‌های ماندگار است و هنگامی به این مدل نیاز است که نتوان ساختار داده‌ای ماندگار (ساختار داده‌ای لازم برای ذخیره کلاس ماندگار) را از ساختار کلاس ماندگار به صورت خودکار یا اتوماتیک تعیین نمود.

۹-۷- مثال تعمیرگاه

برای درک بهتر مدل طراحی و ایجاد مدل کلاس‌ها مثال تعمیرگاه که در فصل قبل توصیف گردید، ارائه می‌شود.

۱) شناسائی کلاس‌ها از شرح موارد کاربری بدست آمده در فصل قبلی استفاده کرده و روش تحلیل گرامری را در مورد آن اعمال می نماییم:

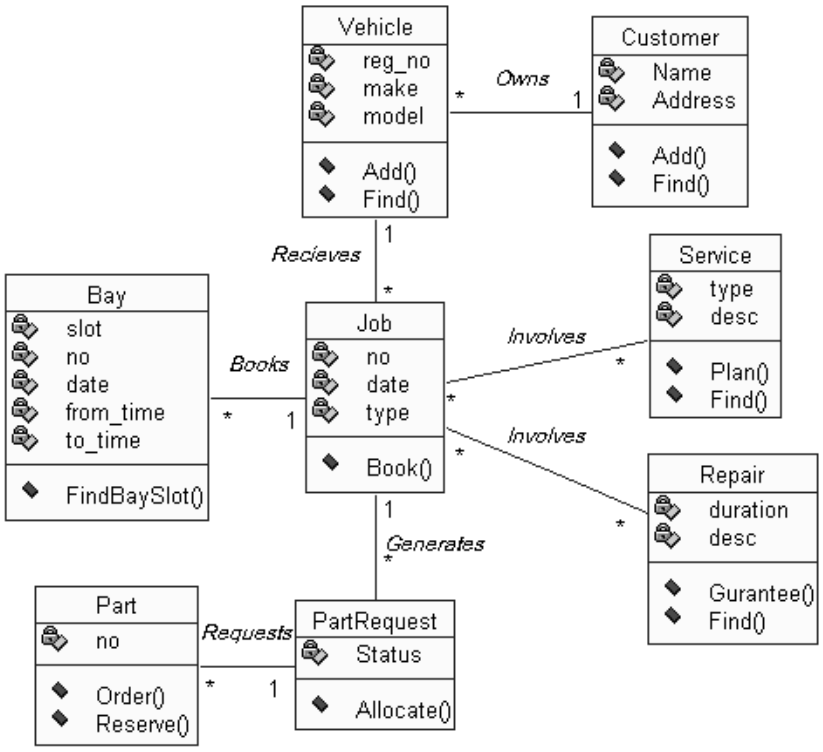
- شناسائی نام‌ها: همانطوریکه بیان شد نامهایی که در دامنه مسئله مفهوم کلیدی داشته و ویژگی‌های ذکر شده در تعریف کلاس و شی را در بردارند، پیدا می نماییم
  - مشتری، خودرو، سرویس، تعمیر، کار.
  - نام و آدرس از صفات مشتری محسوب می‌شوند.
  - شماره ماشین و مدل آن از صفات ماشین محسوب می‌گردند. نوع سرویس(معمولی یا با اولویت) صفت سرویس محسوب می‌شود.
  - چون نباید به تحلیل گرامری اکتفاد نمود می‌توان روابط زیر را اضافه کرد:
    - ماشین یک مالک دارد و آن مشتری است.
    - "کار" در رابطه با ماشین انجام می‌گیرد.
    - یک کار شامل سرویس، تعمیر یا هر دو می‌باشد.
- شناسائی فعل‌ها: اینجا باید اعمال مهم و اینکه هر عمل به کدام کلاس مربوط است را پیدا کنیم. این اعمال و کلاس‌های آنها در جدول ۹-۱ زیر قید شده‌اند.

جدول ۹-۱- فهرست کلاس‌های کاندیدا مثال تعمیرگاه

عمل	کلاس(های) کاندید
پیدا کردن مشتری	مشتری، خودرو
اضافه مشتری	مشتری
پیدا کردن سرویس‌ها	سرویس
پید کردن تعمیرها	تعمیر
رزرو کار	کار

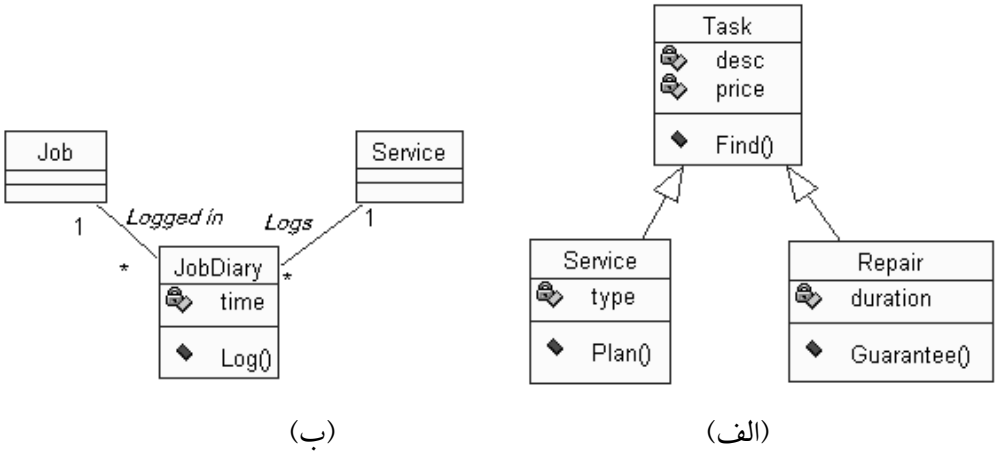
۲) ترسیم برداشت اولیه از نمودار کلاس

۳) تکمیل نمودار و افزودن صفات و اعمال مهم (شکل ۹-۱)



شکل ۹-۱- نمودار کلاس‌های مثال تعمیرگاه

٤) اعمال مفاهیم روابط تجمعی، وراثت و وابستگی (شکل ۹-۲)



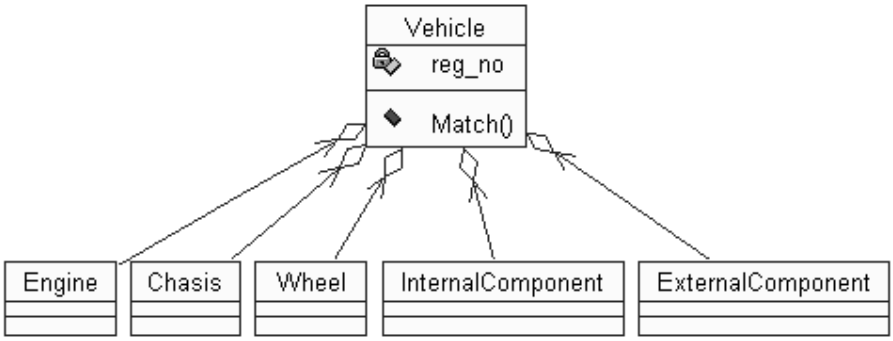
(ب) (الف)

شکل ۹-۲- اعمال رابطه وراثت و رابطه وابستگی به نمودار کلاس‌ها

در شکل ۹-۲-الف از مفهوم وراثت استفاده شده است. چون هر دو کلاس Service و Repair صفات و اعمال مشترکی[1] و متفاوتی دارند پس می‌توان یک Superclass مانند Task معرفی نمود که صفات و اعمال مشترک آنها را دربرگیرد.

مواردی وجود دارد که در آن می‌توان برای رابطه وابستگی صفتهایی یا اعمالی درنظر گرفت. در این موارد این رابطه به صورت کلاس مدلسازی می‌گردد. برای مثال شکل ۹-۲-ب را در نظر بگیرید: در این شکل رابطه بین Job و Service رابطه چند به چند است. حال اگر بخواهیم زمان خاتمه یک سرویس متعلق به یک کار را ثبت نماییم، چکار باید کرد؟ یکی از راه ها این است که این زمان بعنوان صفتی برای یک کلاس سوم (JobDiary) در نظر می‌گیریم. این کلاس وابستگی بین دو کلاس اصلی را نمایش داده و با هرکدام رابطه یک به چند دارد.

شکل ۹-۳ در از رابطه تجمعی استفاده شده است. چنانکه می‌دانیم یک خودرو از اجزائی مانند موتور، شاسی[2]، چرخ‌ها و غیره تشکیل می‌گردد.



شکل ۹-۳- اعمال رابطه تجمعی به نمودار کلاس‌ها

[1] صفات مشترک مانند توصیف(Description) و قیمت اولیه(Price) و اعمال مشترک مانند(Find)

[2] Chasis

۱۰- مدلسازی تعامل و رفتار[1]

همانطوریکه بیان کردیم می‌توان یک سیستم نرم‌افزاری از چند زاویه بررسی نمود. یک زاویه همان تمرکز بر ساختار و روابط حاکم بین اجزاء سیستم است. نمودار کلاس سیستم را از این زاویه مدلسازی می‌نماید. در واقع نمودار کلاس ساختار ایستای سیستم را توصیف می‌نماید. اما آیا أین کافی است؟ مسلماً خیر! زیرا یک سیستم علاوه بر ساختار ایستای آن یک رفتار پویا-مبتنی بر ساختار ایستا-دارد که بیان کننده نحوه و ترتیب ارتباط اجزای مختلف سیستم با یکدیگر می‌باشد. یک خودرو از موتور، چرخ‌ها، شاسی و ... تشکیل می‌شود (ساختار ایستا) همچنین برای اینکه خودرو حرکت کند این اجزاء با ترتیب خاصی همکاری می‌کنند. رفتار پویا بوسیله مجموعه نمودارهای تعامل و حالت نمایش داده می‌شود.

در مدلسازی تعامل، همکاری گروهی از اشیاء در انجام یک عمل معین نشان داده می‌گردد. بطور کلی منظور از یک عمل معین، مورد کاربری می‌باشد و هر نمودار تعامل رفتار یک مورد کاربری را با توجه به اشیاء دخیل در آن و ارتباطات ما بین این اشیاء نشان می‌دهد. نمودار تعامل بر دو نوع است: یکی نمودار ترتیبی و دیگری نمودار همکاری که هر یک توضیح داده خواهد شد.

مدلسازی حالت روی ویژگی‌های بیرونی اشیاء و روابط آنها است. مفهوم کلاس جنبه ایستای یک موجودیت را دربر می‌گیرد در حالیکه مفهوم حالت روی رفتار آن موجودیت تمرکز می یابد. در حقیقت می‌توان اشیاء را به صورت ماشین‌های متناهی[2] که در هر آن در یک حالت بخصوص بسر می برند، تصور کرد. تعداد این حالت‌ها متناهی می‌باشد. بنابراین برای هر شی یک نمودار حالت کشیده می‌شود که بیانگر تمام وضعیت‌هایی را که آن شی می‌تواند بر اثر بروز رخدادهای مختلف به خود بگیرد. برای مثال شی کتاب را در نظر بگیرید اگر به کتاب بعنوان یک FSM نگاه کنیم در می‌یابیم که کتاب حالت‌های گوناگونی دارد: حالت اولیه: وجود کتاب درکتابخانه، حال این کتاب می‌تواند-بعنوان مثال- حالت‌های "امانت داده شده" یا "گم شده" را بخود بگیرد. به رفتار یک شی در زمانهای متفاوت در یک سیستم چرخه حیات شی[3] در آن سیستم گفته می‌شود.

---

[1] Interaction and Collaboration Modeling
[2] Finite State Machines (FSM)
[3] Object Life Cycle

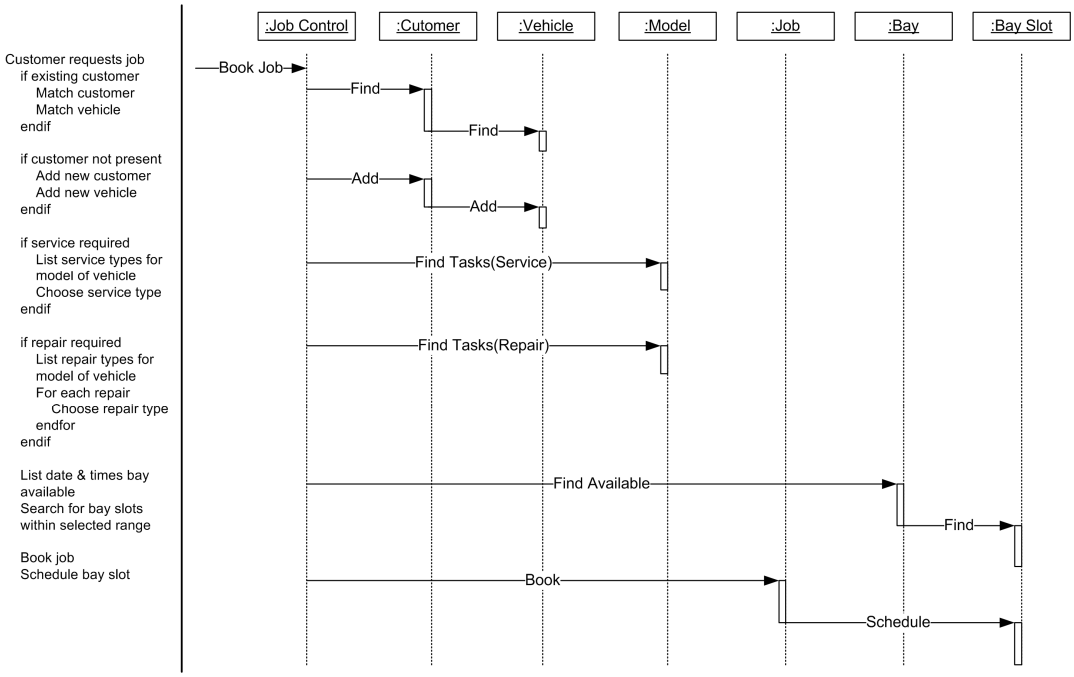
۱۰-۱-نمودار ترتیبی

نمودار ترتیبی معمولاً در بیان مراحل اجرای یک مورد کاربری استفاده می‌گردد. در این نمودار اشیاء به صورت جعبه‌هایی در بالای خطوط منقطع عمودی نمایش داده می‌شود. به هر یک از این خطوط عمودی، خط عمر آن شی گفته می‌شود. این اشیاء می‌توانند مجرد (بدون نام) باشند که نشانگر یک نمونه از یک کلاس است و به صورت ClassName:: نامگذاری می‌شوند. همچنین اشیاء می‌توانند نشان دهنده نمونه واقعی باشند و به صورت ClassName::ActualInstance نامگذاری می‌شوند. پیامی که از یک شی به دیگری فرستاده می‌شود توسط خط جهت داری در میان خطوط عمر دو شی کشیده می‌شود. هر پیام نشان‌دهنده درخواست اجرای یک عمل یا اعلام رخ دادن یک رویداد بوده و می‌تواند شامل مجموعه‌ای از پارامترها و اطلاعات کنترلی باشد. دو بخش از اطلاعات کنترلی روی پیامها حائز اهمیتند. بخش اول یک شرط است که فقط در صورت تحقق پیام فرستاده می‌شود(بین دو علامت[] نوشته می‌شود). بخش دوم علامت تکرار که با * نمایش داده می‌شود و معنای آن ارسال این پیام به چندین شی گیرنده می‌باشد.

برای نمایش بازگشت پیام از فلشی در جهت مخالف استفاده می‌گردد(←——). این فلش بمعنی بازگشت یک پیام است و بهتراست مواقعی که بازگشت از یک پیام مفهوم خاصی را منتقل نمی‌کند این فلش نمایش داده نشود. در سمت چپ نمودار تعامل اشیاء می‌توان-به صورت اختیاری-شرحی با فرم ساخت‌یافته برای مورد کاربری مورد نظر را نوشت. هدف این شرح تسهیل درک نمودار تعامل اشیاء می‌باشد. نمودارهای ترتیبی بسیار ساده و براحتی قابل درک می‌باشند و بعلاوه این نمودارها قابلیت نمایش فرآیندهای همزمان را دارند. برای درک بهتر، مثالی از یک نمودار ترتیبی تعمیرگاه ارائه می‌شود.

۱۰-۲-نمودار تعامل مثال تعمیرگاه

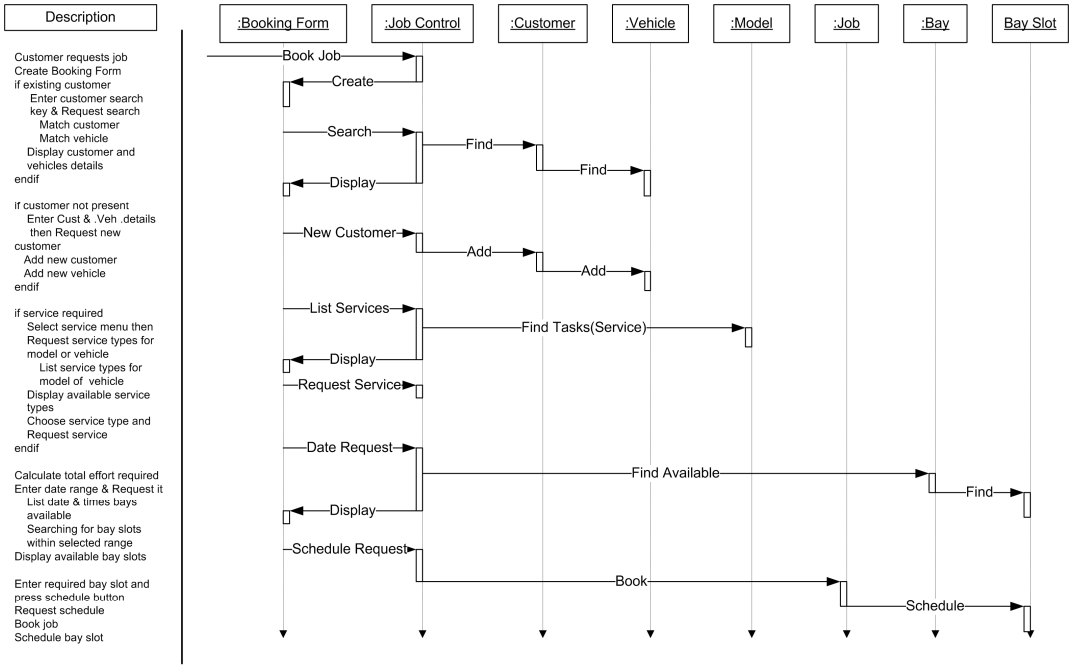
ترسیم نمودار تعامل اشیاء برای مورد کاری "ثبت کار استاندارد برای مشتری" . برای ترسیم نمودار تعامل اشیاء، نخست شرح مورد کاربری مورد نظر را بررسی کرده سپس با مراجعه به نمودار کلاس اشیاء مهم را انتخاب می نماییم. نمونه اولیه این نمودار در شکل ۱۰-۱ نمایش داده شده است.



شکل ۱۰-۱- نمودار اولیه تعامل اشیاء برای مورد کاربری"ثبت کار استاندارد برای مشتری"

همانطوریکه در شکل ۱۰-۱ ملاحظه می‌کنید یک شی جدید-که در نمودار کلاس وجود نداشته- بنام Job Control اضافه گردیده است. هدف از اضافه این شی جدا نمودن اشیاء کاری از تغیرات در واسط کاربر می‌باشد. بدین صورت سطح استفاده مجدد بالا خواهد رفت. در واقع اضافه این شی نمونه‌ای از مواردی که در آن برای طراحی بهتر سیستم یک سری کلاس‌های کمکی به آن اضافه می‌نماییم. با رسیدن پیام Book Job شی Job Control فعال شده و کنترل بقیه اشیاء را بعهده می‌گیرد. در این شکل واسط کاربر نادیده گرفته شده است. برای اضافه آن باید شی Booking Form اضافه گردد. شکل ۱۰-۲ افزوده شدن واسط کاربر را در نمودار ترتیبی نشان می‌دهد.

در شکل ۱۰-۲ بر اثر درخواست ثبت کار مشتری پیام Book Job به شی Job Control فرستاده شده که به دنبال آن این شی پیام Create به شی Booking Form می فرستد و بدین صورت اجرای مورد کاربری آغاز می‌شود.

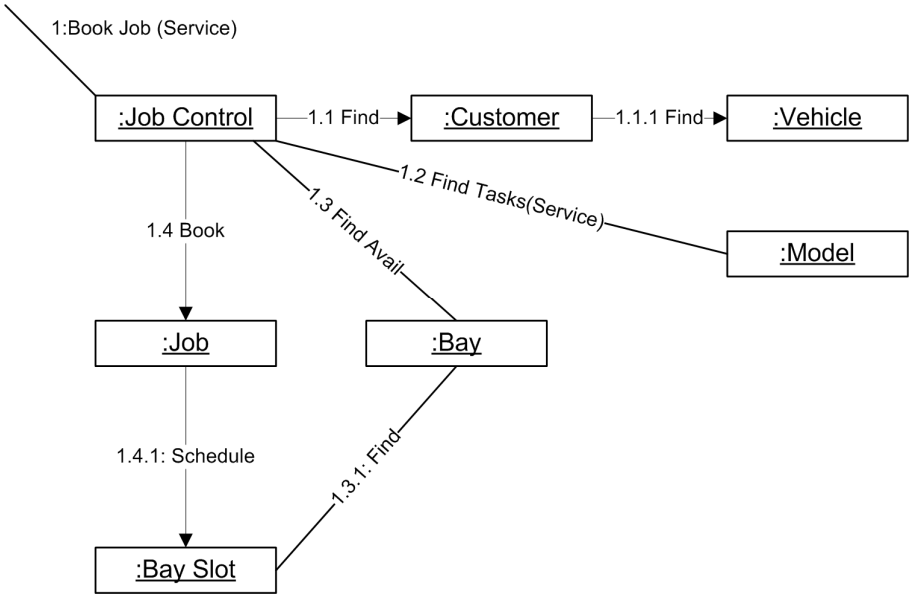


شکل ۱۰-۲- نمودار تعامل مورد کاربری "ثبت کار استاندارد مشتری" با واسط کاربر

۱۰-۳-نمودار همکاری

در این نمودار نیز اشیاء در مستطیل‌ها نمایش داده می‌شوند و فلش‌ها نیز نشان‌دهنده پیام‌های رد وبدل شده بین اشیاء هستند. اما در این نمودارها ترتیب انجام کار با شماره‌بندی پیام‌ها دنبال می‌گردد و شماره بندی بگونه‌ای انجام می‌گردد که هر شماره نشان می‌دهد این عمل از کجا آغاز شده است. در شکل ۱۰-۳ مثالی از یک نموار همکاری دیده می‌شود.

به طور خلاصه می‌توان گفت که از نمودار تعامل در نمایش رفتار اشیاء مختلف در درون یک مورد کاربری استفاده می‌شود. این نمودار همکاری و ارتباط بین اشیاء را بخوبی به تصویر می‌کشد اما وارد جزئیات رفتار اشیاء نمی‌گردد. برای نمایش جزئیات رفتار یک شی در یک مورد کاری باید از مدلسازی حالت استفاده شود. از آنجا که نمودار همکاری نشان‌دهنده نحوه سازماندهی اشیاء می‌باشد، زمینه لازم برای اتخاذ برخی از تصمیمات در رابطه با این سازماندهی فراهم می‌گردد.



شکل ۱۰-۳- نمونه‌ای از یک نمودار همکاری

۱۰-۴-نمودار حالت

هدف از این نمودار، طراحی حالت رفتاری اشیاء است. حالت‌هایی که یک شی در زمان‌های مختلف دارد و ارتباط این حالت‌ها در این نمودار نمایش داده می‌شود. در واقع، این نمودار چرخه زندگی یک شی را نشان می‌دهد. نمودار حالت اغلب برای سه نوع شی مورد استفاده قرار می‌گیرد: الف) اشیایی که حساس به تاریخچه خود هستند ب) اشیایی که دارای پیچیدگی رفتاری هستند و به سادگی قابل درک نیستند. ج) اشیایی که کنترل‌کننده اشیاء دیگر و باید به سیگنال‌های خارجی پاسخ گویند.

قبل از بیان تکنیک‌های مدلسازی حالت به بیان مفاهیم کلیدی آن می‌پردازیم:

- ❖ حالت : برای هر شی تعدادی متناهی از حالت‌ها و جود داشته که از صفات شی و روابط شی با اشیاء دیگر برای نمایش حالت آن استفاده می‌گردد. بنابر این مقادیر فعلی صفات و روابط شی بیانگر حالت فعلی آن می‌باشد. نکته جالب این است که شی اشیاء اصولاً به تاریخچه خود حساسیت دارند بدین معنی که اگر یک شی را ایجاد نمایید و روی آن اعمالی انجام دهید عمل بعدی از نتیجه اعمال گذشته-که در متغیرهای حالت ظاهر می‌شود- متاثر خواهد بود.

❖ انتقال: یک انتقال عبارت از تغییری در حالت شی که بوسیله یک محرک[1] بوجود آمده است. چون حالت‌های گوناگون وجود دارند نیاز به یک عامل تغییر دهنده که باعث انتقال از یک حالت به حالت دیگری می‌گردد، پیدا می‌شود. این عامل(موثر) می‌تواند درخواست انجام یک عمل، یک رویداد، یک شرط نگهبان و یا ترکیبی از اینها باشد. انتقال از هر حالت به حالت دیگر به‌صورت Event [Guard] / Action بیان می‌شود.

❖ شرایط نگهبان[2]: شرطی یا شرایطی که باید محقق گردد تا انتقال مربوطه انجام گیرد.

❖ متغیرهای حالت : مجموعه متغیرهایی که یک حالت را نشان می‌دهند.

۱۰-۵-تکنیک‌های مدلسازی حالت

تکنیک کلی این است که نخست دنبال اشیاء کلیدی رفته و تلاش می نماییم حالات گوناگون شی را پیدا کنیم. باید دنبال الگوهای ساده از رویدادها جستجو نماییم سپس-به صورت تدریجی- الگوهای جایگزین مانند خطاها و استثناها را مطرح می‌کنیم. توجه داشته باشید که یک شی در طول چرخه حیات خود می‌تواند در چندین مورد کاربری شرکت نماید. بعبارت دیگر ما اینجا – بر عکس نمودار تعامل- محدود به موارد کاربری نیستیم. البته از موارد کاربری براس شناسائی حالت‌های اشیاء استفاده می‌کنیم.

گام‌های رسم نمودار حالت عبارتند از:

❖ تعیین و تعریف حالت‌ها

برای هر حالت نیاز است تا خصوصیات تغییرپذیر تعیین شوند. به‌عنوان نمونه باید نشان داده شود که بیشترین تعداد دانشجویانی که می‌توانند درس را انتخاب کنند، ۲۵ است و یا ارتباطی بین حالت درس دادن و پیشنهاد درس وجود وجود دارد.

❖ تعیین رویدادها

تعیین رویدادها از آنجا اهمیت دارد که یک رویداد می‌تواند سبب فعال شدن رویداد دیگر شود و یک فعالیت می‌تواند رویدادی را به شی دیگری ارسال دارد. نمایش کامل رویدادها و تغییراتی

[1] Trigger
[2] Guard

که در حالت‌های یک شی ایجاد می‌شود، بسیار اهمیت دارد، چرا که در صورتی که رویدادی نشان داده نشود بدین معنی است که سیستم در مقابل آن واکنشی تعریف شده ندارد.

❖ تعیین انتقال‌ها

برای هر حالت، باید تعیین شود که چه رویدادی سبب انتقال به چه حالتی می‌شود. در واقع انتقال مشخص می‌کند که در پاسخ به ورود یک رویداد چه اتفاقی می‌افتد و سیستم باید به کدام حالت منتقل شود.

❖ افزودن فعالیت‌ها[1] و عملیات[2]

فعالیت با عملیات از جنبه‌های مختلفی متفاوت هستند. فعالیت‌ها با یک حالت مرتبط هستند در حالیکه عملیات با یک انتقال مرتبط هستند. فعالیت‌ها با ورود به حالت، شروع می‌شوند و برای انجام نیاز به زمان دارند در حالیکه عملیان زمان بسیار کمی نیاز دارند. نکته بسیار مهم در تفاوت این دو، قابلیت توقف فعالیت و غیرقابل توقف بودن عملیات است.

۱۰-۶-نمودار حالت مثال تعمیرگاه

در این مثال کلاس کلیدی همان کلاس کار می‌باشد. با توجه به عملکرد سیستم رویدادهای خارجی و موارد کاربری مربوط به آنها را شناسائی می‌نماییم. شکل ۱۰-۴ فهرست موارد کاربری و رویدادهای مربوط به آنها را در مثال تعمیرگاه نشان می‌دهد.

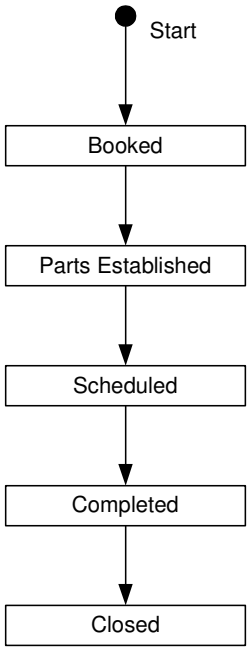
Use-Case	External System Event Name
Book Job for Customer	Job Requested
Establish Parts for Job	Parts Time
Request Parts for Job	Parts Requested
Schedule Job for Day	Schedule Time
Record Job Completion	Job Completed
Customer Arrives	Close Job with Customer

شکل ۱۰-۴- فهرست موارد کاربری و رویدادهای مربوطه

[1] Activities
[2] Activities

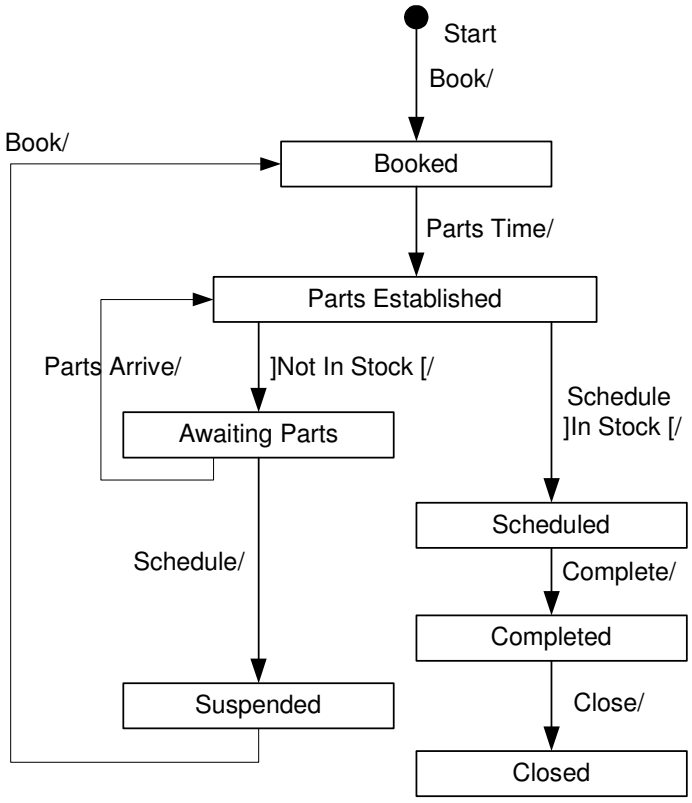
توجه داشته باشید که رویداد Parts Requested هنگامی رخ می‌دهد که کنترل کننده قطعات اعلام نیاز نماید(زیرا قطعات در تعمیرگاه وجود ندارند). نمودار اولیه حالت در شکل ۱۰-۵ نمایش داده شده است. در این سیستم فرض بر این است که "کار" هنگامی زمانبندی می‌گردد که تمام قطعات مورد نیاز در تعمیرگاه باشد. این نیاز باید به صورت صریحی در نمودار ذکر شود، لذا از شرط نگهبان [In Stock] استفاده می‌نماییم (بین حالت‌های Parts Established و Scheduled). همچنین از یک شرط نگهبان دیگر [Not in Stock] استفاده می نماییم.(بین حالت‌های Parts Established و Awating Parts) (شکل ۱۰-٦)

نکته آخر این است که رویداد Parts Requested روی حالت شی Job اثر نمی گذارد. در واقع این شی به رویداد دیگری علاقمند می‌باشد که همان رسیدن قطعات مورد نیاز(Parts Arrive) تا بتواند خود را زمانبندی نماید. بنابر این رویداد رسیدن قطعات مورد نیاز به لیست رویدادها اضافه می‌گردد.



شکل ۱۰-۵- نمودار اولیه حالت برای شی کار

اگر زمانبندی کار فرارسد و قطعات مورد نیاز هنوز تهیه نشده است انجام کار باید به تعویق بیافتد(حالت Suspended) تا دوباره در چرخه ثبت یک کار شرکت نماید.



شکل ۱۰-٦- نمودار نهائی حالت برای شی کار

۱۰-۷-نمودار فعالیت

فلوچارتی است که جریان کنترل را از یک فعالیت به فعالیت دیگر نمایش می‌دهد. نمودار تعامل بر روی مدلسازی جریان‌های کنترلی میان اشیاء تاکید نموده درحالیکه نمودار فعالیت بر روی مدلسازی جریان کنترلی میان فعالیت‌ها که هر کدام منتسب به یک شی هستند، تاکید می‌کند. نمودار فعالیت نوع ویژه‌ای از نمودار حالت محسوب می‌شود که موارد استفاده آن در مدلسازی یک گردش کار[1] و مدلسازی یک عمل[2] است.

قبل از بیان مراحل ایجاد نمودار فعالیت برخی مفاهیم مرتبط با نمودار فعالیت را بیان می‌کنیم.

❖ فعالیت: فرآیند محاسباتی پیوسته و تجزیه‌پذیری که در یکی از حالات ماشین حالت اجرا می‌شود. گاهی نیاز است که یک فعالیت را تجزیه‌ناپذیر تعریف نماییم. آنگاه به فعالیت

[1] Workflow Modeling
[2] Operation Modeling

معمولی «حالت فعالیت»[1] گفته و به فعالیت تجزیه ناپذیر، «حالت کنش»[2] که نشاندهنده اجرای یک کنش است، گفته می‌شود.

- ❖ انتقال: هنگامیکه یک فعالیت خاتمه می‌یابد، کنترل بلافاصله به یک حالت دیگر انتقال پیدا می‌کند.
- ❖ انشعاب و ادغام: برای مدلسازی جریان‌های همزمان می‌توان از سطح همگام سازی استفاده نمود
- ❖ Swimlane: انجام‌دهنده فعالیت را نشان می‌دهد.

۱۰-۸-مراحل ایجاد نمودار فعالیت

مراحل ایجاد نمودار فعالیت به‌صورت زیر نشان داده می‌شود:

- ❖ گردش‌کار مورد نظر را تعیین نمایید. در یک سیستم واقعی مدلسازی همه گردش کارها در یک نمودار امکان‌پذیر نیست
- ❖ اشیاء حرفه کلیدی را انتخاب نمایید. این اشیاء می‌توانند موجودیت‌های واقعی که از واژگان سیستم استخراج شده یا می‌توان در سطح تجریدی بالاتری قرار گیرند، باشند
- ❖ برای هر شی حرفه کلیدی یک Swimlane ایجاد نمایید
- ❖ پیش‌شرط‌های حالت ابتدائی و پس‌شرط‌های حالت پایانی را شناسائی نمایید. بدین‌صورت مرزهای گردش کار معین می‌گردد
- ❖ از حالت ابتدائی آغاز نمایید و فعالیت‌هایی که در طول زمان صورت می‌گیرند با توجه به Swimlane مربوطه در نمودار فعالیت ترسیم نمایید
- ❖ بمنظور بیان رفتار فعالیت‌های پیچیده برای هرکدام یک نمودار فعالیت جداگانه ترسیم نمایید
- ❖ انتقال بین فعالیت‌ها را ترسیم نمایید. از جریان‌های ترتیبی آغاز کرده، سپس جریان‌های شرطی و بالاخره به ترسیم جریان‌های موازی (به صورت انشعاب و ادغام) بپردازید

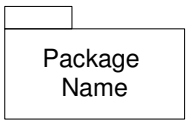
[1] Activity State
[2] Action State

۱۱- مدلسازی مولفه‌ها و استقرار

هنگام مدلسازی سیستم‌های نرم‌افزاری بزرگ به روش شی‌گرا، معمولاً با مدل‌هایی که حاوی تعداد بسیار زیادی از عناصر مدلسازی مانند کلاس‌ها هستند، روبرو خواهیم شد. در این موارد نیاز است تا کلاس‌ها به‌گونه‌ای دسته‌بندی و مدیریت شوند. درک و مدیریت صحیح مدل‌ها مستلزم سازماندهی عناصر این مدل‌ها در گروه‌های بزرگتر است تا بتوان تا حد ممکن پیچیدگی را کاهش داد. برای سازماندهی کلاس‌ها در یک گروه بزرگتر از مفهوم بسته[1] استفاده می‌شود.

۱۱-۱-بسته

بسته، مکانیزمی عمومی برای سازماندهی عناصر مدلسازی در گروه‌های بزرگتر است تا بدین وسیله پیچیدگی وجود و ارتباط کلاس‌های زیاد در نرم‌افزارهای بزرگ کاهش یابد. هر بسته می‌تواند حاوی مجموعه‌ای از کلاس‌ها، موارد کاربری و یا هر نوع عنصر مدل‌سازی دیگر باشد که برای کاهش پیچیدگی نیاز به دسته‌بندی و مدیریت دارند. همچنین بسته‌ها می‌توانند شامل بسته‌های دیگر باشند. شکل ۱۱-۱ نحوهٔ نمایش بسته را نشان می‌دهد.



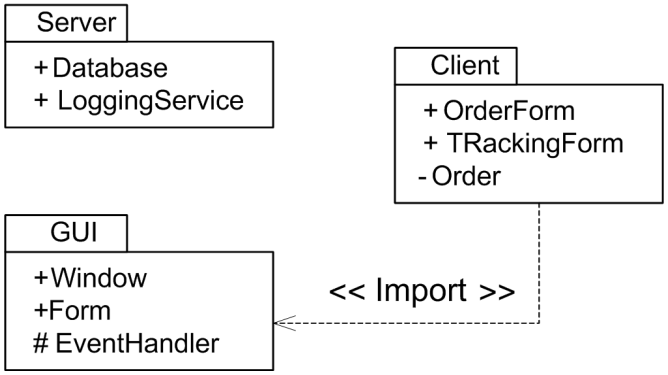
شکل ۱۱-۱-نمایش بسته

۱۱-۱-۱- مفاهیم مرتبط با بسته‌ها

برخی از مفاهیم مرتبط با کلاس برای بسته نیاز صادق است. عناصر بسته می‌توانند دارای قابلیت دستیابی عمومی یا محافظت‌شده یا خصوصی داشته باشند. عناصر عمومی بسته می‌توانند توسط عناصر خارج بسته دستیابی شوند، اما عناصر خصوصی تنها می‌توانند توسط عناصر داخل بسته مورد استفاده قرار گیرند. عناصر با خصوصیت محافظت شده تنها در بسته‌هایی فرزند قابل دسترسی هستند.

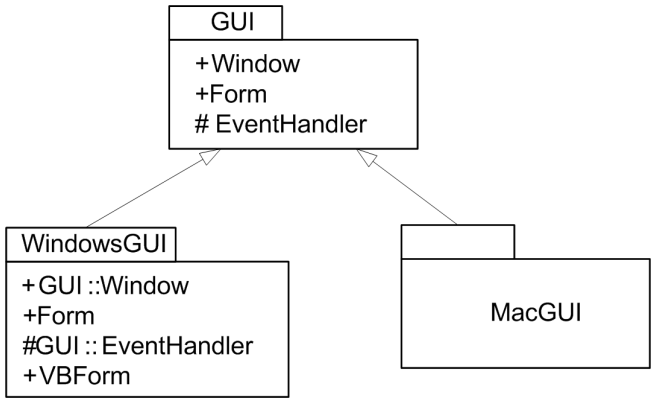
[1] Package

برخلاف موارد کاربری که با استفاده از روابط نظیر uses، include از یکدیگر استفاده می‌کنند، بسته‌ها برای استفاده از یکدیگر از مفهوم «وارد کردن»[1] استفاده می‌کنند. در صورتیکه عنصری از یک بسته به عنصری از بسته دیگر نیاز داشته باشد، باید بسته دوم وارد شود. به‌عنوان مثال، فرض کنید دو کلاس الف و ب در دو بسته جداگانه وجود داشته باشند و هرکدام قابلیت دید «عمومی» را دارند. برای ینکه کلاس الف به کلاس ب دسترسی داشته باشد، باید بسته حاوی کلاس ب توسط بسته حاوی کلاس الف «وارد» شود. شکل ۱۱-۲ نمونه‌ای از وارد کردن بسته GUI توسط بسته Client را نشان می‌دهد. علامت‌های +، - و # قبل از عنصر نشان‌دهندهٔ قابلیت دستیابی عنصر است.



شکل ۱۱-۲- وارد کردن بسته GUI توسط بسته Client

شکل ۱۱-۳ نمونه‌ای از رابطه عام-خاص بین بسته‌ها را نشان می‌دهد. بسته‌ها در این زمینه رفتاری همانند کلاس‌ها دارند.



شکل ۱۱-۳- رابطه عام-خاص در بسته‌ها

[1] Import

علاوه بر «وارد کردن» ارتباط وابستگی[1] نیز برای بسته‌ها معنادار است. اگر بسته‌ای از سرویس‌های بسته دیگری استفاده نماید، گویند بسته اولی به بسته دوم وابسته است. این وابستگی با یک فلش جهت دار نمایش داده می‌شود. شکل ۱۱-۴ نمونه‌ای از رابطه وابستگی بین بسته‌ها را نمایش می‌دهد.

۱۱-۱-۲- کاربرد بسته‌ها در مدلسازی

بسته‌ها دو کاربرد رایج در مدلسازی دارند:

❖ مدلسازی گروه‌هایی از عناصر

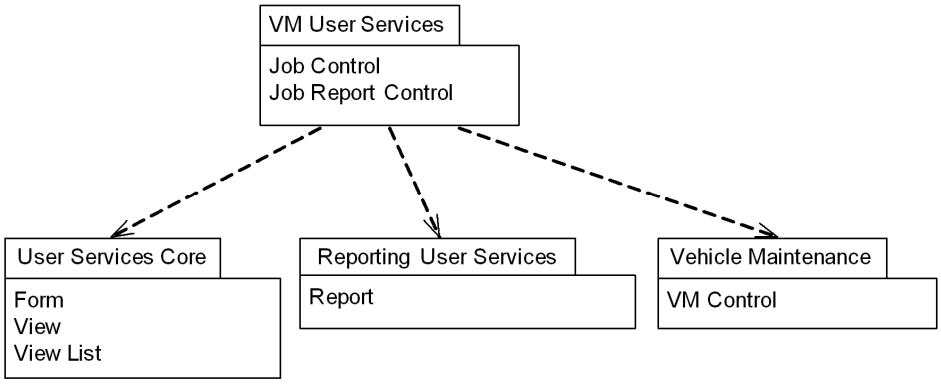
رایجترین استفاده از بسته‌ها در گروه‌بندی عناصر مدلسازی منطقاً مرتبط است به صورتیکه بتوان با هر بسته بعنوان یک واحد رفتار کرد. در این مورد بسته‌ها با کلاس‌ها تفاوت دارند و هر بسته قابلیت در برگیری مجموعه‌ای از عناصر را دارد و می‌تواند شامل بسته‌های دیگر باشد در حالیکه کلاس‌ها این قابلیت را ندارند.

❖ مدلسازی دیدهای معماری

از مهمترین کاربردهای بسته‌ها در مدلسازی، استفاده از آنها به عنوان نمایش‌دهندهٔ عناصر موجود در دید معماری است. برخی مفاهیم معماری همچمون لایه دارای نماد در UML نیستند و از بسته به‌جای آنها استفاده می‌شود. علاوه بر این با استفاده از مفهوم بسته که توانایی نمایش سطح بالاتری از تجرید را دارد می‌تواندبرخی دیدهای معماری را آسان‌تر نمایش داد.

به‌عنوان نمونه می‌توان به مثال تعمیرگاه اشاره نمود که با استفاده از مفهوم بسته می‌توان کلاس‌ها را سازماندهی نمود. شکل ۱۱-۴ نمونه استفاده از بسته برای مثال تعمیرگاه را نشان می‌دهد. چهار بسته که هر یک حاوی کلاس‌های مرتبط با خود هستند. User Service Core حاوی سه بسته برای نمایش و دریافت اطلاعات است. Reporting User Services شامل سرویس Report است که گزارش خدمات را ارائه می‌دهد. VM User Services شامل مجموعه‌ای از بسته‌های کنترل کار و کنترل گزارش کار است که کلاس‌های کنترلی هستند. همچنین سرویس کاری نگهداری ماشین[2] بوسیله یک بسته مستقل فراهم شده است.

[1] Dependency
[2] Vehicle Maintenance or VM



شکل ۱۱-۴- نمونه استفاده از بسته در مثال تعمیرگاه

۱۱-۲-مدلسازی مولفه

همانطوری‌که در بخش‌های قبلی بیان شد، روش موارد کاربری برای استخراج و نمایش نیازهای وظیفه‌مندی یک سیستم به کار می‌رود. بنابراین یک برنامه کاربردی از تعدادی موارد کاربری-که در مدل موارد کاربری دیده می‌شوند- تشکیل می‌شود. حال، بهترین شیوه پیاده‌سازی این موارد کاربری چیست؟ جواب این سوال در روشی که برای مدلسازی منطقی بکار برده می‌شود، نهفته است. در سیستم‌های بسیار ساده نیاز به مدلسازی منطقی وجود ندارد اما در سیستم‌های واقعی نقش مدلسازی منطقی بسیار کلیدی است. مدلسازی منطقی در UML توسط کلاس‌ها انجام می‌شود و مدلسازی فیزیکی توسط مولفه‌ها انجام می‌شود.

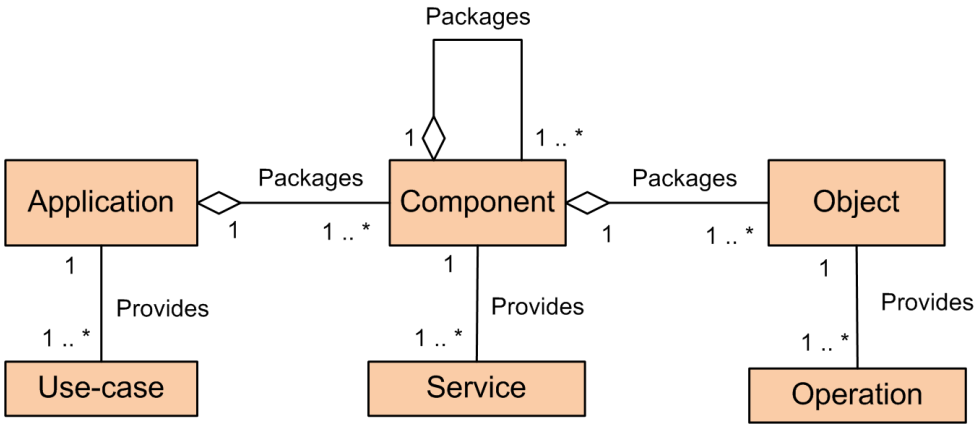
یکی از سوالاتی که از دیر باز در متدولوژی‌های نرم‌افزاری مطرح بوده است عبارتست از اینکه چگونه یک سیستم بزرگ را به سیستم‌های کوچکتر تقسیم کرد؟

متدولوژی‌های شی‌گرا با مطرح کردن مفهوم کلاس تلا ش نمودند به این سوال پاسخ دهند. همچنین به نظر سردمداران تکنولوژی شی‌گرائی کلاس واحد استفاده مجدد-که بزرگترین مزیت این روش حساب می‌شد-نیز هست. به نظر می‌رسد که در سیستم‌های نرم‌افزاری بسیار بزرگ استفاده تنها از مفهوم کلاس بعنوان واحد تجزیه به تنهایی کافی نیست پس به یک ساختار حجیم‌تر هم در سطح طراحی و هم در سطح پیاده‌سازی نیاز است. نکته دوم این است که مکانیزم‌های استفاده مجدد که در زبان‌های برنامه‌نویسی شی‌گرا مطرح است محدود به برنامه‌های خود زبان می‌باشند. بدین علت زبان‌های شی‌گرا در مجسم کردن ایده IC نرم‌افزاری ناموفق عمل کرده است(البته تا حدودی).

۱۱-۲-۱- مفهوم مولفه

یک مولفه عبارتست از قطعه‌ای- معمولاً دودویی- از کد که پیاده‌سازی آن مخفی بوده و می‌توان از آن در نرم‌افزارهایی متفاوت-بوسیله واسط آن- استفاده نمود. بعبارت دیگر مولفه در نرم‌افزار متناظر با IC در سخت‌افزار می‌باشد.

شکل ۱۱-۵ رابطه بین یک برنامه کاربردی، موارد کاربری، مولفه و اشیاء آن ر ا نمایش می‌دهد. مولفه به‌عنوان واحد پیاده‌سازی با اشیاء به‌عنوان واحد منطقی ارتباط تنگاتنگی دارد و حاوی یک یا چند شی است. به‌عبارت بهتر کلاس‌های منطقی با اشیاء که واحد فیزیکی هستند، پیاده‌سازی می‌شوند و مجموعه‌ای از کلاس‌ها که در یک بسته قرار می‌گیرند می‌توانند با یک مولفه پیاده‌سازی شوند.

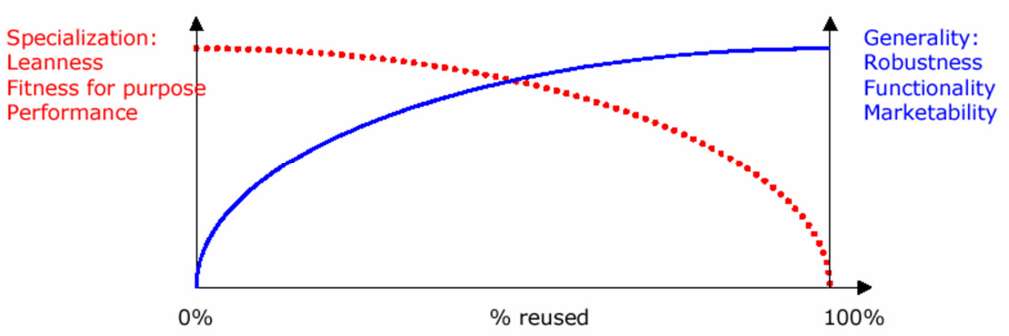


شکل ۱۱-۵- ارتباط مولفه با سایر عناصر موجود در یک برنامه کاربری

با توجه به این مقدمه ارزش مولفه‌ای معلوم می‌گردد چراکه مولفه‌ها با گروه‌بندی کلاس‌های منطقاً مرتبط در سطح طراحی(Packages) و پیاده‌سازی(Components) ساختار حجیم‌تر مورد نیاز را فراهم کرده و با پیروی از یک استاندارد خاص در تعریف واسط‌های خود به ایده IC نرم‌افزاری جامع عمل پوشانده است. بدین صورت می‌توان گفت که بهتر است موارد کاربری به صورت مولفه‌های قابل استفاده مجدد(که فراهم کننده سرویس هستند) پیاده‌سازی شوند. هر مولفه از تعدادی کلاس منطقاً مرتبط تشکیل می‌شوند.

۱۱-۲-۲- استفاده مجدد و پیاده‌سازی مولفه‌ها

میزان استفاده از مولفه‌ها در نرم‌افزار بسته به میزان عمومی‌سازی یا خصوصی‌سازی دارد. هر چه میزان عمومی‌سازی نرم‌افزار بیشتر باشد کارکرد، توانمندی و پذیرش بازار نرم‌افزار افزایش خواهد یافت. همچنین استفاده مجدد نرم‌افزار تا حد ممکن بالا خواهد رفت. اما در مقابل نرم‌افزار اهداف را به‌طور کامل برآورده نساخته، کارایی آن پایین آمده و نرم‌افزار حجیم‌تر ایجاد خواهد شد. شکل ۱۱-۶ توازن بین عمومی‌سازی و خصوصی‌سازی را نشان می‌دهد.



شکل ۱۱-۶- ایجاد توازن در استفاده از مولفه‌ها برای استفاده مجدد

با کاهش استفاده مجدد در نرم‌افزار و خصوصی‌سازی نرم‌افزار، کارایی افزایش می‌یابد. همچنین نرم‌افزار اهداف را به‌طور کامل برآورده خواهد ساخت و کم حجم‌تر خواهد شد. اما کارکرد و پذیرش بازار نرم‌افزار کاهش می‌یابد. توازن بین خصوصی‌سازی و عمومی‌سازی نرم‌افزار نشان‌دهندهٔ میزان قابلیت استفاده مجدد مولفه‌ها است و تعیین این میزان برعهده معمار نرم‌افزار است.

در پیاده‌سازی مولفه‌ها می‌توان به‌ازای هر مولفه یک یا چند واسط پیاده‌سازی نمود و آنها را با مولفه مرتبط ساخت. این روش قابلیت استفاده مجدد از مولفه‌ها را افزایش می‌دهد. همچنین هر زیرسیستم در طراحی به یک یا چند مولفه در پیاده‌سازی تبدیل می‌شوند.

۱۱-۲-۳- برنامه‌نویسی شی‌گرا[1] و توسعه برمبنای مولفه[2]

هدف برنامه‌نویسی شی‌گرا ایجاد یک سیستم نرم‌افزاری متشکل از اشیاء مرتبط است، در حالیکه هدف توسعه برمبنای مولفه ایجاد یک سیستم متشکل از تعدادی مولفه مرتبط است.

[1] Object-Oriented Programming (OOP)
[2] Component-based Development (CBD)

در واقع، هر مولفه در پایین‌ترین سطح، شامل مجموعه‌ای از اشیاء است. مولفه‌ها و کلاس‌ها دارای ویژگی‌های مشترک هستند که در ذیل بیان شده‌اند:

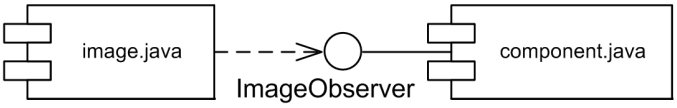
- دارای اسم هستند
- می‌توانند مجموعه‌ای از واسط‌ها را پیاده‌سازی نمایند
- می‌توانند در روابط وابستگی، تعمیم پذیری و تجمعی شرکت نمایند
- قابلیت تو در تو بودن دارند
- می‌توانند در تعاملات شرکت کنند

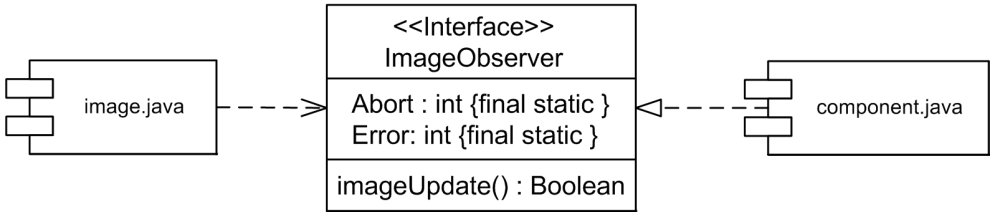
تفاوت‌های بین مولفه‌ها و کلاس‌ها بدین شرح است:

- کلاس‌ها ساختار منطقی سیستم را تشکیل داده در حالیکه مولفه‌ها ساختار فیزیکی سیستم را تشکیل می‌دهند
- مولفه‌ها سطح تجرید بالاتری از کلاس‌ها را نمایش می‌دهند
- یک کلاس می‌تواند دربردارنده صفات و اعمالی که برای سرویس گیرندگان آن، به صورت مستقیم دسترسی‌پذیرند باشد، در حالیکه ساختار داخلی یک مولفه کاملاً کپسوله است

۱۱-۲-۴- نقش واسط در مولفه‌ها

واسط عبارتست از مجموعه‌ای از اعمال که سرویس‌های ارائه شده بوسیله یک کلاس یا یک مولفه را مشخص می‌کند. هر مولفه می‌تواند دارای یک یا بیشتر واسط باشد. در شکل ۱۱-۷ مولفه component.java دارای واسطی به‌نام ImageObserver است که مولفه image.java از آن استفاده می‌کند.





شکل ۱۱-۷- استفاده از واسط برای مولفه و نحوهٔ پیاده‌سازی آن

استفاده از واسط توسط رابطه وابستگی نمایش داده شده است. پیاده‌سازی این واسط در قسمت پایین نمایش داده شده است. ImageObserver به صورت یک کلاس که حاوی دو متغیر و متد imageUpdate است، بیان شده و ارتباط بین component.java و واسط، رابطه «عینیت بخشیدن» می‌باشد.

۱۱-۲-۵- جایگزینی‌پذیری مولفه‌ها[1]

هدف بیشتر ابزارهای مبتنی بر مولفه‌ها فراهم نمودن محیطی است که در آن یک سیستم نرم‌افزاری بوسیله تجمیع تعدادی از قطعات باینری جایگزین‌پذیر ایجاد گردد. مولفه‌ها به‌عنوان یک واحد مستقل می‌توانند در صورت نیاز جایگزین یکدیگر شوند و بدین ترتیب قابلیت ارتقا و بهبود را برای یک نرم‌افزار فراهم سازند. از طرفی نرم‌افزاری که با استفاده از مولفه‌ها توسعه داده شده باشد می‌تواند وظایف و فعالیت‌ها خود را در صورت نیاز توسعه دهد که البته این به نحوهٔ طراحی نرم‌افزار وابسته است.

۱۱-۲-۶- انواع مولفه‌ها

سه نوع مولفه‌ها در UML وجود دارد که عبارتند از:

- مولفه‌های استقرار[2]: این نوع مولفه‌ها برای اجرای سیستم مورد نیاز هستند. از جمله این مولفه‌ها می‌توان با JVM برای نرم‌افزارهای برپایه جاوا و یا مولفه‌های مربوط به DBMS برای نرم‌افزارهای پایگاه داده‌ای اشاره نمود.
- مولفه‌های محصول کاری[3]: شامل مدل‌ها، کدهای منبع و فایل‌های داده که برای ایجاد مولفه‌های استقرار مورد نیاز هستند.
- مولفه‌های اجرائی[4]: این نوع مولفه‌ها در زمان اجرای نرم‌افزار ایجاد می‌شوند. از جمله این مولفه‌ها می‌توان به مولفه‌های +COM، NET. و CORBA اشاره نمود.

مولفه‌ها این قابلیت را دارند که در بسته قرار گیرند و یک یا چند مولفه می‌توانند یک بسته را ایجاد کنند. در این صورت، از روابط وابستگی، تعمیم‌پذیری و تجمعی میان مولفه‌ها استفاده می‌شود تا بین مولفه‌ها ارتباط برقرار شود.

---

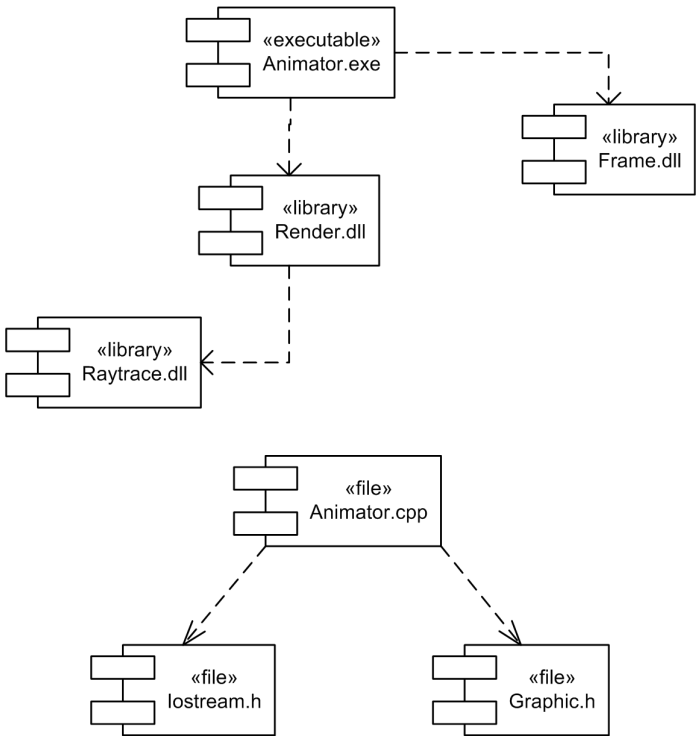
[1] Binary Replaceability
[2] Deployment Components
[3] Work Product Components
[4] Execution Components

۱۱-۲-۷- روش‌های مدلسازی مولفه‌ها

برای مدلسازی مولفه‌ها از روابطی که در قسمت‌های قبلی بیان گردید همانند وابستگی، انجمنی و ... استفاده می‌شود تا ارتباط بین مولفه‌ها نمایش داده شود. در مواردی همچون مدلسازی فایل‌های اجرایی و کتابخانه‌ای می‌بایست برخی قواعد را در مدلسازی رعایت نمود که در ذیل بیان می‌شوند:

- ❖ نحوه تقسیم‌بندی سیستم فیزیکی را با توجه به فاکتورهای تکنیکی، مدیریت پیکربندی و قابلیت استفاده مجدد را مشخص نمایید
- ❖ هر فایل اجرائی یا کتابخانه را به صورت یک مولفه مدلسازی نمایید
- ❖ در صورت لزوم واسط‌های کلیدی سیستم را معین کنید
- ❖ روابط موجود بین فایل‌های اجرائی، کتابخانه‌ها و واسط‌ها را مدلسازی نمایید

در مدلسازی جداول پایگاه داده‌ها، فایل‌های داده‌ای و مستندات، APIها و کدهای منبع نیز چنین قواعدی می‌بایست رعایت شود. شکل ۱۱-۸ دو نمونه از مدلسازی مولفه‌های اجرایی و کد منبع را نشان می‌دهد.

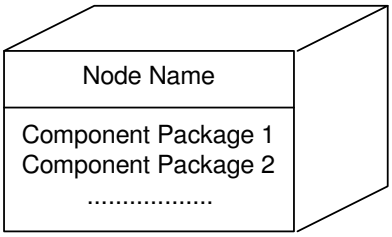


شکل ۱۱-۸- دو نمونه مدلسازی مولفه

۱۱-۳-مدلسازی استقرار

مدلسازی استقرار با استفاده از نمودار استقرار انجام می‌شود. این نمودار ارتباطات فیزیکی ما بین اجزاء سخت‌افزار و نرم‌افزاری یک سیستم را نشان می‌دهد. یک نمودار استقرار برای نشان دادن چگونگی پراکندن اشیاء روی گره های یک سیتسم توزیع شده بکار می‌رود.

در نمودار استقرار یک مفهوم به نام گره[1] وجود دارد. گره، عنصری فیزیکی است که در زمان اجرا وجود داشته و یک منبع محاسباتی را نمایش داده و به صورت کلی دارای حافظه و قابلیت پردازش است. در UML گره‌ها با نماد یک باکس نشان داده می‌شوند. (شکل ۱۱-۹)



شکل ۱۱-۹- نمایش یک گره در UML

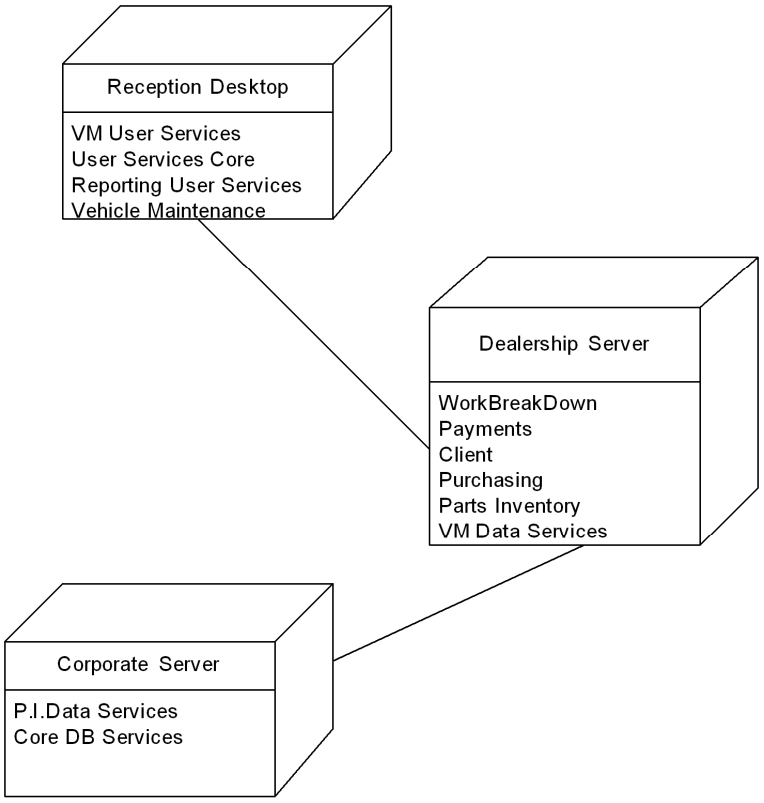
گره‌ها را می‌توان را در بسته‌ها گروه‌بندی نمود و آنها را سازماندهی نمود. همچنین می‌توان از روابط وابستگی، تعمیم‌پذیری و تجمعی میان آنها استفاده کرد. اما رایجترین رابطه میان گره‌ها، رابطه انجمنی است که معمولاً نمایش‌دهنده یک ارتباط فیزیکی میان گره‌ها مانند ارتباط Ethernet، گذرگاه مشترک یا حتی پیوند ماهواره‌ای باشد.

برای مدلسازی استقرار رعایت قواعد زیر می‌تواند منجر به توسعه نمودارهای استقرار مناسب‌تری شود:

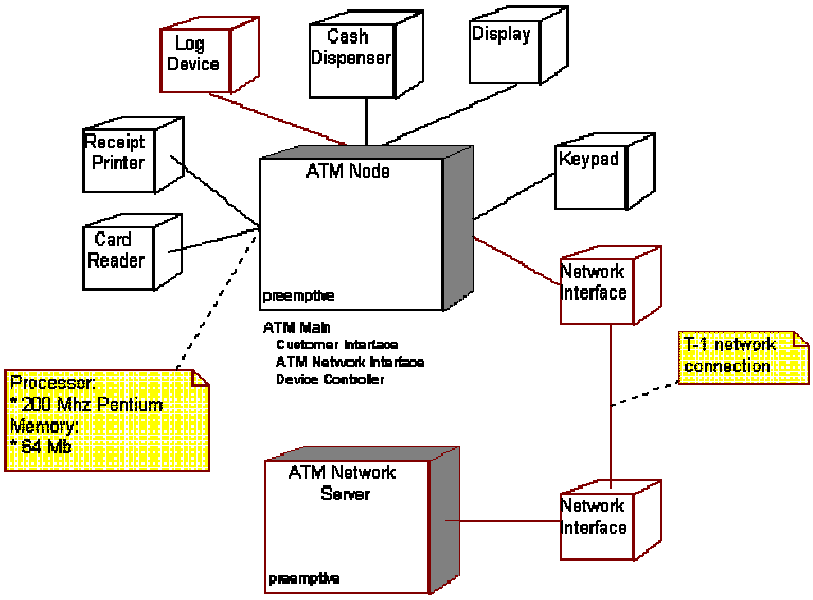
- هرکدام از منابع محاسباتی سخت‌افزاری که دید استقرار را تشکیل می‌دهند به صورت یک گره مدلسازی نمایید
- اگر این منابع، پردازشگرها و یا دستگاه‌هایی مخصوصی باشند که جزئی از واژگان سیستم مورد نظر را تشکیل می‌دهند، از مکانیزم مقوله‌بندی برای بیان آنها استفاده نمایید
- صفات و اعمال هر گره را مشخص نمایید

شکل ۱۱-۹ و شکل ۱۱-۱۰ دو نمونه نمودار استقرار را نشان می‌دهد.

[1] Node



شکل ۱۱-۹- نمودار استقرار سیستم نگهداری خودرو(شامل سرویس و تعمیر و ...)



شکل ۱۱-۱۰- نمودار استقرار نمونه برای ATM

۱۲- روش‌های سریع‌الانتقال (چابک) توسعه نرم‌افزار

در فصل یک در مورد فرآیند توسعه نرم‌افزار و اینکه این فرآیند مجموعه‌ای از فعالیت‌هایی است که نظم از پیش تعریف شده‌ای ندارند و و برای نظم دادن به این فرآیند، متدولوژی‌های توسعه نرم‌افزار مطرح شدند. هر متدولوژی توسعه نرم‌افزار مشخص می‌کند، چه فرآورده‌ای، توسط چه کسی (نقشی) و در چه زمانی تولید شود. هم‌چنین در بخش ۵-۳ در مورد انواع متدولوژی‌های سبک و سنگین نرم‌افزار صحبت شد و اینکه تمام متدولوژی‌های توسعه نرم‌افزار را می‌توان در این دو دسته قرار داد و در فصل‌های گذشته در مورد متدولوژی RUP به‌عنوان یکی از متدولوژی‌های سنگین مطالب متنوعی بیان گردید و این متدولوژی با جزئیات تشریح گردید.

در این بخش به بررس متدولوژی‌های سبک‌وزن و به خصوص متدولوژی‌های چابک خواهیم پرداخت. اما قبل از بررسی این متدولوژی‌های به بیان مشکلات متدولوژی‌های سنگین‌وزن و مقایسه این دو گونه متدولوژی توسعه نرم‌افزار خواهیم پرداخت.

۱۲-۱-متدولوژی سنگین وزن

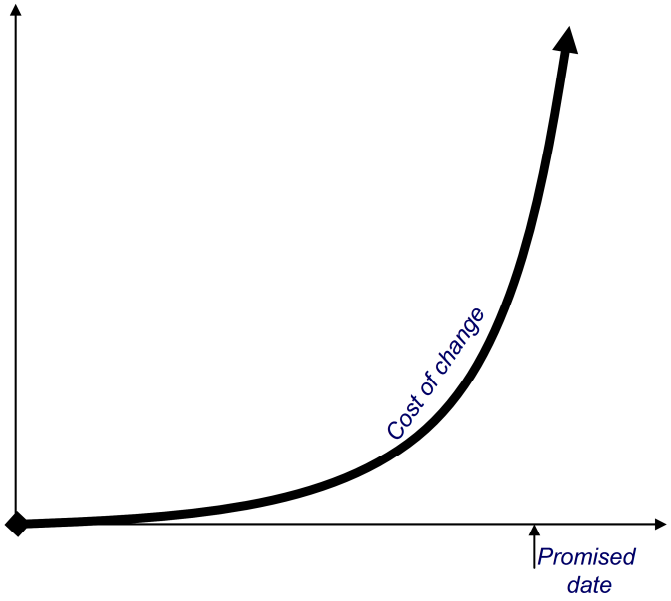
در ۲۵ سال اخیر روش‌های بسیار زیادی برای توسعه نرم‌افزار معرفی شدند اما امروزه تعداد بسیار اندکی از آنها مورد استفاده قرار می‌گیرد. این موضوع تداعی‌گر این مطلب است که متدولوژی‌های توسعه نرم‌افزاری که ارائه می‌شدند، دارای مشکلاتی بودند که آنها را غیرقابل اجرا یا حداقل اجرای آنها را دشوار می‌ساخت و به‌همین دلیل اغلب آنها کمتر مورد استفاده قرار گرفته‌اند. شاید بتوان در یک عبارت کوتاه و کلی «متدولوژی‌های فعلی توسعه نرم‌افزار را بیش از اندازه ماشین‌گرا و مکانیزه دانست». این متدولوژی‌ها بصورت فرآیندی[1] وارد جزئیات غیرضروری می‌شوند و البته به همین دلیل این نوع متدولوژها را سنگین وزن می‌نامند. همین امر (ورود به جزئیات غیرضروری) سبب می‌شود که مشکلات این نوع متدولوژی‌های در اجرا بیشتر نمایان شود، از جمله اینکه:

- مشتریان نرم‌افزارها حاضر نیستند که برای دست یافتن به نرم‌افزارهای مورد نیاز خود مدت زیادی منتظر بمانند. هرچند به‌عنوان تیم توسعه‌دهنده نرم‌افزار می‌توان به مشتری قبولاند که

[1] فرآیند این نوع متدولوژی‌ها بدین صورت است که آنها را وارد جزئیات می‌کند

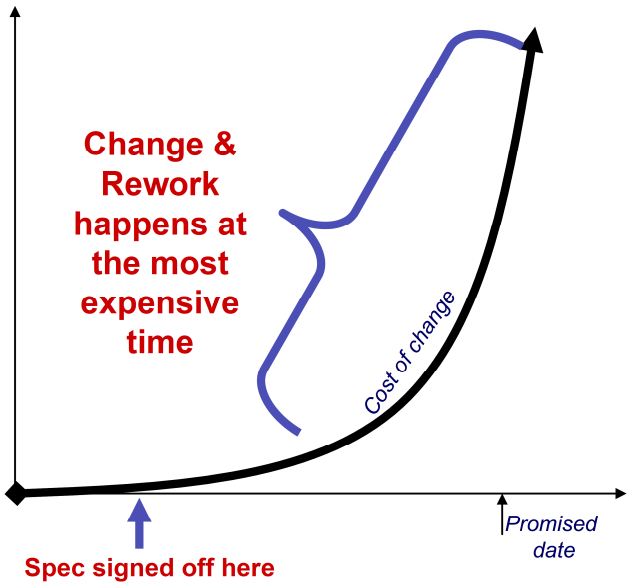
این انتظار خوب است و محصول با کیفیت‌تری تحویل می‌گردد، اما در واقع، اغلب مشتریان راه‌حل بهتر را در زمان کمتر ترجیح می‌دهند و به مستند نمودن و فعالیت‌های افزونه‌ای که در جریان توسعه نرم‌افزار توسط تیم توسعه نرم‌افزار انجام می‌شود، اهمیت نمی‌دهند. آنها ترجیح می‌دهند که این کار پس از توسعه نرم‌افزار و زمانی که نرم‌افزار به آنها تحویل داده شد، انجام شود!

- رقابت بسیار شدید شرکت‌های تولید نرم‌افزار برای ارائه خدمات نرم‌افزاری به کاربران سبب می‌شود که استفاده از متدولوژی‌های سنگین وزن به‌صرفه نباشد. ایجاد انبوهی از مستندات که خواندن آنها برای توسعه بعدی نرم‌افزار، زمان زیادی را از تیم می‌گیرد. اغلب شرکت‌ها توسعه نرم‌افزار نیاز دارند تا توسعه‌های سریع‌تر و با کیفیت‌تر داشته باشند تا توسعه‌های پر مستندتر و با کیفیت‌تر!
- تغییرپذیری بسیار زیاد نرم‌افزارهای امروزی انکارناپذیر است. با افزایش مستندات حجم کاری که برای بروز رسانی مستندات نیاز است نیز بیشتر خواهد شد. با پیشرفت در پروژه و ایجاد تغییرات گسترده در مستندات، بروز نگهداری این مستندات به صورت تصاعدی افزایش می‌یابد. شکل ۱۲-۱ میزان تغییرات را در نرم‌افزار از ابتدای توسعه نرم‌افزار تا زمان تحویل نرم‌افزار را نشان می‌دهد.



شکل ۱۲-۱- میزان رشد تغییرات از ابتدای توسعه نرم‌افزار تا زمان تحویل

توسعه‌دهندگان توسعه نرم‌افزار خصوصا توسعه‌دهندگانی که از متدولوژی‌های سنگین وزن استفاده می‌کنند با تغییرات به‌شدت مقابله می‌کنند تا بتوانند در زمان تعیین شده نرم‌افزار را تحویل دهند. اما نکته بسیار مهمتر، زمان این تغییرات است که بیشتر در انتهای پروژه است، زمانیکه تیم‌های توسعه نرم‌افزار برای جبران کمبود زمانی بیشترین حجم کاری را دارند[1]. شکل ۱۲-۲ میزان رشد تغییرات در انتهای پروژه را نشان می‌دهد و اینکه تغییرات نرم‌افزار در زمانی درخواست می‌شود (انتهای پروژه) که بیشترین هزینه را برای شرکت توسعه‌دهنده نرم‌افزار دارد.



شکل ۱۲-۲- میزان رشد تغییرات در انتهای پروژه

تغییرات درخواست مشتریان در توسعه نرم‌افزار امری بدیهی است و اغلب متدولوژی‌های توسعه نرم‌افزار روش‌هایی برای در نظر گرفتن این تغییرات ارائه می‌دهند. تغییرات نرم‌افزار (نه لزوماً همه تغییرات) سبب می‌شوند که کیفیت نرم‌افزار بهبود یافته و خواسته‌های مشتری بیشتر برآورده شود. اما از طرفی تغییرات (نه لزوماً همه تغییرات) سبب افزایش زمان توسعه نرم‌افزار می‌شوند و به‌همین دلیل توسعه‌دهندگان با آن تاحد ممکن مقابله می‌کنند و سعی می‌کنند نرم‌افزار را در زمان پیش‌بینی شده ارائه دهند. در واقع وقتی پروژه‌ای موفق است که از طرفی محصول با کیفیتی ارائه شود و از طرفی در زمان

[1] واقعیت این است که کمتر اتفاق می‌افتد که تیم توسعه نرم‌افزار از زمانبندی خود جلو باشد و در اسرع وقت نرم‌افزار را تحویل دهد.

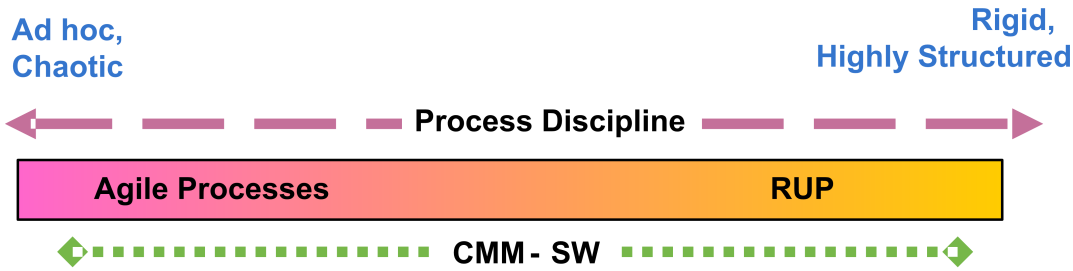
پیش‌بینی شده ارائه شود که اولی به‌معنی پذیرفتن تغییرات و دومی به‌معنی نپذیرفتن تغییرات است. به‌همین دلیل در هر پروژه‌ای باید روش‌های برای در نظر گرفتن تغییرات مناسب و نپذیرفتن تغییرات نامناسب و زمان‌بر وجود داشته باشد.

۱۲-۲-مقایسه متدولوژی‌های سنگین وزن و سبک وزن

این دو دسته متدولوژی تفاوت‌های بسیاری در نوع اجرا، میزان مستنداتی که تولید می‌کنند و ... دارند که آنها را برای پروژه‌های خاصی مناسب می‌کند. در ادامه به بررسی متدولوژی‌های سبک وزن خصوصا متدولوژی‌های چابک و سنگین وزن از نقطه نظرات متفاوت خواهیم پرداخت.

۱۲-۲-۱- روش اجرا

متدلوژی‌های سبک‌وزن بصورت سازگار[1] عمل می‌کنند و با شرایط منطبق می‌شوند اما متدلوژی‌های سنگین وزن بصورت پیشگو[2] عمل می‌کنند و سعی می‌کنند در آغاز همه چیز را پیش‌بینی کنند. استفاده از یک فرایند تقریباً منسجم در متدولوژی‌های سنگین وزن سبب می‌شود که آنها در اجرا به‌صورت از پیش تعریف شده عمل نمایند. شکل ۱۲-۳ تفاوت این دو متدولوژی را از نظر فرآیندی نشان می‌دهد. متدولوژی‌های سبک‌وزن فرآیندهایی بیشتر آشفته و ad-hoc دارند و در مقابل متدولوژی‌های سنگین‌وزن فرآیندهای خوش‌تعریف و خشک دارند. نکته بسیار مهمی که در مورد متدولوژی‌های سبک وزن می‌بایست در نظر داشت، تفاوت بی‌نظمی فرآیندی آنها با عدم وجود متدولوژی است. در این متدولوژی‌ها، تعریف متدولوژی‌ها به گونه‌ای است که اجازه بی‌نظمی تعریف شده‌ای را می‌دهد و منظور از بی‌نظمی عدم وجود نظم نیست!



شکل ۱۲-۳- نظم فرآیندی در متدلوژی‌های سبک‌وزن و سنگین‌وزن

[1] Adaptive
[2] Predictive

تحقیقات بر روی پروژه‌های مهندسی نرم‌افزاری موفق و ناموفق نشان می‌دهد که در اغلب پروژه‌های نرم‌افزاری پیش‌بینی فرایند توسعه نرم‌افزار به پارامترهایی چون مشتری، کارهای قبلی شرکت، خواسته‌ها، اجبارها بستگی دارد که پیش‌بینی را دشوار می‌کند.

۱۲-۲-۲- معیار موفقیت

معیار موفقیت در متدولوژی‌های سبک‌وزن دستیابی به ارزش کاری[1] است در حالیکه در متدولوژی‌های سنگین وزن معیار موفقیت پیش رفتن در راستای طرح اولیه است. در صورتیکه طرح اولیه با تغییر جدی مواجه شود، متدولوژی‌های سنگین وزن با مشکل جدی از نظر هزینه و زمان روبرو خواهند شد. به‌عبارت بهتر، طرح اولیه تاثیر بسیاری در اجرای متدولوژی‌های سنگین‌وزن دارد و انعطاف‌پذیری این متدولوژی‌ها به دیدگاه اولیه‌ای که در پروژه برقرار شده است، برمی‌گردد.

۱۲-۲-۳- اندازه پروژه

اندازه پروژه در متدولوژی‌های سبک‌وزن کوچک است در حالیکه در متدولوژی‌های سنگین وزن می‌تواند بسیار بزرگ باشد. این مورد سبب می‌شود که متدولوژی‌های سبک وزن در پروژه‌های بزرگ مناسب نباشند و بالعکس. اما باید در نظر داشت که تحقیقات نشان داده است که اغلب پروژه‌های مهندسی نرم‌افزار، پروژه‌های کوچک هستند و تعداد پروژه‌های بزرگ کم است. همین موضوع سبب می‌شود که رویکرد استفاده از این متدولوژی‌های در شرکت‌های کوچک و متوسط بیشتر باشد.

۱۲-۲-٤- سبک مدیریت

مدیریت در متدولوژی‌های سبک‌وزن بصورت غیرمتمرکز و آزاد است در حالیکه در متدولوژی‌های سنگین وزن مدیریت بصورت مطلق و استبدادی است. این جمله بدین معنی نیست که در متدولوژی‌های سبک‌وزن هر کسی مدیریت و تصمیم‌گیری می‌کند و یا اینکه در متدولوژی‌های سنگین‌وزن استبدادی بودن اجازه اظهارنظر را به تیم توسعه نمی‌دهد، بلکه این امر نسبی است و این دو نوع متدولوژی نسبت به هم سنجیده می‌شوند. در متدولوژی‌های سنگین‌وزن با توجه به تعریف شده بودن وظایف و فعالیت‌ها، تصمیم‌گیری در مورد تغییر فرآیند اجرا و یا مستندات به نظر مدیریت ارشد بستگی دارد. در

[1] Business Value

متدولوژی‌های سبک‌وزن تصمیم‌گیری در مورد سبک اجرا تا حد زیادی به اعضای تیم و تصمیم‌گیری آنها بستگی دارد. این نوع تصمیم‌گیری امکان دخالت اعضای مختلف و همفکری و همین‌طور همراستایی را فراهم می‌آورد.

۱۲-۲-۵- مستندسازی

مستندسازی در متدولوژی‌های سبک‌وزن بصورت بسیار محدود انجام می‌شود در حالیکه در متدولوژی‌های سنگین وزن مستندسازی بصورت کامل و جامع انجام می‌شود. یکی از اشکالاتی که به متدولوژی‌های سبک وزن گرفته می‌شود، عدم مستندسازی و یا مستندسازی کم آنهاست. اما نکته قابل تامل در این متدولوژی‌ها استفاده از مستندسازی مورد نیاز است. به‌عبارت بهتر، در متدولوژی‌های سبک‌وزن میزان مستندسازی به نیاز پروژه بستگی دارد و وابسته به فرآیند از پیش‌تعریف شده نیست اما مطمئناً برخی از محصولات کلیدی همواره پروژه همواره تولید می‌شوند.

در متدولوژی‌های سنگین‌وزنی همانند RUP نیز دو نوع قالب[1] برای اغلب مستندات وجود دارد: رسمی[2] و غیررسمی[3]. قالب غیررسمی اجازه می‌دهد که برخی چیزهایی که با توجه به پروژه غیرضروری می‌رسد، مستند نشود. مستندسازی در متدولوژی‌های سبک‌وزن بر این اصل است که مستندسازی به‌اندازه نیاز انجام شود. از طرف دیگر، مستندسازی بیشتر نیاز به حجم کاری بیشتر را در زمان وقوع تغییرات طلب می‌کند.

۱۲-۲-٦- تعداد چرخه‌ها[4]

تعداد چرخه‌ها در متدولوژی‌های سبک‌وزن بسیار زیاد است اما زمان آنها کوتاه است در حالیکه در متدولوژی‌های سنگین وزن تعداد چرخه‌ها کم است ولی زمان آنها بسیار زیاد است. عمده‌ترین دلیل استفاده از چرخه‌های کوتاه‌تر دستیابی سریع به نشرهای نرم‌افزار است، اما نکته‌ای با اهمیتی که در نباید فراموش کرد، وجود چرخه‌های زیاد است که تیم توسعه را خسته می‌کند. این مشکل در تمام

[1] Template
[2] Formal
[3] Informal
[4] Cycles

متدولوژی‌هایی که از تکرار افزایشی و تدریجی استفاده می‌کنند، وجود دارد و در روش‌های سبک‌وزن نمود بیشتری پیدا می‌کند. در واقع، باید بین تعداد چرخه‌ها و زمان تکرار توازن ایجاد نمود.

۱۲-۲-۷- اندازه تیم

در متدولوژی‌های سبک‌وزن اندازه تیم کوچک است (بین ۲۰ تا ۳۰ نفر)[1] در حالیکه در متدولوژی‌های سنگین وزن اندازه تیم توسعه بزرگ است. این امر به اندازه پروژه مرتبط است، در پروژه‌های بزرگ تعداد افراد تیم بزرگ بیشتر بوده و اغلب از متدولوژی‌های سنگین‌وزن استفاده می‌شود و در پروژه‌های کوچک‌تر که اغلب اندازه تیم نیز کوچک‌تر است، از متدولوژی‌های سبک‌وزن استفاده می‌شود.

۱۲-۲-۸- برگشت سرمایه

در متدولوژی‌های سبک‌وزن سرمایه خیلی زود در طول پروژه بر می‌گردد در حالیکه در متدولوژی‌های سنگین وزن برای برگشت سرمایه باید تا انتهای پروژه صبر کرد. برگشت سرمایه اثر زیادی در دوام تیم‌های نرم‌افزاری دارد، چرا که اغلب شرکت‌های نرم‌افزار جزء شرکت‌های نرم‌افزار با سرمایه کم یا متوسط محسوب می‌شوند. همین امر در مورد شکست پروژه‌های نرم‌افزار نیز صادق است.

۱۲-۳-بیانیه چابک

در سال ۲۰۰۱، ۱۷ نفر از افرادی که در زمینه‌ی روش‌های چابک فعالیت داشتند از جمله Kent Beck، Martin Fowler، Alistair Cockburn در Snowbird به ابتکار Bob Martin دور هم جمع شدند تا علاوه بر تبادل نظر و بحث به استراحت و اسکی بپردازند. نتیجه ی این گردهمایی، بیانیه‌ای بود که به بیانیه‌ی چابک شهرت یافت. این بیانیه به چهار اصل زیر اشاره دارد.

- ❖ فردگرایی و تعامل برتر از فرآیندها و ابزارها
- ❖ نرم‌افزار قابل اجرا برتر از مستندات مفهومی
- ❖ همکاری با مشتریان برتر از مذاکرات قراردادگرا

[1] در صورتیکه اندازه پروژه بسیار کوچک باشد و تیمی با تعداد نفرات کمتر از ۱۰ نفر تشکیل می‌شود، توصیه می‌شود که از متدولوژی توسعه نرم‌افزار استفاده نشود بلکه محصولات مورد نیاز از یک متدولوژی انتخاب و نرم‌افزار توسعه یابد.

❖ پاسخ به تغییر برتر از دنباله‌روی از طرح

مورد اول به تکیه بر دانش افراد، توانایی و تعاملات بین آنها توجه دارد و بر تاکید بیش از حد بر روی فرآیندها، طرح‌ها و ابزارها انتقاد می‌ورزد. به عنوان مثال، محیط توسعه نرم‌افزاری را در نظر بگیرید که فرآیند و روندهای کاری به طور دقیق مشخص شده‌اند و ابزارهای جدید، پیچیده و با قابلیت به نسبت بالا در این محیط در دسترس افراد است اما تیم توسعه دهنده متشکل از افرادی کم مهارت با تعاملات ضعیف می‌باشد. چنین تیمی در توسعه نرم‌افزار به احتمال قوی دچار مشکل خواهد شد. چرا که کسانی که نرم‌افزار را توسعه می‌دهند، افراد هستند و نه ابزارها و فرایندها. اشتباه نشود، فرآیندها و ابزارها مهم هستند اما مهمتر از آنها افرادی هستند که با آنها کار می‌کنند. البته این موضوع برای مدیریت معمولاً قابل پذیرش نیست، چراکه تفکری مبنی بر رابطه ی مستقیم بین زمان و افراد در ذهن دارند، یعنی می‌توان با افزایش افراد به نتیجه‌ی بهتری رسید در حالی که چنین موضوعی الزاماً صحیح نیست.

مورد دوم بر توجه بیشتر به ساخت نرم‌افزار به نسبت مستندسازی اشاره دارد. اگر از کاربر پرسیده شود که آیا آنها ترجیح می‌دهند، نرم‌افزار قابل اجرا و مناسب در اختیار داشته باشند و یا یک مستند ۱۰۰ صفحه‌ای، چه پاسخی شنیده می‌شود. در ۹۹ تا ۱۰۰ درصد موارد پاسخ نرم‌افزار مناسب خواهد بود. مستندات برای آموزش کاربران و تا حدودی برای نگهداشت سیستم مورد نیاز است. اما نبایستی فراموش شود که هدف اصلی توسعه نرم‌افزار است و نه توسعه مستندات.

در سومین اصل، مشکلی که همواره توسعه‌دهندگان نرم‌افزار با آن روبرو هستند، مورد توجه است و آن عدم رسیدن به درک مشترک و همکاری نامناسب می‌باشد. داشتن قرارداد برای یک پروژه مهم است اما برای ارتباط با مشتری کافی نیست، در توسعه نرم‌افزار نیاز به مشارکت و همکاری متداوم با مشتری وجود دارد.

و مورد پایانی، به پاسخگویی به تغییرات اشاره می‌کند. خواسته‌های مشتری از نرم‌افزار تغییر می‌کند، محیط حرفه‌ای که نرم‌افزار مرتبط به آن است، تغییر می‌کند، تکنولوژی به مرور زمان تغییر می‌کند. تغییر یک واقعیت در توسعه‌ی نرم‌افزار است، واقعیتی که فرآیند توسعه بایستی آن را پشتیبانی کند. داشتن طرحی برای پروژه اشتباه نیست بلکه اگر پروژه‌ای طرح نداشته باشد، جای نگرانی دارد. به هر حال در طرح پروژه، تمهیداتی برای تغییرات احتمالی پیش‌بینی شود.

شاید در مواردی که اشاره شد، نظر بسیاری از توسعه‌دهندگان موافق با بیانیه باشد، اما اینکه در عمل چقدر پایبند به آنها هستند، حرف دیگری است. مدیرانی که بر درستی طرح اولیه اصرار می‌ورزند و تغییرات را نمی‌پذیرند، هنوز وجود دارند. اصولی که اشاره شد، روش‌های چابک به دنبال برآوردن آنها هستند و به قول Scott Ambler، مدل‌سازان چابک آنچه می‌گویند، انجام می‌دهند و آنچه انجام می‌دهند، می‌گویند.

۱۲-۴-متدولوژی‌های چابک

اغلب روش‌های چابک اساس کار خود را بر افراد و توانایی‌های آنها قرار می‌دهند. فردگرایی در این روش‌ها، یکی از ارکان محسوب می‌شود و یکی از دلایل انعطاف‌پذیری این روش‌ها، همین اتکا بر افراد و توانایی‌های آنهاست. برخی تکنیک‌ها مانند برنامه‌نویسی- دونفره با این فردگرایی انطباق پیدا می‌کنند. به طوریکه تیم دو نفره علاوه بر کدنویسی اختیار طراحی و شرکت در آزمون را نیز بر عهده می‌گیرد. این باعث افزایش نقش روز افزون برنامه‌نویسان در توسعه سیستم و بهره‌مندی هر چه بیشتر از تجربیات آنهاست. این موضوع در شرایطی است که برنامه‌نویس در روش‌های سنتی توسعه، اجرا کننده تصمیمات بالادستی دیگر ذینفعان پروژه به حساب می‌آمد. این ماجرا تا بدان جا بود که تخمین غلط مدیر پروژه در زمانبندی به حساب عدم کارآیی برنامه‌نویس نوشته می‌شد. مدیر پروژه ای که به علت دوری از عملیات واقعی توسعه و درگیر نبودن با مشکلات ریز آن، برآورد غلطی از توان اجرایی نیروهای خود دارد، برنامه‌نویسان را متهم به ضعف و عدم کارآیی می‌کند.

Beck به عنوان پایه‌گذار روش XP بر این نکته تاکید دارد و به خاطرات تلخ یکی از دوستان برنامه‌نویس خود اشاره می‌کند. با این اوصاف، شاید بتوان روش‌های چابک را دیدگاهی برخواسته از قشر توسعه‌دهنده سیستم مانند برنامه‌نویسان و طراحان دانست تا دیدگاهی مدیریتی که بیشتر در پی برنامه‌ریزی، بودجه‌بندی و زمانبندی‌های صورت گرفته در یک محیط مجرد است.

تقریباً از سال ۲۰۰۰ به بعد، روش‌های متنوعی در رده‌ی روش‌های چابک معرفی شده اند. این روش‌ها اغلب سعی بر پایبندی به اصول چابکی دارند. نمون ای از این روش‌ها در ذیل آمده است.

- Extreme Programming (XP)
- Feature Driven Development (FDD)
- Scrum
- Adaptive Software Development (ASD)

- Dynamic Software Development Method (DSDM)
- Crystal Family

در میان این روش‌ها طبق آمار ارائه شده روش XP، پرکاربردترین روش شناخته شده است. این روش را به طور اجمالی در بخش‌های بعدی بررسی می‌کنیم.

۱۲-۵-متدولوژی XP

متدولوژی XP یکی از متدولوژی‌های توسعه نرم‌افزار است که بر پایه اصولی مثل سادگی، ارتباط و بازخورد استوار است. این متد برای استفاده در تیم‌های کوچک که به توسعه سریع نرم‌افزار در یک محیط تغییرپذیر نیاز دارند طراحی شده است. ایده اصلی این متد متعلق به Kent Beck نویسنده کتاب Extreme Programming Explained می‌باشد که سال‌ها در زمینه توسعه نرم‌افزار با متدهای شی‌گرا کار کرده است. در واقع XP نتیجه تجربه او در استفاده از متدهای شی‌گرا بویژه استفاده از زبان SmallTalk بود. SmallTalk اولین زبان محبوب شی‌گرا بود. XP موفق شد زیرا هدف اصلی آن رضایت مشتریان بود. این متد به توسعه دهندگان این امکان را می دهد که بدون هیچ تردیدی به تغییر نیازهای مشتریان پاسخ دهند این متد شبیه یک جدول معماست که از تکه های زیادی تشکیل شده و وقتیکه این تکه ها در کنار هم قرار گیرند تشکیل یک شکل می‌دهند. XP بر توسعه مبتنی بر آزمایش استوار است و به همه نقش‌ها از جمله مشتریان توجه خاص دارد و بعنوان یک اصل بر تعامل و همکاری متقابل بین نقش‌های کلیدی در توسعه نرم‌افزار تکیه می کند و تلاش می‌کند که محصول در حداقل زمان ممکن به مشتری تحویل داده شود. در چند سال اخیر پروژه‌های بسیار زیادی با این متدولوژی انجام شده است و اکثر آنها موفقیت آمیز بوده‌اند و این موفقیت موجب شده است که متدولوژی XP در بین توسعه دهندگان نرم‌افزار محبوبیت خاصی پیدا کند و محققان برای کار کردن روی آن تشویق شوند چنانکه در چند سال اخیر چندین کنفرانس بین المللی در مورد این موضوع تشکیل شده است. این متدولوژی شامل ۱۲ فعالیت اصلی و پنج فاز است که به بررسی آنها می‌پردازیم.

۱۲-۵-۱- ارزش‌های XP

XP چهار ارزش معرفی می‌کند که تاثیر گذار بر فعالیت‌ها، نقش‌ها و محصولات در این روش هستند. این چهار ارزش ارتباط، سادگی، بازخورد و شجاعت هستند.

- **ارتباط**

اولین ارزش XP ارتباط است. بسیاری از مشکلات پروژه تا زمانی که بین افراد در رابطه با آنها، گفتگو صورت نگیرد، حل نمی‌شوند. گاهی اوقات یک برنامه‌نویس تغییری بنیادی در طراحی ایجاد می‌کند و به کسی نمی‌گوید. گاهی برنامه‌نویس از مشتری سوال نمی‌پرسد و بر مبنای نظر خود تصمیمی را لحاظ می‌کند. در مقابل، مشتری از برنامه‌نویس پرسش نمی‌کند و روند پیشرفت پروژه اشتباه گزارش می‌شود. در بعضی شرایط نیز، روابط نامطلوبی بین افراد وجود دارد و پروژه را دچار مشکل می‌کند. برنامه‌نویسی را در نظر بگیرد که به اعلام اخبار ناگوار از پروژه به مدیر، محکوم می‌شود و در مقابل مشتری اطلاعات مهمی را به برنامه‌نویس می‌گوید ولی برنامه‌نویس از آنها چشم پوشی می‌کند. برخی فعالیت‌ها در XP نیازمند روابط مناسب بین افراد است و XP بر این نکته تاکید می‌ورزد. در این روش نقشی وجود دارد که یکی از وظایف آن نظارت بر صحیح بودن روابط بین افراد پروژه است، این نقش مربی نام دارد.

❖ سادگی

دومین ارزش XP سادگی است و انتظار می‌رود که ساده ترین چیزی که می‌تواند که کار می‌کند، استفاده شود. دستیابی به سادگی، معمولاً به آسانی به دست نمی‌آید. دلیل این مطلب آنست که معمولاً به احتمالات آینده فکر می‌شود، اگر این ویژگی نیز درخواست شود؟ اگر جایی دیگر نیز بدان نیاز شود؟ و نگرانی‌هایی از این دست معمولاً تیم را به سمت و سوی توسعه‌ی سیستمی پیچیده‌تر از آنچه مورد نظر است، می‌کشاند و از عواقب آن نرسیدن به زمان‌های مورد نظر و کار و هزینه‌ی بیشتری که صورت پذیرفته است. سادگی و ارتباط رابطه‌ی نزدیکی با یکدیگر دارند. با روابط مناسبتر، می‌توان دقیقتر سیستم و نیازمندی‌های آن را فهمید و در نتیجه از پیچیده شدن بیش از حد سیستم جلوگیری کرد و ساده‌ترین روش‌ها را برای توسعه سیستم به کار برد. در مقابل صحبت در رابطه با سیستمی که ساده‌تر باشد، آسان‌تر است و ارتباطات آسانتر و موفقتر صورت می‌گیرد.

❖ بازخورد

در XP تاکید بر روی مانیتور کردن وضعیت سیستم به طور مرتب، دقیقه به دقیقه و روز به روز است. نمونه‌ای از این موضوع، آزمون واحدهاست. برنامه‌نویسان برای واحدهایی که توسعه می‌دهند، مورد آزمون‌هایی را می‌نویسند. مشتریان داستان‌های کاربری[1] را می‌نویسند و برنامه‌نویسان آنها را تخمین

[1] User Stories

می‌زنند. با توجه به این تخمین، در مورد کیفیت داستان‌هایشان اطلاع کسب می‌کنند. آزمون کنندگان نیز سیستم را آزمون و از سیستم بازخوردهایی را دریافت می‌کنند.

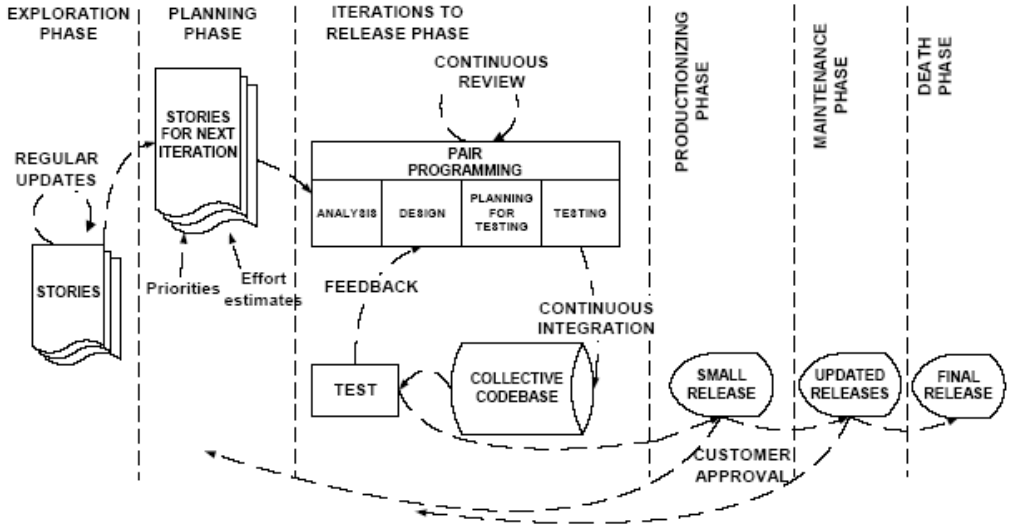
بازخورد وابستگی‌هایی با ارتباط و سادگی دارد. در صورت بازخورد بیشتر در رابطه با سیستم، ارتباطات آسانتر است. وقتی در حال گفتگو در مورد سیستم باشیم، متوجه خواهیم شد که چه مواردی را باید مورد آزمون و اندازه‌گیری قرار دهیم. به همین شکل، آزمون سیستم‌های ساده‌تر، آسانتر است. زمانیکه برای سیستم، موارد آزمون نوشته می‌شود، نگرشی از میزان سادگی سیستم بدست می‌آید.

❖ شجاعت

زمانی که در توسعه سیستم، اشکالی مانند مشکل در طراحی کلان سیستم شناخته شود و به تبع آن مشکل در آزمایش سیستم بروز کند، تیم می‌بایست انسجام خود را حفظ کرده و درصدد رفع مشکل مربوطه باشد. نکته‌ی دیگر در این رابطه، بیرون انداختن کدی است که به نظر مناسب نوشته نشده است. اگر برنامه‌نویس در پایان روز به این نتیجه رسید که کد نوشته است به اندازه‌ی مطلوب مناسب نوشته نشده و طراحی قابل قبول را ندارد، بایستی آن را کنار گذاشته و فردا از ابتدا بر روی آن بخش تمرکز کند.

۱۲-۵-۲- مدل فرآیندی

مدل فرآیندی که مورد استفاده XP است در شکل ۱۲-۴ آمده است. این مدل از پنج فاز تشکیل شده است که در ادامه به تشریح هر یک از فازها خواهیم پرداخت.



شکل ۱۲-۴- مدل فرآیندی روش XP

❖ اکتشاف[1]

در این فاز مشتری مشغول نوشتن داستان‌های کاربری (فرآورده‌های XP را ببینید) بر روی کارت‌های داستان به منظور نشر اولیه می‌شود. در این مدت برنامه‌نویسان به بررسی تکنولوژی مورد استفاده پرداخته و سعی می‌کنند برخی عملیاتی که سیستم نیاز دارد را در قالب تکنولوژی انجام دهند. برای این هدف، ممکن است از متخصصین تکنولوژی خارجی استفاده شود، چراکه برای کسانی که با تکنولوژی آشنا نیستند ممکن است انجام عملیاتی تحت تکنولوژی جدید به راحتی مقدور نباشد و یا مواردی از قلم بیفتد.

برنامه‌نویسان همچنین بایستی توجه با ایده‌های معمارانه داشته باشند. شاید نیاز به پیاده‌سازی ساده‌ای از برخی معماری‌های مدنظر برای سیستم باشد. در نهایت توجه به این نکته مهم است که در پایان تکرار اول در روش XP بایستی اسکلت سیستم بدست آید.

❖ برنامه‌ریزی[2]

هدف اصلی فاز برنامه‌ریزی، موافقت بر روی تاریخی است که مجموعه با ارزش‌ترین داستان‌های کاربری بایستی انجام شوند. فعالیت بازی طراحی، برای این منظور صورت می‌پذیرد. اگر آمادگی در طول اکتشاف بدست آمده باشد، برنامه‌ریزی در یک یا دو روز صورت پذیرد.

❖ تکرار برای انتشار[3]

زمانبندی به تکرارهای یک تا چهار هفته‌ای شکسته می‌شود. تکرار اول، معماری را مدنظر قرار می‌دهد. داستان‌های کاربری بایستی بایستی به گونه‌ای انتخاب شوند که نمای کلی از سیستم به وجود آید حتی اگر حاصل تنها اسکلتی از سیستم باشد. سوالی که بایستی مطرح شود این است که ارزشمندترین موارد برای نشر اول کدام ها هستند؟

در طول گذر از تکرارها متوجه انحراف از طرح خواهیم شد. ممکن است داستان‌هایی اضافه و یا برخی حذف و اصلاح شوند. در این شرایط نیازمند تغییر و به روز رسانی طرح هستیم. در پایان تکرار آخر، آماده ی رفتن به فاز بعدی که فاز تولید است، هستیم.

---

[1] Exploration
[2] Planning
[3] Iterations to release

❖ تولید[1]

در این مرحله، نشر آماده اعتبارسنجی و در استفاده قرار گرفتن است. آزمون‌های جدیدی برای این منظور پیاده‌سازی و اعمال می‌شوند. آزمون‌ها معمولاً به طور موازی انجام می‌شوند. در این مرحله کارایی سیستم بررسی و سعی به تنظیم آن با توجه به اطلاعات طراحی می‌شود.

❖ نگهداشت[2]

پس از هر نشری نیازمند پشتیبانی از محصول هستیم. ممکن است وظیفه‌مندی‌های جدیدی به سیستم افزوده شود. ممکن است تلاش‌هایی برای بهبود کد نشر قبلی و یا تکنولوژی جدیدی که در نشر بعدی مورد استفاده قرار می‌گیرد، صورت پذیرد.

توسعه‌ی سیستمی که هم اکنون در استفاده نیست با سیستمی که در حال استفاده توسط کاربران است، متفاوت است. ممکن است فرآیند توسعه به علت مسائل به وجود آمده در محصول دچار وقفه شود. بایستی به تغییراتی که در نظر گرفته می‌شود، توجه ویژه ای شود. سیستم هم اکنون شامل داده‌هایی است که در تغییر سیستم بایستی در نظر گرفته شوند.

❖ مرگ[3]

در اینجا مشتری دیگر نیازمندی مدنظر ندارد و یا ویژگی‌هایی را می‌خواهد که اقتصادی تشخیص داده نمی‌شود. زمان آنست که مستنداتی برای سیستم تولید شود.

۱۲-۵-۳- نقش‌ها و مسئولیت‌ها

در XP نقش‌های مختلفی برای انجام وظایف و دستیابی به اهداف متفاوت در فرآیند توسعه و فعالیت‌ها در نظر گرفته شده‌اند. این نقش‌ها به شرح ذیل هستند:

❖ برنامه‌نویس[4]

برنامه‌نویسان علاوه بر نوشتن کد برنامه، به کمک مشتری موارد آزمون را می‌نویسند و کد برنامه را تا حد ممکن ساده و پایدار نگاه می‌دارند. موضوعی که در موفقیت XP اهمیت دارد، ارتباط مناسب و موثر هر برنامه‌نویس با دیگر برنامه‌نویسان و اعضای تیم است.

---
[1] Productionization
[2] Maintenance
[3] Death
[4] Programmer

❖ مشتری[1]

مشتری آزمون‌های وظیفه‌مندی و داستان‌ها را می‌نویسد و تصمیم‌گیری آنکه چه موقع یک نیاز برآورده شده است، بر عهدهٔ مشتری است. علاوه بر این، مشتری اولویت پیاده‌سازی نیازمندی‌ها را تعیین می‌کند.

❖ آزمون‌کننده[2]

آزمون‌کننده به مشتری در نوشتن آزمون‌های وظیفه‌مندی مناسب‌تر کمک می‌رساند. وی به طور منظم موارد آزمون را اجرا کرده و نتایج آزمون را اعلام می‌دارد. مسئولیت ابزارهای آزمون نیز بر عهده‌ی وی می‌باشد.

❖ پیگیری‌کننده[3]

وی در طول پروژه بررسی می‌کند که تخمین‌های صورت گرفته، چقدر صحیح می‌باشند تا تخمین‌های آینده بهبود یابند. همچنین میزان پیشرفت هر تکرار را بررسی می‌کند تا میزان منابع و زمان را برای رسیدن به هدف ارزیابی کند و بررسی آنکه نیازی به تغییر در فرآیند باشد.

❖ مربی[4]

فردی است که مسئول کل فرآیند می‌باشد. این فرد بایستی درکی شفاف از XP داشته باشد تا بتواند دیگر اعضای تیم را راهنمایی کند.

❖ مدیر[5]

مدیر وظیفه‌ی تصمیم‌گیری را برعهده دارد. برای انجام این وظیفه، وی با اعضای تیم در ارتباط است تا وضعیت کنونی را بدست آورد و سختی‌ها و کمبودهای موجود را شناسایی کند.

❖ مشاور[6]

مشاور عضوی خارجی است که در یک زمینه‌ی فنی، دانش مورد نیاز را دارد. مشاور اعضای تیم را در حل مشکلات در زمینه مربوطه یاری می‌کند.

---

[1] Customer
[2] Tester
[3] Tracker
[4] Coach
[5] Big Boss
[6] Consultant

۱۲-۵-۴- فرآورده‌های XP

مجموعه‌ای از فرآوردها که در متدولوژی XP تولید می‌شوند، در ذیل ارائه شده است.

- **داستان‌های کاربری**[1]

معمولاً بشکل متنی بوده و توسط مشتریان نوشته می‌شوند و از طریق آنها نیازمندی‌های سیستم مشخص می‌شود.

- **طرح تکرار**

مجموعه‌ای از داستان‌های کاربری ست که توسط مشتری انتخاب می‌شوند و در یک تکرار که معمولاً دو هفته طول می‌کشد، تولید می‌شود. طرح‌های تکرار با توجه به اولویت مشخص شده توسط مشتری اجرا می‌شوند و انتخاب آنها براساس بودجه تعیین‌شده توسط توسعه‌دهندگان خواهد بود.

- **طرح نشر**[2]

مجموعه‌ای از طرح‌های تکرار را در قالب یک نقشه کلی برای رسیدن به نشرها نمایش می‌دهد. در واقع نشان می‌دهد که نشرها چه‌موقع و چگونه تولید می‌شوند.

- **وظیفه**[3]

زیرمجموعه‌ای از داستان‌های کاربری هستند. وظایف از نظر فنی و کاری اولویت بالایی دارند و باید سریع انجام شوند. در مرحله طرح‌ریزی تکرارها (فعالیت‌های XP را ببینید) مشخص می‌شوند.

- **Metaphore**

نشان‌دهنده یک تصویر کلی از سیستم است. برای هر عنصر در سیستم یک نام در نظر گرفته می‌شود و ارتباط بین عناصر درگیر در سیستم از طریق Metaphore مشخص می‌شود.

- **Spike**

یک راه‌حل ضربتی[4]، برنامه ساده‌ایست که بوسیله آن می‌توان راه‌حل‌های بالقوه را کشف کرد. در مواردی که داستان‌های کاربری حساس و مهمند از این راه‌حل استفاده می‌شود.

---

[1] User Stories
[2] Release Plan
[3] Task
[4] Spike Solution

۱۲-۵-۵- فعالیت‌های XP

برای تحقق ارزش‌های ذکر شده، فعالیت‌هایی در نظر گرفته شده است که به شرح ذیل می‌باشند.

❖ **بازی طراحی**[1]

یک تعامل محصور[2] بین مشتری و برنامه‌نویس در حین اجرای پروژه بدست می‌آید. برنامه‌نویس کار لازم برای پیاده‌سازی گزارش‌های مشتری را تخمین می‌زند و مشتری در مورد حوزه و زمان نشرها تصمیم‌گیری می‌کند. در واقع، مشتری براساس اطلاعات و تخمین‌هایی که برنامه‌نویس در اختیارش قرار می‌دهد، طرح را بروزرسانی می‌کند. علاوه بر این، برای برنامه‌ریزی نیاز به تصمیم‌گیری در موارد زیر وجود دارد:

- **محدوده**. چه میزان از مسئله بایستی در قالب سیستم توسعه داده شود تا محصول ارزشمند باشد. فردی در رابطه با حرفه، این توانایی را دارد که بیان کند، چه میزان کافی نیست و چه میزان بیش از حد است.
- **اولویت**. در اینجا بایستی تعیین شود که بین دو بخش A و B، کدامیک بایستی ابتدا توسط برنامه‌نویسان توسعه داده شود.
- **ترکیب نشرها**[3]. بایستی مشخص شود که چه میزان از سیستم توسعه یابد تا برای حرفه ای که بدون نرم افزار است، مفید واقع شود که این تصمیم‌گیری اگر به تنهایی توسط برنامه‌نویسان صورت گیرد، امکان اشتباه دارد.
- **روزهای نشر**. روزهای مهم نشر نرم افزار کدام‌ها هستند.
- **نشرهای کوچک**[4]. نشرها بایستی تا حد امکان کوچک و در بر دارنده‌ی ارزشمندترین نیازهای حرفه باشند. البته این بدان معناست که نشر بایستی ناقص نباشد یعنی نیمی از یک ویژگی در نشر پیاده‌سازی شده باشد.

❖ **طراحی ساده**

[1] Planning game
[2] Close Interaction
[3] Composition of releases
[4] Short releases

طراحی سیستم بایستی تا حد ممکن ساده باشد، تا ارزش سادگی که قبلاً بحث شد، برآورده کند.

❖ آزمایش

برنامه‌نویسان موارد آزمون را برای اطمینان از بخش کدی که می‌نویسند، تهیه می‌کنند که البته اینکار برای بخش‌هایی انجام داده می‌شود که امکان خطا وجود داشته باشد.

❖ بهبود[1]

در زمان پیاده‌سازی یک ویژگی برنامه، از طرف برنامه‌نویسان این سوال مطرح می‌شود که آیا می‌توان برنامه را ساده‌تر نوشت؟ در اینجا زمانیکه یک ویژگی اضافه می‌شود، برنامه‌نویسان سعی در ساده سازی برنامه دارند. نمونه ای از این ساده سازی، تبدیل کدهای کپی شده به متدهاست. این تغییرات نبایستی باعث تغییر در رفتار سیستم شود. در واقع، تلاش در جهت بهبود ساختاری سیستم است.

❖ زوج برنامه‌نویسی[2]

تولید کد توسط دو برنامه‌نویس بر روی یک کامپیوتر با یک صفحه کلید و یک ماوس صورت می‌گیرد. در هر جفت، دو نقش وجود دارد. یک نفر بر نحوه ی پیاده سازی کد با استفاده از کامپیوتر تمرکز می‌کند و دیگری به مورد آزمون‌های ممکن، نحوه ی ساده‌تر کردن طراحی و موضوعاتی از نگاه بالاتر تمرکز دارد. دو نفری که در یک روز در قالب زوج برنامه‌نویس هستند، روز بعدی می‌توانند زوج خود را عوض کنند و هیچ مانعی در انتخاب زوج وجود ندارد.

❖ مالکیت جمعی[3]

در برخی روشهای توسعه بخش‌هایی از کد که توسط فرد خاصی توسعه داده شده است، در مالکیت وی است و تغییرات در آن قسمت توسط وی یا با اجازه ی وی صورت می‌گیرد. در این شرایط، اگر فردی که مالکیت بخشی از کد را دارد، دیگر با تیم توسعه همکاری نکند و یا مشکلی

[1] Refactoring
[2] Pair programming
[3] Collective Ownership

برایش پیش آید، توسعه دچار مشکل خواهد شد. در XP، مالکیت کد برای همه اعضاست و هر کس می‌تواند تغییرات مدنظرش را اعمال کند.

❖ یکپارچه‌سازی مداوم[1]

کدهای نوشته شده حداکثر پس از یک روز بایستی یکپارچه شوند. یک روش آن است که کد یکپارچه شده بر روی یک ماشین قرار گیرد و در زمانی که این ماشین آزاد است، زوج‌های برنامه‌نویس کد توسعه داده شده‌ی خود را با استفاده از این ماشین به کد یکپارچه اضافه کنند.

❖ ۴۰ ساعت کار در هفته[2]

افراد در تیم نبایستی در طول هفته بیش از ۴۰ ساعت کار کنند و دلیل آن شاداب و سرزنده ماندن افراد در توسعه سیستم است.

❖ مشتری مستقر در سایت[3]

مشتری بایستی در کنار تیم توسعه حضور داشته باشد و به سوالات توسعه‌دهندگان پاسخ دهد. منظور از مشتری فردی است که پس از آنکه سیستم توسعه یافت، از آن استفاده خواهد کرد. اگر شما مشغول توسعه‌ی یک سیستم حسابداری هستید، این فرد بایستی یک حسابدار باشد.

❖ استانداردهای کد نویسی[4]

اگر قصد بر آنست که تمام برنامه‌نویسان، بتوانند در تمام بخش‌های سیستم فعالیت توسعه را انجام دهند. یک روز بر روی یک بخش کد و روز بعد بخشی دیگر و در راستای فعالیت مالکیت جمعی بر کد حرکت کنیم، نیازمند داشتن استانداردهای برنامه‌نویسی در میان تیم توسعه دهنده‌ی نرم‌افزار هستیم.

۱۲-۶-متدولوژی Feature Driven Development

متدولوژی FDD در سال ۲۰۰۲ توسط Palmer و Felsing ارائه شده است. این روش تمام فرآیند توسعه نرم‌افزار را پوشش نمی‌دهد و بیشتر روی دو فاز طراحی و پیاده‌سازی متمرکز می شود . این روش

[1] Continuous integration
[2] 40 hour week
[3] On-site customer
[4] Coding standards

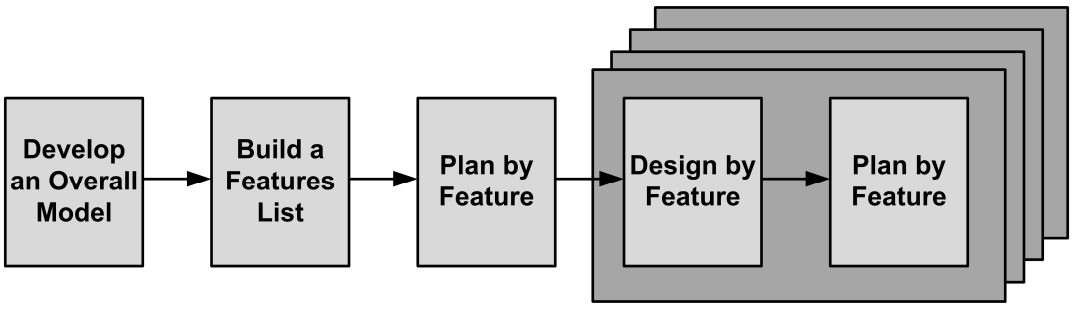
برای استفاده بهمراه سایر فعالیت های یک پروژه توسعه نرم‌افزار طراحی شده است و هیچ مدل فرآیند خاصی لازم ندارد. متدولوژی FDD مبتنی بر توسعه تکراری با انتخاب بهترین و موثرترین فعالیت‌هاست و بر روی جنبه های کیفی تاکید دارد و شامل نشرهای محسوس و پیگیری دقیق پیشرفت پروژه است. این روش شامل مجموعه‌ای از نقش‌ها، محصولات، اهداف و خطوط زمانی لازم در پروژه است.

فرآیند توسعه شامل تکرارهای بسیار کوتاه است و همه خواص مورد توسعه بصورت واضحی توضیح داده می‌شوند که برای کاربر قابل فهم است. خواص مطرح شده در این روش از تکنیک‌های شی‌گرا پشتیبانی می‌کنند. در هر زمانی در طول پروژه می‌توان تخمین زد که تا پایان کار چقدر فاصله داریم. این کار توسط فرآورده‌های زمانی خاصی که در این روش وجود دارد محقق می‌شود.

تعیین میزان پیشرفت پروژه در همه روش‌های چابک از اهمیت ویژه ای برخوردار است و در این روش نیز تدابیر خاصی برای تشخیص میزان پیشرفت پروژه وجود دارد. گزارشات و مدل‌های ویژه‌ای در طول فرآیند توسعه تهیه می‌شوند که میزان کار انجام شده و میزان پیشرفت پروژه را نشان می‌دهند. در عین حال زمان اتمام هر تکرار و هر فاز تخمین زده می‌شود و چون فرآیند توسعه به بخش‌های کوچک تقسیم شده است این تخمین چندان دور از ذهن نیست.

۱۲-۶-۱- فرآیند FDD

FDD شامل پنج فرآیند ترتیبی است که از طریق آنها فعالیت‌های طراحی و پیاده‌سازی انجام می‌شود. شکل ۱۲-۵ مدل فرآیند FDD را نشان می‌دهد. قسمت تکراری فرآیند FDD (طراحی و ساخت) از توسعه چابک حمایت می‌کند. هر تکرار از یک خاصیت، معمولا ۲ تا ۳ هفته زمان می‌برد.



شکل ۱۲-۵- مدل توسعه FDD

❖ ایجاد مدل سراسری

فعالیت‌های این فاز عبارتند از:

- ایجاد یک مدل سراسری، زمانی آغاز میشود که متخصصان دامنه از نیازمندی‌های سیستم اطلاع کافی پیدا کرده باشند.
- از طریق یک WalkThrough که توسط متخصصین دامنه نمایش داده می‌شود، اعضای تیم و معمار یک توصیف سطح بالا از دامنه را می بینند.
- دامنه سراسری به چندین دامنه تقسیم می شود که برای هر یک از آنها یک WalkThrough با جزییات زیاد وجود دارد.
- سپس اعضای هر دامنه یک Object Model دم‌دستی تهیه می کنند.
- مجموعه این Object Model ها مدل سراسری را تشکیل می‌دهند.

❖ ایجاد فهرست خصوصیات

فعالیت‌های این فاز عبارتند از:

- بوسیله Walk Throughها، Object modelها و مستندات نیازمندی‌های سیستم یک لیست جامع از خواص سیستم ایجاد می‌شود.
- تمامی وظایف مورد انتظار کاربر در آن لیست شده است.
- وظایف مربوط به هر دامنه مشخص شده است.
- این لیست توسط کاربران و ذی‌نفعان بازبینی شده و تکمیلی می گردد.

❖ برنامه‌ریزی با خصوصیات

فعالیت‌های این فاز عبارتند از:

- طرح‌ریزی بر اساس خواص، شامل ایجاد یک طرح سطح بالاست که در آن مجموعه خواص بر اساس اولویت‌شان مرتب شده‌اند.
- در این فرآیند باید زمانبندی صورت گیرد و فرسنگ شمارهای خاصی برای تحویل خواص ایجاد گردد.
- کلاس‌های شناسائی شده در فرآیند توسعه یک مدل سراسری در این مرحله به یک توسعه‌دهنده خاص بنام صاحب کلاس[1] تحویل داده می‌شوند.

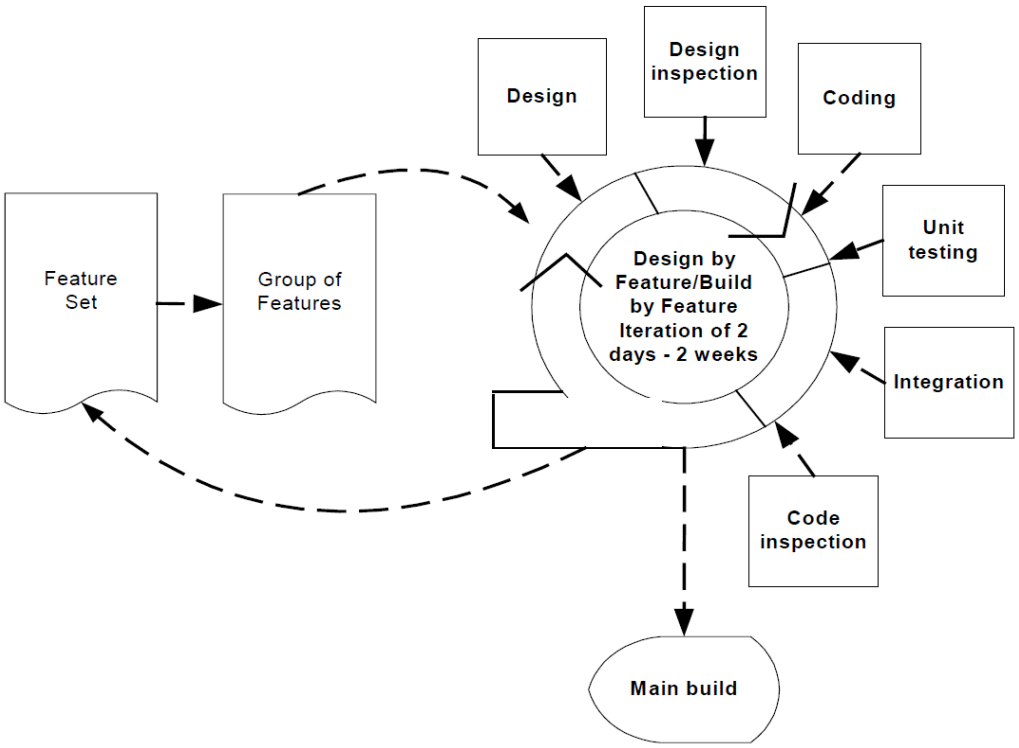
[1] Class owner

❖ طراحی و ساخت با خصوصیات

فعالیت‌های این فاز عبارتند از :

- یک گروه کوچک از محموعه خواص برای پیاده‌سازی انتخاب می‌شوند و صاحبان کلاس‌ها یک تیم را برای توسعه این خواص انتخاب می کنند.
- این فاز یک فرآیند تکراری است که در طول آن خواص انتخاب شده پیاده‌سازی می‌شوند.
- هر تکرار بین چند روز تا دو هفته طول می کشد.
- این تکرار ممکن است شامل موارد طراحی، بازبینی طراحی، کدنویسی، آزمایش، یکپارچگی‌سازی و بازبینی باشد.

در پایان هر تکرار یک محصول اصلی خواهیم داشت و کار با خواص جدید ادامه پیدا می کند. شکل ۱۲-۶ فرآیندهای طراحی با خصوصیات و ساخت با خصوصیات را نشان می‌دهد.



شکل ۱۲-۶- فرایندهای طراحی با خصوصیات و ساخت با خصوصیات FDD

۱۲-۶-۲- نقش‌ها و مسئولیت‌ها

FDD نقش‌های خود را به سه دسته کلی تقسیم می کند:

- نقش‌های کلیدی

- نقش‌های حمایتی
- نقش‌های اضافی

شش نقش کلیدی در FDD عبارتند از:

- مدیر پروژه
  - مسئول و مدیر مالی پروژه است. مدیر پروژه تنها کسی است که تصمیم نهایی را در مورد حیطه، زمانبندی و کارکنان پروژه می‌گیرد. یکی از عمده‌ترین مسئولیت‌های مدیر پروژه، حفاظت از تیم پروژه از مشکلات خارجی و ایجاد شرایط مناسب برای کار کردن اعضای تیم در کنار یکدیگر است.
- معمار ارشد
  - مسئول طراحی کلی سیستم و برگزاری کارگاه برای تیم است. معمار اصلی تنها کسی است که در مورد ویژگی‌های طراحی تصمیم نهایی را می‌گیرد.
- مدیر توسعه
  - مسئول هدایت فعالیت‌های روزمره توسعه و حل برخی اختلاف‌نظرها در زمان توسعه نرم‌افزار است
- برنامه‌نویس ارشد
  - برنامه‌نویس ارشد فردی است که تجربه توسعه نرم‌افزار داشته و در تحلیل نیازمندی‌ها و طراحی مشارکت می‌کند. همچنین این نقش، مسئول هدایت تیم‌ها در طراحی و توسعه خصوصیات جدید است. انتخاب خصوصیت جدید برای توسعه از مسئولیت‌های این نقش است.
- صاحب کلاس[1]
  - تحت هدایت برنامه‌نویس ارشد فعالیت نموده و طراحی، برنامه‌نویسی، آزمایش و مستندسازی را انجام می‌دهد. با توجه به اینکه FDD از شی‌گرایی حمایت می‌کند، به صاحب کلاس مجموعه‌ای از کلاس‌ها تخصیص داده می‌شود تا آنها را پیاده‌سازی نماید. بنابراین صاحب کلاس عضوی از تیم خصوصیات است.

[1] Class Owner

- متخصص دامنه
  - این نقش می‌تواند کاربر، ذی‌نفع، تحلیل‌گر یا ترکیبی از این افراد باشد. وظیفه متخصص دامنه ارائه دانش در مورد نیازمندی‌های مختلف سیستم در حال طراحی است. در واقع، این نقش برعهده فرد یا افرادی است که نیازمندی‌های را می‌شناسند.

پنج نقش حمایتی عبارتند از :

- مدیر نشر
  - این نقش پیشرفت فرآیند توسعه را بازبینی گزارشات برنامه‌ریز اصلی و تشکیل جلسات متعدد کنترل می‌کند و یک گزارش به مدیر پروژه ارائه می‌دهد.
- مشاور زبان[1]
  - افراد در مورد فراگیری زبان و تکنولوژی جدید مورد استفاده در پروژه مسئولیت دارند و این نقش در این مورد بسیار مفید است.
- مهندس ساخت[2]
  - شخصی است که مسئول راه‌اندازی، نگهداری و اجرای فرآیند ساخت است، مدیریت نسخه‌ها، کنترل سیستم و انتشار مستندات از وظایف اوست.
- مسئول ابزار[3]
  - این نقش مسئول تولید ابزارهای کوچک باری توسعه، آزمایش و تبدیل داده در پروژه است نگهداری پایگاه داده وب سایت‌ها از دیگر وظایف این نقش است.
- مدیر سیستم
  - پیکربندی، مدیریت و رفع عیب سرورها شبکه و محیط‌های آزمایش از وظایف این نقش است.

سه نقش اضافی که در همه پروژه‌ها وجود دارند عبارتند از :

- آزمایش‌کننده

[1] Language Guru
[2] Build Engineer
[3] Toolsmith

- آزمایش‌کننده بررسی می‌کند که آیا سیستم در جهت رفع نیازهای مشتری پیش می‌رود یا خیر؟ این نقش ممکن است یک تیم مستقل یا قسمتی از تیم توسعه باشد.

- مستقرکننده[1]
  - این نقش در مورد استقرار نشرها مسئولیت دارد و ممکن است یک تیم جداگانه یا قسمتی از تیم توسعه باشد.
- نویسنده فنی
  - مستندات کاربران توسط این نقش فراهم می‌شوند این نقش ممکن است یک تیم جداگانه یا قسمتی از تیم توسعه باشد.

هر عضو می‌تواند چندین نقش بازی کند و هر نقش ممکن است به چند عضو نسبت داده شود.

۱۲-۶-۳- بهترین تجربیات

متدولوژی FDD شامل مجموعه‌ای از بهترین تجربیات است که می‌توانند در طی فرآیند توسعه بر مبنای FDD مورد استفاده قرار گیرند. توسعه‌دهندگان FDD توصیه می‌کنند که تمام این تجربیات مورد استفاده قرار گیرند تا از خواص این متدولوژی به بهترین وجه استفاده شود. این تجربیات عبارتند از:

- Domain Object Modeling
  استفاده از استخراج و توضیح دامنه مسأله می‌تواند کمک شایانی به درک مسئله نماید
- Developing By Feature
  توسعه و بررسی میزان پیشرفت پروژه از طریق دنبال کردن پیاده‌سازی لیست وظایف و خواص مشخص شده
- Individual Class Ownership
  برای هر کلاس شخصی وجود داشته باشد که مسئول سازگاری، کارایی و صحت آن باشد
- Feature Teams
  اشاره به تیم کوچکی که به‌صورت پویا شکل گرفته‌اند، دارد
- Inspection
  استفاده از معروفترین و بهترین مکانیزهای شناسایی خطاها
- Regular Builds
  تضمین اینکه همیشه یک سیستم قابل اجرا و قابل نمایش دادن، وجود دارد

[1] Deployer

- Configuration Management
  داشتن تاریخچه تغییرات و نسخه‌های مختلف (بهمراه کد منبع)
- Progress Reporting
  روند اجرای فعالیت‌ها به صورت کامل و در سطوح مختلف سازمانی گزارش شود

۱۲-۷-متدولوژی SCRUM

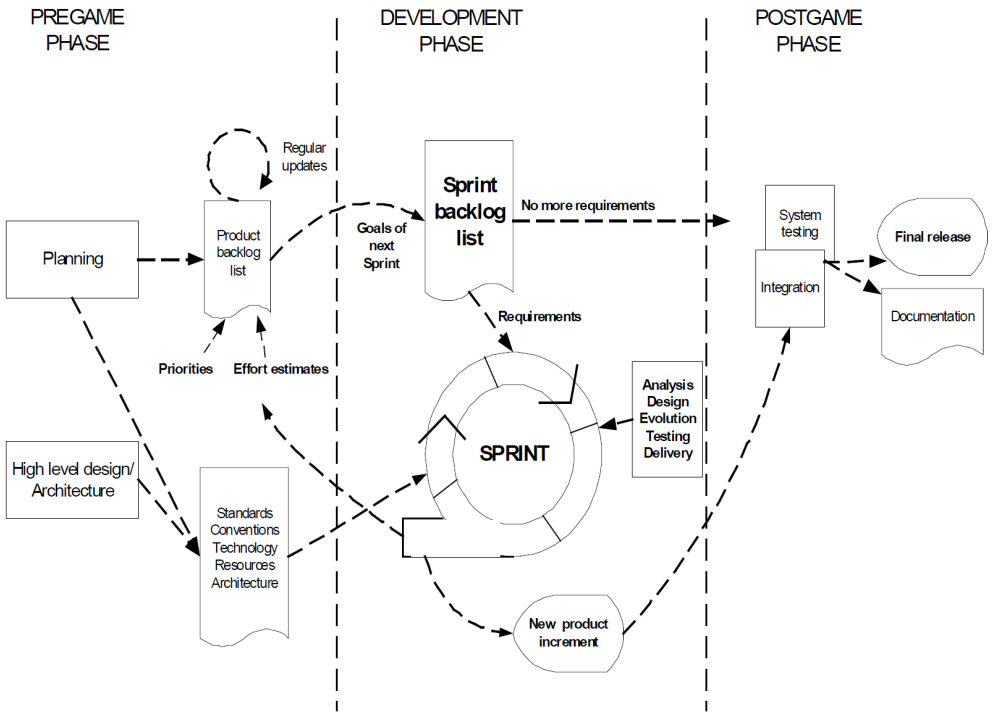
روش Scrum نخستین بار در سال ۱۹۸۶ توسط Takeuchi و Nonaka مطرح شد. لفظ Scrum از یک استراتژی در بازی راگبی گرفته شده است. این روش برای مدیریت فرآیند توسعه سیستم ها مورد استفاده قرار می گیرد.

تاکید اصلی این روش روی اصول انعطاف‌پذیری، سازگاری و سودمندی است. اما روش Scrum هیچ تکنیک خاصی را برای توسعه نرم‌افزار تعریف نمی کند. Scrum بر روی این مطلب متمرکز می شود که: چگونه اعضای تیم باید عمل کنند تا سیستم تولید شده، در یک محیط کاملاً تغییرپذیر انعطاف‌پذیری کافی داشته باشد. ایده اصلی Scrum این است که توسعه سیستم‌ها شامل چندین متغییر محیطی و تکنیکی است (نیازها ، زمان ، منابع و تکنولوژی) که احتمالاً در طول فرآیند توسعه تغییر می کنند و این مطلب فرآیند توسعه را پیچیده و غیرقابل پیش‌بینی می کند و انعطاف‌پذیری فرآیند توسعه سیستم را می‌طلبد تا بتواند در مقابل تغییر نیازها واکنش مطلوبی از خود نشان دهد. Scrum به پیشرفت فعالیت‌ها مهندسی موجود کمک می کند. این روش شامل یک سری فعالیت‌های مدیریتی است که بوسیله آنها تمامی نقص‌ها و موانع موجود در فرآیند توسعه کشف می‌شوند. همچنین Scrum این امکان را می‌دهد که پروژه‌ای بزرگ را به پروژه‌ای کوچک تقسیم کنیم و به صورت موازی و همزمان این پروژه‌ا را انجام دهیم.

۱۲-۷-۱- فرآیند Scrum

فرآیند توسعه Scrum شامل ۳ فاز است که در شکل ۱۲-۷ نمایش داده شده است و عبارتند از:

- PreGame
- Development
- Post Game



شکل ۱۲-۷- فرآیند Scrum

در ادامه این فازها و نقش هر یک از آنها در فرآیند توسعه تشریح خواهد شد.

❖ فاز PreGame

این فاز خود شامل دو فاز برنامه‌ریزی و طراحی سطح بالا است.

❖ برنامه‌ریزی[1]

فاز برنامه‌ریزی شامل تعریف سیستم در حال توسعه است و به عنوان یک محصول مهم در این فاز یک product backlog list (قسمت فرآورده‌ها را ببینید) طراحی می‌شود که شامل تمامی نیازهای شناخته شده در سیستم است. نیازهای در این فاز اولویت‌بندی می‌شوند و کار لازم برای پیاده‌سازی آنها تخمین زده می‌شود و product backlog list با شناسایی نیازهای جدید به روز می‌شود و تغییر می‌کند. علاوه بر این، این فاز شامل فعالیت‌هایی مانند: تعریف تیم پروژه ، ابزارها و سایر منابع، تشخیص خطرات، تعیین نیازهای آموزشی و بررسی روش‌های مدیریت است. در هر تکرار product backlog list توسط تیم توسعه بازبینی می‌شود تا برای استفاده و تکرار بعدی آماده شود. بطور کلی فاز برنامه‌ریزی شامل موارد زیر است:

- ایجاد یک backlog list جامع
- تعریف تاریخ تحویل و وظیفه‌مندی‌های هر نشر جدید

[1] Planning

- انتخاب نشرهای ضروری برای توسعه سریع آنها
- تعریف تیم پروژه برای ساخت نشر جدید
- شناسایی خطرها و پیش‌بینی برای کنترل مناسب آنها
- بازبینی موارد موجود در backlog
- تایید یا انتخاب مجدد ابزارهای توسعه
- تخمین هزینه هر نشر

❖ معماری/ طراحی سطح بالا[1]

در فاز معماری، یک طراحی سطح بالا از سیستم که همه موارد موجود در product backlog list را در بر می‌گیرد، انجام می‌شود. طراحی انجام شده مورد بازبینی قرار گرفته و به عنوان یک طرح برای پیاده‌سازی مورد استفاده قرار می‌گیرد. علاوه بر این طرح‌های مقدماتی برای نشرها در این فاز تولید می‌شود. به طور کلی فعالیت‌های زیر در این فاز انجام می‌شوند:

- بازبینی موارد موجود در backlog
- شناسایی تغییرات لازم برای پیاده سازی موارد موجود در backlog
- تحلیل دامنه مسئله برای طراحی مدل دامنه
- تصحیح معماری سیستم برای پشتیبانی از نیازهای جدید
- تعیین مشکلات مربوط به پیاده‌سازی نیازهای تغییر یافته

❖ فاز توسعه

به این فاز ، فاز Game نیز گفته می‌شود. این فاز بعنوان یک جعبه سیاه رفتار می‌کند. متغییرهای محیطی و تکنیکی که در فاز قبلی شناسایی شدند و ممکن است در طول فرآیند توسعه تغییر کنند توسط فعالیت‌های انجام شده در قسمت sprint از این فاز کنترل می‌شوند.

هدف Scrum از کنترل این متغییرها این است که فرآیند توسعه در مقابل تغییر نیازها انعطاف‌پذیری داشته باشد. در فاز توسعه، سیستم در sprintها توسعه داده می‌شود sprintها چرخه‌های تکراری هستند که فعالیت‌های خاصی در آنها برای تولید محصول انجام می‌شود معماری و طراحی سیستم در طول یک

[1] High level design /architecture

sprint تکامل می‌یابند. یک sprint ممکن است از یک هفته تا یک ماه طول بکشد در واقع در یک sprint کار توسعه انجام می شود.

❖ **فاز Postgame**

در این فاز نشرها خاتمه می‌یابند. وقتی به این فاز می‌رسیم که توافق اساسی روی متغییرهای محیطی مانند نیازها صورت گرفته باشد و دیگر هیچ نیاز جدیدی وجود نداشته باشد. هم اکنون سیستم برای انتشار نشرها آماده است در این فاز فعالیت‌هایی مانند یکپارچه‌سازی، آماده‌سازی سیستم و مستندسازی انجام می‌شود.

۱۲-۷-۲- فرآورده‌ها

فرآورده‌های Scrum به سه دسته اصلی تقسیم می‌شوند:

❖ **Product backlog**

شامل یک صف اولویت‌بندی شده وظیفه‌مندی‌های تکنیکی و کاری است که باید توسعه داده شوند. برای هر مورد مشخص شده در این فرآورده، مواردی مانند وضعیت، اولویت و تخمین کاری مربوط به آن مورد وجود دارد.

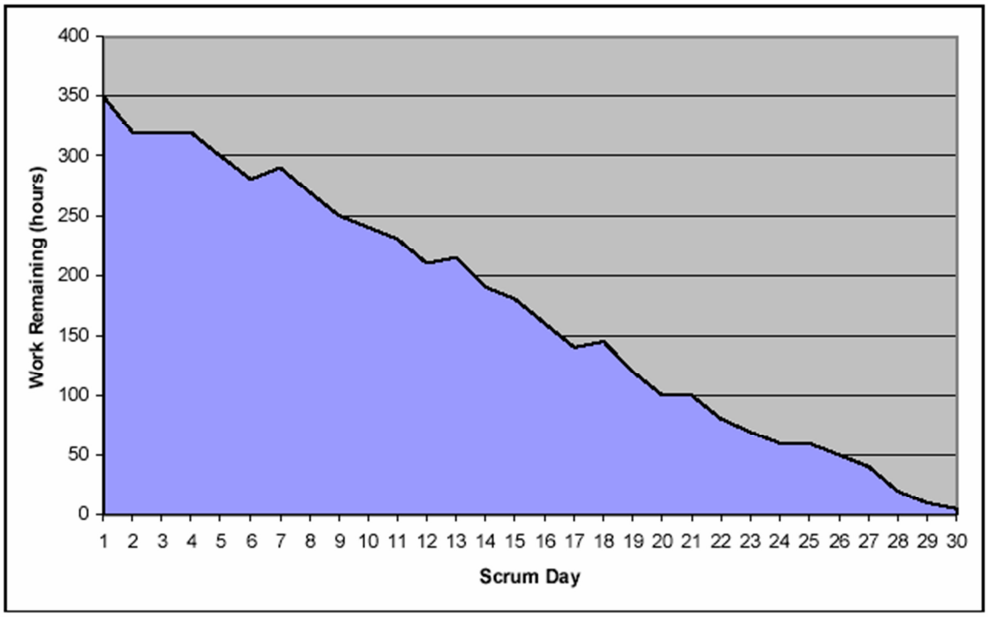
❖ **Sprint backlog**

مجموعه‌ای از موارد کاری و تکنیکی که برای تکرار جاری[1] زمانبندی شده‌اند، در آن وجود دارند. نیازها در این فرآورده به وظایف تبدیل می‌شوند. برای هر وظیفه یک شرح کوتاه وجود دارد و مشخص می‌شود که چه کسی مسئول انجام آن است و همچنین وضعیت و تعداد ساعات باقیمانده تا تکمیل شدن هر وظیفه در این فرآورده مشخص می‌شود. backlog باید بصورت روزانه به روز شود.

❖ **Sprint burndown chart**

ساعات باقیمانده برای تکمیل شدن همه وظایف مربوط به یک sprint را در قالب یک گراف بصورت برجسته نمایش می‌دهد. در شکل زیر یک مثال ساده از آن دیده می‌شود. شکل ۱۲-۸ این نمودار را نشان می‌دهد.

[1] Current Iteration



شکل ۱۲-۸- نمودار Sprint BurnDown Chart در متدولوژی Scrum

۱۲-۷-۳- نقش‌ها و مسئولیت‌ها

شش نقش مختلف در فرآیند توسعه Scrum وجود دارد که هر یک اهداف و وظایف خاصی دارند. این نقش‌ها عبارتند از:

- **Scrum Master**

این نقش یک نقش مدیریتی جدید است که در Scrum معرفی شده است.

این نقش باید اطمینان بدهد که پروژه بر مبنای فعالیت‌ها، متغییرها و قوانین Scrum در حال انجام است و مطابق طرح پیشرفت می‌کند. رئیس scrum در طول پروژه با مشتریان و تیم توسعه در تعامل است.

- **Product Owner**

مالک محصول بطور رسمی در قبال مدیریت و کنترل پروژه و تهیه backlog مسئولیت دارد. او توسط رئیس Scrum، مشتری و مدیر انتخاب می‌شود. او تصمیم نهایی را در مورد تولید backlog اتخاذ می‌کند و همچنین در تخمین کار لازم برای موارد موجود در backlog دخالت می‌کند.

- **Scrum Team**

تیم توسعه وظیفه دارد که در مورد فعالیت‌های ضروری تصمیم‌گیری کند و به نحوی کار سازماندهی را انجام دهد که به اهداف هر sprint برسیم. از جمله وظایف اصلی تیم توسعه می‌توان به موارد زیر اشاره کرد:

- تخمین میزان کار لازم برای پروژه

- ایجاد و طراحی backlog مربوط به محصول
- بازبینی backlog مربوط به محصول
- پیش بینی موانع موجود و تلاش برای رفع آنها

❖ **Customer**

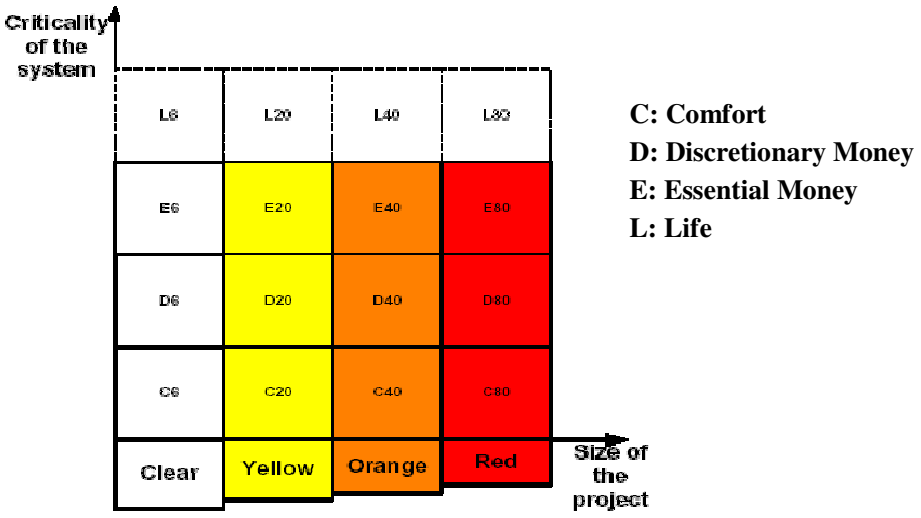
مشتری در فعالیت‌های مربوط به تولید موارد موجود در Backlog سهیم است.

❖ **Management**

تصمیم‌گیری نهایی در پروژه بر عهده مدیر است. علاوه بر این مدیر در تنظیم اهداف و نیازها دخالت می‌کند.

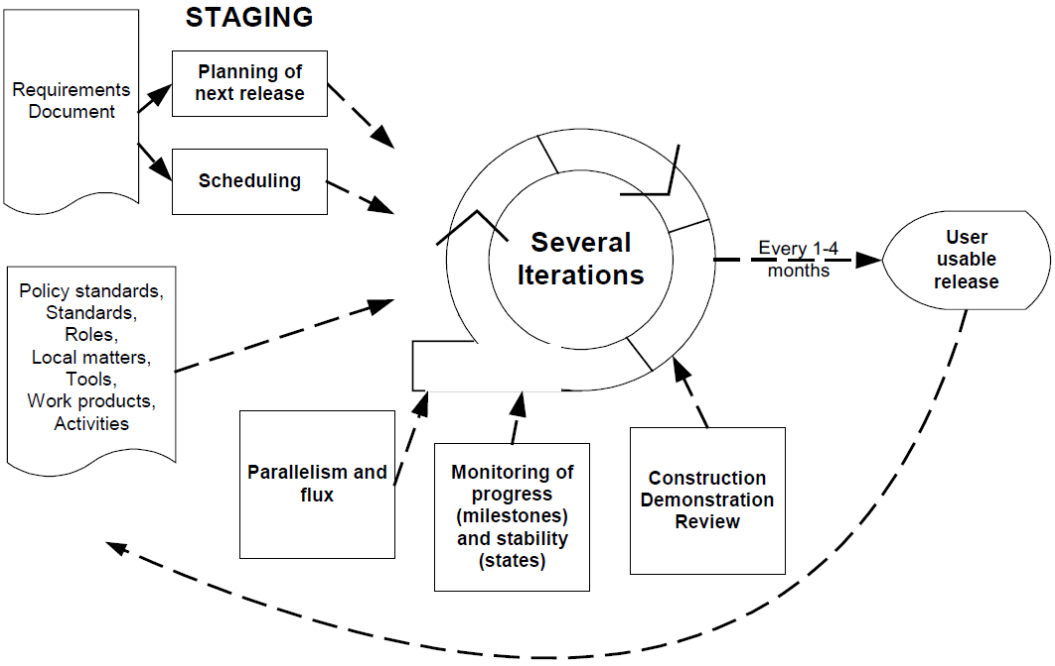
۱۲-۸-متدولوژی‌های خانواده Crystal

شامل مجموعه‌ای از متدولوژی‌های متفاوت است که مناسبترین آنها برای هر پروژه منحصر به فرد انتخاب می‌شود. دارای اصولی هستند که متدولوژی‌ها را برای شرایط مختلف موجود در پروژه‌ها سفارشی می‌کند. روش Crystal پیشنهاد می‌کند که یک متدولوژی مناسب براساس اندازه و میزان بحرانی‌بودن پروژه انتخاب شود. برای این کار هر عضو از خانواده Crystal با یک رنگ مشخص می‌شود که میزان سنگینی متدولوژی را نشان می‌دهد. رنگ تاریکتر نشان دهنده متدولوژی سنگین‌تر است. شکل ۱۲-۹ متدولوژی‌های خانواده Crystal را به‌همراه درجه سنگینی آنها نشان می‌دهد.



شکل ۱۲-۹- ابعاد متدولوژی‌های خانواده Crystal

در متدولوژی‌های خانواده Crystal تمامی پروژه‌ها از توسعه افزایشی با بازه زمانی حداکثر ۴ ماه استفاده می‌کنند. این متدولوژی‌های تأکید زیادی روی ارتباطات و همکاری بین افراد درگیر در پروژه است و هیچ فعالیت یا ابزاری را برای توسعه محدود نمی‌کنند. به‌عنوان نمونه می‌توان از فعالیت‌های XP و Scrum با هم در این روش استفاده کرد. شکل ۱۲-۱۰ نمونه‌ای از چرخه در Crystal نارنجی را نشان می‌دهد. این نوع Crystal شامل چند تیم است.



شکل ۱۲-۱۰- یک تکرار در Crystal نارنجی

۱۲-۹-مزایا و معایب متدولوژی‌های چابک

متدولوژی‌های چابک نسبت به سایر متدولوژی‌های مزایای و معایب خاص خود را دارند. مزایای این متدولوژی‌های را می‌توان به صورت زیر عنوان کرد:

- تأکید روی شرکت‌دادن مشتری در پروژه‌ها است که در پروژه‌های کاربردی بسیار مفید است
- تأکید روی کارگروهی و ارتباط متقابل که در بالا بردن راندمان کاری نقش مهمی دارد
- همه افراد درگیر در پروژه در قبال کیفیت محصول مسئولند
- سنجش مستمر کارهای انجام شده از مزایای بسیار مفید این روش‌ها است
- توسعه افزایشی که با روش‌های مدرن توسعه نرم‌افزار سازگار است
- طراحی ساده و روشن برای فرآیند توسعه

- بازبینی‌های مستمر که به بالا رفتن کیفیت کار برنامه‌نویسان کمک می‌کند

عمده‌ترین معایب متدولوژی‌های چابک عبارتند از:

- بدلیل کمبود فعالیت‌های طراحی در این روش‌ها، اگر کد بیش از چند هزار خط باشد ممکن است فرآیند توسعه با موانع خطرناکی برخورد کند
- کمبود مستندات مربوط به طراحی در این روش‌ها آنها را به پروژه‌های کوچک محدود می‌کند و قابلیت‌های استفاده مجدد را در آنها محدود می‌کند
- کمبود فرآیندهای بازبینی ساخت‌یافته
- کمبود فرآیند طراحی منظم و استفاده از بازبینی‌های غیر ساخت‌یافته باعث اتلاف زمان و هزینه می‌شود
- کمبود طرح کیفیت. در این روش‌ها هیچ نوع طرح استانداردی برای ارزیابی کیفیت وجود ندارد
- کمبود راهنماهای آموزشی برای استفاده از این روش‌ها

۱۳- الگوهای طراحی

در سال‌های اخیر توسعه سریع نرم‌افزار همواره یکی از دغدغه‌های شرکت‌های نرم‌افزار و ذی‌نفعان نرم‌افزار بوده است. تابحال روش‌های متفاوتی برای دستیابی به این هدف در فصل‌های گذشته مورد بررسی قرار گرفت که روش‌های چابک از جمله آخرین روش‌ها است. اما جنبه دیگری از توسعه سریع در نرم‌افزار می‌تواند از طریق استفاده مجدد مفاهیم، کدها، کلاس‌ها و حتی واسط نرم‌افزار اتفاق بیفتد. چیزی که در حال حاضر، شرکت‌های نرم‌افزاری بسیاری از آن سود می‌برند و سعی به استفاده از کارهای قبلی خود برای توسعه نرم‌افزار فعلی خود دارند. این موضوع به‌عنوان الگو[1] در نرم‌افزار شناخته شده که سابقه‌ای طولانی در نرم‌افزار و حتی سایر علومی دارد که با تجربه اجرایی همراه هستند.

استفاده از الگوها عمدتاً در زمان پیاده‌سازی نرم‌افزار و بیشتر توسط برنامه‌نویسان مطرح و مورد استفاده قرار می‌گیرد، اما استفاده از الگوها در طراحی و حتی معماری نرم‌افزار سبب ایجاد فضای تازه‌ای در توسعه نرم‌افزار شده است. در ادامه به بررسی این فضای پیش‌رو در مهندسی نرم‌افزار خواهیم پرداخت. به همین منظور ابتدا به بررسی عوامل ایجاد الگوهای طراحی، ارائه تعریف الگو، بیان تاریخچه و ساختارهای متداول ارائه شده برای الگوها خواهیم پرداخت. سپس ضدالگوها که مفاهیم جدیدی در الگوهای طراحی هستند را مورد کنکاش قرار خواهیم داد.

۱۳-۱-عوامل ایجاد الگوهای طراحی

استفاده از شی‌گرایی در توسعه نرم‌افزار منجر به تحولات بسیاری در حوزه نرم‌افزار شد. شی‌گرایی ابتدا در پیاده‌سازی نرم‌افزار و سپس در طراحی و معماری نرم‌افزار مطرح شد و سبب استفاده از ایده‌هایی که لزوماً سابقه‌ای در برنامه‌نویسی رویه‌ای نداشتند، شد. از جمله این موارد می‌توان به الگوهای طراحی اشاره نمود که در برنامه‌نویسی رویه‌ای امکان بکارگیری نداشته و به‌همین دلیل سابقه‌ای آن به بعد از استفاده از شی‌گرایی در مهندسی نرم‌افزار برمی‌گردد. به‌عبارت بهتر، خصوصیات شی‌گرایی (تجرید، محصورسازی، واحدبندی و سلسله مراتب) اجازه داده است تا الگوها بتوانند در مهندسی نرم‌افزار مورد استفاده قرار گیرند اما عواملی سبب شده‌اند که برنامه‌نویسان، طراحان و حتی معماران نرم‌افزار به سمت

[1] Pattern

استفاده از الگو تمایل پیدا نمایند را می‌توان به‌صورت ذیل عنوان نمود. این عوامل در گذر زمان ایجاد شده و با تجربیاتی که در توسعه نرم‌افزار شی‌گرا انجام شد، حاصل شده‌اند.

- اغلب روش‌های تحلیل و طراحی شی‌گرا تاکید بسیاری بر استفاده از نمادها در طراحی دارند
  - برای درک بهتر مفاهیم شی‌گرا مورد استفاده در توسعه نرم‌افزار نیاز است که نمادها و مفاهیم مشترک و یکسانی توسط برنامه‌نویسان، طراحان و سایر ذی‌نفعان مورد استفاده قرار گیرد. این موضوع دلیل ایجاد زبان یکپارچه UML است. نمادها و مفاهیم همچنین برای مستندسازی و ذکر خصوصیات می‌توانند مورد استفاده قرار گیرند تا افراد بعدی که مستندات نرم‌افزار را می‌خوانند، بتوانند به درک درستی از چگونگی و مفاهیم مورد استفاده در نرم‌افزار دست یابند. همین امر سبب شده است که ایجاد الگو از کارهای گذشته ساده‌تر انجام شود.
- تحلیل و طراحی شی‌گرا متمرکز بر رسم نمودارهای مختلف است
  - این موضوع که شی‌گرایی بر مدلسازی تصویری تاکید دارد را می‌توان به‌عنوان یکی از نقاط قوت این روش نام برد، چرا که مدلسازی تصویری به‌عنوان یکی از بهترین تجربیات توسعه نرم‌افزار مطرح است. اما تحلیل و طراحی شی‌گرا تنها رسم نمودار نیست و نقاشی خوب دلیل بر طراحی خوبی نیست. در طراحی شی‌گرا مفاهیم استفاده شده، نشان‌دهندهٔ تفکرات طراح هستند که برنامه‌نویس می‌بایست آنها را پیاده‌سازی نماید. استفاده از ارتباطات بین کلاس‌ها که به‌صورت نمودار ارائه می‌شود، کمک می‌کند تا الگوهای طراحی بهتر تشخیص داده شوند. (قسمتی از نمودارها با توجه به قواعد ارتباطی شبیه یکدیگر هستند)
- طراحی شی‌گرای خوب نیاز به سال‌ها تجربه دارد
  - موضوع مهمی که سبب شده است که الگوهای طراحی بیشتر مورد توجه قرار گیرند، نیاز به استفاده از تجربیات قبلی طراحان است. در واقع، کسب تجربه در طراحی شی‌گرا نیازمند تجربه بسیار در توسعه نرم‌افزار است و دانستن طراحی نرم‌افزار به اندازه دانستن گرامر زبان اهمیت دارد. الگوهای طراحی همانطور که در ادامه خواهیم دید کمک می‌کنند که این تجربیات منتقل شوند.

- بیشترین استفاده مجدد در هنگام طراحی اتفاق می‌افتد
  - سال‌ها تجربه در مهندسی نرم‌افزار نشان داده است که بیشترین حجم استفاده مجدد در طراحی و پیاده‌سازی نرم‌افزار اتفاق می‌افتد. در واقع، طراحی اولین مرحله از توسعه نرم‌افزار است (طراحی معماری نیز شامل این موضوع است) که می‌توان از فعالیت‌های گذشته در توسعه نرم‌افزار استفاده نمود. تجربه طراح کمک شایانی به چگونگی استفاده مجدد می‌نماید. الگوهای طراحی سبب افزایش قابلیت استفاده مجدد می‌شوند.

این عوامل سبب شدند که در گذر زمان الگوهای طراحی ایجاد و مورد استفاده قرار گیرند. در واقع، مجموعه‌ای از ساختارهایی که در هر نرم‌افزار تکرار می‌شوند، پیدا شده و سپس مورد استفاده قرار گرفتند. این ساختارهای تکرار شونده در سیستم‌های شی‌گرا که با تعریف دقیق‌تر آنها را الگو خواهیم نامید، سبب ارتقای تجرید، انعطاف‌پذیری، واحدبندی و ظرافت[1] شدند و حاوی اطلاعات ارزشمند طراحی هستند.

در اغلب موارد ساختارهای تکرارشده در یک نرم‌افزار شی‌گرا نمی‌تواند به‌صورت مستقیم توسط نرم‌افزار دیگر مورد استفاده قرار گیرد (با توجه به تفاوت مسئله، حوزه و راه‌حل) به‌همین دلیل شناسایی، دسته‌بندی و چگونگی استفاده از این ساختارهای تکرار شوند عمده‌ترین مسئله در توسعه نرم‌افزار شی‌گرا است.

۱۳-۲-تعریف الگوی طراحی

تعاریف مختلفی برای الگو ارائه شده است که یکی از بهترین تعاریف توسط Drik Riehle و Heniz Zullighoven ارائه شده است که بصورت گسترده‌ای مورد استفاده قرار گرفته است. در این تعریف، الگو «تجریدی از یک شکل عینی که تکرار را در زمینهٔ دلخواه نگاه می‌دارد» عنوان شده است. بعنوان نمونه، در توسعه نرم‌افزار این تکرار بیشتر شامل مسائل طراحی می‌شود و الگوها با مسائل طراحی درگیر هستند. هر الگو نشان می‌دهد که چگونه مسئله خاصی را با راه‌حل خاصی حل می‌شود. اما الگو بیشتر از یک راه‌حل است و بیانگر این است که مسئله مورد نظر در زمینهٔ خاصی اتفاق می‌افتد که در آن علاقه‌مندی‌های دیگری نیز وجود دارند. در واقع، راه‌حل پیشنهادی یک الگو حاوی نوعی ساختار است

[1] Elegance

که بین علاقه‌مندی‌های خاص یا اجبارها توازن برقرار می‌نماید تا بهترین راه‌حل برای مسئله در زمینه مورد نظر ارائه شود. هر الگوی طراحی شامل مجموعه‌ای از وابستگی‌ها، ساختارها، تعاملات و قراردادها[1] است.

هر الگو نام‌ها و ساختار طراحی را به‌صورت صریح تعیین می‌کند و راه‌حلی نمونه برای مسئله‌ای در زمینه خاص مطرح می‌کند. در واقع، الگوها در تجربیات قبلی پیدا شده‌اند و ابداع نشده‌اند و راهی برای تسهیل ارتباطات بین ذینفعان هستند. یک الگوی خوب کارهای زیر را انجام می‌دهد:

- مسئله‌ای را حل می‌کند: الگوها راه‌حل‌ها را نشان می‌دهند نه راهبرد یا مفاهیم.
- یک مفهوم اثبات شده است: الگوها راه‌حل‌هایی اثبات شده برای انجام مسئله هستند.
- راه‌حل مشهود نیست: روش‌های حل مسئله زیادی سعی به اشتقاق راه‌حل از مفاهیم اولیه می‌نمایند. بهترین الگوها راه‌حلی غیر مستقیم برای یک مسئله تولید می‌کند.
- یک رابطه را توصیف می‌کنند: الگوها نه تنها ماژول‌ها را توصیف می‌کنند بلکه ساختارهای سیستم و مکانیزم‌ها را عمیق‌تر توصیف می‌کنند.

۱۳-۳-طبقه‌بندی الگوها

عموماً الگوها به سه دسته تقسیم می‌شوند اما به دلیل پذیرش زیاد کتاب Gang of Four و اینکه این کتاب اولین کتابی که در زمینه الگوهای معماری ارائه شده است اغلب تنها بر روی یک نوع الگو یعنی الگوهای طراحی تمرکز دارند و به هر نوع الگوی که در زمینه معماری باشد، الگوی طراحی گویند. طبقه‌بندی‌های متفاوتی از الگوها ارائه شده است که Frank Buschmann and et. al. در کتاب خود سه دسته از الگوهای معماری را به صورت زیر معرفی می‌نمایند:

- **الگوهای معماری**[2]: الگوی معماری، طرح یا سازوکار ساختاری پایه را برای سیستم نرم‌افزار بیان می نماید. این الگو مجموعه‌ای از زیرسیستم‌های تعریف شده را فراهم می‌آورد، وظایفشان را مشخص می‌کند و دارای قوانین و خطوط راهنمائی برای سازماندهی ارتباط بین آنهاست.
- **الگوهای طراحی**[3]: الگوی طراحی، مدلی برای تصحیح زیرسیستم یا مولفه‌های سیستم نرم‌افزاری یا ارتباط بین آنها را بیان می‌کنند. این الگو ساختارهای تکراری معمول در مولفه‌های

[1] Conventions
[2] Architectural Patterns
[3] Design Patterns

ارتباطی را توصیف می‌کند که در واقع مسائل طراحی عمومی را بدون داشتن زمینه (هر الگو دارای زمینه خاص است) خاص حل می‌کند.

- **Idiom** :Idiom یک الگوی سطح پائین مخصوص زبان برنامه‌نویسی است. یک idiom چگونگی پیاده‌سازی جنبهٔ خاصی از مولفه‌ها یا ارتباط بین آنها را با استفاده از خصوصیات زبان داده شده توصیف می‌کند.

تفاوت بین این سه نوع الگو در سطوح تجرید و جزئیات آنهاست. الگوهای معماری، راهبردهای سطح بالایی هستند که با مولفه‌های بزرگ و خصوصیات عمومی و مکانیزم‌های سطح بالای سیستم مرتبط است. این الگوها ساختارهای کلی یک سیستم نرم‌افزاری را تحت تاثیر قرار می‌دهد و بر تمام جنبه‌های سیستم نرم‌افزاری تاثیر می‌گذارند. الگوهای طراحی تاکتیک‌های سطح میانه هستند که برخی ساختارها و رفتارهای موجودیت‌ها و ارتباطات بین آنها را مشخص می‌کنند. این الگوها ساختار کلی سیستم را تحت تاثیر قرار نمی‌دهند، اما در عوض ریزمعماری‌های[1] زیر سیستم‌ها و مولفه‌ها را مشخص می‌کند. به عبارت بهتر، این الگوها بر هر زیرسیستم تاثیر می‌گذارند و از الگوهای معماری تاثیر می‌پذیرند. Idiom ها تکنیک‌های برنامه‌نویسی مخصوص زبان و مخصوص پارادایم هستند که جزئیات خارجی و داخلی ساختار یا رفتار یک مولفه را توصیف می‌کند.

Riehle و Zullighoven نیز طبقه‌بندی خاصی را برای الگوها ارائه نموده‌اند. این دو محقق، الگوها را به سه دسته ذیل تقسیم نموده‌اند:

- **الگوهای مفهومی** : شکل این الگوها با اصطلاحات و مفاهیم دامنهٔ یک کاربرد توصیف می‌شود.
- **الگوهای طراحی**: شکل این الگوها با ساختارهای طراحی نرم‌افزار مثل اشیاء، کلاسها، وراثت، اجتماع و ارتباطات از نوع use توصیف می‌شود.
- **الگوهای برنامه‌نویسی**: شکل این الگوها بوسیله ساختارهای زبان برنامه‌نویسی توصیف می‌شود.

با مقایسه دو دسته طبقه‌بندی بالا می‌توان الگوهای برنامه‌نویسی را معادل با idiom در نظر گرفت. برای دو نوع دیگر از الگوهای ذکر شده در بالا، نویسندگان طبقه‌بندی اول، مجموعه خود را براساس دیدگاه

[1] Micro-architecture

معماری بیان نمودند. درحالیکه نویسندگان دوم از این دیدگاه آنها را طبقه‌بندی نمودند که آیا آنها زبان را از فضای مسئله استخراج می‌کنند یا از فضای راه‌حل استخراج می‌کنند.

۱۳-٤-تاریخچه الگوی طراحی

استفاده فعلی از اصطلاح الگو از نوشته‌های کریستوفر الکساندر[1] معمار که کتاب‌های بسیاری در زمینه برنامه‌ریزی شهری و معماری ساختمان دارد، گرفته شده است. او در کتاب A Pattern Language سعی به ارائه زبانی برای الگوهای طراحی مورد استفاده در ساختمان نمود تا بتواند سرعت طراحی ساختمان را افزایش دهد. در سال ۱۹۸۷ دو محقق بنام‌های Ward Cunningham و Kent Beck با زبان برنامه‌نویسی Smalltalk و روی طراحی واسط کاربر کار می‌کردند که تصمیم گرفتند از برخی از ایده‌های الکساندر برای توسعه یک زبان کوچک پنج الگویی استفاده نمایند. آنها در مقاله‌ای که در کنفرانس OOPSLA'87 ارائه نمودند، الگوهای طراحی را معرفی نمودند.

پس از معرفی رسمی الگوها، Jim Coplien با استفاده از نتیجهٔ کار منتشر شده دو محقق ذکر شده، کاتالوگی از idiom های زبان C++ را مهیا نمود و در کتابی آن را منتشر نمود. از سال ۱۹۹۰ تا ۱۹۹۲ اعضای مختلف Gang of Four[2] با یکدیگر ملاقات نموده و کارهایی را در زمینه تهیه یک کاتالوگ از الگوها انجام دادند و در کارگاه آموزشی شی‌گرایی در سال ۱۹۹۱ آنها را معرفی نمودند. در آگوست سال ۱۹۹۳، Kent Beck و Grady Booch با یکدیگر در کلورادو ملاقات نمودند که در واقع این ملاقات پایه‌گذار گروه Hillside شد. کمی بعد از آن کتاب معروف «الگوهای طراحی» توسط Gang of Four منتشر شد که اولین کتاب معتبر در زمینه الگو است.

۱۳-۵-ساختار الگوی طراحی

هر الگو دارای قالب و ساختاری است که نشان‌دهندهٔ خصوصیات آن الگو است. ساختارهای متعددی برای بیان الگوهای طراحی پیشنهاد شده است که هر یک سعی به مستندسازی خصوصیاتی از الگوها دارند که بتوانند در توسعه نرم‌افزار از آنها استفاده نمایند. هر الگو دارای چهار بخش اصلی است که در همه ساختارهای پیشنهادی برای الگو وجود دارند که عبارتند از:

- نام الگو: معمولاً برای الگو نامی انتخاب می‌شود تا گویای عملکرد الگو باشد

[1] Christopher Alexander
[2] John Vlissides, Ralph Johnson, Richard Helm, and Erich Gamma

- مسئله: نشان می‌دهد که چه‌وقت یک الگو می‌بایست مورد استفاده قرار گیرد. مسئله می‌تواند شامل بیان مشکلات طراحی نیز باشد.
- راه‌حل: عناصر طراحی، ارتباط بین آنها، مسئولیت هر یک و نحوهٔ همکاری بین آنها را بیان می‌کند. در این قسمت، راه‌حل دقیق نشان داده نمی‌شود بلکه الگو مانند یک قالب کلی می‌ماند که می‌تواند به مسائل بیان شده اعمال شود.
- نتایج: توازن‌های ایجاد شده و نتایج اعمال الگو در این قسمت بیان می‌شود.

در ادامه به بررسی چند ساختار پیشنهادی برای الگو خواهیم پرداخت.

۱۳-۵-۱- ساختار پیشنهادی الکساندر

الکساندر در کتاب A Pattern Language ساختاری اولیه‌ای را برای نگهداری اطلاعات مربوط به الگوها و استفاده از آنها پیشنهاد داده است که می‌توان آن را از جمله اولین ساختارهای مطرح در حوزه الگو دانست. این ساختار شامل نام الگو، زمینه الگو، مسئله (اجبارهای) که الگو حل می‌کند، راه‌حلی (پیکربندی و چگونگی راه‌حل) که الگو ارائه می‌دهد و بیان ارتباط زمینه، مسئله و راه‌حل است. زمینه الگو پیش شرایطی که بنظر می‌رسد تحت آن، مسئله و راه‌حل آن تکرار می‌شوند و راه‌حل ارائه شده برای آن مسئله مطلوب است. زمینه، نشان‌دهندهٔ کاربرد الگو است. می‌توان اینطور عنوان نمود که پیش‌شرایط نشان‌دهندهٔ پیکربندی اولیه سیستم قبل از اعمال الگو هستند. به عبارت دیگر، زمینه نشان‌دهندهٔ فضایی است که الگو در آن فضا تکرار می‌شود. تکرار الگو به معنی تکرار مسئله و راه‌حل الگو است.

۱۳-۵-۲- ساختار پیشنهادی Polti

ساختار دیگری که برای الگو پیشنهاد شده است، توسط Georges Polti و Lucille Ray در کتاب Thirty-Six Dramatic Situations ارائه شده است که نسبت به ساختار پیشنهادی الکساندر دیدگاه متفاوتی ارائه می‌دهد. این ساختار عبارتند از: نام الگو، عناصر الگو، توصیف الگو، نمونه‌های الگو، موارد استفاده الگو و جزئیات الگو. توصیف الگو را می‌توان معادل زمینه الگو در ساختار الکساندر دانست. وجود عناصر الگو سبب کاربردی‌تر شدن الگوهای مطرح شده با این ساختار می‌شود. همچنین بیان جزئیات الگو سبب می‌شود که تمام زوایای دیگری که الگو بر آنها اثر دارد، مورد بررسی قرار گیرند.

۱۳-۵-۳- ساختار پیشنهادی الگوی GoF

Gang of Four قالبی را برای الگوهای معماری ارائه داده‌اند که اغلب مورد استفاده قرار می‌گیرد و به «قالب GoF» مشهور است. خصوصیات پیشنهادی GoF اغلب مواردی که قالب‌های دیگر پوشش داده نشده است، در بردارد. خصوصیات پیشنهادی GoF برای هر الگو عبارتند از:

- **نام**: هر الگو باید دارای یک نام مشخص باشد. نام الگو اجازه می‌دهد تا یک کلمه یا عبارت کوتاه برای اشاره به الگو، خصوصیات و دانشی که الگو بهمراه دارد، مشخص شود. نام الگوی خوب، واژگانی برای اشاره به خصوصیات مفهومی الگو شکل می‌دهد. برخی اوقات الگو بیش از یک نام دارد، در این مورد بهتر است دیگر اسامی الگو مستندسازی شده تا بتوانند در صورت نیاز مورد استفاده قرار گیرند. برخی از اسامی مورد استفاده برای الگوها، علاوه بر نام مورد نظر، نشان‌دهندهٔ طبقه‌ای از الگوها هستند.
- **طبقه‌بندی**[1]: طبقه‌بندی براساس کاری که الگو انجام می‌شود. الگوها از نظری کاری که انجام می‌دهند به سه دسته زیر تقسیم می‌شوند:
  - تولیدکننده[2]: بر روی فرآیند ایجاد اشیا و کلاس‌ها تمرکز دارند
  - ساختاری: بر روی ترکیب کلاس‌ها و اشیا تمرکز دارند
  - رفتاری: بر روی رفتار کلاس‌ها و اشیا و توزیع وظیفه تمرکز دارند

به نظر الکساندر بهترین الگوها، الگوهایی هستند که تولیدکننده هستند. الگوهای تولیدکننده فعال و پویا هستند. این الگوها نشان می‌دهند چگونه می‌توان چیزی را ایجاد نمود و می‌توانند در معماری‌های سیستم حاصل که در آن مشارکت داشته‌اند، مشاهده شوند. برخلاف این الگوها، الگوهای غیر تولیدکننده، ثابت و انفعالی هستند. این الگوها پدیده‌های تکراری را بدون نیاز به گفتن اینکه چطور آنها را مجدداً تولید نمود، توصیف می‌کنند. اغلب تلاش می‌شود که الگوهای تولیدکننده مستند شوند بدین دلیل که آنها نه تنها خصوصیات سیستم‌های خوب را نشان می‌دهند بلکه یاد می‌دهند چگونه آنها را بسازیم. جدول ۱۳-۱ طبقه‌بندی برخی از الگوهای طراحی را که توسط GoF انجام شده است، نشان می‌دهد.

[1] Classification
[2] Creational

Patterns Classification		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

جدول ۱۳-۱- طبقه‌بندی الگوهای طراحی توسط GoF

- **منظور**[1]: جمله‌ای که قصد مسئله مورد نظر الگو را شرح می‌دهد. به عبارت دیگر، در مسئله الگو به اهداف و موضوعاتی که الگو می‌خواهد در یک زمینه و اجبار خاص به آن بپردازد، توجه می‌شود. اغلب، اجبارهای الگو با اهداف مطرح شده در مسئله در تضاد هستند و الگو می‌بایست به روش‌هایی بین آنها توازن ایجاد نماید. بیان منظور الگو به‌صورت شفاف کمک می‌کند تا درک بهتری از کاری که الگو برای آن ایجاد شده، بدست آید.
- **نام‌های مشابه**: در صورتی که الگو دارای اسامی شناخته شده دیگری است، قید می‌شود.
- **انگیزه**[2]: سناریویی که مشکل طراحی و اینکه چگونه کلاس‌ها و اشیای موجود در الگو مشکل را حل می‌کنند، نشان می‌دهد. این سناریو کمک می‌کند تا درک بهتری از توصیفاتی مربوط به الگو داشته باشید. عمده‌ترین قسمت هر الگو بخش انگیزه آن است که نشان می‌دهد الگو توانایی حل مسئله را دارد و چگونگی حل مسئله توسط الگو تشریح می‌شود.
- **کاربردپذیری**[3]: وضعیتی که الگو می‌تواند به کار برده شود و همچنین نحوهٔ شناسایی این وضعیت در اینجا بیان می‌شود. نمونه‌هایی از طراحی‌های ضعیف که الگو می‌تواند آنها را تقویت نماید و تشخیص اینکه در چه حالت‌هایی الگو می‌تواند بکار برده شود در این بخش بیان می‌شود.

[1] Intent
[2] Motivation
[3] Applicability

- **ساختار**: نمایشی گرافیکی از کلاس‌ها در الگو با استفاده از نمادگذاری OMT (با توجه به اینکه نماد UML در سال ۱۹۹۵هنوز نهایی نشده بود، می‌توان از UML استفاده نمود). برای نمایش تعاملات می‌توانید از نمودار ترتیبی یا نمودار همکاری استفاده نمائید. کلاس‌های کلیدی و ارتباط بین آنها در این بخش نمایش داده می‌شود.
- **شرکاء**[1]: کلاس‌ها و/یا اشیایی که در الگوی طراحی و انجام وظایفش مشارکت می‌کنند. برای هر یک از شرکاء تعریفی که نشان‌دهندهٔ نقش آنهاست، بیان می‌شود.
- **همکاران**: نحوهٔ همکاری شرکاء در انجام وظایف الگوی طراحی در این قسمت بیان می‌شود. در واقع، مفاهیم و ساختاری که در بخش‌های انگیزه و ساختار به آنها پرداخته شده‌اند و در الگو مشارکت می‌کنند، در بخش قبل و این قسمت تشریح می‌شوند.
- **نتایج**: در این بخش در مورد اینکه الگو چگونه اهداف مورد نظر خود را برآورده می‌سازد، چه توازن‌ها و نتایجی با به کار بردن الگو بدست می‌آید و چه جنبه‌هایی از ساختار سیستم می‌توانند متفاوت باشند، صحبت می‌شود و نحوهٔ پاسخ به این سوالات نشان داده می‌شود.
- **پیاده‌سازی**: نکات فنی، نقاط ضعف و دیگر نکاتی که باید در استفاده از الگو مد نظر قرار داد، در اینجا مورد بررسی قرار می‌گیرند.
- **نمونه کد**: بخش‌های از کد الگو به زبان C++ یا Smalltalk در این قسمت بیان می‌شود.
- **استفاده شناخته شده**[2]: در این قسمت استفاده‌هایی که از این الگو شده که شناخته شده هستند و همچنین کاربردهای الگو در سیستم‌های موجود بیان می‌شود. این قسمت به ارزیابی الگو با بررسی اینکه آیا این الگو براستی راه حل اثبات شده‌ای برای یک مسئله تکراری است، کمک می‌کند.
- **الگوهای مرتبط**: ارتباطات پویا و ایستای بین این الگو و دیگر الگوها در همان زبان یا سیستم را نشان می‌دهد. اغلب الگوهای مرتبط دارای نیروهای مشترک، زمینه ابتدایی و متن سازگار با دیگر الگوها هستند.

[1] Participants
[2] Known Uses

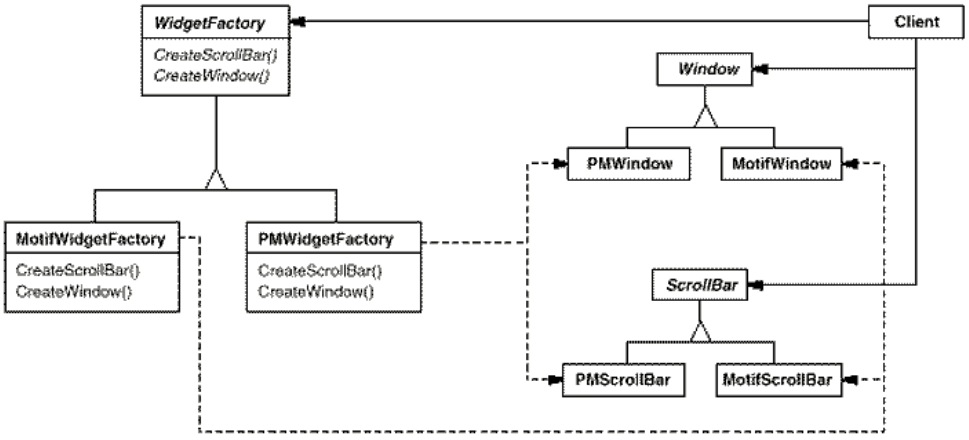
۱۳-۶-بررسی چند الگوی طراحی

در جدول ۱۳-۱ فهرست الگوهای طراحی معرفی شده توسط GoF به‌همراه دسته‌بندی آنها ارائه شده است. در ادامه به بررسی چند الگوی مطرح در طراحی نرم‌افزار خواهیم پرداخت. در بیان هر الگو از ساختار GoF برای بیان الگو استفاده شده است.

۱۳-۶-۱- الگوی طراحی Abstract Factory

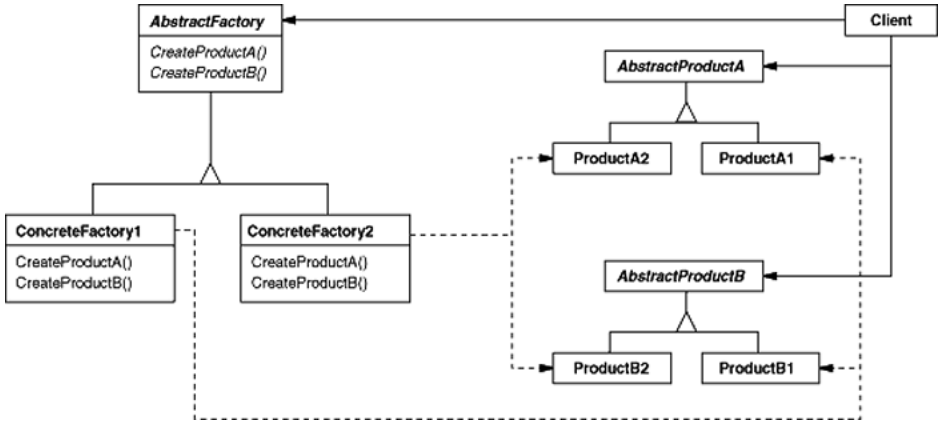
- **طبقه‌بندی**: ایجادکننده
- **منظور**: رابطی برای ایجاد خانواده‌ای از اشیای مرتبط یا وابسته بدون مشخص‌کردن کلاس‌های آنها فراهم می‌کند
- **نام‌های مشابه**: Kit
- **انگیزه**: در نرم‌افزارهایی که نیاز به حمایت از چند نوع واسط کاربری وجود داشته باشد (هر واسط کاربری مجموعه‌ای از widget ها همانند Button، Scroll bar و Window می‌باشد) نیاز است تا نرم‌افزار به‌گونه‌ای انعطاف‌پذیر طراحی شود تا برنامه‌نویس مجبور به کدنویسی برای هر یک از واسط‌های کاربری نشود. برای حل این مشکل، کلاس مجردی به‌نام WidgetFactory تعریف می‌شود که واسطی برای تعریف هر نوع widget در اختیار می‌گذارد. همچنین یک کلاس مجرد برای هر نوع از widget ها (از قبیل button و Scroll bar) وجود داشته و یک زیرکلاس عینی از Widget برای هر نوع واسط کاربری پیاده‌سازی شده است. واسط WidgetFactory دارای متدی برای ایجاد و برگشت شی جدید از هر کلاس مجرد Widget است که سرویس‌گیرنده‌ها این متد را فراخوانی می‌کنند، بدون اینکه از کلاس عینی که مورد استفاده قرار می‌دهند، مطلع باشند. به‌این ترتیب سرویس‌گیرنده‌ها مستقل از کلاس واسط کاربری خواهند شد. شکل ۱۳-۱ کلاس‌های نمونه‌ای بیان شده و ارتباط بین کلاس‌ها را نمایش می‌دهد. دو واسط کاربر MotifWidgetFactory و PMwidgetFactory از کلاس مجرد WidgetFactory ایجاد شده‌اند تا Widget مورد نظر برای هر واسط کاربری در آنها دیده شود. کلاس مجرد Window (نوع Widget) و دو کلاس عینی PMWidnow و MotiftWindow که برای هر دو نوع واسط کاربری پیاده‌سازی شده‌اند. سرویس‌گیرنده‌ها، با فراخوانی متدهای

موجـود در Widgetfactory، درخواسـت ایجـاد Widget (در اینجـا Scroll bar یـا Window) مـی‌کننـد. درخواسـت براسـاس نـوع واسـط کـاربری درخواسـتی بـه یکـی از کـلاس‌هـای MotifWidgetFactory یـا PMWidgetFactory هـدایت مـی‌شـود و ایـن کـلاس‌هـا بـسته بـه درخواست Widget مورد نظر را ایجاد و برگشت می‌دهند.



شکل ۱۳-۱- نمونه استفاده از الگوی Abstract Factory

- **کاربردپذیری**: وقتی از الگوی Abstract Factory استفاده نمائید که:
  - سیستم می‌بایست مستقل از چگونگی تولید، ترکیب و نمایش محصولاتش باشد
  - سیستم می‌بایست توسط یک یا بیشتر محصول پیکربندی شود
  - خانواده‌ای از اشیا مرتبط به‌گونه‌ای طراحی شده‌انـد کـه مـی‌بایـست از یکـدیگر اسـتفاده کنند و شما مجبور این اجبار را در نظر بگیرید
  - وقتی می‌خواهید کتابخانه‌ای از محصولات مختلف ایجاد کنید و تنهـا واسـط آنهـا را در اختیار قرار دهید
- **ساختار**: شکل ۱۳-۲ ساختار الگوی Abstract Factory را نشان مـی‌دهـد. بـا توجـه بـه شـکل و توضیحاتی که قبلاً داده شد می‌توان محصولات بیشتری به غیر از ProductA و ProductB را در AbstractFactory قرار داد و به همـین ترتیـب نیـاز اسـت تـا کـلاس‌هـای ConcreteFactory1 و ConcreteFactory2 نیز تغییر یابند. هـر ConcreteFactory هماننـد یـک واسـط کـاربر مـی‌مانـد. پیاده‌سازی هر محصول به اندازه تعداد واسط‌هایی که نیاز است، انجام می‌شود.



شکل ۱۳-۲- ساختار الگوی طراحی Abstract Factory

- **شرکاء**: برای هر کلاسی که در شکل ۱۳-۱ ارائه شده، تعریف کوتاهی به‌همراه نمونه کلاس (از شکل ۱۳-۱) بیان می‌شود
  - AbstractFactory (WidgetFactory)
    - واسطی برای عملیات تعریف می‌کند که اشیاء محصول مجرد را ایجاد می‌کند
  - ConcreteFactory (MotifWidgetFactory, PMWidgetFactory)
    - عملیات مورد نیاز برای ایجاد اشیاء محصول را پیاده‌سازی می‌کند
  - AbstractProduct (Window, ScrollBar)
    - واسطی برای هر نوع شی متعلق به محصول ایجاد می‌کند
  - ConcreteProduct (MotifWindow, MotifScrollBar)
    - شیی از محصول را تعریف می‌کند که با استفاده از ConcreteFactory مربوطه ایجاد می‌شود. در واقع، واسط AbstractProduct را پیاده‌سازی می‌کند
  - Client
    - درخواست سرویس می‌کند و تنها واسط‌های تعریف شده توسط کلاس‌های AbstractFactory و AbstractProduct را مورد استفاده قرار می‌دهد
- **همکاران**: معمولاً یک نمونه از کلاس ConcreteFactory در زمان اجرای نرم‌افزار ایجاد می‌شود. این نمونه، اشیاء هر محصول را که دارای پیاده‌سازی خاص هستند، ایجاد می‌کند. برای ایجاد اشیاء محصولات دیگر، می‌بایست ConcreteFactory دیگری ایجاد شود. AbstractFactory تنها ایجاد اشیاء محصول را به ConcreteFactory ارسال می‌دارد.

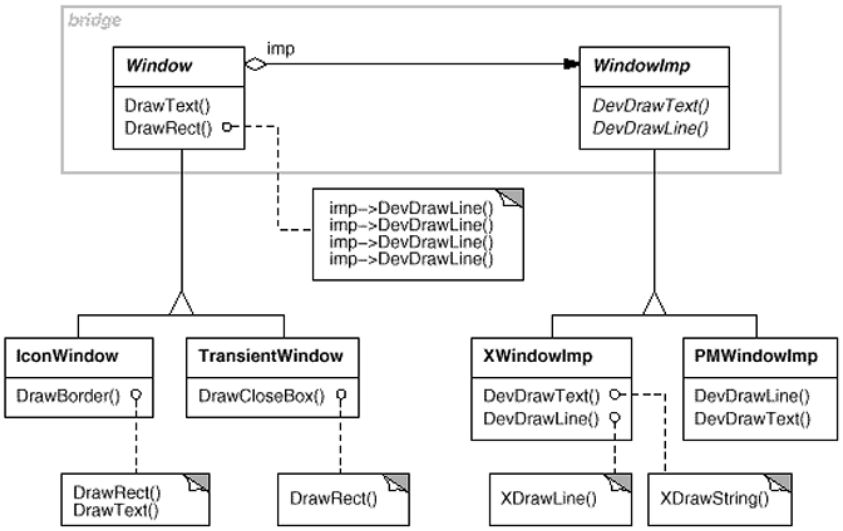
- **نتایج**: الگوی AbstractFactory دارای مزایا و معایب زیر است:
  - انعطاف‌پذیری را افزایش می‌دهد
  - تجرید را افزایش می‌دهد
  - توسعه آن برای حمایت از محصولات جدید دشوار است
- **پیاده‌سازی**
  - استفاده از پارامترها راهی برای کنترل اندازه واسط است
  - این الگو در واقع مجموعه‌ای از الگوهای Factory Method و ConcreteFactory اغلب یک Singleton (الگوی Singleton) است

۱۳-۶-۲- الگوی طراحی Bridge

- **طبقه‌بندی**: ساختاری
- **منظور**: برای جداسازی واسط مجرد(منطقی) از پیاده‌سازی‌اش (فیزیکی) مورد استفاده قرار می‌گیرد
- **نام‌های مشابه**: Handle/Body
- **انگیزه**: وقتی یک کلاس مجرد نیاز به چند پیاده‌سازی دارد معمولاً از وراثت استفاده می‌شود. در این صورت یک کلاس مجرد، واسط را به‌صورت مجرد تعریف نموده و چندین زیرکلاس عینی آن را به روش‌های مختلف پیاده‌سازی می‌کنند. اما این روش همیشه انعطاف‌پذیر نیست. وراثت یک پیاده‌سازی را به یک کلاس مجرد به‌صورت دائم نگاشت می‌کند و اصلاح و استفاده مجدد را دچار مشکل می‌کند.

  الگوی Bridge راه‌حل مناسبی برای این مسئله است و با استفاده قرار دادن پیاده‌سازی‌ها در یک کلاس دیگر این کار را انجام می‌دهد. به‌عنوان مثال در صورتیکه بخواهید یک window قابل حمل که هم بر روی سیستم‌های XWindows تحت یونیکس کار کند و هم بر روی سیستم عامل ویندوز داشته باشید، الگوی Bridge با قرار دادن یک کلاس مجرد از Window و پیاده‌سازی‌های آن در یک کلاس سلسله مراتبی مجاز مشکل را حل می‌کند. در واقع، یک سلسله مراتب کلاس برای واسط‌های Window (,IconWindow ,Window

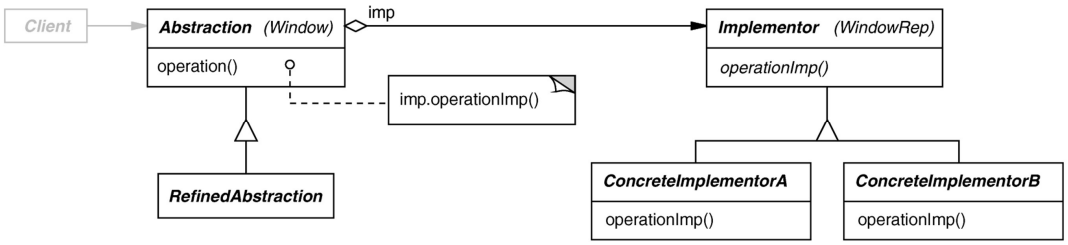
TransientWindow) و یک سلسله مراتب برای پیاده‌سازی Window مخصوص به سیستم عامل وجود دارد که ریشه آن WindowIm است. زیرکلاس XWindowImp نمونه‌ای از پیاده‌سازی Window تحت سیستم عامل یونیکس است. شکل ۱۳-۳ این مسئله را نشان می‌دهد.



شکل ۱۳-۳- نمونه استفاده از الگوی Bridge

تمام عملیات زیرکلاس Window برحسب عملیات مجرد واسط WindowImp پیاده‌سازی شده‌اند. این کار سبب می‌شود تجریدهای Windows از پیاده‌سازی‌های وابسته به سیستم عامل جدا شود. به ارتباط بین Window و WindowImp یک Bridge گویند.

- **کاربردپذیری**: وقتی از الگوی Bridge استفاده نمائید که:
  - نیاز است واسط و پیاده‌سازی‌اش باید با یکدیگر مجزا باشند
  - نیاز به یک واسط واحد برای تعامل با دیگر کلاس‌ها باشد
  - تغییر در پیاده‌سازی‌های کلاس نباید تاثیری رو سرویس‌گیرنده‌ها داشته باشد
  - نیاز است که پیاده‌سازی کلاس کاملاً از دید سرویس‌گیرندگان پنهان بماند
- **ساختار**: شکل ۱۳-٤ ساختار الگوی طراحی Bridge را نمایش می‌دهد. هر سرویس‌گیرنده به کلاس Abstraction متصل می‌شود. Abstraction با استفاده از متدهای Implementor پیاده‌سازی می‌شود. پیاده‌سازی‌های متفاوت زیرکلاس Implementor هستند.



شکل ۱۳-٤- ساختار الگوی طراحی Bridge

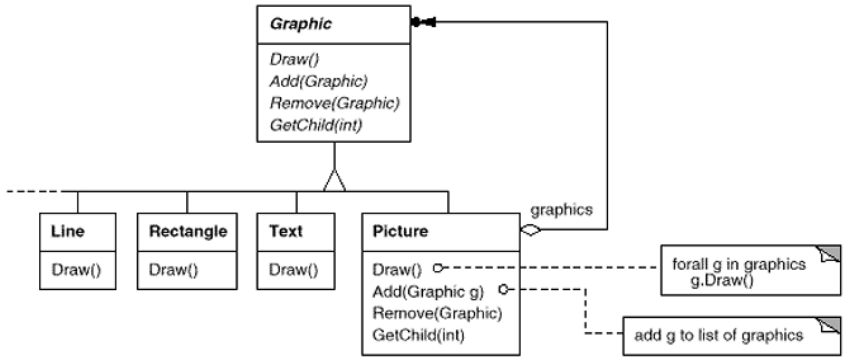
- **شرکاء**: برای هر کلاسی که در شکل ۱۳-٤ ارائه شده، تعریف کوتاهی به‌همراه نمونه کلاس (از شکل ۱۳-۳) بیان می‌شود
  - Abstraction (Window)
    - واسط تجرید را تعریف می‌کند و ارجاعی به یک شی از نوع Implementor نگاه می‌دارد
  - RefinedAbstraction (IconWindow)
    - واسط تعریف شده توسط Abstraction را توسعه می‌دهد
  - Implementor (WindowImp)
    - واسطی برای کلاس‌های پیاده‌سازی تعریف می‌کند. اجباری نیست که این واسط دقیقاً با واسط Abstraction متناظر باشد. در واقع، دو واسط می‌توانند کاملاً متفاوت باشد. واسط Implementor تنها عملیات اولیه را فراهم می‌کند و Abstraction عملیات سطح بالا را براساس این عملیات اولیه تعریف می‌کند.
  - ConcreteImplementor (XWindowImp, PMWindowImp)
    - واسط Implementor را پیاده‌سازی و پیاده‌سازی عینی‌اش را تعریف می‌کند
- **همکاران**: کلاس Abstraction درخواست‌های سرویس‌گیرنده را برای شی Implementor ارسال می دارد
- **نتایج**: الگوی Bridge مزایای زیر را ارائه می‌دهد:
  - واسط مجرد و پیاده‌سازی از یکدیگر مجزا می‌شوند
  - پیاده‌سازی می‌تواند مجزا و متغیر باشد

- واسط باید به‌گونه‌ای عمومی باشد که همه رابط‌هایی که پیاده‌سازی می‌شوند را حمایت کند

- **پیاده‌سازی**
  - تنها یک پیاده‌ساز باید وجود داشته باشد
  - ایجاد پیاده‌ساز صحیح دشوار است
  - اشتراک پیاده‌سازها باید مد نظر قرار گیرد

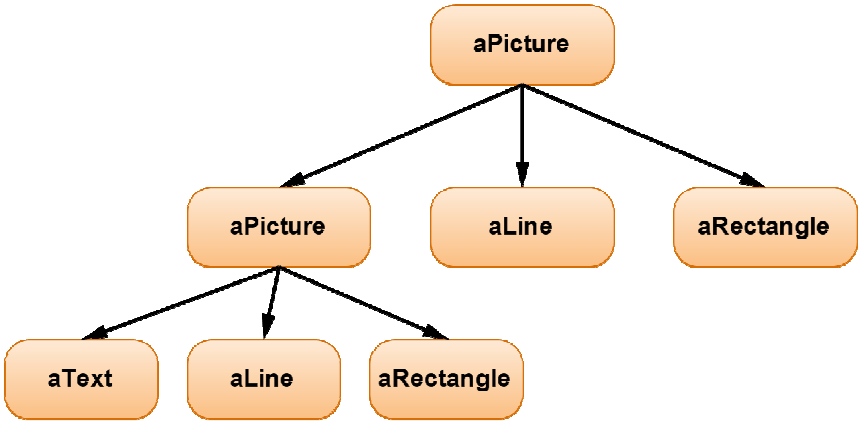
۱۳-۶-۳- الگوی طراحی Composite

- **طبقه‌بندی**: ساختاری
- **منظور**: ترکیب اشیا در ساختاری درختی برای نمایش سلسله مراتب Part-Whole
- **نام‌های مشابه**: ندارد
- **انگیزه**: نرم‌افزارهای گرافیکی همانند ویرایشگرهای متن نیازمند نگهداری اطلاعات مربوط به عناصر پیچیده‌ای که خود از عناصر اولیه مانند متن و شکل تشکیل شده‌اند، هستند. با ترکیب این اشیاء، نگهداری آنها نیز دشوارتر می‌شود. یک طراحی ساده می‌تواند بدین گونه باشد که کلاس‌های عناصر اولیه با کلاس‌های عناصری که می‌توانند عناصر اولیه را در خود جای دهند، متفاوت باشد. با این روش، کد نرم‌افزار می‌بایست بین این نوع کلاس تفاوت قائل شود در حالیکه ممکن است از نظر کاربر این دو نوع با یکدیگر متفاوت نباشند. این تفاوت سبب پیچیده شدن بیش از حد نرم‌افزار می‌شود. شکل ۱۳-۵ نمونه‌ای از این ارتباط را نشان می‌دهد.



شکل ۱۳-۵- ارتباط نمونه بین عناصر گرافیکی

با استفاده از الگوی Composite می‌توان این مشکل را به‌گونه‌ای حل نمود تا پیچیدگی نرم‌افزار کاهش یابد. کلید این کار استفاده از یک کلاس مجرد برای عناصر اولیه و عناصر ترکیبی است. به‌عنوان نمونه در مثال ذکر شده، این کلاس می‌تواند Graphic باشد. کلاس Graphic می‌تواند شامل متدهایی نظیر Draw باشد که مخصوص عناصر گرافیکی است. همچنین این کلاس شامل عملیاتی است که در عناصر ترکیبی مشترک است همانند دستیابی به فرزندان عنصر پیچیده. بدین‌ترتیب هر کلاس می‌تواند مجموعه‌ای از عناصر اولیه و پیچیده باشد. شکل ۱۳-۶ نمونه‌ای از ارتباط بین عناصر ساده و ترکیبی را نشان می‌دهد.

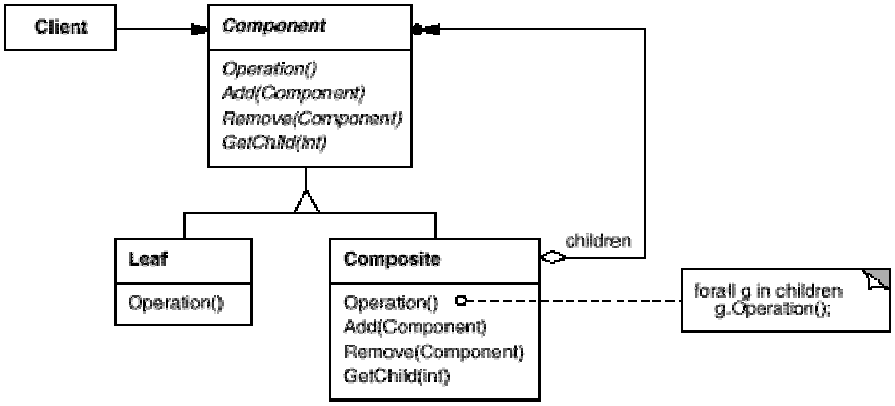


شکل ۱۳-۶- نمونه‌ای از ارتباط بین عناصر با استفاده از الگوی Composite

هر عنصر aPicture می‌تواند یک خط (aLine)، متن (aText)، مستطیل (aRectangle) یا یک شکل ترکیبی باشد.

- **کاربردپذیری**: وقتی از الگوی Composite استفاده نمائید که:
  - اشیا نیاز به ترکیب به‌صورت بازگشتی[1] دارند
  - نیاز به نمایش رابطه Part-Whole دارید
  - تفاوتی بین اشیای ترکیبی و ساده وجود ندارند
  - اشیا در ساختار می‌توانند واحد در نظر گرفته شوند
- **ساختار:** شکل ۱۳-۷ ساختار الگوی Composite را نشان می‌دهد. هر کلاس Component می‌تواند شامل برگ یا یک کلاس Composite دیگر باشد.

[1] Recursive



شکل ۱۳-۷- ساختار الگوی طراحی Composite

- **شرکاء:** برای هر کلاسی که در شکل ۱۳-۷ ارائه شده، تعریف کوتاهی به‌همراه نمونه کلاس (از شکل ۱۳-۶) بیان می‌شود
  - Component (Graphic)
    - واسطی برای اشیاء در ترکیب تعریف می‌کند
    - رفتار پیش‌فرض را برای هر واسط مشترک برای همه کلاس‌ها تاحد ممکن پیاده‌سازی می‌کند
    - واسطی برای دستیابی و مدیریت مولفه‌های فرزند تعریف می‌کند
  - Leaf (Rectangle, Line, Text, etc.)
    - اشیاء برگ را در ترکیب‌شان نشان می‌دهد (برگ، گره‌ای است که فرزند ندارد)
    - رفتار اشیاء اولیه را در ترکیب تعریف می‌کند
  - Composite (Picture)
    - رفتار مولفه‌هایی که دارای فرزند هستند را تعریف می‌کند
    - مولفه‌های فرزند را ذخیره می‌کند
    - عملیات مرتبط با فرزندان را در واسط Component پیاده‌سازی می‌کند
  - Client
    - با استفاده از واسط Composite عملیات خود را بر روی اشیاء انجام می‌دهد

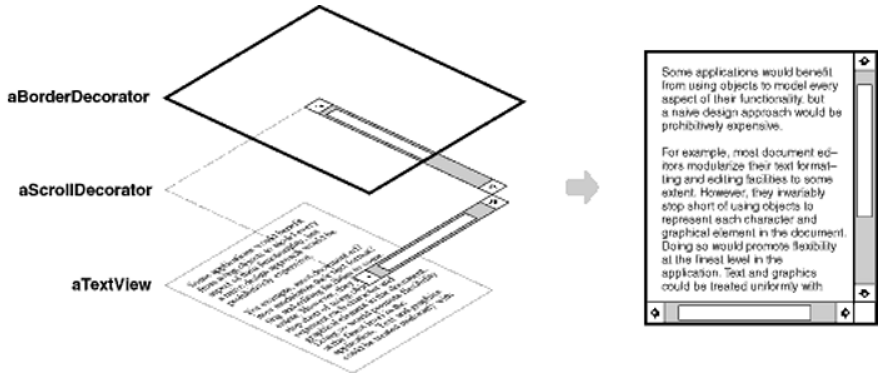
- **همکاران:** سرویس‌گیرندگان با استفاده از واسط کلاس Composite با اشیاء موجود در ساختار ترکیبی ارتباط برقرار می‌کنند. اگر شی مورد نظر برگ باشد، درخواست به صورت مستقیم مدیریت می‌شود و اگر شی مورد نظر ترکیبی باشد، درخواست به مولفه‌های فرزند ارسال می‌شود و منجر به عملیات بیشتری می‌شود.
- **نتایج**: با استفاده از الگوی طراحی Composite این نتایج حاصل می‌شود:
  - همسانی: بدون در نظر گرفتن پیچیدگی همه اشیا یکسانند
  - قابلیت توسعه: مولفه‌های جدید براحتی می‌توانند افزوده شوند
  - سربار: سربار تعداد اشیای جدید را کاهش می‌دهد
- **پیاده‌سازی**
  - روشی برای شناسایی والدین و فرزندان باید استفاده شود
  - برای والدین و فرزندان از یک واسط استفاده می‌شود
  - حذف یک فرزند مسئولیت دارد! (این فرزند می‌تواند فرزندانی داشته باشد و سبب ایجاد اشیای یتیم[1] شود)

۱۳-۶-۴- الگوی طراحی Decorator

- **طبقه‌بندی**: ساختاری
- **منظور**: افزودن وظیفه جدید به یک شی به صورت پویا
- **نام‌های مشابه**: Wrapper
- **انگیزه**: برخی اوقات نیاز است تا مسئولیت‌هایی به اشیاء مستقل و نه به کل کلاس افزوده شود. به‌عنوان مثال در نرم‌افزارهای گرافیکی نیاز است خصوصیاتی شبیه به Scrolling یا Border به برخی از اشیاء اعمال شود. در این حالت، روش متداول استفاده از وراثت است. بدین ترتیب که Border از کلاس دیگری به ارث رسیده و به هر زیرکلاس اعمال می‌شود. این روش نامنطعف است، زیرا انتخاب Border به صورت ایستا و در زمان برنامه‌نویسی صورت می‌پذیرد و قابلیت تغییر ندارد. از طرفی انتخابی برای اینکه Border باشد یا نباشد، وجود ندارد.

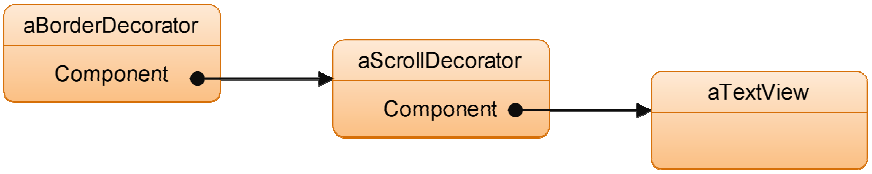
[1] Orphan

روش دیگر قرار دادن مولفه در شی دیگری است که Border را اضافه می‌کند. این روش همان الگوی Decorator است و شی ایجاد شده Decorator نام دارد. Decorator به واسط مولفه اعمال می‌شود و نحوهٔ نمایش آن برای کاربران به‌صورت شفاف[1] است. Decorator درخواست‌ها را به مولفه‌ها ارسال می‌دارد و می‌تواند شامل فعالیت‌های اضافی مانند رسم یک Border باشد. شفافیت Decorator اجازه می‌دهد تا از این الگو به صورت متعدد استفاده شود و وظایف مختلف افزوده شود.



شکل ۱۳-۸- استفاده از الگوی Decorator در ایجاد Border و Scroll Bar

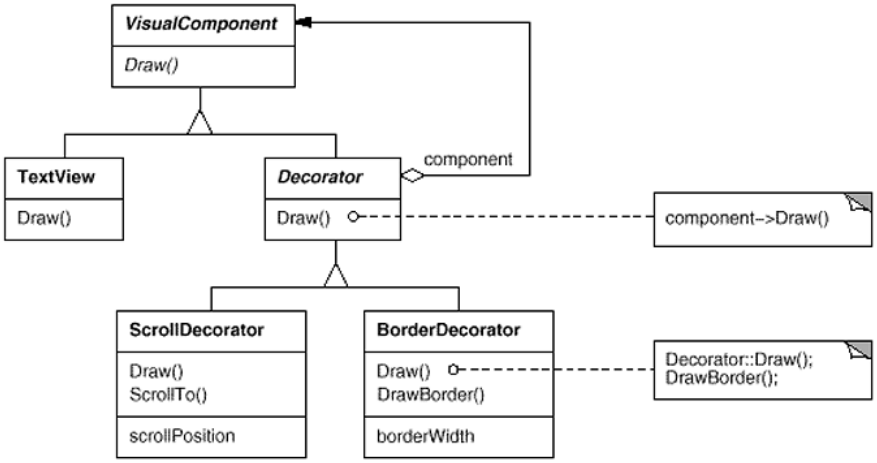
در شکل ۱۳-۸ نمونه‌ای از استفاده از الگوی Decorator را مشاهده می‌کنید. شی TextView متنی را در پنجره نمایش می‌دهد. TextView به‌صورت پیش‌فرض فاقد Scroll Bar است، زیرا همیشه به آن نیاز ندارد و وقتی نیاز باشد تا متن Scroll شود، می‌توان از ScrollDecorator استفاده نمود. برای استفاده از Border نیز به‌همین ترتیب می‌توانیم از BorderDecorator استفاده نمائیم. شکل ۱۳-۹ نحوهٔ ارتباط بین اشیاء را نشان می‌دهد. با این روش aBorderDecorator تمام درخواست‌ها را دریافت نموده و برای aScrollDecorator ارسال می‌دهد و به‌همین ترتیب برای aTextView ارسال می‌شود.



شکل ۱۳-۹- استفاده از الگوی Decorator و ارتباط بین اشیاء

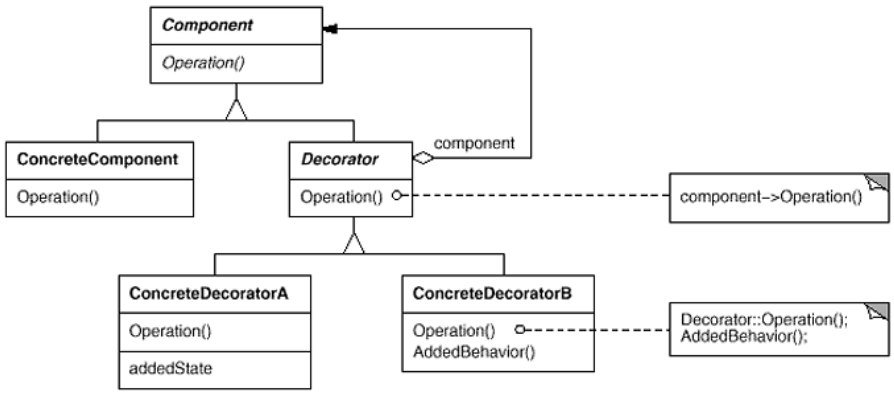
[1] Transparent

شکل ۱۳-۱۰ ارتباط بین اشیاء مختلف مثال نمونه را نشان می‌دهد، کلاس ScrollDecorator و BorderDecorator زیرکلاس‌های Decorator هستند که خود تجریدی برای تمام کلاس‌هایی است که مولفه‌های دیگر را تزئین می‌کنند.



شکل ۱۳-۱۰- ارتباط بین اشیاء مختلف در الگوی Decorator

- **کاربردپذیری**: وقتی از الگوی Decorator استفاده نمائید که:
  - نیاز به افزودن وظیفه جدید به یک شی به صورت پویا، شفاف و بدون تاثیر بر دیگر اشیا باشد
  - نیاز به حذف وظیفه خاصی از یک شی به‌صورت پویا باشد
  - توسعه کلاس به زیرکلاس‌ها غیرممکن باشد
- **ساختار**: شکل ۱۳-۱۱ ساختار الگوی Decorator را نشان می‌دهد.

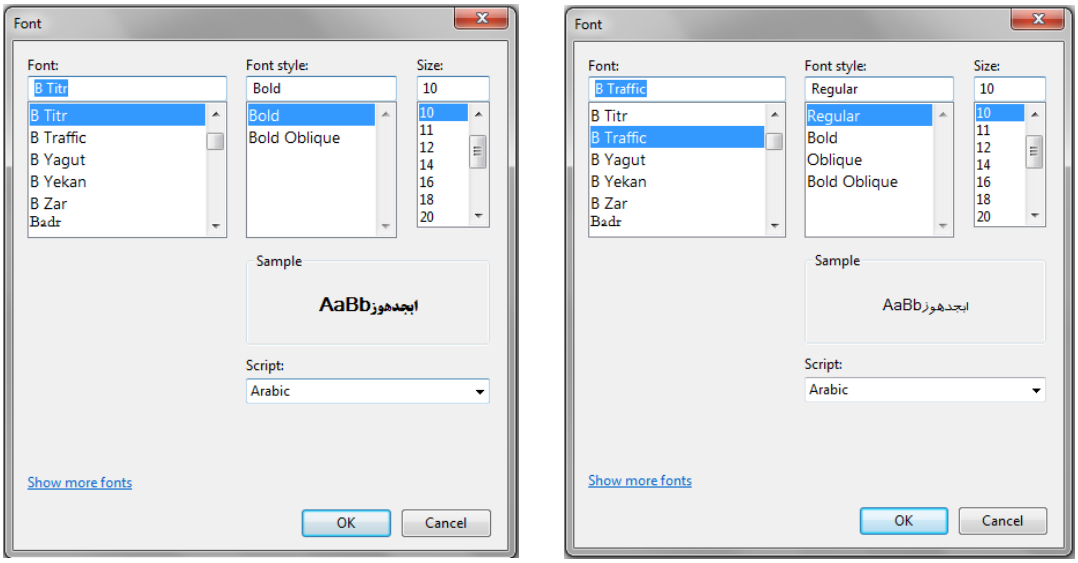


شکل ۱۳-۱۱- ساختار الگوی طراحی Decorator

- **شرکاء:** برای هر کلاسی که در شکل ۱۳-۱۱ارائه شده، تعریف کوتاهی به‌همراه نمونه کلاس (از شکل ۱۳-۱۰) بیان می‌شود
  - Component (VisualComponent)
    - واسطی برای اشیاء تعریف می‌کند که می‌توان مسئولیت‌هایی به‌صورت پویا به آنها اضافه نمود
  - Component (VisualComponent)
    - شیی که مسئولیت‌های اضافه می‌توان به آن ضمیمه شود
  - Decorator
    - ارجاعی به شی Component نگاه می‌دارد و واسطی برای هماهنگی با واسط Component تعریف می‌کند
  - ConcreteDecorator (BorderDecorator, ScrollDecorator)
    - مسئولیت‌ها را به مولفه می‌افزاید
- **همکاران**: Decorator درخواست‌ها را به شی Component ارسال می‌دارد. ممکن است برخی عملیات را قبل از ارسال یا بعد از آن انجام دهد.
- **نتایج**: این الگو دو قابلیت عمده و دو ضعف عمده دارد:
  - وظایف می‌توانند در زمان اجرا افزوده/حذف شوند
  - موجب کاهش رشد ساختار سلسله‌مراتبی می‌شود
  - سبب انسداد واسط می‌شود، چرا که Decorator و مولفه‌هایش یکسان نیستند
  - ترکیب decorator ها بسیار دشوار است
- **پیاده‌سازی**
  - مطابقت واسط decorator ها باید در نظر گرفته شود
  - برای decorator از کلاسی سبک و مجرد استفاده نمائید
  - در صورتی‌که نیاز به افزودن یک وظیفه دارید، نیاز به تعریف کلاس برای decorator ندارید

۱۳-۶-۵- الگوی طراحی Modiator

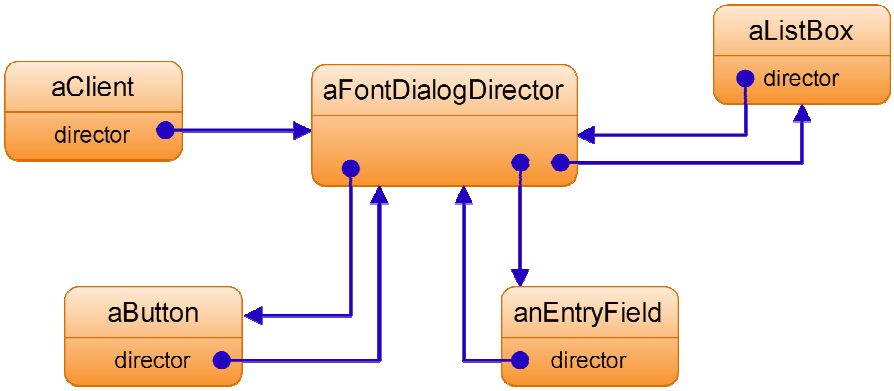
- **طبقه‌بندی**: رفتاری
- **منظور**: تعریف شیی که چگونگی تعامل بین اشیاء دیگر را پنهان سازد. این الگو اتصال سست[1] بین عناصر را با استفاده از آدرس‌دهی پویا بین آنها ارتقاء می‌دهد.
- **نام‌های مشابه**: ندارد
- **انگیزه**: در نرم‌افزارهای شی گرا که ارتباط بین اشیاء با توجه به تقسیم شدن مسئولیت‌ها برخی اوقات بسیار پیچیده می‌شود، اغلب نیاز است تا اشیاء از وجود دیگری مطلع باشند و به‌نوعی به‌هم وابسته می‌شوند. به‌عنوان مثال شکل ۱۳-۱۲ دو پنجره انتخاب فونت نمایش داده شده است که در آن با انتخاب فونت و خصوصیات آن، پیش نمایش آن در نمایش داده می‌شود. به‌ازای هر فونت برخی خصوصیات قابل اجرا هستند به‌عنوان مثال Font Style دو فونت مختلف می‌تواند متفاوت باشد. بنابراین بین شی Font و Font Style ارتباط تنگاتنگی وجود دارد و همین ارتباط بین Font و Size و نیز Font و Script وجود دارد. این ارتباط سبب می‌شود تا عناصر روی Form یعنی ListBox مربوط به فونت، ListBox مربوط به Font Style، ListBox مربوط به Size و Combo مربوط به Script به‌یکدیگر وابسته باشند.



شکل ۱۳-۱۲- پنجره انتخاب فونت در نرم‌افزار Notepad

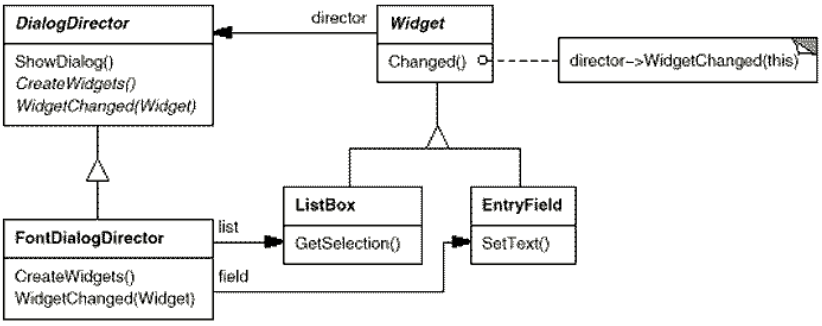
[1] Loose Coupling

در این مثال ساده وابستگی بین عناصر سبب می‌شود تا برنامه‌نویس در کد برنامه نسبت به رفع این مشکل اقدام کند که این امر منجر به کاهش انعطاف‌پذیری خواهد شد. برای پرهیز از مشکل می‌توان از الگوی Mediator استفاده نمود. شی Mediator مسئول کنترل و هماهنگ‌سازی تعامل بین گروهی از اشیاء است. این شی، به‌عنوان یک واسطه بین اشیاء عمل می‌کند و اجازه نمی‌دهد اشیاء به‌صورت صریح با یکدیگر تعامل داشته باشند. اشیاء تنها به واسطه متصل می‌شوند و همین باعث می‌شود که ارتباطات بین اشیاء کاهش یابد. شکل ۱۳-۱۳ استفاده از الگوی Mediator را برای مثال پنجره فونت نشان می‌دهد.



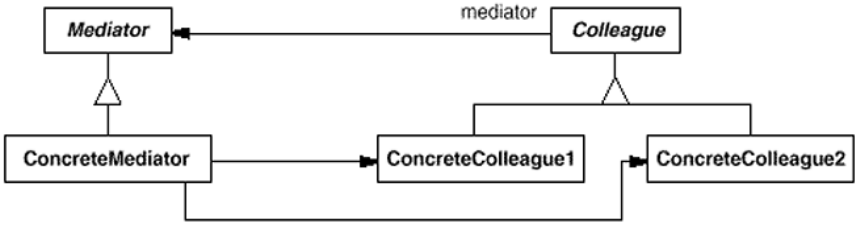
شکل ۱۳-۱۳- استفاده از الگوی Mediator در مثال پنجره انتخاب فونت

شکل ۱۳-۱۴ ارتباط بین اشیاء مختلف با استفاده از الگوی طراحی Mediator در مثال پنجره فونت را نشان می‌دهد. هر شی در پنجره به‌عنوان یک Widget محسوب می‌شود. DialogDirector کلاس مجردی است که رفتار کلی پنجره را تعریف می‌کند. سرویس گیرندگان متد ShowDialog را فراخوانی می‌کنند تا پنجره بر روی صفحه نمایش داده شود. متد CreateWidget متد مجردی است که برای ایجاد Widget (هر چیزی درون پنجره) مورد استفاده قرار می‌گیرد. متد WidgetChanged متد مجرد دیگری است که Widget ها آن را فراخوانی می‌کنند تا به سرپرست‌شان (Director) اطلاع دهند که تغییر نموده‌اند. بدین ترتیب DialogDirector همانند یک Mediator بین Widget ها و FontDialogDirector عمل می‌کند و تغییرات آنها را به اطلاع یکدیگر می‌رساند. همچنین ارتباط مستقیم بین Widget های روی پنجره (همانند ListBox, Button, Combo) با یکدیگر برداشته شده است.

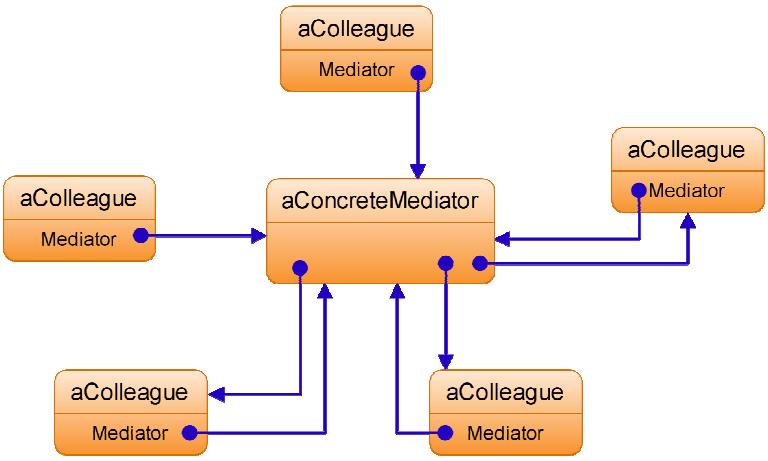


شکل ۱۳-۱۴- استفاده از الگوی طراحی Mediator

- **کاربردپذیری**: وقتی از الگوی Mediator استفاده نمائید که:
  - تعدادی شی با یکدیگر به‌صورت خوش‌تعریف تعامل دارند اما تعامل آنها پیچیده است
  - استفاده مجدد یک شی به‌خاطر تعامل زیاد آن با دیگر اشیا دشوار باشد
  - رفتاری بین چند کلاس توزیع شده و باید بدون استفاده از زیرکلاس‌ها سفارشی شود
  - توسعه کلاس به زیرکلاس‌ها غیرممکن باشد
- **ساختار**: شکل ۱۳-۱۵ساختار و شکل ۱۳-۱۶ ارتباط اشیاء در این الگو را نشان می‌دهد.



شکل ۱۳-۱۵- ساختار الگوی Mediator



شکل ۱۳-۱۶- ساختار نمونه اشیاء با استفاده از الگوی Mediator

- **شرکاء:** برای هر کلاسی که در شکل ۱۳-۱۵ ارائه شده، تعریف کوتاهی به‌همراه نمونه کلاس (از شکل ۱۳-۱۴) بیان می‌شود
  - Mediator (DialogDirector)
    - واسطی برای ارتباط با اشیاء Colleague تعریف می‌کند
  - ConcreteMediator (FontDialogDirector)
    - رفتارهای مشترک را برای ارتباط اشیاء Colleague پیاده‌سازی می‌کند
    - اشیاء Colleague خود را می‌شناسد و نگهداری می‌کند
  - Colleague classes (ListBox, EntryField)
    - هر کلاس Colleague شی Mediator خود را می‌شناسد
    - هر کلاس Colleague با Mediator خود ارتباط دارد
- **همکاران**: Colleague درخواست‌ها را به شی Mediator ارسال و دریافت می‌دارند
- **نتایج**: این الگو دارای نقاط قوت و ضعف زیر است:
  - تعداد زیرکلاس‌ها را کاهش می‌دهد
  - اتصال سست بین کلاس‌های مجاور را افزایش می‌دهد
  - ارتباط چند به چند را به ارتباط یک به یک تبدیل می‌کند
  - یک کنترل‌کننده مرکزی ایجاد می‌کند
- **پیاده‌سازی**
  - نیازی به تعریف کلاس مجرد Mediator نیست زیرا کلاس‌های همکار می‌توانند با یک Mediator کار کنند

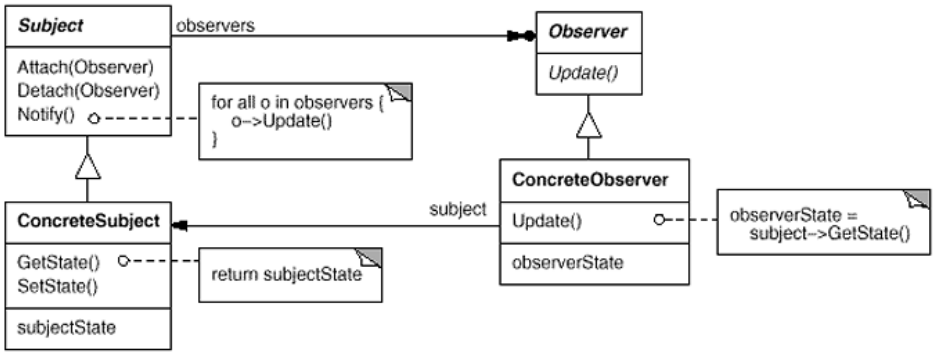
۱۳-۶-۶- الگوی طراحی Observer

- **طبقه‌بندی**: رفتاری
- **منظور**: تعریف یک رابطه یک به چند بین اشیاء بصورتی‌که وقتی یک شی حالتش را تغییر می‌دهد، بقیه اشیا مطلع شده و بروز شوند
- **نام‌های مشابه**: Dependents, Publish-Subscribe

- **انگیزه**: در نرم‌افزارهای شی‌گرا که ارتباط بین اشیاء با توجه به تقسیم شدن مسئولیت‌ها برخی اوقات بسیار پیچیده می‌شود، نیاز است تا سازگاری بین اشیاء مختلف حفظ شود. به‌عنوان مثال در یک نرم‌افزار صفحه گسترده که اطلاعات در قالب داده‌های خام وارد شده و سپس با استفاده از نمودارهای مختلف نمایش داده می‌شود، می‌بایست بین اطلاعات خام و نمودارها ارتباط برقرار شود تا در صورت تغییر اطلاعات، نمودارها نیز تغییر یابند.

الگوی Observer می‌تواند ارتباط بین اشیاء مختلف را به‌صورت مناسب برقرار نماید تا اشیاء از تغییرات یکدیگر مطلع شوند. اشیاء کلیدی این الگو Subject و Observer هستند. هر Subject می‌تواند به تعداد نامحدودی Observer متصل باشد. تمام Obserever‌ها از تغییر Subject مطلع می‌شوند.

- **کاربردپذیری**: وقتی از الگوی Observer استفاده نمائید که:
  - تجریدی دارای دو جنبه بوده و یکی به دیگری وابسته باشد
  - تغییر در یک شی سبب تغییر دیگر اشیا می‌شود و نمی‌دانید چه تعداد شی دیگر نیاز به تغییر دارند
  - یک شی نیاز به مطلع ساختن شی دیگر دارد و در مورد چگونگی آن شی اطلاعی ندارد
- **ساختار**: شکل ۱۳-۱۷ ساختار الگوی Observer را نشان می‌دهد.



شکل ۱۳-۱۷- ساختار الگوی Observer

- **شرکاء:** برای هر کلاسی که در شکل ۱۳-۱۷ ارائه شده، تعریف کوتاهی بیان می‌شود
  - Subject
    - واسطی برای افزودن و حذف اشیاء Observer فراهم می‌کند

- اشیاء Observer خود را می‌شناسد و هر تعداد Observer می‌تواند به یک Subject متصل شود
  - Observer
    - واسطی بروز شونده برای اشیاء تعریف می‌کند که با تغییر Subject حالتش تغییر می‌کند
  - ConcreteSubject
    - حالت مورد نظر اشیاء ConcreteObserver را ذخیره می‌کند
    - پس از تغییر حالت، اعلامی به اشیاء Observer ارسال می‌دارد
  - ConcreteObserver
    - ارجاعی به شی ConcreteSubject نگاه می‌دارد
    - حالتش را ذخیره می‌کند که باید با شی مورد اشاره‌اش سازگار باشد
    - واسط بروزشونده‌ای برای Observer پیاده‌سازی می‌کند تا حالتش را با شی مورد اشاره‌اش سازگار نگاه دارد
- **همکاران**: ConcreteSubject در صورت بروز تغییر Observerهای خود را مطلع می‌سازد تا حالت خود را سازگار نگاه دارند
- **نتایج**: این الگو دارای نقاط قوت و ضعف زیر است:
  - واحدبندی: Subject و Observerها می‌توانند مستقل باشند
  - قابلیت توسعه: می‌توان هر تعداد Observer تعریف نمود
  - قابلیت سفارشی شدن: Observerهای متفاوت، دیدهای مختلف را نمایش می‌دهند
  - بروزرسانی غیرمنتظره: Observerها اطلاعی از یکدیگر ندارند
  - سربار بروزرسانی: بروزرسانی سربار زیادی ایجاد می‌نماید
- **پیاده‌سازی**
  - نگاشت Subjectها به Observerهای متناظرشان راهی ساده برای در جریان بودن Subject از بروزرسانی Observer است
  - ارجاع معلق به Subjectهای حذف شده باید مورد توجه قرار گیرند
  - از پروتکل‌های بروزرسانی ویژه Observer (مدل‌های push and pull) اجتناب نمائید

۱۳-۷-ضدالگوها

ایده ضدالگوها از آنجا ناشی شده است که اغلب کارهای منتشر در مهندسی نرم‌افزار به راه‌حل‌های سازنده و موثر تمرکز دارند. ضدالگوها بر راه‌حل‌های منفی و ناموفق تمرکز دارند و مجموعه‌ای از راه‌حل‌های متداول که برای حل یک مسئله اتفاق می‌افتند و بطورقطعی نتایج منفی یا ناموفق تولید نموده‌اند را توصیف می‌نماید. همانطور که نیاز به دانستن الگوها در زمینه خاص وجود دارد، دانستن وجود یا عدم وجود ضدالگو می‌تواند به توسعه بهتر نرم‌افزار کمک کند. در واقع، بسیاری از مهندسان نرم‌افزار می‌خواهند بدانند «آیا راهی که می‌روند به شکست منجر می‌شود». دانستن جواب این سوال می‌تواند در اجرای پروژه نقش موثری ایفا نماید.

به‌عنوان مثالی ساده از ضدالگوها می‌توان به موارد زیر اشاره نمود:

- پنج پروژه از شش پروژه به شکست منجر شده‌اند!
- یک سوم پروژه‌های نرم‌افزاری رها شده‌اند!
- بودجه و زمان واقعی انجام پروژه بیش از دو برابر از آنچه تخمین زده می‌شود، است!!!

این موارد برای مدیر پروژه می‌توانند هشداردهنده باشند و نشان‌دهندهٔ مسائلی هستند که مدیر پروژه باید به آن فکر کند. از طرفی الگوها نیز می‌توانند به ضدالگو تبدیل شوند. این امر بدین دلیل است که اهداف و مسائلی که در گذشته مدنظر بوده‌اند، در حال حاضر کمتر مورد توجه قرار می‌گیرند.

به‌عنوان نمونه در جدول ۱۳-۲ نمونه‌ای از الگوهایی که قبلاً رایج بوده‌اند و در حال حاضر به ضدالگو تبدیل شده‌اند بهمراه الگوهایی که امروز متداول هستند، ارائه شده است. در دهه قبل مسائلی از قبیل برنامه‌نویسی ساخت‌یافته و معماری Client/Server به‌عنوان یک الگو مطرح بودند و سیستم‌های بسیاری با این روش‌ها توسعه پیدا نمودند، اما در حال حاضر استفاده از CBD و معماری سرویس‌گرا به‌عنوان یک الگو مطرح هستند و استفاده از برنامه‌نویسی ساخت‌یافته و معماری Client/Server جایی در سیستم‌های امروز ندارند و به عنوان یک ضدالگو مطرح شده‌اند.

با پیشرفت مفهوم ضدالگو در مهندسی نرم‌افزار می‌توان ارتباط مستحکم‌تری بین الگوها و ضدالگو یافت نمود. در واقع، الگو زمانی به ضدالگو تبدیل می‌شود که زمینه مسئله تغییر می‌کند و سبب می‌شود تا راه‌حل‌های پیشنهادی الگو جواب‌گو نباشد. در این حالت ضدالگو مطرح می‌شود که نشان می‌دهد که در صورت استفاده از الگوی قبلی دچار شکست خواهیم شد.

الگوهای دیروز (ضدالگوهای امروز)	الگوهای امروز
❖ برنامه‌نویسی ساختاریافته ❖ طراحی بالا به پائین ❖ معماری Client/Server ❖ تولید کد از مدل	❖ فناوری مولفه‌ها ❖ اشیا توزیع‌شده ❖ معماری سرویس‌گرا ❖ استفاده مجدد از نرم‌افزار ❖ عامل‌های نرم‌افزار[1] ❖ رابط Web

جدول ۱۳-۲- مقایسه الگوهای قدیمی و جدید

تابحال مجموعه‌ای از ضدالگوها توسط نویسندگان مختلف ارائه شده است که این فهرست همواره در حال بروز شدن است. فهرستی از ضدالگوهای مطرح شده عبارتند از:

- Ambiguous viewpoint
- Boat Anchor
- Continuous obsolescence
- Cut-and-Paste Programming
- Dead End
- Functional decomposition
- Golden Hammer
- Input Kludge
- Lava Flow
- Mushroom Management
- Poltergeists
- Spaghetti Code
- The Blob
- Walking through a Minefield

همانند الگوها برای ضدالگوها طبقه‌بندی‌های مختلفی ارائه شده است که از جمله مطرح‌ترین آنها به طبقه‌بندی ارائه شده توسط Brown است که ضدالگوها را به سه دسته مدیریتی، معماری و توسعه تقسیم نموده است. ضدالگوهای مدیریتی تمرکز بر مفاهیم مدیریت پروژه نرم‌افزاری دارند. ضدالگوهای توسعه در توسعه نرم‌افزار مورد استفاده قرار می‌گیرند و ضدالگوهای معماری در حوزه معماری نرم‌افزار کاربرد دارند. در ادامه به بررسی دو ضدالگوی مطرح توسعه خواهیم پرداخت.

[1] Agents

۱۳-۷-۱- ضدالگوی توسعه Lava Flow

- **مشکل**: استفاده از کد قدیمی و اطلاعات طراحی فراموش‌شده در طراحی دایم‌التغییر
- **نمونه**: به‌عنوان نمونه‌ای از این ضدالگو می‌توان به موارد زیر اشاره نمود:
  - طراح ارشد استعفا نموده است
  - طراح جدید روش بهتری ارائه نموده است، اما بدلیل عدم آشنایی با کد نمی‌تواند آنها را حذف نماید
- **علل**: از جمله عواملی که سبب می‌شود ضدالگوی Lava Flow روی دهد، عبارتند از:
  - توزیع کنترل‌نشده کدهای اتمام نشده یا اصلاح نشده
  - بخش‌ها و مسیرهایی که برای تست در کد قرار داده شده بودند، حذف نشدند
  - شکاف در معماری بخاطر استفاده از فناوری قدیمی. شکاف در معماری وقتی بوجود می‌آید که معماری اولیه پس از شروع توسعه دچار تغییرات شگرف شود
  - فقدان معماری یا معماری بسیار ضعیف که کنار گذاشته شود
  - فقدان مدیریت پیکربندی در توسعه نرم‌افزار
- **راه‌حل**: برای حل مشکل این ضدالگو می‌توان از یک یا بیشتر راه‌حل زیر استفاده نمود:
  - استفاده از سیستم مدیریت پیکربندی سبب می‌شود که کدهای کهنه شناسایی و حذف شوند
  - تکمیل‌نمودن طراحی قدیمی یا ایجاد یک طراحی جدید
  - استفاده از یک معماری سالم برای پیش‌بردن مراحل توسعه
  - استفاده از رابط در سطح سیستم که خوش-تعریف، پایا و بدرستی مستند شده باشد

۱۳-۷-۲- ضدالگوی توسعه Blob

- **مشکل**: استفاده از سبک طراحی رویه‌ای منجر به ایجاد کلاسی با مسئولیت بسیار زیاد می‌شود در حالیکه بقیه کلاس‌ها تنها داده نگهداری می‌نمایند. در واقع، این کلاس همان کلاسی است که قلب معماری خواهد بود و پردازش و دیگر داده‌ها را انحصاری می‌نماید. حجم بسیار این کلاس و سنگینی بیش از حد سبب عدم انعطاف نرم‌افزار خواهد شد.

- **علل**: از جمله عواملی که ایجاد این ضدالگو می‌شوند، عبارتند از:
  - فقدان دیدگاه معماری شی‌گرایی
  - عدم وجود التزام به معماری
  - طراح رویه‌ای معمار ارشد است
  - عدم تمایل توسعه‌دهندگان در طراحی کلاس‌های جدید و کارآمد
- **راه‌حل**: مجموعه راه‌حل‌هایی که می‌توان برای حل این مشکل استفاده نمود، عبارتند از:
  - توزیع مجدد وظیفه‌مندی
  - جداسازی اثر تغییرات
  - تعیین یا طبقه‌بندی خصوصیات و عملیات
  - حذف «ارتباطات بسیار دور»، زائد یا غیرمستقیم
  - حذف تمام ارتباطات موقت

۱۴- شبکه‌های پتری

نمایش رفتار سیستم‌های نرم‌افزاری همواره یکی از مهمترین مسائل حوزه مهندسی نرم‌افزار و به‌خصوص مدلسازی نرم‌افزار بوده است. در فصل ۱۰ به بررسی نمایش تعامل و رفتار سیستم پرداخته شد و نمودار ترتیبی، فعالیت و حالت به‌عنوان روش‌هایی برای مدلسازی رفتار معرفی شدند. این روش‌ها که جزئی از UML هستند، توانایی‌های کافی برای نمایش رفتار سیستم را ارائه می‌دهند. اما مسئله‌ای که مدلسازی و به‌خصوص مدلسازی رفتار را به یک مسئله مهم در مهندسی نرم‌افزار تبدیل می‌کند، اثر مستقیم روش مدلسازی بر ارزیابی سیستم است. هر چه مدل دارای عناصر بیشتری باشد، ارزیابی دشوارتر می‌شود و در صورتی‌که مدل از پشتوانه ریاضی برخوردار باشد، ارزیابی ساده‌تر خواهد شد. در میان روش‌های مدلسازی، شبکه‌های پتری به‌عنوانی یکی از روش‌های با قابلیت ارزیابی بالا مطرح هستند که در ادامه این فصل به بررسی این شبکه‌ها خواهیم پرداخت.

۱۴-۱-روش‌های مدلسازی رفتار سیستم

به‌طور کلی سه روش برای مدلسازی رفتار وجود دارد:

- نمودار حالت[1] و نمودارهای UML
- نمودار انتقال حالت[2]
- شبکه‌های پتری[3]

نمودار حالت و بقیه نمودارهای UML برای نمایش رفتار در فصل ۱۰ مورد بررسی قرار گرفتند. در نمودار انتقال حالت، سیستم توسط مجموعه‌ای از حالات نشان داده می‌شود که رویدادهای خارجی سبب حرکت روی حالات می‌شوند. با توجه به اینکه اغلب سیستم‌ها دارای حالات مختلف هستند، نمودار انتقال حالت بزرگ و پیچیده خواهد شد. این نمودار دارای دو عنصر حالت و انتقال است. عمده‌ترین مشکلات نمودار انتقال حالت را می‌توان به‌صورت زیر عنوان نمود:

- وقتی تعداد حالات و رویدادها افزایش یابد، پیچیدگی به‌صورت تصاعدی افزایش می‌یابد

[1] Statechart
[2] State Transition Diagram
[3] Petri Net

- ابهام در نمودار سبب افزایش پیچیدگی و تضعیف ارزیابی می‌شود
- با توجه به عدم وجود پشتوانه ریاضی توانایی ارزیابی کامل نمودار وجود ندارد

با وجود نقاط ضعف و مشکلات روش‌های مدلسازی رفتار در ارزیابی، استفاده از شبکه‌های پتری بسیار مورد توجه قرار گرفته است. شبکه پتری، مبتنی بر نظریه گراف بوده و با استفاده از قواعدی منطقی جریان فعالیت‌ها در سیستم را نمایش می‌دهد. چارچوب ریاضی شبکه پتری سبب می‌شود تا توانایی تحلیل، تایید صحت و ارزیابی مدل‌ها را داشته باشد. شبکه پتری قادر به توصیف سیستم‌هایی است که شامل مجموعه‌ای از رخدادهای گسسته و پراکنده هستند، از جمله این رخدادها می‌توان به موارد زیر اشاره نمود:

- همزمانی و تعارض
- ترتیب‌ها، شاخه‌های شرطی و چرخه‌ها
- همگام‌سازی[1]
- اشتراک منابع محدود و انحصار متقابل
- الگوهای مخابراتی، کنترلی و جریان‌های اطلاعاتی

۱۴-۲-تاریخچه شبکه‌های پتری

نظریه شبکه‌های پتری توسط Carl Adam Petri در سال ۱۹۶۲ در رساله دکتری ایشان ارائه شد. او از شبکه‌های پتری برای نمایش ارتباطات علت و معلول استفاده نمود. پس از آن تا سال ۱۹۷۰ این نظریه در دانشگاه MIT بسط داده شد و در مقالات و کنفرانس‌های متعددی ارائه شد. با حرکت محققان به استفاده از این روش برای ارزیابی، شناختی خوبی از شبکه پتری ایجاد گردید تا اینکه در سال ۱۹۷۵ اولین کنفرانس شبکه پتری و روش‌های مرتبط برگزار گردید.

شبکه‌های پتری در طول زمان کامل‌تر شدند و مفاهیم کاربردی‌تر و جدیدی‌تری به مفاهیم مورد استفاده آن افزوده شد. از جمله این مفاهیم می‌توان به افزوده شده زمان قطعی، زمان تصادفی و رنگ اشاره نمود. برخی از این مفاهیم منجر به تولید شبکه‌های خاصی شدند که از آن جمله می‌توان به شبکه‌های پتری رنگی اشاره نمود.

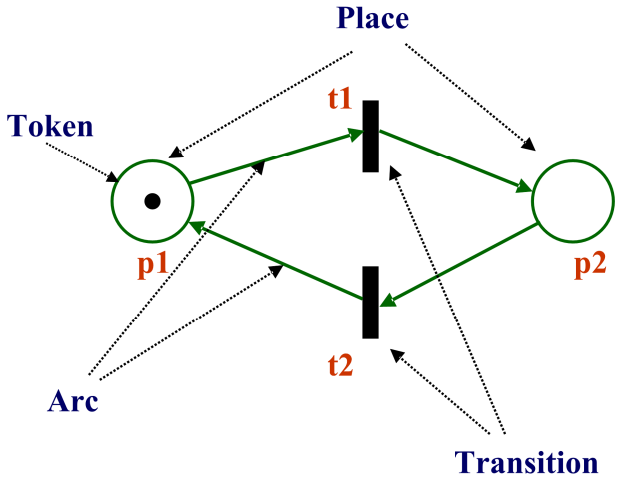
[1] Synchronization

۱۴-۳-عناصر شبکه پتری

هر مدل شبکه پتری با استفاده از سه عنصر مدل می‌شود

- **مکان**[1]: حالت سیستم را نشان می‌دهند
- **انتقال**[2]: رویدادهایی را که سبب تغییر حالت سیستم می‌شوند را نشان می‌دهند
- **کمان**[3]: ارتباط بین حالات را نشان می‌دهند

همچنین در هنگام نمایش اجرای شبکه پتری از نشانه[4] برای بیان وضعیت فعلی شبکه پتری استفاده می‌شود. در واقع، نشانه‌ها در مکان‌ها قرار می‌گیرند. شکل ۱۴-۱ عناصر شبکه پتری و ارتباطات بین آنها را نمایش می‌دهد.



شکل ۱۴-۱- عناصر شبکه پتری

۱۴-۴-تعریف شبکه پتری

اساس شبکه‌های پتری بر پایه گراف، بنا نهاده شده است. اگر بخواهیم یک توصیف غیر رسمی از آن داشته باشیم می‌توان گفت که یک گراف جهتدار دو قسمتی است که از دو عنصر مکان و انتقال تشکیل شده است. یک توصیف رسمی از شبکه‌های پتری به شکل زیر می‌تواند باشد.

شبکه پتری، یک پنج تایی مرتب به شکل (P,T,I,O,M_0) است که:

[1] Place
[2] Transition
[3] Arc
[4] Token

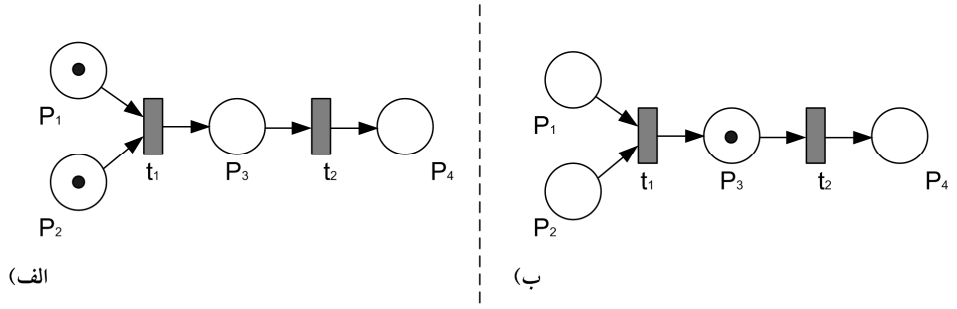
- P یک مجموعه متناهی از مکان‌ها است
- T یک مجموعه متناهی از انتقال‌ها است
- مجموعه ورودی‌ها ($P \cap T = \varnothing$)
- مجموعه خروجی‌ها
- مجموعه حرکات: $M_0 : P \Rightarrow N_0$ تابع علامتگذاری اولیه[1] نامیده می‌شود

توابع I و O نمایش‌دهنده ارتباط میان مکان‌ها و انتقال‌ها هستند. اگر رابطه $I\text{-}(P,T) > 0$ برقرار باشد در اینصورت کمانی از مکان P به انتقال T وجود دارد. در این حالت به مکان P، یک مکان ورودی گفته می‌شود. اگر رابطه $I+(P,T) > 0$ برقرار باشد در اینصورت کمانی از انتقال T به مکان P وجود دارد. در این حالت به مکان P، یک مکان خروجی گفته می‌شود. توابع تلاقی، اعداد طبیعی را به کمان‌ها منتسب می‌کنند که به این اعداد وزن کمان‌ها گفته می‌شود. زمانیکه هر مکان ورودی از انتقال T حداقل به تعداد وزن کمانی که آن را به انتقال T وصل می‌کند حاوی مهره باشد، در اینصورت گفته می‌شود که انتقال T فعال[2] شده است. یک انتقال فعال می‌تواند آتش شود. در چنین حالتی انتقال مورد نظر به مقدار وزن هر یک از کمان‌های ورودی از هر یک از مکان‌های ورودی مهره برداشته و به مقدار وزن هر یک از کمان‌های خروجی در هر یک از مکان‌های خروجی مهره ایجاد می‌کند.

تعداد اولیه مهره‌ها در شبکه با استفاده از تابع M_0 که نشان‌دهنده تعداد مهره‌ها در هر مکان از شبکه است مشخص می‌شود. به تعداد مهره‌ها در شبکه، علامتگذاری شبکه می‌گویند. به M_0، علامتگذاری اولیه می‌گویند. زمانی که یک انتقال آتش می‌شود ممکن است علامتگذاری شبکه تغییر کند. شکل ۲-۱٤ یک شبکه پتری را نشان می‌دهد که از ٤ مکان و ۲ انتقال تشکیل شده است. شکل مذکور شبکه پتری را قبل و بعد از آتش شدن انتقال t_1 نشان می‌دهد. بعد از آتش شدن انتقال t_1، یک مهره از مکان P_1 و P_2 برداشته می‌شود و یک مهره در مکان P_3 ایجاد می‌شود.

کمان ورودی به کمانی گفته می‌شود که از مکان به یک انتقال وارد می‌شود و نشان‌دهندهٔ شرطی است که باید برآورده شود تا رویداد اتفاق بیفتد. کمان خروجی به کمانی گفته می‌شود که از انتقال به مکان وارد می‌شود و نشان‌دهندهٔ شرایط حاصل از وقوع رویداد است.

[1] Initial marking
[2] Fire



شکل ١٤-٢- یک شبکه پتری قبل و بعد از آتش شدن یک انتق

در شبکه‌های پتری، شلیک شدن به‌صورت اتوماتیک و پس از فعال شدن انتقال انجام می‌شود. ممکن است که چند انتقال فعال شوند، اما در هر زمان تنها یک شلیک انجام می‌شود. حالت شبکه با توزیع نشانه‌ها در مکان‌ها نمایش داده می‌شود و به همین دلیل، حرکت نشانه‌ها و شلیک شدن سبب ایجاد حالت‌های مختلف می‌شوند.

انواع حالاتی که در شبکه‌های پتری وجود دارند، عبارتند از:

- حالت اولیه[1]: توزیع اولیه نشانه‌ها
- حالت قابل دستیابی[2]: حالت قابل دستیابی از حالت اولیه
- حالت نهایی[3]: حالتی که هیچ انتقالی فعال نباشد (حالت مرده)
- حالت خانه[4]: حالتی که همواره امکان بازگشت به آن وجود داشته باشد. این حالت از هر حالتی قابل دستیابی است

١٤-٥-خصوصیات رفتاری شبکه‌های پتری

شبکه‌های پتری دارای مجموعه‌ای از خصوصیات رفتاری هستند که نشان‌دهندهٔ وضعیت آنهاست. این خصوصیات می‌توانند به حالت اولیه بستگی داشته باشند یا اینکه مستقل از حالت اولیه شبکه پتری باشند. در ادامه برخی از این خصوصیات رفتاری شبکه‌های پتری مورد بررسی قرار خواهد گرفت.

- **قابلیت دستیابی**[5]: این خصوصیت نشان می‌دهد که آیا همه حالت‌های موجود در شبکه پتری قابل دستیابی بوده و می‌توانند اجرا شوند. حالت M_n قابل‌دستیابی از حالت M_0 است، اگر ترتیبی

[1] Initial State
[2] Reachable State
[3] Final State
[4] Home State
[5] Reachability

از شلیک‌هایی وجود داشته باشد که از M_0 شروع شده و به M_n برسد. وجود حالت‌های غیرقابل دستیابی نشان می‌دهد که قسمت‌هایی از مسئله به‌درستی درک و مدلسازی نشده است.

- **کران‌دار بودن و مطمئن بودن**[1]: در صورتیکه تعداد نشانه‌ها در هر مکان قابل دستیابی از حالت اولیه، از تعداد خاصی کمتر باشد به آن شبکه کران‌دار گویند. در صورتیکه تعداد نشانه‌ها K باشد به آن شبکه k-bounded گویند. شبکه ۱-bounded را شبکه مطمئن گویند. در صورتیکه شبکه مطمئن می‌باشد می‌توان اطمینان حاصل کرد که با هر ترتیبی از شلیک‌ها، سرریز در ثبّات‌ها و بافرها به‌وجود نمی‌آید.
- **زنده بودن**[2]: در صورتی شبکه پتری زنده است که بتوان هر حالت را با ترتیبی از شلیک‌های مناسب فعال نمود. زنده‌بودن شبکه پتری معادل بدون بن‌بست بودن است. شبکه‌های پتری دارای سطوح مختلف زنده‌بودن هستند (L0 = مرده، L1، L2، L3، L4 = زنده)
- **معکوس‌پذیری**[3]: یک شبکه پتری معکوس‌پذیر است اگر برای هر حالت M_n که از M_0 قابل دستیابی است، M_0 نیز از M_n قابل دسترسی باشد. در صورتی $\acute{M}$ حالت خانه است اگر برای هر حالت M قابل دستیابی از M_0، $\acute{M}$ از M قابل دستیابی باشد.
- **ماندگار**[4]: یک شبکه پتری ماندگار است اگر در هر انتقال دوتایی، شلیک شدن یکی سبب غیرفعال شدن دیگری نشود.
- **منصف بودن**[5]: در صورتیکه در شبکه هر انتقال سرانجام شلیک شود و انتقالی وجود نداشته باشد که شلیک نشود. در واقع، اگر در شبکه چرخه بی‌نهایت وجود داشته باشد و برخی از انتقالات شلیک نشوند، شبکه از حالت منصف بودن خارج می‌شود اما در حالتیکه شبکه بی‌نهایت باشد و همه انتقالات سرانجام شلیک می‌شوند، شبکه دارای انصاف است. دو نوع شبکه دارای انصاف وجود دارد
  - Bounded-Fairness: اگر تعداد دفعاتی که یک انتقال می‌تواند شلیک شود در حالیکه انتقال دیگر شلیک نشده باشد، محدود باشد

[1] Boundedness and Safeness
[2] Liveness
[3] Reversibility
[4] Persistence
[5] Fairness

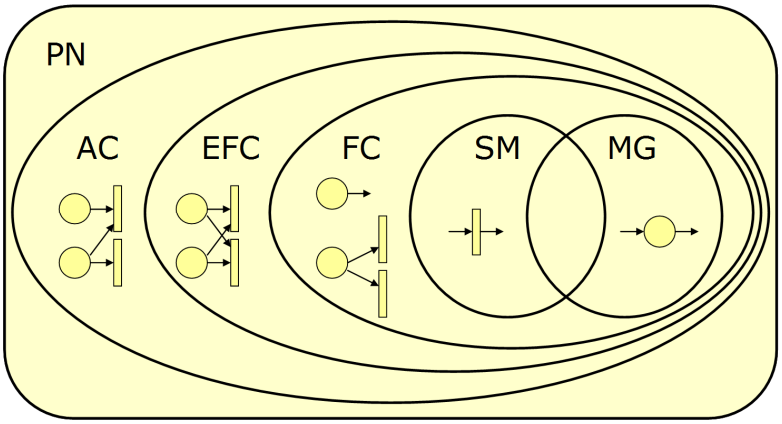
- Unbounded-Fairness: اگر تعداد دفعاتی که یک انتقال می‌تواند شلیک شود در حالیکه انتقال دیگر شلیک نشده باشد، نامحدود باشد

۱۴-۶-زیرنوع‌های شبکه پتری

انواع مختلفی از شبکه‌های پتری از ابتدای معرفی این تئوری، معرفی و مورد استفاده قرار گرفته‌اند که هر یک خصوصیات و ویژگی‌های خاص خود را دارند، از آن جمله می‌تواند به انواع زیر اشاره نمود:

- Ordinary PNs
- State machine
- Marked graph
- Free-choice
- Extended free-choice
- Asymmetric choice (or simple)

ارتباط بین این زیرنوع‌های شبکه پتری در شکل ۱۴-۳ نمایش داده شده است



شکل ۱۴-۳- ارتباط بین انواع شبکه‌های پتری

۱۴-۷-شبکه‌های پتری رنگی

شبکه‌های پتری رنگی توسط Kurt Jenson به عنوان یک مدل توسعه یافته از شبکه‌های پتری معرفی شده است. علاوه بر مکان‌ها، انتقال‌ها و مهره‌ها در این شبکه مفاهیم رنگ، گارد[1] و عبارت[2] معرفی می‌شوند. مقادیر داده‌ای در این شبکه‌ها توسط مهره‌ها حمل می‌شوند. شبکه‌های پتری رنگی ارائه‌دهنده مدل‌های دقیقتری از سیستم‌های پردازشی غیرهمگام پیچیده هستند. در این شبکه‌ها، برخلاف شبکه‌های

[1] Guard
[2] Expression

پتری، مهره‌ها از یکدیگر قابل تمیز هستند. زیرا هر یک از مهره‌ها دارای صفاتی به عنوان رنگ هستند. در یک شبکه پتری رنگی، گارد، یک عبارت بولی است که به یک انتقال منتسب می‌شود و شرایطی برای فعال‌شدن کمان ورودی ایجاد می‌نماید. هر یک از مکان‌ها، کمان‌ها و انتقالات می‌توانند بسته به رنگی که دارند، دارای گارد مخصوص به خود باشند و وقتی حاصل گارد «درست» باشد، عملیات انجام می‌شود.

۱۴-۸-زمان در شبکه پتری

در شبکه‌های پتری می‌توان به هر نشانه، مقداری زمانی منتسب کرد، که به این مقدار زمانی زمانمهر[1] گفته می‌شود. زمانمهر بیانگر اولین زمانی است که پس از برآورده شدن ورودی‌ها، نشانه می‌تواند شلیک شود. وقتی انتقال در حالت آماده است که زمان نشانه‌های مکان‌های ورودی انتقال کوچکتر و یا مساوی با زمان فعلی باشند و وقتی این زمان بیشتر باشد، شلیک انجام می‌شود. در شبکه‌های پتری مفهوم زمان از طریق عنصری با نام ساعت سراسری[2] معرفی می‌شوند. مقادیری که این ساعت اختیار می‌کند بیان‌کننده زمان مدل است. این زمان می‌تواند یک عدد صحیح باشد که نشانگر زمان گسسته است یا می‌تواند یک عدد حقیقی باشد که بیان‌کننده زمان پیوسته است..

برای آنکه بتوان یک رویداد که r واحد زمانی طول می‌کشد را در یک شبکه پتری مدل کرد، باید انتقال متناظر با رویداد ، زمانمهرهایی برای مهره‌های خروجی تولید کند که r واحد زمانی بزرگتر از مقدار ساعت سراسری باشد که در آن انتقال رخ داده است. این موضوع بدان معنا است که مهره‌های تولید شده تا r واحد زمانی در دسترس نیستند. در شبکه‌های پتری مبتنی بر زمان، وقتیکه در زمان فعلی مدل هیچ انتقالی آماده نیست، زمان سیستم به جلو کشیده می‌شود تا حداقل یک انتقال در حالت آماده باشد.

۱۴-۹-کاربرد شبکه پتری در مهندسی نرم‌افزار

با توجه به سادگی و پشتوانه قوی ریاضی شبکه‌های پتری، می‌توان از این نوع شبکه‌ها در ایجاد مدل قابل اجرا از معماری استفاده نمود. با استفاده از مدل قابل اجرای ایجاد شده از این شبکه‌ها پتری می‌توان

[1] Timestamp
[2] Global clock

ارزیابی از رفتار و سایر نیازمندی‌های غیر وظیفه‌مندی در مهندسی نرم‌افزار داشت. همچنین برای ارزیابی سایر محصولات مهندسی نرم‌افزار مانند نمودار مورد کاربری، ترتیبی، فعالیت می‌توان از شبکه‌های پتری استفاده نمود. در واقع، شبکه پتری می‌تواند برای ارزیابی اغلب نمودارهای UML بکار گرفته شده است. این محصولات، با اجرای الگوریتم‌هایی به شبکه پتری تبدیل می‌شوند و سپس خصوصیات کیفی[1] مختلف مورد بررسی قرار می‌گیرد. خصوصیات کیفی کارایی و قابلیت اطمینان از جمله مهمترین خصوصیاتی هستند که با تبدیل نمودارها به شبکه پتری به‌صورت کمی مورد ارزیابی قرار می‌گیرند.

در زمینه تبدیل نمودارهای UML به شبکه‌های پتری فعالیت‌های بسیاری صورت پذیرفته است که به‌طور خلاصه می‌توان این فعالیت‌ها را به‌صورت ذیل جمع‌بندی نمود:

- برای ارزیابی کارایی و قابلیت اطمینان، اطلاعات اضافی به توصیفات معماری اضافه می‌شود
- روش‌های کمی نتایج را به توصیفات معماری بازخورد می‌دهند
- بیشتر روش‌ها از سطح خودکارسازی بالایی برخوردار هستند
- بیشتر روش‌ها اجتماع مدل نرم‌افزار با مدل شبکه پتری را در سطح متوسط انجام می‌دهند

در ادامه به بررسی تبدیل چند نمودار UML به معادل شبکه پتری خواهیم پرداخت.

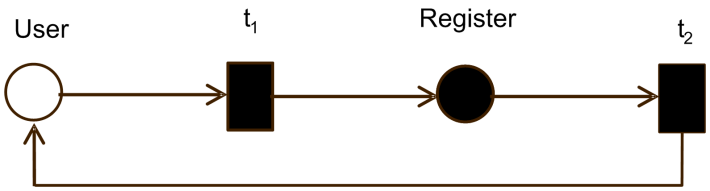
۱۴-۹-۱- تبدیل نمودار موارد کاربری

برای تبدیل نمودار موارد کاربری به شبکه پتری، ابتدا هر یک از موارد کاربری به‌همراه عامل‌ها به شبکه پتری با روش زیر تبدیل می‌شود:

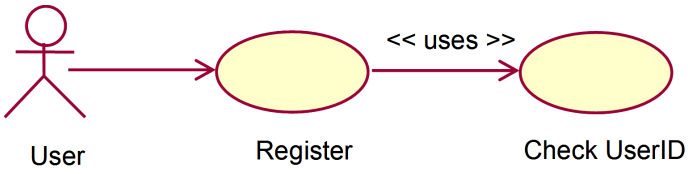
- هر کاربر و مورد کاربری به یک مکان نگاشت می‌شوند
- ورودی هر مکان، انتقالی با یک گارد است. گارد شرط مربوط به صدا زدن مورد کاربری توسط کاربر را نشان می‌دهد
- یک انتقال برای برگشت نیز وجود دارد
- مکان مربوط به هر مورد کاربری با شبکه پتری حاصل از نمودار ترتیبی آن جایگزین می‌شود. این مکان‌ها با دایره توپر نشان داده می‌شوند، تا از سایر مکان‌ها مجزا شوند

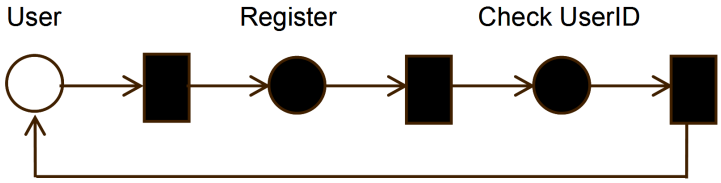
پس از انجام مراحل فوق، شبکه‌های پتری حاصل، ترکیب می‌شوند. (کنار هم قرار داده می‌شوند).

[1] Quality Attributes



شکل ۱۴-۴- تبدیل نمودار موارد کاربری به شبکه پتری



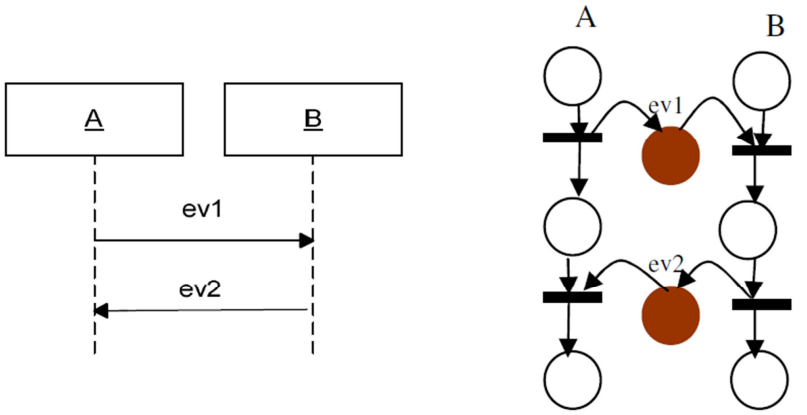


شکل ۱۴-۵- تبدیل نمودار موارد کاربری به‌همراه عبارت Uses

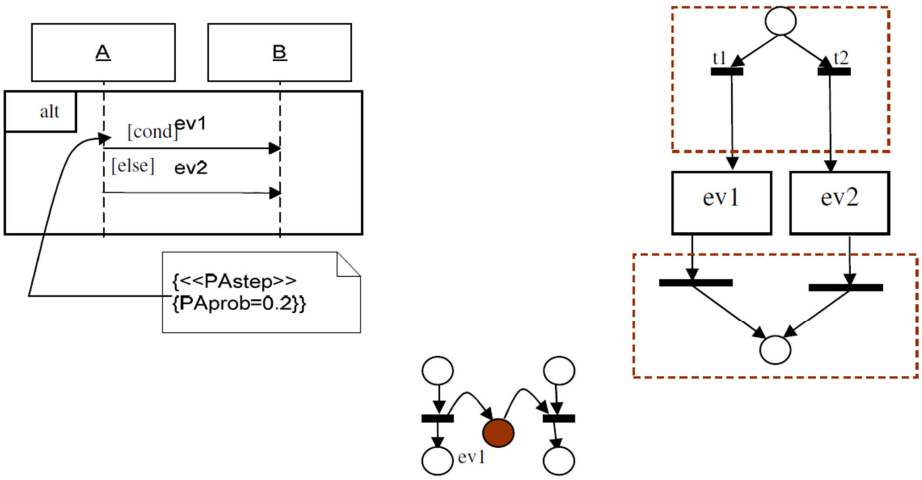
۱۴-۹-۲- تبدیل نمودار ترتیبی

برای تبدیل نمودار موارد ترتیبی به شبکه پتری، مراحل زیر به‌ترتیب انجام می‌شود:

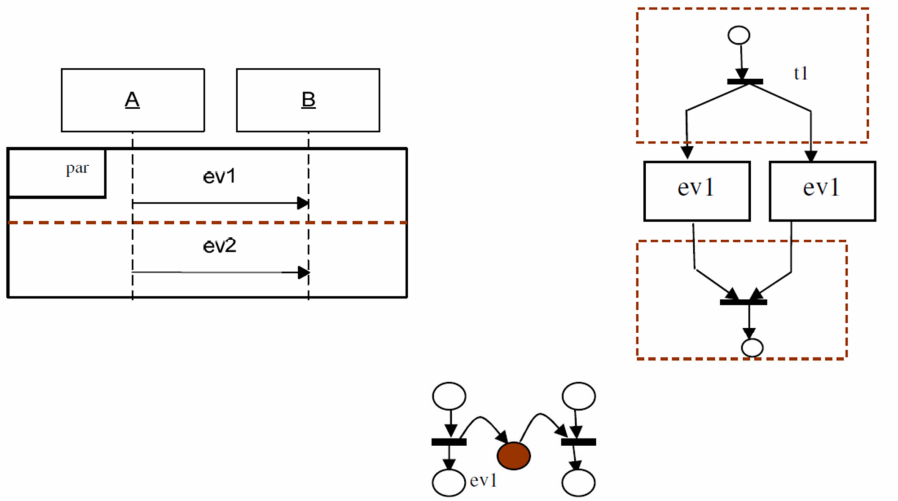
- به‌ازای هر پیام موجود در نمودار ترتیبی، مولفه‌های فرستنده و گیرنده آن به یک زیرسیستم شبکه پتری تبدیل می‌شوند
- شبکه‌های پتری حاصل، مطابق با ترتیب و ارتباط بین پیام‌ها ادغام می‌شوند
- در نهایت برای شبکه پتری حاصل، نشانه‌گذاری اولیه انجام شود



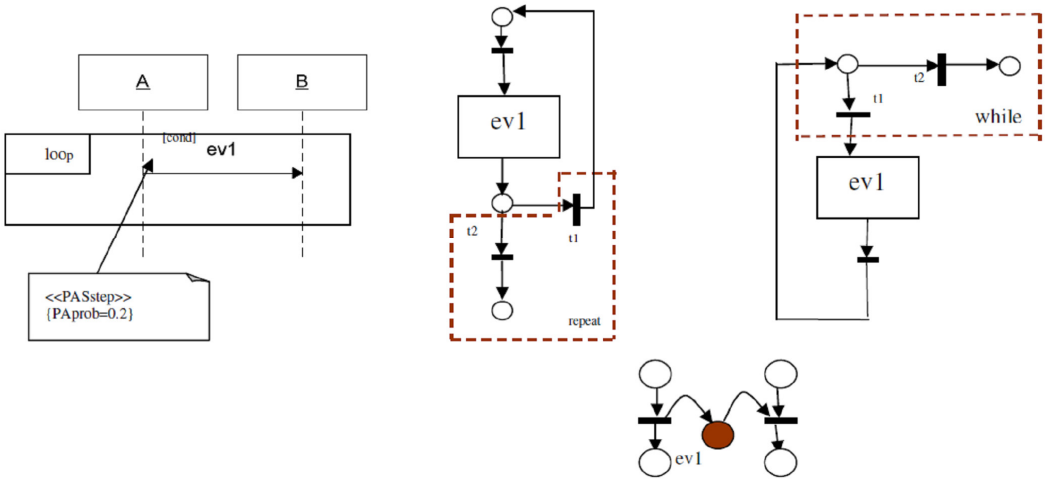
شکل ۱۴-۶- تبدیل ساختار ترتیب در نمودار ترتیبی به شبکه پتری



شکل ۱۴-۷- تبدیل ساختار انتخاب در نمودار ترتیبی به شبکه پتری



شکل ۱۴-۸- تبدیل ساختار توازی در نمودار ترتیبی به شبکه پتری



شکل ١٤-٩- تبدیل ساختار تکرار در نمودار ترتیبی به شبکه پتری

١٤-٩-٣- تبدیل نمودار مولفه

برای تبدیل نمودار موارد مولفه به شبکه پتری، مراحل زیر به‌ترتیب انجام می‌شود:

- هر یک از مولفه‌ها به‌صورت مجزا به شبکه پتری تبدیل می‌شوند
- شبکه‌های پتری حاصل مطابق با نوع ارتباط بین مولفه‌ها با هم ترکیب می‌شوند
- رفتار هر مولفه با عبارت مسیر[1] نشان داده می‌شود. به‌عبارت بهتر، عبارت مسیر شامل ترتیب میان عملیات یک مولفه است. همچنین فرض بر این است که کلیه عملیات انجام شده بوسیله هر مولفه، ترتیب فراخوانی آنها و دفعات اجرا و انتخاب این عملیات مشخص است.
- در نهایت برای شبکه پتری حاصل، نشانه‌گذاری اولیه انجام شود

جدول ١٤-١ فهرستی از عملگرهای عبارت مسیر را ارائه می‌دهد.

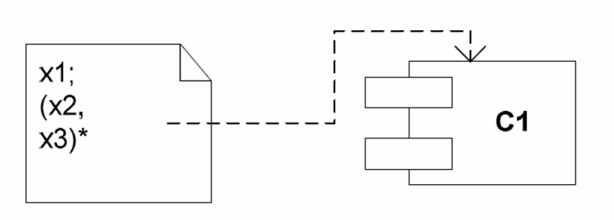
عملگر	شرح
؛	ترتیب اجرای عملیات عملگر
،	انتخاب یک عمل از بین مجموعه‌ای از عملیات
+	تکرار یک یا بیشتر عملیات و عملگر
×	تکرار صفر یا بیشتر عملیات و عملگر

[1] Path expression

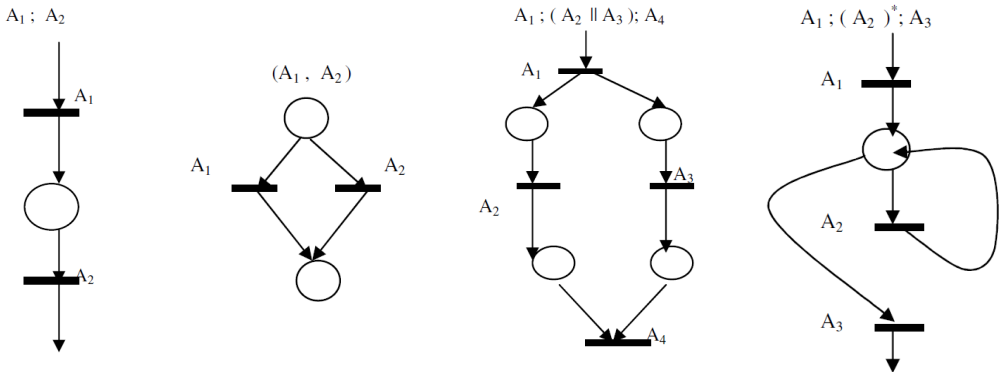
عملگر	شرح
||	اجرای موازی عملیات
( )	عملگر مورد نظر بر کل عملیات داخل پرانتز اعمال می‌شود

جدول ۱۴-۱- عملگرهای عبارت مسیر

شکل ۱۴-۱۰ نمونه‌ای از نمودار مولفه پالایش شده با عبارت مسیر را نشان می‌دهد که رفتار مولفه به نمودار مولفه اضافه شده است.



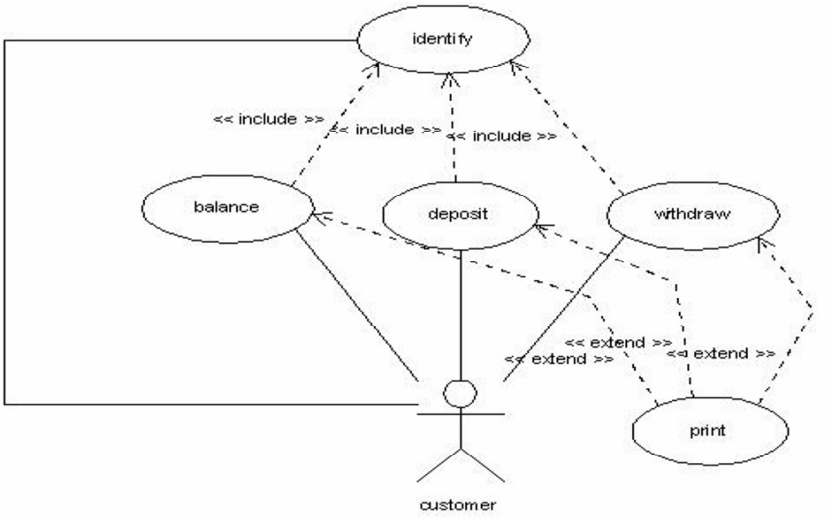
شکل ۱۴-۱۰- نمونه‌ای از نمودار مولفه پالایش شده با عبارات مسیر



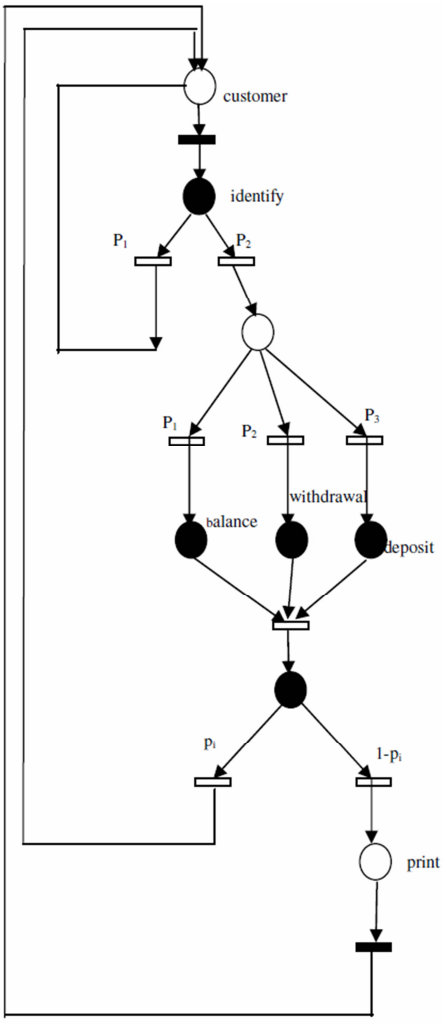
شکل ۱۴-۱۱- نمونه‌ای از تبدیل مولفه با عملگرهای مسیر خاص

۱۴-۱۰- نمونه ATM

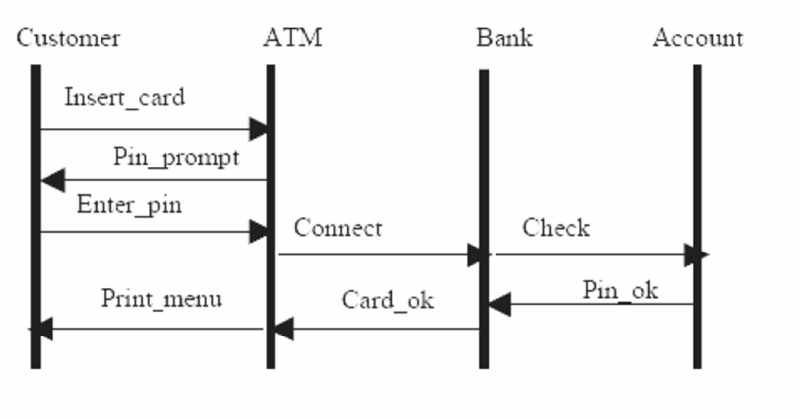
مسئله ATM در فصل‌های قبلی مورد بررسی قرار گرفت و نمودارهای مختلف آن ارائه شد. در این قسمت، با استفاده از تبدیلات بیان شده، نمودار موارد کاربری، ترتیبی و مولفه آن به شبکه پتری تبدیل خواهد شد. در ادامه پس از بیان هر نمودار مربوط به ATM، نمودار تبدیل آن نیز ارائه شده است.



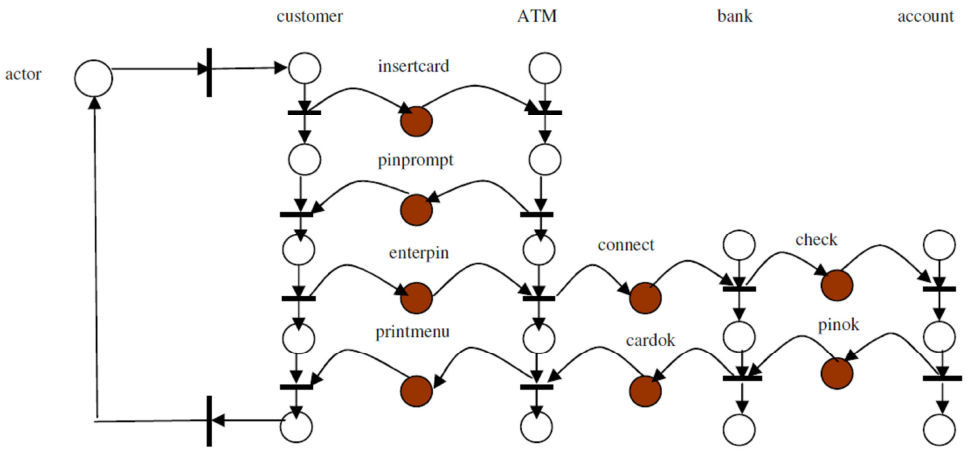
شکل ۱۴-۱۲- نمونه نمودار مواردی کاربری ATM



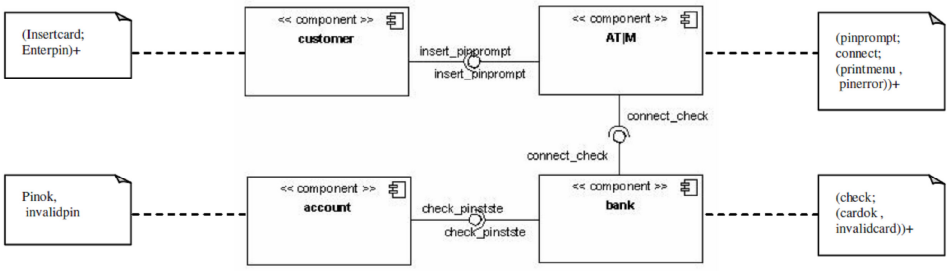
شکل ۱۴-۱۳- شبکه پتری معادل موارد کاربری ATM



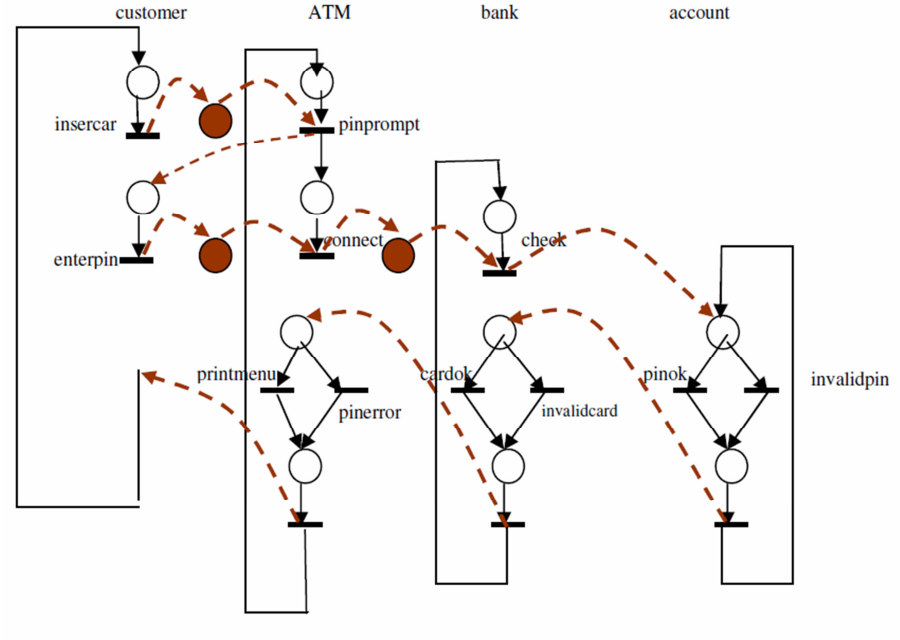
شکل ۱۴-۱۴- نمونه نمودار ترتیبی ATM



شکل ۱۴-۱۵- شبکه پتری معادل نمودار ترتیبی ATM



شکل ۱۴-۱۶- نمونه نمودار مولفه ATM



شکل ۱۴-۱۷- شبکه پتری معادل نمودار مولفه ATM

۱۴-۱۱- کاربرد شبکه پتری در مدلسازی فرآیند

شبکه‌های پتری علاوه بر مهندسی نرم‌افزار در مدلسازی فرآیندهای کسب و کار نیز می‌توانند مورد استفاده قرار گیرند. از عمده‌ترین مزایای استفاده از شبکه‌های پتری در نمایش رفتار فرآیند می‌توان به موارد زیر اشاره نمود:

- بیان صریح حالات و عملیات فرآیند
- ارائه یک مدل اجرایی از فرآیند با استفاده از شبکه پتری
- نمایش دقیق پیش‌شرط‌ها و پس‌شرط‌های عملیات
- مدلسازی ساده‌تر و دقیق‌تر
- قابلیت ارزیابی و رفع مشکلات

۱۵- توسعه برپایه عامل

پیچیدگی ذاتی نرم‌افزار سبب شده است که راه‌حل‌های بسیاری برای حل آن ارائه شود. در فصل‌های گذشته روش‌هایی چون تجرید، تجزیه و سازماندهی بیان گردید که سعی به حل پیچیدگی و یا کاهش آن دارند. این روش‌ها هر چند می‌توانند قسمتی از مسئله پیچیدگی را حل نمایند، اما در محیط‌های مدرن پیچیدگی به شکل‌های دیگری نیز ظاهر می‌شود که روش‌های ذکر شده فاقد توانایی لازم برای کاهش پیچیدگی آنها هستند. از جمله موارد پیچیدگی در محیط‌های مدرن می‌توان به هوشمندی، تعامل‌پذیری[1]، سازگاری، ناهمگنی سکوها و محیط توزیع شده نرم‌افزار اشاره نمود. چنین محیط‌هایی نیاز به نرم‌افزارهایی دارند که

- نیاز به کاربر نداشته باشند و مستقل از او واکنش دهند
- بتوانند به سود ما تصمیم‌گیری نمایند
- با دیگر سیستم‌ها براحتی تعامل برقرار کند
- در محیط‌های شبکه‌ای مختلف قابلیت اجرا داشته باشند
- با شرایط جدید محیط سازگار شوند

هر یک از این ویژگی‌ها سبب توسعه علوم و دانش حوزه خود شده‌اند، اما برآورده نمودن همه، سبب ایجاد زمینه جدیدی به‌نام «سیستم‌های چند عامله» شده است. سیستم‌های چند عامله مفهوم جدیدی در حوزه نرم‌افزار نیستند و سابقه طولانی در این حوزه دارند، اما با توجه به ماهیت‌شان کمتر مورد توجه قرار گرفته‌اند. در این بخش به بررسی عامل‌های و توسعه سیستم‌های مبتنی بر عامل خواهیم پرداخت.

۱۵-۱-عامل[2]

موجودیتی نرم‌افزاری (یا سیستم کامپیوتری) است که مناسب محیط خاصی طراحی شده و قادر به انجام اعمال انعطاف‌پذیر و مستقل برای رسیدن به اهداف درنظر گرفته شده برای آن سیستم می‌باشد. در واقع، عامل‌ها، موجودیت‌هایی قابل تشخیص برای حل مسئله با محدوده و رابط خوش‌تعریف هستند و برای محیط‌های خاصی مناسب هستند.

[1] Interoperability
[2] Agent

آنها بسته به حالت‌شان ورودی‌هایی را از طریق سنسورها دریافت می‌کنند و را از طریق مجریان[1] می‌گذارند. هر عامل برای برآورده نمودن اهداف خاصی طراحی شده است، به همین دلیل نمی‌توان عامل‌های یک محیط را در محیط‌های دیگر مورد استفاده قرار داد و این امر پیچیدگی طراحی عامل‌های را بیشتر می‌کند.

خصوصیات بیشماری برای عامل ذکر گردیده است که از عمده‌ترین خصوصیان عوامل می‌توان به موارد زیر اشاره نمود:

- خودگردانی[2]
  - اعمال اصلی بدون دخالت انسان یا عامل دیگری انجام می‌شود. ارتباط میان عامل‌ها با دیگر اجزا به صورت سلسله مراتبی نیست
- پیش‌فعال بودن[3]
  - یک عامل نباید حتماً منتظر فراخوانی باشد بلکه باید تحت شرایط خاص محیط فعال شود
- واکنش‌دهی[4]
  - عامل باید در برخی سیستم‌ها محیط خود را حس کرده و به تغییراتی که در آن رخ می‌دهد پاسخ دهد
- ارتباط با دیگران[5]
  - ممکن است یک عامل نرم‌افزاری عامل دیگری را فراخوانی کند، درباره آن استنتاج انجام دهد و یا با آن به مذاکره بپردازد
- یادگیری
  - براساس تغییراتی که در محیط رخ می‌دهد رفتار خود را تغییر می‌دهد
- انعطاف‌پذیری
  - رفتارهایی که عامل انجام می‌دهد، از پیش تعیین‌شده نیستند

[1] effectors
[2] Autonomous
[3] Pro-activeness
[4] Reactivity
[5] Communicative

- هدف‌گرا[1]
  - تنها به محیط خود پاسخ نمی‌گوید بلکه بدنبال دستیابی به اهداف خود است
- متحرک[2]
  - توانایی جابه‌جا کردن خود از سیستمی به سیستم دیگر را دارد
- شخصیت[3]
  - دارای حالت بوده و شخصیتی مستقل است

۱۵-۱-۱- عامل‌ها و اشیاء

عامل‌ها خصوصیاتی شبیه به اشیاء دارند، از این رو لازم است که تفاوت‌ها و شباهت‌های آنها مورد بررسی قرار گیرد. از عمده‌ترین شباهت‌های بین عامل‌ها و اشیاء می‌توان به موارد زیر اشاره نمود:

- هر دو موجودیت‌هایی را تعریف می‌کنند
- عامل‌ها را اشیا فعال می‌نامند
- هر دو ساختار داخلی‌شان را بسته‌بندی می‌کنند
- به وسیله ارسال پیام با یکدیگر ارتباط برقرار می‌کنند

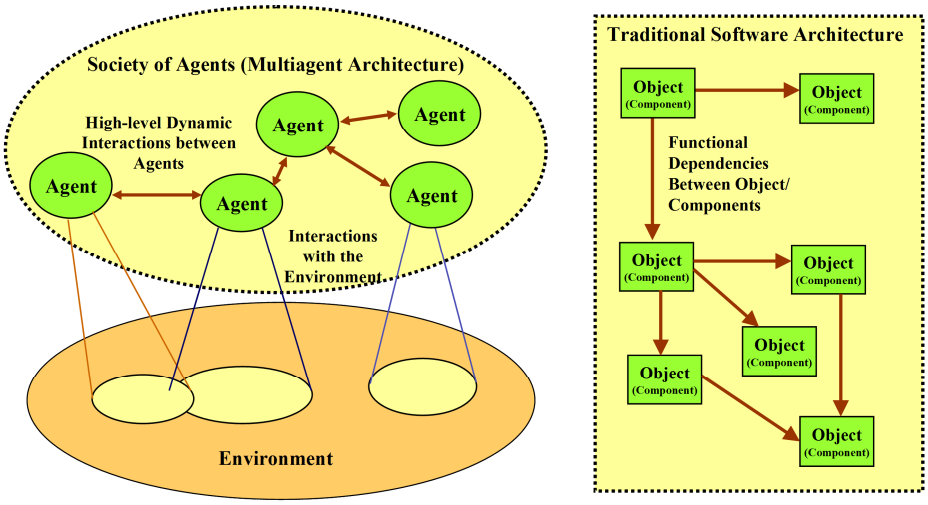
از عمده‌ترین تفاوت‌های بین این دو می‌توان به موارد زیر اشاره نمود:

- عامل‌ها پیش‌فعال هستند
  - اشیا قبل از آنکه فعال شوند باید درخواست دریافت کنند
- عامل‌ها از نظر ساختار درونی انعطاف‌پذیرند
  - اشیا در واقع متدها و خواص را بسته‌بندی می‌کنند. اگر یک تابع به صورت خصوصی در شی تعریف شده باشد اشیا دیگر امکان دسترسی به آن را ندارند
- هر عامل یک کنترل مخصوص به خود دارد
  - کنترل در سیستم‌های شی‌گرا به صورت مرکزی انجام می‌شود

[1] Goal-oriented
[2] Mobile
[3] Character

- هر عامل می‌تواند در هر زمان که تشخیص بدهد و با توجه به شرایط با عامل دیگر ارتباط برقرار کند
  - در روش شی‌گرا ارتباط‌ها به صورت مقطعی و از پیش تعیین شده در سیستم وجود دارند که به وسیله کلاس‌های سلسله مراتبی تعریف می‌شوند

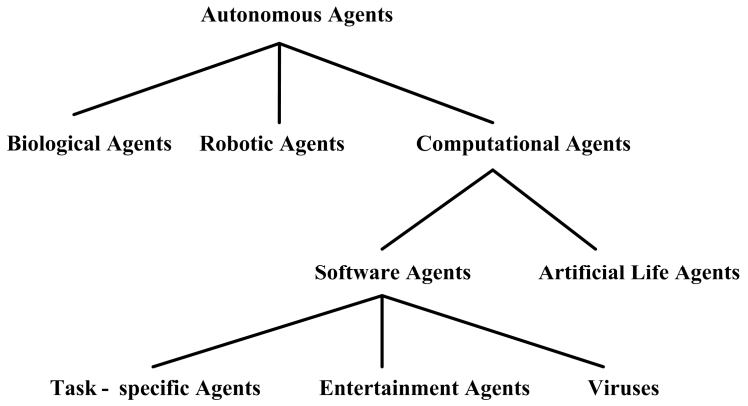
شکل ۱۵-۱ تفاوت‌های بین عامل‌ها و اشیاء را در اجرا نشان می‌دهد. عامل‌ها نسبت به اشیاء انعطاف بیشتری در اجرا نشان می‌دهند.



شکل ۱۵-۱- مقایسه بین عامل‌ها و اشیاء

۱۵-۱-۲- طبقه‌بندی عامل‌ها

طبقه‌بندی‌های متنوعی برای عامل‌ها ارائه شده است که هر یک با توجه به دیدگاه خاصی انجام شده است. عامل‌ها از نظر محیط، نوع عامل، هوشمندی، نوع کار و سایر پارامترها می‌توانند دسته‌بندی شوند. شکل ۱۵-۲ انواع عامل‌ها را از نظر نوع ارائه می‌دهد. عامل‌های نرم‌افزاری که جزء عامل‌های محاسباتی محسوب می‌شوند به سه دسته تقسیم می‌شوند که ویروس‌های هوشمند یکی از آنها هستند.



شکل ۱۵-۲- طبقه‌بندی عامل‌ها از نظر نوع

۱۵-۱-۳- سیستم‌های چندعامله

سیستمی نرم‌افزاری که از مجموعه‌ای از عامل‌ها که با یکدیگر تعامل دارند، رقابت می‌کنند و همکاری دارند، تشکیل شده است. در این سیستم‌ها عامل‌ها می‌توانند دارای اهداف متفاوتی باشند. در واقع، عامل‌ها در فضایی که سیستم در اختیار آنها قرار می‌دهد، استفاده نموده و فعالیت‌های خود را انجام می‌دهند. در مسیر انجام عملیات، عامل‌ها همانند انسان‌ها با دیگر عامل‌ها تعامل دارند و اطلاعات ردو بدل می‌کنند.

۱۵-۲-مهندسی نرم‌افزار عامل‌گرا

طراحی سیستم‌های چند عامل‌ها و نرم‌افزارهایی که از عامل استفاده می‌کنند با نرم‌افزارهای متداول متفاوت است. در واقع برای استفاده از عامل‌ها نمی‌توان از روش‌های متداول تحلیل و طراحی استفاده نمود، چرا که روش‌های متداول نمی‌توانند به سوالات ذیل پاسخ گویند:

- چگونگی تجزیه مسئله
- چگونگی انتساب وظایف به عامل‌ها
- چگونگی هماهنگی، کنترل و ارتباط عامل‌ها
- چگونگی حل تعارض بین اهداف عامل‌ها
- چگونگی تصمیم‌گیری عامل در مورد سایر عامل‌ها و حالتشان

این سوالات و سوالاتی نظیر آنها پیچیدگی طراحی نرم‌افزار عامل‌گرا را نشان می‌دهند. برای توسعه سیستم‌هایی که از عامل استفاده می‌کنند از متدولوژی‌های عامل‌گرا استفاده می‌شود. این متدولوژی تنها برای توسعه یک سیستم برپایه عامل مورد استفاده قرار می‌گیرند و با توجه به دستیابی به چنین هدفی طراحی و پیاده‌سازی می‌شوند. همچنین نکته‌ای که می‌بایست در نظر داشت این است که متدولوژی عامل‌گرا علاوه بر پاسخ به سوالات قبلی، باید روشی شفاف و منظم برای تحلیل، طراحی و توسعه سیستم‌های چندعامله ارائه دهند.

متدولوژی‌های عامل‌گرا به دو دسته اصلی تقسیم می‌شوند که عبارتند از:

- متدولوژی‌هایی که از روش‌های شی‌گرا ناشی شده‌اند
  - از جمله معروفترین متدولوژی‌های برگرفته شده از شی‌گرا می‌توان به AUML، GAIA، ADEPT و Prometheus اشاره نمود. از جمله مزایای استفاده از این متدولوژی‌ها برای توسعه سیستمی برپایه عامل می‌توان به موارد زیر اشاره نمود:
    - برخی عامل را شی فعال می‌نامند و ادعا می‌کنند که روش‌های شی‌گرایی به‌تنهایی برای توسعه عامل‌ها کافی هستند
    - روش‌های شی‌گرا متداول و شناخته شده هستند
    - یادگیری این متدلوژی‌ها توسط مهندسان نرم‌افزار سریعتر انجام شده و هزینه کلی تولید سیستم کاهش می‌یابد
    - امکان استفاده از برخی مدل‌های موجود در روش‌های شی‌گرا مانند موارد کاربری در عامل‌ها وجود دارد

این متدولوژی‌های معایب خاص خود را نیز دارند که از آن جمله می‌توان به موارد زیر اشاره نمود:

- عامل‌ها تجرید بالاتری نسبت به شی دارند، بنابراین نحوه شکستن مسئله در روش‌های مبتنی بر عامل نسبت به شی‌گرا متفاوت می‌باشد
- در روش‌های شی‌گرا تکنیکی برای مدل کردن حالت‌هایی مانند هدف‌گرایی و پیش‌فعال بودن وجود ندارد

- مدل کردن نحوهٔ ارتباط میان عامل‌ها با استفاده از متدلوژی‌های شی‌گرا امکان‌پذیر نیست
- متدولوژی‌هایی که از مهندسی دانش ناشی شده‌اند
  - در این متدولوژی‌ها بیشتر توجه به فرایند مهندسی دانش در توسعه سیستم متمرکز است. فرایند مهندسی دانش شامل فرآیند استخراج[1]، ساختاربندی[2]، شکل‌دهی[3] و عملیاتی نمودن[4] دانش است که در واقع هدفی است که عامل دنبال می‌کند. از جمله این متدولوژی‌های می‌توان به DESIRE، MAS-CommonKADS و Tropos اشاره نمود. عمده‌ترین مزیت استفاده از این متدولوژی‌ها برای توسعه سیستمی برپایه عامل «امکان استفاده از ابزارها و کتابخانه‌های هستان‌شناسی و روش‌های با قابلیت استفاده مجدد در حل مسائل» است و مهمترین مشکل این روش‌ها عدم در نظر گرفتن ویژگی‌های توزیعی عامل‌ها است که می‌تواند توسعه سیستم را دچار مشکل نماید.

۱۵-۲-۱- متدولوژی توسعه GAIA

متدولوژی GAIA از معروفترین متدولوژی‌های توسعه عامل برپایه شی‌گرا است. این متدولوژی اولین بار توسط Jennings و Wooldridgeدر سال ۱۹۹۹ معرفی شد و نسخه نهایی توسط Zambonelli، Jennings و Wooldridge در سال ۲۰۰۳ ارائه شد. متدولوژی GAIA همانند همه متدولوژی‌های شی‌گرا، از نیازمندی‌های توسعه شروع می‌کند و بعد از تشخیص نیازمندی‌ها سعی به طراحی نرم‌افزاری با دیدگاه عامل دارد. در واقع، این متدولوژی سعی دارد تیم توسعه‌دهنده را برای طراحی یک سیستم چندعامله خوش-تعریف هدایت کند و با توجه به شناختی که تیم توسعه از متدولوژی شی‌گرا دارند، از مفاهیم شی‌گرایی نیز برای طراحی و توسعه سیستم استفاده نموده است. برخلاف برخی نظرات که اعتقاد دارند که متدولوژی‌هایی که پایه شی‌گرایی دارند نمی‌توانند برای نرم‌افزارهای مبتنی بر عامل بزرگ استفاده شوند، GAIA توانایی مدلسازی و حل پیچیدگی سیستم‌های چندعامله را نیز دارد.

[1] Eliciting
[2] Structuring
[3] Formalizing
[4] Operationalizing

از مهمترین نقاط ضعف این متدولوژی می‌توان به موارد زیر اشاره نمود:

- مسائل و مشکلات پیاده‌سازی در نظر گرفته نمی‌شوند
- تعیین نیازمندی‌ها و مدل‌سازی آنها مسکوت مانده است
- توسعه با این روش، ترتیبی است

۱۵-۲-۲- متدولوژی Troops

این متدولوژی اساساً عامل‌گرا طراحی شده است و به‌همین دلیل در توسعه نرم‌افزارهای عامل‌گرا مورد توجه قرار گرفته است. با توجه به عامل‌گرا بودن این متدولوژی، نمادگذاری[1] عامل‌ها، اهداف و مسائلی از این قبیل در فرآیند توسعه دیده شده است که یکی از مهمترین نقاط قوت این متدولوژی است. از مهمترین ویژگی‌های متدولوژی Troops می‌توان به حرکت تدریجی فرایند توسعه عامل‌ها اشاره نمود. در واقع این متدولوژی با تعداد کمی از عناصر شروع می‌کند و تدریجاً آنها را افزایش می‌دهد. در هر بار افزایش، جزئیات به‌صورت تدریجی افزوده می‌شوند و تعامل و وابستگی‌ها بررسی و بروز می‌شوند. هر گام از متدولوژی می‌تواند منجر به معرفی/حذف عناصر یا ارتباطات شود که این بسته به شناخت طراحان و توسعه‌دهندگان نرم‌افزار از عواملی است که می‌بایست پیاده‌سازی شوند.

از مهمترین نقاط ضعف این متدولوژی می‌توان به موارد زیر اشاره نمود:

- برای طراحی نرم‌افزارهای عامل پیچیده که ارتباطات بین ذینفعان پیچیده است، مناسب نیست
- تضمین کیفیت تنها با استفاده از الگوهای طراحی!
- راهنمایی‌هایی بسیار کمی در مورد گام‌ها و چگونگی انجام آن می‌توان یافت نمود
- تحویل‌دادنی‌های آن خوش-تعریف نیستند

[1] notions