بسم الله الرحمن الرحیم

# درس مهندسی نرم افزار پیشرفته

مـدرس : رضا قائمی

# فهرست

# ۱- مشکلات تولید نرم افزار و بررسی مسئله پیچیدگی در نرم افزار

امروزه نرم افزار نقش دوگانه ای را بازی می کند. در یک نقش به عنوان محصول نهائی[۱] و نقش دیگری به عنوان تولید کننده محصول نهائی است. در نقش اول، نرم افزار آن پتانسیل بالقوهٔ سخت افزار را به فعلیت می رساند و در این نقش- در کاربردهای گوناگونی که مورد استفاده قرار می گیرد از تلفن همراه گرفته تا کامپیوترهای بزرگ[۲]- به عنوان تبدیل کننده (تولید، مدیریت، بازیابی، بهنگام سازی و نمایش) اطلاعات عمل می نماید. این اطلاعات میتواند به سادگی یک بیت وبه پیچیدگی یک شبیه سازی چند رسانه ای[۳] باشد. اما در نقش دوم، نرم افزار به عنوان ابزار اساسی کنترل سیستم های کامپیوتری(سیستم عامل)،کنترل شبکه های کامپیوتری و طراحی و تولید نرم افزارهای دیگر(ابزارهای و محیطهای برنامه نویسی) عمل می کند. بعقیده صاحبنظران، نرم افزار یکی از نیروهای اصلی و محرک قرن بیست و یکم خواهد بود، زیرا مهمترین محصول آن قرن را پردازش مینماید که همان اطلاعات است. امروزه، نرم افزار یک عامل حیاتی در گردش کار موسسات، کارخانجات، صنعت حمل و نقل، پزشکی، بانکداری، شبیه سازی سیستمهای علمی و صنعتی و دیگر موارد است. همچنین کاربردهای نرم افزار از نمایش بهتر و قابل استفاده تر اطلاعات شخصی گرفته تا مدیریت اطلاعات سازمانهای بزرگ و فراهم کردن یک بستر اطلاعاتی قوی(اینترنت) که بوسیله آن ایده دهکده جهانی تحقق گردیده، گسترش یافته است.

## بحران نرم افزار

در نیمه دوم قرن بیستم، نقش نرم افزار دستخوش تغییرات زیادی شده است. پیشرفت شگرف سخت افزار، تغییرات اساسی در معماری سیستمهای کامپیوتری و افزایش شگفت انگیز ظرفیت حافظه های 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 تولید نرم افزار و ناتوانی این روشها در

---

[۱] End-Product

[۲] Mainframes

[۳] Multimedia Simulation

کنترل پیچیدگی نرم افزار باعث بوجود آمدن معضلاتی در تولید آن شد، که به آنها اصطلاح "بحران نرم افزار[1]" اطلاق می شود.

علایم و نشانه های این بحران عبارت بودند از:

ناتوانی نرم افزار در بهره گیری کامل از پیشرفت سریع و قدرت و اطمینان پذیری رو به افزایش سخت افزار.

ناتوانی روشهای تولید نرم افزار در پاسخگویی به افزایش تقاضا روی سیستمهای کامپیوتری پیچیده.

هزینه های هنگفتی که برای تولید نرم افزار پرداخته می شد.

تأخیری(احیاناً تا سالها) که در تولید نرم افزار رخ می داد.

عدم تامین مشخصات و نیازمندیهای مورد نظر کاربر.

کیفیت پایین و نامطمئن و ناکارا بودن نرم افزار.

قدرت ما در نگهداری سیستمهای موجود منوط به کیفیت طراحی و وجود منابع مناسب است.

از آنجاییکه -معمولا- سطح اغلب طراحیها پایین است و منابع مناسب وجود ندارد، نگهداری پر هزینه می گردد.

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

## متدولوژی و ضرورت توجه به آن

چنانکه بیان نمودیم، یکی از علل اساسی بحران نرم افزار عدم وجود روشهای مناسبی برای تولید نرم افزار 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 نرم افزاری را به خود جلب نموده و در نتیجه متدولوژی های گوناگون ارائه گردیده است.

برای روشنتر شدن بحث به بیان فرق بین متدولوژی و متد می پردازیم:

یک متد عبارت است از فرآیندی منظم برای تولید مجموعه ای از مدلها که هر کدام بخشی از سیستم نرم افزاری در حال تولید(یا توسعه) را تشریح نموده و با یک علامت گذاری روشن نمایش شده اند.

---

[1] Software Crisis

متدولوژی عبارت از مجموعه ای از متدها که در تمام چرخه حیات سیستم نرم افزاری اعمال شده و بر یک نوع نگرش کلی درباره جهان نرم افزار متکی باشند. توجه داشته باشید که تعریف ذکر شده از واژه متدولوژی با تعریف آن در علم و فلسفه فرق می نماید. در واقع اینجا مقصود از متدولوژی، متدولوژی تولید نرم افزار است. به نظر می رسد یک متدولوژی تولید نرم افزار باید حد اقل ۷ ویژگی زیر را داشته باشد:

۱- ارئه تعاریفی از مفاهیم اولیه بکاررفته در متدولوژی: بعبارت دیگر دیدگاه های اصلی یک متدولوژی درباره روش حل مسئله باید روشن باشد. مثلاً برای ایجاد یک ساختمان باید مولفه های بکاررفته در آن مانند آجر و آهن تعریف شده باشد. مثال دیگر: هنگامیکه متدولوژی شئ گرائی نرم افزار را بعنوان مجموعه ای از اشیاء مستقل که با یکدیگر به صورت هوشمندانه همکاری می نماید تلقی می کند، باید بتواند تعریف روشنی از شئ ارائه نماید.

۲- ارائه مدلی برای فرآیند تولید: مدل فرآیند[۱] عبارت از الگوی تولید نرم افزار است. بعبارت روشن تر مدل فرآیند گامهای لازم برای تولید نرم افزار(همان نحوه و ترتیب استفاده از متدها) و چگونگی انتقال از یک گام به گام دیگر را بیان می نماید. انتظار می رود بین متدولوژی و فرآیند تولید سنخیت و سازگاری وجود داشته باشد. برای مثال متدولوژی ساخت یافته که نرم افزار را بصورت مجموعه ای از گامهای متوالی تعریف می نماید با فرآیند تولید "آبشاری"که تولید نرم افزار 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 شئ گرائی که بر مفهوم "کلاس" 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 می باشد.

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

---

[۱] Process Model

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

٥- معرفی تکنیکهایی برای پیاده سازی 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 نرم افزاری پیچیده را بوجود می آورند. همچنین بوسیله روشها، فرآورده هایی تولید شده که بعنوان وسائل ارتباطی بین افراد تیم عمل نمایند و بالآخره بوسیله این روشها می توان نقاطی را در طول زمان تولید معرفی نمود که در آن مدیریت روند پیشرفت پروژه را ارزیابی کرده و خطرات بالقوه را کنترل می نماید.

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 شئ گرائی انتظار می رود که معیارهایی برای مقایسه بین دو مدل کلاس یا تعیین میزان قابلیت استفاده مجدد از یک مولفه را معرفی نماید. وجود این معیارها می تواند بحث استاندارد پذیری و قابلیت اعتماد و کارایی را گسترش دهد.

٧- وجود ابزار اتوماتیک برای کمک به تولید و اجرای مدلهای مبتنی بر متدولوژی[3]: این ویژگی جزو متدولوژی نیست و لی برای تضمین بکارگیری کارا و صحیح و تسهیل در امر بهره برداری از آن ضروری می باشد.

ولی با توجه به آشنائی نسبتاً دیرینه بشر با روشهای مهندسی و اینکه مهندسی سخت افزار با مشکلات مهندسی نرم افزار مواجه نشده است، این سؤال مطرح میگردد که آیا استفاده از روشهای سنتی مهندسی در تولید نرم افزار، مشکلات آنرا برطرف نمی سازد؟

در پاسخ باید گفت که مسلماً استفاده از روشهایی کلی[4]-که بشر تا به امروز شناخته- در کنترل پیچیدگی سیستمهای نرم افزاری-چنانکه خواهیم دید-باید کارساز باشد و لی این روشها بسیار کلی هستند. به عبارت دیگر اگر جزئیات بکاربردن روش بخصوصی را نیز مد نظر داشته باشیم، خواهیم دید

---

[1] Standard Notation

[2] Software Metrics

[3] CASE Tools

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

که در تولید نرم افزار بعنوان یک محصول نهائی امکان استفاده از روشهایی است که در ساخت و تولید محصولاتی دیگری همچون سخت افزار به کاربرده میشود، مشکل خواهد بود اما بکارگیری این روشها بعنوان یک ایده در نظر سازندگان نرم افزار هنوز از جلوه خاص برخوردار بوده و زمینه چالش در این مورد را فراهم می نماید.

## تفاوت روش تولید نرم افزار و سخت افزار

*فرآیند تولید نرم افزار یک فرآیند مهندسی[1] است نه یک فرآیند ساختن[2] سنتی.*

برای بیان تفاوت بین این دو فرآیند باید گفت:

در سخت افزار 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 تولید نرم افزار امکان ساخت فیزیکی محصول بر اساس محصولات موجود کوچکتر از طریق سرهم بندی آنها بسادگی وجود ندارد. بلکه برای دستیابی به محصول نهائی نرم افزار لازم است یک *فرآیند مهندسی* که برای هر کاربرد جدید منحصر به فرد است را دنبال نمود. این مسأله از متفاوت بودن طبیعت نرم افزار بعنوان یک محصول منطقی که بیشتر زاییده فکر انسان بوده در مقابل سخت افزار بعنوان یک محصول فیزیکی که مواد اولیه موجود در آن تابع قوانین فیزیکی مشخصی است، ناشی می شود.

از سوی دیگر بدلیل این تفاوت ذاتی بین نرم افزار و سخت افزار پیچیدگیهای خاصی در ابعاد مختلف از جمله در تعریف نرم افزار، طراحی، پیاده سازی، 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 وابزار خاصی را برای کنترل و فائق آمدن بر این پیچیدگی مطرح نموده اند و از این جمله متدولوژی شئ گرایی که در ادامه خواهیم دید.

## شیوه مقابله با بحران نرم افزار

برای حل مشکلات نرم افزار نیاز به روشهای فنی و سیستماتیک[3] برای کنترل پیچیدگی نرم افزار( و در نتیجه بحران نرم افزار) احساس می شود. همچنین نیاز به یک فرآیند تولید مدون که چهارچوب گامهای مورد نیاز برای اعمال این روشها- که نتیجه آن تولید یک نرم افزار با کیفیت عالی

---

[1] Engineering Process

[2] Manufacturing Process

[3] Technical and Systematic Methods

و قابلیت اعتماد بالا در حالیکه از نظر اقتصادی مقرون بصرفه باشد- وجود دارد. البته جایگاه ابزارهایی[1] که بکاربردن این روشها را در چهارچوب یک فرآیند تولید معین آسانتر می نمایند، قابل اغماض نیست.

یکی از روشهای مدعی نو آوری در زمینه کنترل پیچیدگی نرم افزار، شئ گرایی است.

روشهای ارائه شده قبل از روش شئ گرایی(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 شئ گرایی بوده است- را نداشته اند.

تکنیکهای سنتی موجود توان پاسخگویی به مثالهایی از سیستمهای بزرگ و واقعی را نداشته، مثلا در مورد طراحی ساخت یافته[2], که در عمل بیشتر از بقیه روشهای طراحی مورد استفاده قرار گرفته است، کاربردهای عملی نشان داده که برنامه نویسی ساخت یافته هنگامیکه تعداد خطوط برنامه از مرز ۱۰۰۰۰۰ خط فراتر رود، این روش در کنترل برنامه با مشکل مواجه می گردد.

سیستمهای بزرگ و کاربردهای جدیدی که روز به روز همراه با پیشرفت سخت افزارها مطرح شده اند نیازمند ساختاری برای مدیریت مدلهای پیچیده تری بوده اند که تنها مکانیزمهای موجود در تکنولوژی شئ گرایی پتانسیل برخورد با گستردگی و پیچیدگی این سیستمها را دارند. ایده اصلی برای فراهم آوری این پتانسیل تکیه بر مفهوم قابلیت استفاده مجدد[3] و تعریف وساخت مؤلفه های استاندارد با قابلیت کاربرد در محیط های مختلف است.

همانطوری که بیان نمودیم روشها، باید در چهارچوب فرآیندها اعمال گردند.

فرآیندی که در تولید سیستمهای سنتی به کار می رود فرآیند آبشاری می باشد. به نظر می رسد 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 شئ گرایی 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 نرم افزار عدم وجود روشهای مناسبی برای تولید نرم افزار بوده ، و این بمعنی ناتوانی روشهای موجود در کنترل پیچیدگی نرم افزار است. پس بیاید پیچیدگی نرم افزار را مورد بررسی قرار دهیم.

---

[1] Computer-Aided Software Engineering(CASE) Tools

[2] Structured Design

[3] Reusability

## پیچیدگی ذاتی نرم افزار

در واقع می توان سیستمهای نرم افزاری را با توجه به پیچیدگی شان به دو دسته کلی طبقه بندی کرد:

سیستمهای نرم افزاری معمولی(غیر پیچیده): این سیستمها معمولا بوسیله یک نفر طراحی، پیاده سازی و نگهداری می شوند. چنین سیستم هایی وقتی نیازهای مورد نظر کاربران خود را براورده نمی سازند با سیستمهای جدید جایگزین می گردند، در واقع این سیستمها ارزش استفاده مجدد یا اصلاح را ندارند.

اما در مقابل دسته فوق، دسته دیگری از سیستمهای نرم افزاری وجود دارد که به آنها نرم افزارهای با بنیه صنعتی[1] اطلاق می شود. این دسته جزو سیستمهای پیچیده به شمار می آیند که بارزترین ویژگی آنها این است که درک همه جزئیات آن از عهده قدرت ذهنی یک نفر خارج است. در واقع اصطلاح بحران نرم افزار در رابطه با تولید مشکلات اینگونه سیستمها مطرح شده است. مثال هایی از این سیستمها عبارتند از سیستمهای بلا درنگ[2]، پایگاه داده هایی که باید میلیونها رکورد ذخیره نمایند در حالیکه دستیابی صدها کاربر به صورت همزمان به داده ها با کارایی بالا را فراهم کند، سیستمهای اطلاعاتی بزرگ و سیستمهای هوشمند.

چنانچه قبلا بیان کردیم یک تفاوت اساسی بین نرم افزار و دیگر ساخته های دست بشر وجود دارد بدین صورت که نرم افزار یک محصول منطقی است که بیشتر زاییده فکر انسان بوده لذا بین عناصراولیه آن قوانین بنیادی حاکم نیست. در حالیکه محصولات دیگر ساخت بشر بگونه ای هستندکه عناصر اولیه تشکیل دهنده آن از قوانین فیزیکی معینی پیروی می نمایند.

با توجه به این مطلب، دو نکته درباره پیچیدگی سیستمهای نرم افزاری 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 نرم افزاری 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 غیر قانونمند را بیشتر توضیح می دهیم: در صنعت عمران و ساختمان سازی اگر مشتری از یک شرکت ساختمانی درخواست اضافه یک طبقه زیر زمینی به یک ساختمان صد طبقه را نماید، مسلما یک نوع شوخی تلقی خواهد شد!، در حالیکه در صنعت نرم افزار چنین درخواستی کاملا طبیعی بنظر می رسد!(البته از نظر مشتری). علاوه بر این، مشتریان چنین استدلال می کنند 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 نرم افزاری بزرگ است به عبارت دیگر *نمی توان این پیچیدگی را از بین برد بلکه باید آن را کنترل نمود.*

## عوامل پدید آورنده پیچیدگی ذاتی

اما چرا پیچیدگی یک خاصیت جدائی ناپذیر برای سیستمهای نرم افزاری بزرگ است؟
در واقع، این پیچیدگی از ٤ فاکتور ناشی می شود:

### ۱) پیچیدگی خود مسأله

معمولا سیستمهای نرم افزاری بزرگ حاوی عناصری است که پیچیدگی آنها اجتناب ناپذیر بوده، این پیچیدگی ناشی از :

*وجود نیازمندیهای[۱] گوناگون و مختلف و گاهی حتی متضاد*

چنانکه می دانیم نیازمندیها به دو گروه کلی تقسیم میشوند: نیازمندیهای وظیفه ای[۲] که عبارتست از عملکرد خام سیستم و نیازمندیهای غیر وظیفه ای[۳] که معمولا به صورت ضمنی بیان می شوند مانند کارایی[٤]، قابلیت اعتماد[٥] و هزینه است. مثلا یک سیستم هوشمند مانند روبوت را در نظر بگیرید: مشخص است که درک عملکرد چنین سیستمی به اندازه کافی سخت است (نیازمندیهای وظیفه ای) حال اگر به این نیازمندیها، نیازمندیهای غیر وظیفه ای نیز اضافه گردد همان پیچیدگی غیر قانونمند بوجود خواهد آمد.

*ناتوانی کاربر و مهندس نرم افزار در درک صحیح یکدیگر*

معمولا برای کاربران توصیف نیازهای مورد نظر خود به صورتی قابل فهم برای توسعه دهنده[٦] سیستم خیلی مشکل است. در موارد بسیاری کاربران فقط ایده های مبهمی از آنچه می خواهند سیستم نرم افزاری حاوی آن باشد، را دارند. البته بدین معنی نیست که کاربر یا توسعه دهنده سیستم(یا هر دو) مقصرند، بلکه -به طور کلی- این مشکل ناشی از عدم وجود آشنائی کافی هر دو از زمینه های کاری یکدیگر است. در واقع کاربران و توسعه دهندگان سیستم دارای دو دید مختلف نسبت به مسأله و راه

---

[۱] Requirements

[۲] Functional Requirements

[۳] Non-Functional Requirements

[٤] Efficiency

[٥] Reliability

[٦] System Developers

حل آن هستند و حتی اگر کاربر شناخت کافی از نیازهای خود داشته باشد، ما هنوز به ابزاری که بتواند این نیازها را به صورت صحیح و دقیق بیـــــــــان نماید احتیاج داریم (البته در راستای حل مشکل تعیین نیازمندیها [۱] به کمک ابزارهایی مانند موارد کاربری [۲] و کارتهای CRC [۳] 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 [۴] 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 توسعه دهندگان سیستم نسبت به مسأله و راه های حل آن بیشتر می شود و می توانند سوالهای بهتری درباره جنبه های مبهم رفتار سیستم طرح نمایند و بدین صورت دید کاملتری نسبت به سیستم پیداخواهند کرد.

## ۲) مشکل کنترل فرآیند تولید

سیستمهای نرم افزاری روز به روز پیچیده تر می شوند. امروزه ما شاهد سیستمهایی نرم افزاری بزرگ هستیم که حجم آنها با صد هزار خط یا حتی یک ملیون خط اندازه گیری می شود. چنانکه قبلا گفتیم درک کامل و دقیق چنین سیستمهای بزرگی از عهده یک نفر خارج است. حتی اگر سعی کنیم سیستم را به واحدهای [۵] 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 ایده آل باید سعی کنیم از یک تیم با حد اقل افراد استفاده نماییم ولی صرف نظر از اندازه تیم، توسعه مبتنی بر تیمها همیشه با مشکلات مهمی روبرو بوده است، زیرا افراد بیشتر، به معنی ارتباطات پیچیده تر ودر نتیجه هماهنگی بین افراد تیم مشکلتر می شود 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.

## ۳) استاندارد نبودن نرم افزار

---

[۱] Capturing Requirements

[۲] Use Cases

[۳] Class, Responsibilities, and Collaborators

[۴] System Prototypes

[۵] Modules

معمولا 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 می نماید یا در صنعت سخت افزار مثلا برای ساختن یک برد از ICهای استاندارد ساخت شرکتهای دیگر استفاده می شود و هرگز خود شرکت سازنده اقدام به ساختن همه قطعات مورد نیاز این بُرد نمی کند ولی متاسفانه چنین اتفاقی در صنعت نرم افزار به فراوانی رخ می دهد! . اینها مثالهایی از استفاده مجدد[۱] بوده است، چیزی که صنعت نرم افزار شدیداً به آن نیازمند است. البته در این راستا گامهای خوبی برداشته شده است(در قسمت مربوط به مؤلفه ها[۲] 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 نرم افزار از این نظر خیلی عقب مانده است!. لذا تولید سیستمهای نرم افزاری یک کار طاقت فرسا به شمار می آید.

## ٤) مشکل توصیف رفتار سیستمهای پیچیده

از نظر رفتار می توان سیستمها را -به طور کلی- به دو گروه تقسیم نمود:

سیستمهای پیوسته[۳]: رفتار این سیستمها بوسیله یک تابع پیوسته توصیف می گردد، که توسط آن می توان رفتار سیستم و عکس العمل آن در مقابل رویدادهای گوناگون را پیش بینی کرد.

سیستمهای گسسته[٤]: این سیستمها از تعداد متناهی حالت تشکیل شده است که این تعداد در سیستمهای گسسته پیچیده معمولا عدد بزرگی است. ویژگی بارز این سیستمها این است که نمی توان 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 می تواند غیر قطعی[٥] باشد و در بدترین شرایط این رویداد می تواند حالت سیستم را خراب نماید.

با توجه به اینکه کامپیوترهای دیجیتال سیستمهای گسسته هستند واینکه نرم افزار روی این دستگاه ها اجرا می گردد، پس ما با یک سیستم گسسته سروکار داریم.

---

[۱] Reuse

[۲] Components

[۳] Continuous Systems

[٤] Discrete Systems

[٥] Non-Deterministic

حال به این مثال توجه نمایید: فرض کنید یک هواپیما بوسیله یک کامپیوتر کنترل می شود، 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 می دهد را در نظر نگرفته اند احتمال بروز چنین حالتهایی وجود دارد.
با توجه به مثال فوق و اینکه نرم افزار یک سیستم گسسته است ، می بینیم که این خاصیت یکی از عوامل افزایش پیچیدگی سیستمهای نرم افزاری است. پس برای رسیدن به نرم افزار با کیفیت و قابلیت اطمینان بالا چه باید کرد؟ این هدف جز با انجام آزمایش های گوناگون و جامع تحقق نمی یابد.
لذا فاز تست در چرخه تولید نرم افزار از اهمیت بسزائی برخوردار است.
بنابر آنچه بیان شد، مشکل اصلی نرم افزار 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.

## 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 می دانیم کامپیوتر شخصی از مؤلفه های اصلی زیر تشکیل می شود:
واحد پردازش مرکزی(CPU)، نمایشگر، صفحه کلید و حافظه جانبی مانند دیسک سخت. هر کدام از این مؤلفه ها به نوبه خود از مؤلفه های کوچکتر تشکیل می گردد مثلا CPU از واحد کنترل، ALU، ثباتها و ... تشکیل می شود، همچنین ALU از مجموعه ای از ثباتها تشکیل شده است.
اینجاست که طبیعت سلسله مراتبی سیستمهای پیچیده را می توان مشاهده نمود.
در واقع علت درستی عملکردکامپیوتر شخصی عبارت از فعالیت گروهی و مشترک مؤلفه های اصلی آن است. این مؤلفه های جداگانه با هم یک *واحد منطقی* تشکیل می دهند.
مسلماً قدرت ما در استدلال بر نحوه عملکردکامپیوتر شخصی از طریق قدرت ما بر تجزیه این سیستم به اجزای کوچکتر که می توان هر کدام را به صورت جداگانه و مستقل از بقیه اجزاء بررسی نمود، حاصل شده است. لذا می توانیم -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 می دهند که هر سطحی روی سطح پایین تر از آن بنا شده است. به هر سطحی از سطوح تجرید که نگاه کنیم می بینیم که اولا این سطح بخودی خود قابل فهم است و ثانیا این سطح سرویس هایی به سطوح بالاتر را ارائه می دهد. ما-معمولا- 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.

**•گیاهان**

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

یک گیاه از سه قسمت اصلی تشکیل می شود: ریشه ها، ساقه ها و برگها که هر کدام نیز از قسمت های کوچکتر دیگری تشکیل می شوند. به علاوه این قسمتهای کوچکتر به قسمتهای ریزتر تقسیم می شوند 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 که از همکاری اجزای مختلف گیاه بوجود می آید و هر سطح آن دارای پیچیدگی خاص خود است.

همه اجزائی که در یک سطح واحدی از سطوح تجرید واقعند با هم دیگر بوسیله راه هایی که بخوبی تعریف شده است، ارتباط پیدا می کنند. برای مثال، بالاترین سطح تجرید را در نظر بگیرید: در این سطح -که از سه مؤلفه اساسی ذکر شده تشکیل می شود- ریشه ها وظیفه جذب کردن آب و نمک های معدنی از خاک را دارند،ریشه ها با ساقه ها ارتباط برقرار می کنند که بوسیله ساقه ها مواد خام جذب شده به برگها منتقل می گردد. برگها به نوبه خود این مواد خام را بوسیله عمل فتوسنتز به غذا تبدیل می کنند.

نکته دیگری که مشاهده می شود، وجود یک مرز مشخص بین درون و برون یک سطح تجریدی است، لذا می توان گفت که آثاری که از یک برگ بروز می نماید -که نتیجه فعالیت های 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.

---

[1] Abstraction Levels

همچنین وجود عناصر مشترک در ساختارهای مختلف یک گیاه ملاحظه می گردد. برای مثال سلول گیاهی را در نظر بگیرید: در واقع، سلول گیاهی به عنوان بلوک اصلی تشکیل دهنده همه ساختارها در گیاه عمل می کند 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 می شود. ساختمان جوامع بشری و ساختمان ماده نمونه های دیگری از سیستمهای پیچیده است که در مورد آنها نکات مشابهی می توان مطرح نمود.

## ویژگیهای سیستمهای پیچیده

با توجه به نمونه های ذکر شده پنج ویژگی مشترک بین تمامی سیستمهای پیچیده را مشاهده می نماییم، که عبارتند از:

۱. *در اغلب سیستمهای پیچیده، پیچیدگی به صورت سلسله مراتب ظاهر می شود. 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 می شود. این روند تجزیه ادامه می یابد تا به پایینترین سطح از مؤلفه های اولیه برسیم.*

این حقیقت که اغلب سیستمهای پیچیده دارای ساختار تقریبا تجزیه پذیر[2] و سلسله مراتبی[3] هستند، یک عامل اساسی است که درک ، توصیف و حتی دیدن این گونه سیستمها و اجزای آن را برای ما آسان می نماید. در واقع می توان گفت که ذهن انسان سیستمهایی را می تواند بهتر درک و تحلیل کند که دارای ساختار سلسه مراتبی باشند.

نکته دیگر این است که معماری یک سیستم پیچیده تابعی است از مؤلفه های سیستم باضافه روابط سلسله مراتبی که بر این مؤلفه ها حاکمند. اما دربارهٔ طبیعت این مؤلفه های اولیه می توان گفت:

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

لذا مؤلفه هایی که از دید یک طراح اولی هستند می توانند از دید طراح دیگر در سطح بالایی از تجرید باشند.

---

[1] Economy in Expression

[2] Nearly Decomposable

[3] Hierarchic

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

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

این حقیقت در جدا کردن خواصی که دارای نرخ تغییر زیاد بوده[4] (ساختار داخلی مؤلفه ها) از خواصی که دارای نرخ تغییر پایین[5] (ارتباط بین مؤلفه ها) است، کمک می نماید.
این تفاوت بین ارتباطات درون مؤلفه ای و ارتباطات برون مؤلفه ای یک مرز مشخص و روشن بین ساختار داخلی یک مؤلفه و ارتباط آن با بیرون معرفی می نماید که به ما کمک می کند که هر مؤلفه را به صورت تقریبا جداگانه از بقیه سیستم بررسی و تحلیل نماییم.
همانطوری که در نمونه ها ملاحظه کردیم، بسیاری از سیستمهای پیچیده بوسیله اقتصاد در بیان پیاده سازی می شوند. با توجه به این مطلب خاصیت چهارم را بیان می کنیم:

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

٥. *معمولا سیستمهای پیچیده که به صورت محکم و استوار عمل می کنند حاصل تکامل سیستمهای ساده ای هستند که به درستی عمل می کردند. سیستمهای پیچیده که از ابتدا به صورت پیچیده طراحی می شوند، هرگز کار نخواهند کرد.*

اشیائی که که در یک دوره تکامل سیستم، پیچیده به شمار می آیند 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

## پیچیدگی سازمان یافته و سازمان نایافته[1]

- **شکل اصلی سیستمهای پیچیده[2]:**

در سیستمهای پیچیده دو نوع سلسله مراتب قابل مشاهده است:

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

ساختار شئ[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 می گیرند.تجربه نشان داده است که موفقترین سیستمهای نرم افزاری پیچیده آنهایی بودند که در طراحی شان ساختار کلاس و شئ و خواص پنجگانه سیستمهای پیچیده مراعات شده است.

حال که ما نحوه طراحی سیستمهای پیچیده را می دانیم، پس چرا تولید نرم افزار هنوز با مشکلات جدی روبرو است؟

در واقع ایده پیچیدگی *سازمان یافته* (که استفاده از شئ گرائی و راهنما های مدل شئ[5] باعث سازمان یافتگی می شود)، یک ایده نسبتاً نو است و هنوز برای به کاربردن صحیح و کارای آن نیاز به تحقیقات و بررسی های بیشتر وجود دارد.

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 نرم افزاری پیچیده را شروع می کنیم، 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 می کنیم از راه فرآیند تحلیل و طراحی این پیچیدگی را سازمان دهی نماییم، و لی باید برای اینکار به چیزهای زیادی همزمان فکر کنیم زیرا-چنانچه قبلا توضیح دادیم- سیستم های کامپیوتری، سیستم های گسسته هستند بنابراین با فضای حالت بسیار بزرگ و پیچیده روبرو خواهیم شد که

---

[1] Organized and Disorganized Complexity

[2] Canonical Form of a Complex Systems

[3] Class Structure

[4] Object Structure

[5] Object Model

متاسفانه درک آن برای یک نفر کار غیر ممکنی است. تجارب روانشناسی نشان می دهد که حد اکثر تعداد *قطعه های اطلاعاتی* که مغز انسان همزمان می تواند درک نماید ۲±۷ است.

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

از طرفی پیچیدگی سیستم های نرم افزاری روز به روز در حال افزایش است، و از طرفی دیگر قدرت پردازش همزمان مغز انسان محدود است. چگونه می توان 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 می کنیم سپس هر کدام از آنها را به تنهایی حل می کنیم. 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 شئ و اصول اساسی آن را در قسمت بعدی مطرح خواهیم نمود.

[1] Divide and Rule

# ۲- معرفی اصول شئ گرائی برای مقابله با پیچیدگی

## زمینه تاریخی

با توجه به پیچیدگی روز افزون سیستمهای نرم افزاری، روش های مقابله با آن نیز به نوبه خود تکامل یافته اند. در روزهای اولیه عصر کامپیوتر نقش نرم افزار در یک سیستم کامپیوتری نقش ثانویه تلقی شده و هزینه اساسی طراحی یک سیستم کامپیوتری برای سخت افزار پرداخت می شد. چون در آن روزها قابلیت سخت افزار بسیار محدود بوده، برنامه ها ساده و کوچک بوده و زبان رایج همان زبان ماشین بود. سپس برای تسهیل در نوشتن برنامه های بزرگتر زبان اسمبلی ابداع شد. با اینکه در آن زمان سخت افزار همه منظوره وجود داشت اما نرم افزارها تک منظوره بودند بدین معنی که برای کاربرد بخصوصی طراحی شده اند[۱]. بیشتر نرم افزارها بوسیله یک نفر طراحی، پیاده سازی، تست ، نگهداری و حتی اجرا می شد لذا ظاهرا نیازی به مستند سازی نبوده است، و با توجه به طبیعت شخصی این محیط، فرآیند طراحی به صورت ضمنی در ذهن برنامه نویس صورت می گرفت 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. سخت افزار سریعتر، قابل اطمینان تر و ارزانتر شده است و این پیشرفت شگفت انگیز سخت افزار باعث اقتصادی شدن فرآیند خودکار سازی بسیاری از کاربردهای صنعتی و تجاری و این بمعنی افزایش تقاضا برای سیستمهای نرم افزای پیچیده تر است. در مقابل این فشار زبآنهای سطح بالا[۲] بعنوان ابزارهای مهم تولید سیستم های نرم افزاری وارد صحنه شده اند. در واقع این سیستمها بازدهی برنامه نویسان منفرد و تیمهای نرم افزاری را افزایش داده بودند.

در دهه ٦۰ و ۷۰ میلادی توجه پیشگامان رشته نرم افزار به "طراحی" بعنوان ابزار مهم مقابله با پیچیدگی سیستمهای نرم افزاری معطوف گشته است. لذا در این دو دهه و بعد از آن روشهای زیادی ابداع شده است که مهمترین آنها طراحی ساخت یافته است.

با توجه به تقسیمی که آقایSommerville پیشنهاد کرده است، سه طبقه بندی کلی برای روشهای طراحی نرم افزار وجود دارد.

---

[۱] Custom-Build Software

[۲] High Level Languages

### طراحی ساخت یافته(یا بالا به پایین)

در این روش هر واحد در سیستم یک گام از یک فرآیند کلی را نمایش می دهد. به عبارت دیگر سیستم نرم افزاری به صورت مجموعه ای متوالی از توابع دیده می شود. تاکید بیشتر این روش روی فرآیندهای سیستم و نه روی داده های 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 می شود. این روش مانند طراحی ساخت یافته 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 شئ گرایی

نگرش شئ گرایی 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 شئ[2] است که ضمن مدل سازی اشیاء واقعی(با توجه به میزان تجرید مورد نظر) قابلیت تلفیق ساختمان داده ها و رفتار و اعمال روی آن را در یک واحد به نام شئ داراست.

در این نگرش دنیای نرم افزار را به عنوان سیستمهایی میتوان در نظر گرفت که شامل مجموعه ای 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 نرم افزار در دنیای واقعی میسر میشود.

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

تجرید یا چکیده سازی[3]

پنهان سازی جزئیات یا محصور سازی[4]

واحد بندی[5]

سلسله مراتب[1]

---

[1] Data-Driven Design

[2] Object

[3] Abstraction

[4] Encapsulation

[5] Modularity

اینک به ارائه توضیحی درباره هرکدام از این مفاهیم می پردازیم

## تجرید(چکیده سازی، انتزاع)

تجرید عبارتست از فرآیند متمرکزشدن و تاکید کردن روی ویژگیها و رفتارهای اصلی و ذاتی یا مهم یک شئ- نسبت به یک دید مشخص- و نادیده گرفتن ویژگیها موقت یا غیر مهم آن شئ-البته نسبت به همان دید- است.

این فرآیند به ساختن یک مدل یا یک دید از آن شئ منتهی می گردد که به این مدل نیز تجرید(یا به عبارت دقیقتر مجرد) گفته می شود.

- **نقش تجرید در کنترل پیچیدگی**

اصل تجرید به مدلساز یا طراح سیستم کمک می کند 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 بی شماری 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. وبدین ترتیب می توان 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 می آید.

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 شئ انواع گوناگونی از تجرید وجود دارد. بعنوان مثال وقتیکه می خواهیم 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 شئ واحدی که همان انسان است. در واقع هر متخصص از زاویه تخصص خود به بدن انسان می نگرد لذا هر کدام، سایر جزئیات دستگاه بدن بعنوان یک مکانیزم همه جانبه را نادیده می گیرد، وفقط روی جنبه مورد نظر خود متمرکز می شود.

---

[1] Hierarchy

§ تجرید همواره با نمود خارجی[1] از یک شئ سروکار دارد. به عبارت دیگر تجرید روی اینکه یک شئ چیست و چه کار میکند تاکید مینماید و جزئیات مربوط به نحوه پیاده سازی آن شئ 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 قطعه های 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 شئ فایل را در نظر بگیرید، یک فایل دارای اسم، اندازه و محتویات است. خود این ویژگیهای ثابتند و لی مقادیر آنها متغیرند زیرا اسم، اندازه و محتویات در طول حیات فایل تغییر می یابد.

§انواع تجرید بشرح زیر می باشد:

۱. تجرید موجودیت[4]:یک مدل مفید از یک موجودیت واقعی(شئ)-که در قلمرو مسأله مطرح است- را نمایش می دهد.

۲. تجرید رفتار[5]:یک مجموعه عمومی از اعمال[6] که عملکرد یکسانی دارند را نمایش می دهد. مانند عمل اضافه کردن به لیست.

۳. تجرید مجازی[7]: عبارتست از مجموعه ای 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 می گیرند. 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 می نماید.

---

[1] Outside View

[2] Static

[3] Dynamic

[4] Entity Abstraction

[5] Action Abstraction or Procedural Abstraction

[6] Operations

[7] Virtual Abstraction

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(مثلا این شکل  برای نمایش انسان)
- نمایش گرافیکی از رفتار یک سیستم.

## پنهان سازی جزئیات[1] (محصور سازی)

پنهان سازی جزئیات عبارت از عدم پذیرش تاثیرات ناخواسته و یا کنترل نشده و محدود کردن طرق دسترسی و استفاده از یک شئ است.

این فرآیند با جداسازی ویژگیهای اصلی یک شئ -که برای اشیاء دیگر قابل دسترسی است- از جزئیات پیاده سازی این شئ -که می بایستی از بقیه اشیاء مخفی شود- انجام می پذیرد. بعبارت ساده تر پنهان سازی جزئیات همان فرآیند مخفی سازی جزئیات پیاده سازی یک شئ است.

برای درک بهتر این مطلب از مفاهیم OOP کمک می گیریم:

می دانیم که مفهوم تجرید با ساختار Class در اغلب زبانهای شئ گرا پیاده سازی می شود که در این زبآنها هر کلاس باید دو قسمت داشته باشد:

واسط[2]: در واقع همان تجرید یک شئ بوده زیرا در این قسمت توصیف رفتار مشترک بین همه نمونه های کلاس بیان می شود. سرویس های یک کلاس تنها بوسیله واسط آن دسترسی پذیرند.

پیاده سازی: در این قسمت رفتار مورد نظر کلاس پیاده سازی می شود.

### ♦ 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 شئ محدود می شود و نگهداری برنامه آسانتر می شود.

---

[1] Encapsulation

[2] Interface

همچنین با استفاده ازپنهان سازی جزئیات، تغییرات در پیاده سازی یک شئ-تا وقتی که واسط آنرا تغییر نکرده باشد-می تواند به آسانی و باقابلیت اعتماد بالا انجام شود و بدون اثرگذاری روی اشیاء استفاده کننده از سرویس های این شئ است.

نکات زیر درباره پنهان سازی مطرح است:

قاعده محصور سازی ایجاب می کند که ارتباط بین اشیاء تنها از راه واسط ها باشد. به عبارت دیگر هیچ قسمتی از یک سیستم نرم افزاری نباید به جزئیات داخلی یک قسمت دیگر و ابسته باشد.

گاهی بین دو مفهوم تجرید و محصور سازی خلط می شود، برای بیان رابطه این دو با یکدیگر باید گفت که می توان از تجرید به عنوان مکانیزم تعیین جزئیاتی که باید پنهان شود، استفاده نمود اما خود عمل پنهان سازی جزئیات(که شامل طراحی واسط و پنهان سازی پیاده سازی)را محصور سازی گویند.

محصور سازی یک مفهوم نسبی است: آن چیزی که در یک سطح از تجرید مخفی است می تواند نمود خارجی یک سطح دیگری باشد.

## واحد بندی[1]

قبل از بیان مفهوم واحد بندی مفاهیم زیرا توضیح می دهیم

- **واحدها**

واحدها عبارت از واحد تشکیل دهنده ساختار فیزیکی سیستم نرم افزاری است. به بیان دیگر در یک سیستم نرم افزاری طراحی شده به روش OO ، کلاس ها و شئ ها ساختار منطقی سیستم را تشکیل می دهند و با گروه بندی تجریدهای منطقا مرتبط در یک واحد، ساختار فیزیکی سیستم مشخص می گردد.

برای درک بهتر این مفهوم طراحی یک بورد کامپیوتری را در نظر بگیرید، در این فرآیند ابتدا با استفاده ازگیت های منطقی(NAND, AND, NOR,..) رفتار مورد نظر را پیاده سازی می نماییم(سطح طراحی منطقی). اما برای پیاده سازی این طرح باید از ICهای موجود که عبارت از بسته های استاندارد شده این گیتها هستند استفاده شود. در این مثال بسته های IC با واحدهای نرم افزاری متناظرند.

- **انسجام[2]**

---

[1] Modularity

[2] Cohesion

انسجام عبارتست از خاصیتی متعلق به درجه ارتباط عملکردی[1] 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 می کنند پسA یک واحد *کاملا منسجم*[2] است. به هر اندازه که عناصر واحد A وظایف گوناگونی را انجام می دهند و به هر اندازه که ارتباط این وظایف با همدیگر ضعیف باشد، به همان اندازه درجه انسجام ضعیفتر خواهد بود.

## ♦ وابستگی[3]

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 نسبتا کامل عملکرد واحد B باشد پس درجه وابستگی بین A و B بالا[4] خواهد بود. اما اگر این رابطه ضعیف است انگاه درجه وابستگی بین A و B ضعیف[5] خواهد بود.

حال می توان واحد بندی را تعریف نمود:

سیستمی را واحدبندی شده گویند که به مجموعه ای از واحدهای منسجم و معنی دار که وابستگی بین آنها حد اقل 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.

بعبارت دیگر فرض کنید مسئله P را به زیر مسئله های P1 ، P2 و P3 قابل شکستن می باشد. 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 نمی شود.

بنابراین می توان نوشت:

E(P) > E(P1) + E(P2) + E(P3), E: تلاش

---

[1] Function Relatedness

[2] Completely Cohesive

[3] Coupling

[4] A and B are Highly Coupled

[5] A and B are Loosley Coupled

نکات زیر درباره واحد بندی مطرح است:

§اگر شرایط بیان شده در تعریف واحد بندی 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 اگرn بالا رود آنگاه با واحدهای بسیار کوچک روبرو خواهیم بود که خود یک مشکل پدید می آورد. یکی از وظایف طراح اینست که متوسط n را بدست آورد(n بر اساس سرویسهای لازم در سیستم تعیین می گردد).

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 و اقع سختی این کار متناظر با سختی انتخاب مجموعه کلاس های 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 واحد بندی است. اگر این کار بدرستی انجام شود و همچنین واحدهای بدست آمده به خوبی مستند شوند آنگاه مزیت سوم واحد بندی روشن می شود زیرا تقسیم برنامه به واحدهای خوش تعریف[1] که به خوبی مستند شده باشند در آسان کردن درک برنامه و نگهداری آن نقش مهمی ایفا می نماید.

## سلسله مراتب[2]

تا به حال سه مفهوم از مفاهیم اساسی مدل شئ مطرح نموده و نقش هر کدام در کنترل پیچیدگی نرم افزار را بیان نمودیم. برای رعایت پیوستگی مطالب و روشن شدن نقش سلسله مراتب، تسلسل منطقی مکانیزمهای مدل شئ برای مقابله با پیچیدگی سیستمهای نرم افزاری را بیان می کنیم: 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 معمولا 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 واسط ها است. همچنین واحد بندی از طریق ارائه راهی بمنظور دسته بندی تجریدهای منطقا مرتبط کمک شایانی می کند. اما این به تنهایی کافی نیست و اغلب مجموعه ای از

[1] Well-Defined Modules

[2] Hierarchy

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 می یابد.

سلسله مراتب به صورت زیر تعریف می شود:

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 شئ(PART-OF)[2] می باشد.

وراثت مهمترین شکل سلسله مراتب IS-A بوده و یک عنصر اساسی در سیستمهای شئ گرا 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 دیگر وراثت 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 عمومی تر(کلاس پدر)را نمایش می دهد.

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 می توان 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 می شود.

## ♦ نقش سلسله مراتب در کنترل پیچیدگی

با سازمان دهی قطعه های اطلاعاتی منفصل(تجریدها) در سلسله مراتب های IS-A و PART-OF درک ما نسبت به سیستم بهبود می یابد. اساسا ذهن انسان به گونه ای شکل گرفته که ترجیح می دهد برای مدیریت بهتر یک مسأله آنرا در لایه های 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[5] از فعل و انفعالاتی[1] که بین آنها رخ میدهد بیان شده اند، را نمایش می دهد.

---

[1] also Kind-Of or Generalization/Specialization Realtionship

[2] also Aggregation or Wole/Part Relationship

[3] Single Inheritance

[4] Multiple Inheitance

[5] Patterns

اهمیت سلسله مراتبIS-A در این است که افزونگی[۲] موجود در سیستم را مدیریت نموده و بوسیله آن سیستمهایی با خاصیت اقتصاد در بیان را می توان پیاده سازی کرد.
یک نکته قابل توجه است که استفاده از وراثت با پنهان سازی تام، تعارض دارد زیرا مستلزم دسترسی مستقیم کلاس فرزند به بعضی از عملیات و داده های اختصاصی[۳] کلاس پدر است.

## مزایای مدل شئ و کاربردهای آن

§هدف نهایی تکنولوژی Object-Oriented انجام فرآیند تولید نرم افزار شبیه به روشی که در تولید سخت افزار استفاده می گردد، یعنی از طریق گروه بندی Object ها در لایه های مختلفی از تجرید است.

§تکنیکهای سنتی موجود توان پاسخگویی به پیاده سازی گونه های مختلف از سیستم های پیچیده امروزی را ندارند. این سیستمها نیازمند ساختاری برای مدیریت مدلهای پیچیده اند. مکانیزمهای موجود ر تکنولوژی شئ گرایی پتانسیل برخورد با گستردگی و پیچیدگی سیستمهای تجارتی امروز را دارند.

§از طریق کاربرد اشیاء به بعنوان و احد 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 می دهد. 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 تولید نرم افزار را نیز کاهش خواهد یافت.

§محصور سازی و جداسازی لایه های معماری نرم افزار باعث ایجاد مقیاس پذیری و قابلیت توسعه تدریجی یک سیستم می شود.

§قابلیت انعطاف برای اجرای اشیاء بصورت توزیع شده در زمانیکه یک سیستم بزرگ از مجموعه ای از اشیاء تعریف شده است بوجود خواهد آمد. اشیاء نیازمند به قدرت پردازش بالا در سایتهای قویتر قرار می گیرند و بر اساس نیازمندیهای جغرافیایی این توزیع انجام می گیرد.

§قابلیت استفاده مجدد: امروزه تمایل زیادی به سمت ساخت عناصر نرم افزاری استاندارد که می توان از آنها-بدون تغییر- در تولید نرم افزارهای مختلف استفاده نمود(معادل نرم افزاری IC). ساخت مؤلفه ها[۴] که بر دیدگاه شئ گرایی مبتنی است می تواند زمینه ساز چنین تحولی باشد.

---

[۱] Interactions

[۲] Redudancy

[۳] Private Members

[۴] Components

# ۳- آشنائی با مفاهیم اولیه شئ گرائی

## مفاهیم اساسی

### § شئ[1]

کلمه Object در فارسی تحت عنوان مفهوم، ایده عینیت، هدف و شئ ترجمه شده است که متاسفانه هیچیک از این عناوین بیانگر معنی دقیقی از کلمه Object نیستند. اما شاید نزدیکترین کلمه معادل فارسی همان "شئ" یا "چیز" باشد و البته مقصود ما از شئ یک مفهوم کلی است بگونه ای که دارای هویت بوده و قادر به بروز رفتار و ثبت حالات(وضعیت) خود باشد. مثلا یک ماشین نمونه کاملی از یک شئ است. انسان، درخت، موجودات زنده و حتی عناصر بی جان نیز گونه ای 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[عکس العمل] )و نیز حالاتی در آنها وجود دارد که گویا یکی از وضعیتهای کلی است که آن شئ می تواند در آن بسر ببرد. بسته به فضا و محیطی که به آن می اندیشیم و سطح تجرید مورد علاقه می توان اشیاء را تمییز داده و بگونه ای کاملاً روشن تبیین نمود.

از نظر تکنیک شئ گرائی، یک شئ دارای سه مشخصه ذاتی زیر می باشد:

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

در زبانهای OOP هویت یک شئ با یک اسم منحصر بفرد(یا یک Handle) نمایش داده می شود.

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

۳- رفتار[4]: رفتار؛ چگونگی عمل و عکس العمل یک شئ در قالب تغییر حالت در مقابل دریافت و یا ارسال پیام است، را نشان می دهد.

---

[1] Object

[2] Identity

[3] State

[4] Behaviour

مثالهایی از اشیاء عبارتند از:

§موجودیتهای خارجی(External Entities)

§اسباب(Things)

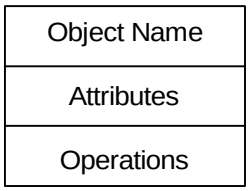
§نقشها(Roles): مدیر، کارمند، معمار نرم افزار

§واحدهای سازمانی(Organization Units)

§مکانهای فیزیکی

§ساختارها

در زبانهای OO یک شئ به صورت زیر نمایش داده می شود:




شکل ٤-١ نمایش شئ در زبانهای OO

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 می توان گفت که:

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

اگر به جهان بنگریم اشیاء زیادی پیدا می کنیم که عبارتند از نمونه هائی از یک مفهوم کلی تر که آنرا کلاس(Class) می نامیم. برای مثال ماشین پیکان که رنگ زرد دارد و متعلق به آقای فلان یک شئ است. این شئ نمونه ای از یک مفهوم کلیتر که همان ماشین می باشد.

### § کلاس[1]

مجموعه ای از اشیاء که دارای ساختار و رفتار مشترک باشند را یک کلاس می نامیم. مزیت گروه بندی اشیاء در مفهوم کلاس مدیریت بهتر و قابلیت استفاده مجدد است. در واقع یک الگوی کلی داریم که بر حسب نیاز اشیائی از آن برداشت می نماییم. چنانکه می دانیم Abstract Data Type(ADT) امکانی را فراهم می آورد که باعث شناسائی(معرفی) اشیاء از طریق بیان ساختار و رفتار آنها بدون نیاز به پیاده سازی آن ساختار یا رفتار می شود.

در زبانهای OO کلاس از دو قسمت تشکیل شده:

Class Decleartion **à** ADT
Class Body **à** Behaviour Implementation

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 OOP هر شئ کلاس خود را می شناسد.

### § نمونه[2]

یک نمونه به یک مورد مشخص از یک کلاس اشاره می نماید. عمل تعریف یک شئ در OOP را Instantation گویند. برای مثال: Studenet s1, s2;

### § ارتباط بین اشیاء

مکانیزم ارتباط بین اشیاء و بهرمندی از سرویس ها(عملیات) آنها از طریق تبادل پیام[3] صورت می گیرد.

در شئ گرائی اصل Encapsulation از طریق محدود کردن راه استفاده اشیاء از یکدیگر در مکانیزم تبادل پیام اِعمال می گردد.

---

[1] Class

[2] Instance

[3] Message Passing

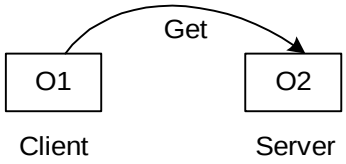
برای تصور بهتر تبادل پیام به مفهوم Client/Server بعنوان مدلی مناسب برای نشان دادن ارتباط توجه کنید.

شئ ارسال کننده پیام تقاضادهنده[۱] حساب شده و شئ دریافت کننده پیام سرویس دهنده محسوب می گردد[۲].

مثال: شئ O1 پیام Get به شئ O2 می فرستد(شکل ٤-۲)

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

O2.Get( param_list );




شکل ٤-۲ رابطه C/S

**§نقش ها**[۳]: برای تعیین رفتار یک شئ می توان از مسئولیت (نقش) آن شئ استفاده نمود.

**§واسطها**[٤]: واسطها نحوه استفاده از یک کلاس بدون نیاز به شناسائی جزئیات پیاده سازی آن را به ما نشان می دهند. در C++ خود تعریف کلاس(که معمولا در Header File قرار می گیرد) واسط حساب می شود.

سه نوع واسط وجود دارد:

۱- عمومی[٥]: برای همه استفاده کنندگان قابل دسترسی است .

۲- اختصاصی[٦] : تنها برای همان کلاس و دوستان آن کلاس قابل دسترسی است .

۳- حفاظت شده[۷]: تنها برای خود کلاس، دوستان کلاس و زیر کلاسها[۸] قابل دسترسی است .

---

[۱] Client

[۲] Server

[۳] Roles

[٤] Interfaces

[٥] Public

[٦] Private

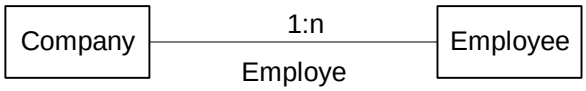
[۷] Protected

[۸] Subclasses

## رابطه بین کلاسها

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

۱- **رابطه انجمنی**[۱]: نوعی وابستگی معنایی[۲] بین کلاسهای متفاوت که با حذف وابستگی عملاً هیچ




ارتباط بین دو کلاس وجود نخواهد داشت. مثال:

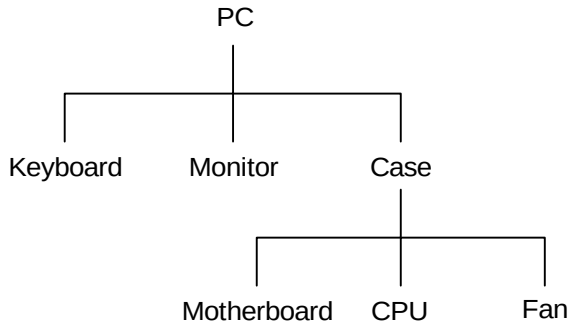
شکل ٤-۳

در این رابطه دو مسئله اهمیت دارد:

۱- نوع وابستگی

۲- درجه وابستگی

۲- **رابطه تجمعی**[۳]: زمانی که یک شئ از تعدادی اشیاء دیگر تشکیل می گردد، این رابطه را تجمعی(جزئی-از) گویند.




شکل ٤-٤

[۱] Association

[۲] Semantic Relationship

[۳] Aggregation

۳- **رابطه وراثت**[۱] : وراثت عبارت از رابطه بین چند کلاس که در آن یک کلاس در ساختار و رفتار یا هر دو با یک کلاس (وراثت یگانه[۲]) یا چند کلاس (وراثت چندگانه[۳]) دیگر شرکت دارد.

بعبارت دیگر وراثت عبارتست از سلسله مراتبی از تجریدها[٤] که در آن کلاس فرزند[٥] خصوصیات کلاس پدر[٦] را به ارث می برد.

در واقع کلاس فرزند یک تخصیص[۷] از کلاس پدر را نمایش داده و همزمان کلاس پدر یک تعمیم[۸] از کلاس فرزند به حساب می آید.

مثال:

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

مثال وراثت یگانه:


[۱] Inheritance or Generalization/Specialization or IS-A Relationship

[۲] Single Inheritance

[۳] Multiple Inheritance

[٤] Abstract Hierarchy

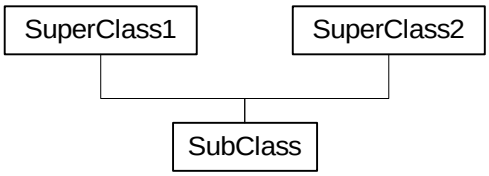
[٥] Subclass

[٦] Superclass

[۷] Specialization

[۸] Generalization

مثال وراثت چندگانه:




## روش پیدا کردن کلاسها

به صورت کلی، روشهای تجزیه و تحلیل شئ گرا به دو دسته تقسیم می شوند:

۱- مبتنی بر داده[۱]

۲- مبتنی بر وظیفه[۲]

تفاوت اصلی این دو روش در نقطه شروع شناسائی کلاسها می باشد.

در روش اول مبنای کار، شناسائی کلاسها بر اساس ساختمان داده های مورد نیاز مسئله می باشد. در حالیکه در روش دوم کلاسها بر اساس وظیفه ها شناسائی می شوند. بعنوان نمونه از روش دوم روش(Class, Responsibilties, & Collaborators or CRC) را توضیح می دهیم.

## روش CRC

در این روش هر کلاس در مسئله متناظر با یک کارت۲×۳ اینچ بوده که دارای ۳ فیلد نام کلاس، وظایف کلاس[۳] و همکاران کلاس[٤] در انجام وظایف خود می باشد.

---

[۱] Data-Driven

[۲] Responsibility-Driven

[۳] Class Responsibities

[٤] Class Collaborators

| Class Name | |
|---|---|
| Responsibilities | Collaborators |
| | |

شکل ٤-٥ نمونه ای از کارت CRC

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

در حقیقت، در بیشتر مسئله ها یک سری کلاسهای مشهود وواضح وجود داشته که شناسائی آنها می تواند مبنای شناسائی کلاسهای دیگر باشد.

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

مثال۱: وقتی یک مشتری به بانک مراجعه می کند برای انجام امور بانکی احتیاج به همکاری اشیاء دیگر دارد. مثلا حساب باز می کند، پول برداشت می کند و ... با توجه به این وظایف، اشیاء خود را نشان می دهند.

مثال۲: مسئله معروف تولید کننده[1] و مصرف کننده را در نظر بگیرید: در این مسئله تولید کننده یک Item تولید می نماید که مصرف کننده آنرا مصرف می کند. برای هماهنگی این دو کلاس به یک بافر میانی نیاز داریم(که خود یک کلاس است). کارتهای این مسئله به صورت زیرند.

[1] Produce & Consumer Problem

| Consumer | |
|---|---|
| Responsibilities | Collaborators |
| GetItem()<br>ConsumeItem() | Buffer |

| Producer | |
|---|---|
| Responsibilities | Collaborators |
| ProduceItem()<br>PutItem() | Buffer |

| Buffer | |
|---|---|
| Responsibilities | Collaborators |
| SeizeBuffer()<br>ReleaseBuffer() | |

شکل ٤-٦ 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 Itemدر بافر نیاز به همکار دارد که خود بافر است. بافر طوری است که یک عنصر بیشتر در آن جا می گیرد و غیر فعال[1] نیز هست.

[1] می توان اشیاء را به دو قسمت تقسیم نمود: فعال(Active) و غیر فعال(Passive). شئ فعال روی اشیاء دیگر تاثیر می گذارد(با فرستادن پیام) و شئ غیر فعال تاثیر می پذیرد. اگر شئ فعال و غیر فعال باشد آنگاه Agent نامیده می گردد.

# ٤- معماری عناصر و مقوله بندی اشیاء[1]

## مقدمه

معماری عبارتست از تعیین ساختار کلی سیستم و روشهایی که این ساختار را قادر به تامین کلیه ویژگیهای کلیدی سیستم می سازد.

معماری نرم افزار را می توان با استفاده از مثالی از معماری یک ساختمان توصیف کرد. یک ساختمان معمولاً از دید مشتری دارای یک بعد می باشد. اما معمار ساختمان را از یک یا چند زاویه مختلف برای مشتری تصویر می کند. از طرفی افراد زیادی روی قسمتهای مختلف یک ساختمان کار می کنند تا آنرا تکمیل نمایند. از جمله این افراد می توان به نجار، برق کار، لوله کش و بنا اشاره کرد. هر یک از این کارکنان به یک نقشه تخصصی ساختمان نیاز دارند که با توجه به آن کار خود را انجام دهند. در ضمن این نقشه ها هم باید یا یکدیگر سازگاری و همخوانی داشته باشند تا هر یک از افراد بتوانند بدرستی کار خود را انجام دهند بنابراین معمار تمامی این نقشه ها را با توجه به اصول ساخت و ساز تهیه نموده و سپس مهندس ساختمان با توجه به مصالح مورد استفاده، نوع کار و زمان تحویل کار جزئیات این نقشه ها را مشخص می نماید و هر نقشه را در اختیار کارکنان همان قسمت می گذارد. در موقع ساخت ساختمان خیلی از کارکنان برای گرفتن یک دید کلی از کار به نقشه های معماری رجوع می نمایند، اما برای انجام جزئیات کار از نقشه هایی که مهندس ساختمان تعیین کرده استفاده می کنند.

یک سیستم نرم افزاری نیز همانند یک ساختمان یک موجودیت واحد است، اما معمار نرم افزار ترجیح می دهد این سیستم را از زاویه های گوناگون بررسی کرده و نمایش دهد. نگرش به سیستم از جهات مختلف به درک بهتر سیستم کمک می نماید. مجموعه این نگرش های گوناگون معماری سیستم را تشکیل می دهد.

## الگوهای[2] معماری

یکی از مهمترین مباحثی که در معماری وجود دارد، مبحث استفاده از الگوها می باشد. در فرآیند معماری این سوال همواره پیش می آید که چگونه می توان یک معماری خوب را تولید نمود؟ بواسطه معماریهایی که برای سیستمهای مختلف انجام گرفته که می توان آنها را نوعی تمرین دانست یکسری قوانین و الگوها جمع آوری شده است که در معماری به کار می روند. هر الگوی معماری در حقیقت برای پاسخ به یک مشکل ایجاد شده است وبنابراین می توان در سیستمهایی که خواص

---

[1] Software Architecture and Stereotyping

[2] Patterns

مفهومی مشترک داشته و همان مسئله را دارند مورد استفاده قرار بگیرد. استفاده از الگوهای معماری معتبر می تواند باعث توسعه قابلیت استفاده مجدد در سطح طراحی گردد و قابلیت اطمینان نیز را بالا ببرد.

برخی از الگوهای معماری مطرح عبارتند از لایه بندی[۱] و معماری دلال[۲] بوده که در زیر مروری بر آنها خواهیم داشت.

## الگوی لایه بندی

در این الگو سیستم به صورت تعدادی از لایه ها سازمان دهی می گردد. هر لایه مجموعه ای از سرویسهای معین که بوسیله واسط این لایه برای لایه های بالاتر قابل استفاده بوده فراهم می آورد.

هر لایه از تعدادی مولفه تشکیل شده که همکاری گروههی این مولفه ها بوجود آورنده رفتار لایه می باشد. این الگو وابستگی ها را کاهش می دهد بطوریکه لایه های پایین تر از جزئیات و واسطهای لایه های بالاتر اطلاعی ندارند.همچنین این الگو می تواند به شناسائی بخشهای قابل استفاده مجدد و تصمیم گیری در مورد مولفه های قابل خریداری و مولفه های قابل ساخت کمک نماید. می توان به معماریهای متمرکز، معماری Cllient/Server و معماری 3-Tier بعنوان نمونه های از لایه بندی در دنیای نرم افزار اشاره نمود. یک برنامه کاربردی از نظر منطقی به سه قسمت کلی (واسط کاربر، منطق کاری[۳] یا منطق برنامه و سرویسهای داده ای) تقسیم می شود. تفاوت این معماریها در نحوه ارتباط این قسمتها با یکدیگر می باشند. برای مثال در معماری متمرکز این سه قسمت با یکدیگر شدیداً آمیخته شده که باعث ساختن سیستمهای غیر انعطاف پذیر و غیر قابل نگهداری می باشد. در حالیکه در معماری دوم دو لایه داریم: لایه Client شامل واسط کاربر و منطق کاری بوده و لایه Server سرویسهای داده ای را بعهده گرفته است و بالاخره در معماری سوم این لایه ها از یکدیگر کاملاً جدا هستند که این ویژگی قابلیت استفاده مجدد در این سیستمها را بالا می برد. بعنوان یک مثال اخیر می توان از متدولوژی شئ گرای Select Perspective نام برد. این متدولوژی از معماری 3-Tier بهره برده و معماری خود را به صورت زیر مطرح می نماید:

معماری Perspective بر ایده سرویس مبتنی بوده که عبارتست از مجموعه ای منطقا مرتبط از وظایف که بوسیله یک واسط قابل دسترسی می باشند.

---

[۱] Layers Architecture

[۲] Broker Architecture

[۳] Business Logic

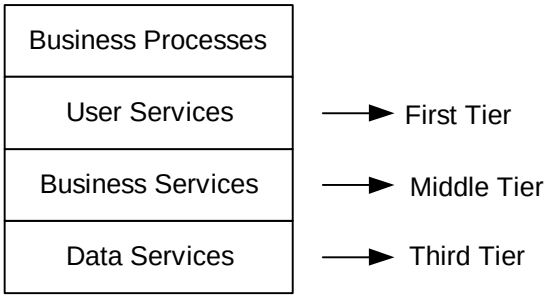
در معماری Perspective (شکل ٤-١) چهار لایه وجود دارد به شرح زیر می باشد:

١- فرآیندهای کاری[١]: فرآیند کاری عبارتست از مجموعه از فعالیت ها که یک یا چند ورودی دریافت کرده و خروجی ارزشمندی برای کاربر ایجاد می کند.

٢- سرویسهای کاربر[٢]: وظیفه این لایه فراهم نمودن سرویسهای ارتباط با کاربر(واسط کاربر)

٣- سرویسهای کاری[٣]: سرویسهای عمومی که منطق کاری سیستم را تشکیل می دهند. این سرویسها داده ها را از سرئیسهای کاربر و سرویسهای داده ای را گرفته و مورد پردازش قرارداده و اطلاعات تولید می نماید.

٤- سرویسهای داده ای: این سرویس داده های مورد نیاز فرآیندهای کاری متفاوت را فراهم می نمایند. سرویسهای داده ای عمل پردازش د اده ها به صورت مستقل از نحوه ذخیره سازی آنها انجام می دهند.




شکل ٤-١ معماری Perspective

## الگوی Broker

در این الگو که بیشتر مناسب سیستمهای OO بوده مجموعه ای از اشیاء توزیع شده روی ماشینهای یا شبکه های مختلف با یکدیگر از راه یک دلال به صورت شفاف[٤] ارتباط برقرار می نمایند. مقصود از شفاف بودن این است که صدا کننده لازم نیست بداند شئ صدا زده شده روی یک ماشین دیگر یا شبکه دیگر است یا خیر.

---

[١] Business Process

[٢] User Services

[٣] Business Services

[٤] Transparent

## مقوله بندی[1]

کلاسها همان واحدهای اساسی که در مولفه ها[2] گروه بندی می شوند. مولفه ها بعنوان فراهم کننده سرویس عمل می نمایند. در یک سیستم نرم افزاری 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 می توان کامپیوتر را یک شئ فرض نمود و لی در یک سطح پایینتر خود کامپیوتر از تعدادی اشیاء تشکیل شده است. مقوله بندی اشیاء به ما کمک می کند که سطوح انتزاعی را بشناسیم و بدین صورت تمام افراد تیم می توانند در یک سطح واحدی از انتزاع فعالیت نمایند. بعبارت دیگر Stereotyping روشی برای میزان کردن سطح انتزاع است[4]

روشهای مختلفی برای طبقه بندی(مقوله بندی) اشیاء وجود داشته که یکی از آنها -که مورد نظر Perspective است- بشرح زیر می باشد:

می توان اشیاء بر اساس سرویسهایی راکه ارائه می دهند به سه نوع زیر طبقه بندی نمود:

§**اشیاء واسط/کاربری**[5]: 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 می باشد.

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

§ **اشیاء کاری** [6]: سرویسهای مورد احتیاج نیازمندیهای کاری[7] را فراهم می نمایند. بعبارت ساده تر منطق کاری(منطق برنامه) از همکاری این اشیاء با یکدیگر بوجود می آید. این اشیاء با بکاربردن قواعد کاری[8] و استفاده از سرویسهای داده ای، داده ها را به اطلاعات تبدیل می نمایند.

---

[1] Stereotyping

[2] Components

[3] Abstraction

[4] Abstract Tuner

[5] User/Interface Objects

[6] Business Objects

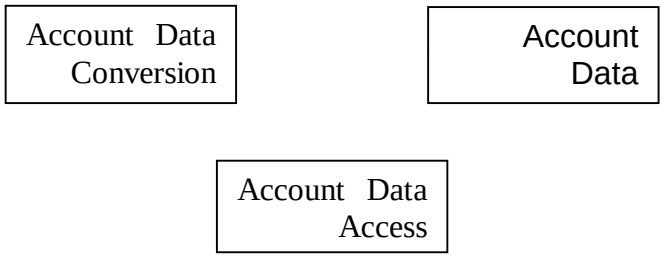
[7] Business Requirement

[8] Business Rules

مثال : در سیستم هتل داری ، اشیائی مثل رزرو کردن(Reservation) و (Customer Order) و ... مشاهده می شوند که سرویسهای مورد انتظار از سیستم را فراهم می نمایند[1].
اگر شئ فقط در همان برنامه کاربردی[2] استفاده شود شئ محلی[۳] و اگر خارج از آن قابل استفاده باشد شئ عمومی[٤] گویند .

§ **اشیاء داده ای**[٥] : برای فراهم نمودن سرویسهای داده ای این اشیاء با یکدیگر همکاری کرده و با استفاده از سرویسهای سیستم پایگاه داده ها[6] (ذخیره/ بازیابی /بهنگام سازی)، داده های مورد نیاز لایه های بالاتر را تامین می کنند. این لایه داده ها را به صورت اشیاء پایا[۷] ذخیره و بازیابی می نماید اگر DBMS مورد استفاده از نوع OO باشد مشکلی پدید نمی آید اما اگر از مدل رابطه ای پیروی می کند آنگاه این لایه عمل نگاشت اشیاءپایا به مدل رابطه ای را انجام می دهند. در واقع هدف از وجود این لایه حفاظت لایه سرویسهای کاری از تغییرات در تکنولوژی DBMS ها می باشد.

مثال:




شکل ۲-۵

یک طبقه بندی دیگر که خانم Wirfs-Brock در روش RDD[۸] که یک روش وظیفه گرا بوده، ارائه نموده بشرح ذیل می باشد:

۱. اشیاء کنترل کننده[۹]: وظیفه اساسی این اشیاء فراخوانی و کنترل اشیاء دیگر.

---

[۱] System Functionality

[۲] Application

[۳] Local Object

[٤] Corporate Object

[٥] Data Objects

[٦] Database Management System(DBMS)

[۷] Persistent Objects

[۸] Respnsibility-Driven Design

[۹] Controller Objects

۲. اشیاء هماهنگ کننده[۱]: باعث آغاز یک فعالیت شده و یا باعث هماهنگی بین اشیاء Client و اشیاء Server می شوند.

۳. اشیاء واسط[۲]: این اشیاء ارتباط بین سیستم و محیط خارج از خود را برقرار می نمایند.

٤. اشیاء فراهم کننده سرویس[۳]: اشیائی هستند که سرویسهای بخصوصی فراهم می نمایند مانند Buffer.

۵. اشیاء نگهدارنده اطلاعات[٤]: شبیه اشیاء داده ای در تقسم قبلی می باشند.

٦. اشیاء ساختار[۵]: برای نگهداری ارتباط بین اشیاء استفاده می شود.

## کارتهای CRC و مقوله بندی

می توان از کارتهای CRC برای تشخیص طبقه بندی(مقوله بندی)اشیاء استفاده نمود. به مثال زیر توجه کنید:

مثال: در یک سیستم بانکی مشتری می خواهد مبلغی از حساب خود برداشت نماید. می توان این مبلغ را شئ تصور نمود. در واقع عمل برداشت مبلغ تراکنش[٦] حساب می شود. در کارتهای CRC می توان از مفهوم وراثت بهره برد. برای مثال یک تراکنش مالی(Financial Transaction) 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 می نماید. در شکل۵-۳ کارت CRC این مثال نمایش شده است.

---

[۱] Coordinators Objects

[۲] Interface Objects

[۳] Services Provider Objects

[٤] Information Holder Objects

[۵] Structure Objects

[٦] Transaction

| Withdrawal Transaction | |
|---|---|
| Supercalss:Financial Transaction | |
| Responsibilities | Collaborators |
| Knows account<br>Knows amount<br>Perfoms Withdraw<br>Logs Transaction<br>Initiates Dispensing cash | Cash Dispensy |

شکل ۵-۳

در آغاز تراکنش Amount را از Account کم می کند سپس در فایل Log می نویسد که چه کسی در چه زمانی چه مبلغی از چه حسابی برداشته است و بالآخره به دستگاه پیغام (Initiates Dispensing Cash) می دهد که پول را تحویل دهید.

در پشت کارت CRC اطلاعات طبقه بندی را یاداشت می نماییم(شکل ۵-٤)

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

شکل ۵-٤ پشت کارت CRC، Withdraw Transaction که در آن طبقه بندی این کلاس دیده می شود.

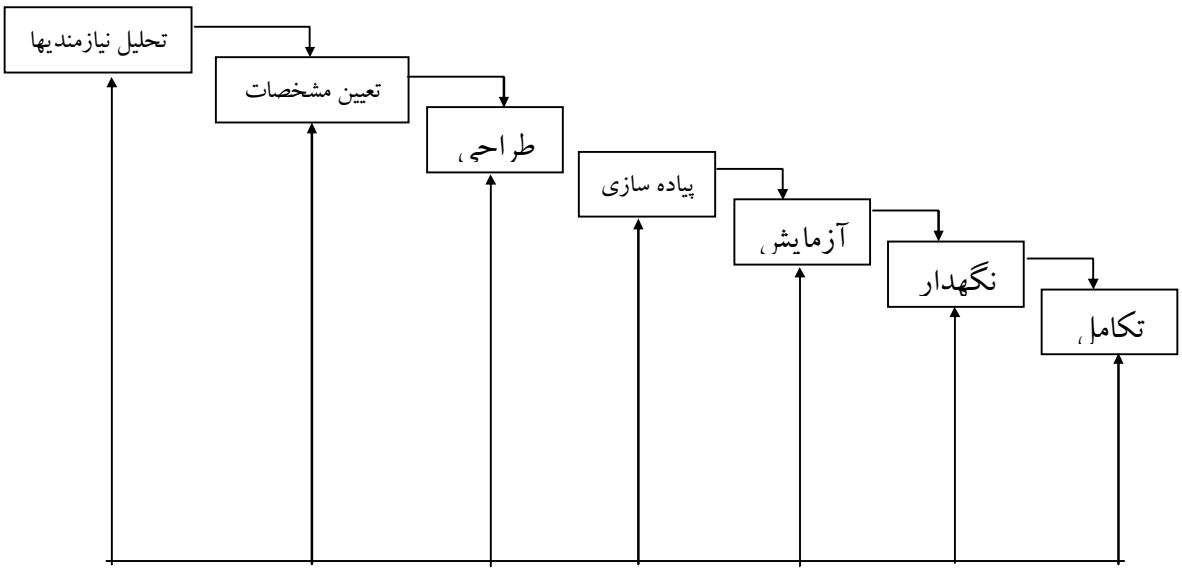
این اطلاعات شامل هدف (وظیفه اصلی) و طبقه بندی شئ می باشد.

# ۵- فرآیند تولید نرم افزار با تکیه بر دیدگاه شئ گرا

## مقدمه

فرآیند تولید سنتی(آبشاری[۱]) شامل مراحل تحلیل نیازمندیها، تعیین مشخصات، طراحی، پیاده سازی، آزمایش، نگهداری و تکامل[۲] 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 می شوند؛ برای کشف یک خطا در پیاده سازی که ریشه آن اشتباهی در طراحی یا گردآوری نیازمندیها بوده، باید همه مستندات طراحی بررسی شوند (که حجم آن در پروژه های 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 پروژه هایی که در آن خطاهای طراحی دیر کشف می شود، پی می بریم!

بطور خلاصه می توان گفت که در این روش با گذشت زمان هزینه ریسک بالاتر می رود.




شکل ۵-۱ مراحل تولید در روش آبشاری

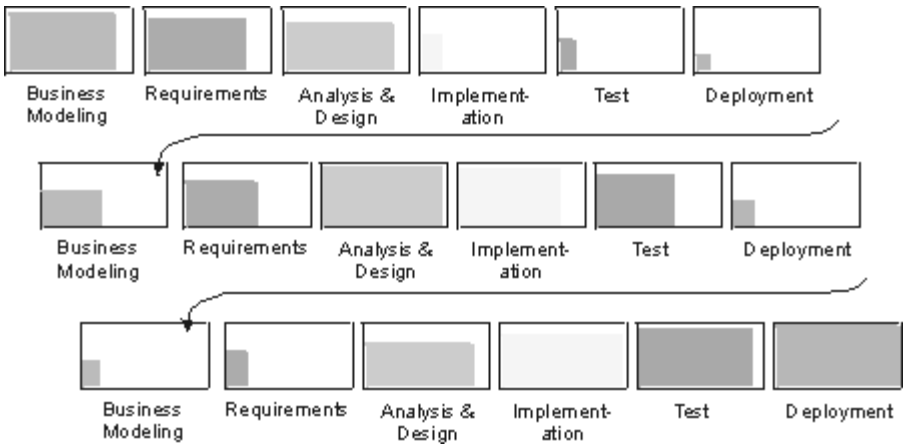
[۱] Waterfall Approach

[۲] Integration

در واقع روش آبشاری برای پروژه های کوچک(کوتاه مدت) یا پروژه هایی که بیشترجزئیات طراحی آن به خوبی شناخته شده مناسب است اما برای پروژه های بزرگ یا پروژه های جدید و غیر سنتی(مانند بودن هزینه ، برگشت به عقب و وقت گیر بودن مشکلات اصلی این مدل است .

در واقع روش آبشاری برای پروژه های کوچک(کوتاه مدت) یا پروژه هایی که بیشترجزئیات طراحی آن به خوبی شناخته شده مناسب است اما برای پروژه های بزرگ یا پروژه های جدید و غیر سنتی(مانند سیستمهای هوشمند و سیستمهای اطلاعاتی بزرگ) این روش جوابگو نیست.

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 پروژه ها استفاده نماییم. بدین صورت ما مقدار کمی از نیازمندیها را مشخص نموده سپس در مورد همین مقدار، مراحل مدیریت ریسک، تحلیل، طراحی، پیاده سازی و آزمایش را انجام می دهیم. 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 ۵-۲ )




شکل ۵-۲ روش تکرار و توسعه تدریجی

چنانکه می بینیم در این روش ریسک ها به صورت زود هنگام تشخیص می شوند و هرگاه که ممکن است تلاش می کنیم با آنها مقابله نماییم.

## § ویژگی های روش تکرار و توسعه تدریجی

۱. تشخیص زود هنگام خطاهای که در درک مسأله، تحلیل یا طراحی رخ می دهند.
۲. تشخیص زود هنگام ناسازگاری های موجود بین تحلیل نیازمندیها، طراحی و پیاده سازی.

۳. با توجه به شیوه عمل تکراری کاربر می تواند همیشه بر روند پیشرفت پروژه نظارت داشته باشد. همچنین می تواند نظر خود را ارائه نماید که در تشخیص بهتر نیازمندیهای مسأله کمک می نماید.
٤. بوسیله این روش تیم توسعه سیستم می تواند روی قسمتهای مهمتر پروژه متمرکز شود و از پرداختن به قسمتهای کم اهمیت تر پرهیز نماید.
۵. آزمایش تکراری و مستمر امکان تشخیص بهتر روند پیشرفت پروژه را به ما می دهد.
٦. بارکاری[1] تیم ها-بخصوص آزمایش کنندگان-روی چرخه تولید پروژه به صورت متوازن توزیع می شود.

در فرآیندهای تولید نرم افزار بر مبنای روش شئ گرا غیر از بکاربردن مفهوم توسعه تدریجی از روش تحلیل موارد کاربری و معماری نرم افزار استفاده می گردد.

در واقع روش توسعه تدریجی با یک بعدی بودن فرآیند تولید نرم افزار(مثلاً روش آبشاری) سازگاری نداشته و یک روش دو بعدی می طلبد. برای بیان این مفهوم یک متدولوژی شئ گرا به نام Unified Software Development Process (USDP) را معرفی کرده و در فصل بعدی به بیان چرخه حیات نرم افزار از دیدگاه این متدولوژی خواهیم پرداخت.

## Unified Software Development Process

USDP یک فرآیند تولید مهندسی نرم افزار است که یک روش سیستماتیک و منظم برای ترتیب انجام فعالیتها در یک پروژه نرم افزاری را پیشنهاد می نماید.

هدف این فرآیند، تولید نرم افزارهایی با کیفیت عالی که علاوه بر برآوردن نیازهای کاربران خود، در زمان و با هزینه پیش بینی شده تولید شوند.

USDP از مدل شئ گرایی حمایت نموده و از روش های مدرن توسعه نرم افزار مانند:

- استفاده از روش تکرار و توسعه تدریجی
- معماری مبتنی بر مولفه ها
- راهبری بر مبنای موارد کاربری
- مدلسازی بصری نرم افزار
- کنترل تغییرات
- بررسی کیفیت نرم افزار

پشتیبانی می نماید.

---

[1] Workload

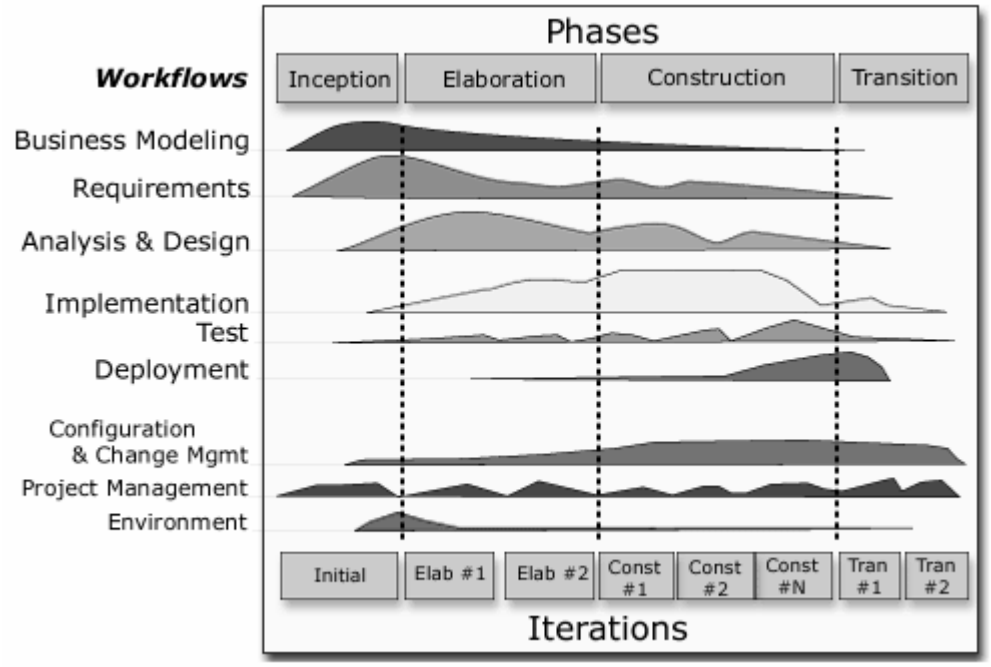
همانطوری که گفتیم USDP یک فرآیند تولید دو بعدی است(شکل ۵-۳ ):

§محور عمودی: این محور گردش کارهای اصلی را نشان می دهد.

§محور افقی(زمان): این محور ساختار چرخه تولید نرم افزار در RUP در بستر زمان را نشان می دهد.

بعد اول، جنبه ایستای سیستم را نمایش می دهد که شامل فعالیت ها، گردش کارها، فرآورده ها و کارکنان است. این فرآیندها متناظر با مراحل فرآیند تولید سنتی هستند که شامل جمع آوری نیازمندیها، تحلیل و طراحی، پیاده سازی و آزمایش است. به عبارت دیگر این بعد *فرآیندهای خرد*[۱] که بو سیله آنها روش تکرار و توسعه تدریجی پیاده سازی می شود را نشان می دهد.

بعد دوم، جنبه های پویای سیستم را نمایش می دهد. این جنبه ها به صورت چرخه ها[۲]، فازها[۳]، تکرارها[۴] و فرسنگ شمارها[۵] بیان می شوند.




شکل ۵-۳ نمونه ای از فرآیند تولید دو بعدی(USDP)

به عبارت دیگر این بعد فرآیندهای کلان[۶] را نشان می دهد.

---

[۱] Micro Processes

[۲] Cycles

[۳] Phases

[۴] Iterations

[۵] Milestones

[۶] Macro Processes

# ٦- بررسی اجمالی مراحل تولید نرم افزار بر مبنای متدولوژی USDP[1]

در فصل قبل مقدمه ای بر متدولوژی USDP آورده ایم. در این فصل به بیان ساختار دو بعدی و چرخه حیات نرم افزار از دیدگاه این متدولوژی خواهیم پرداخت

## ساختار ایستا

وظیفه یک فرآیند تولید نرم افزار این است که معین می کند چه کسی(Who)، چه چیزی را باید انجام دهد(What)، به چه صورت(How)، و در چه زمانی(When) است.

RUP برای این چهار سؤال، چهار عنصر مدلسازی زیر را ارائه می نماید:

١. کارکنان[2]: Who
٢. فعالیتها[3]: How
٣. فرآورده ها[4]: What
٤. گردش کارها[5]: When.

نمادهای این عناصر در شکل ٦-١ نمایش داده شده است.

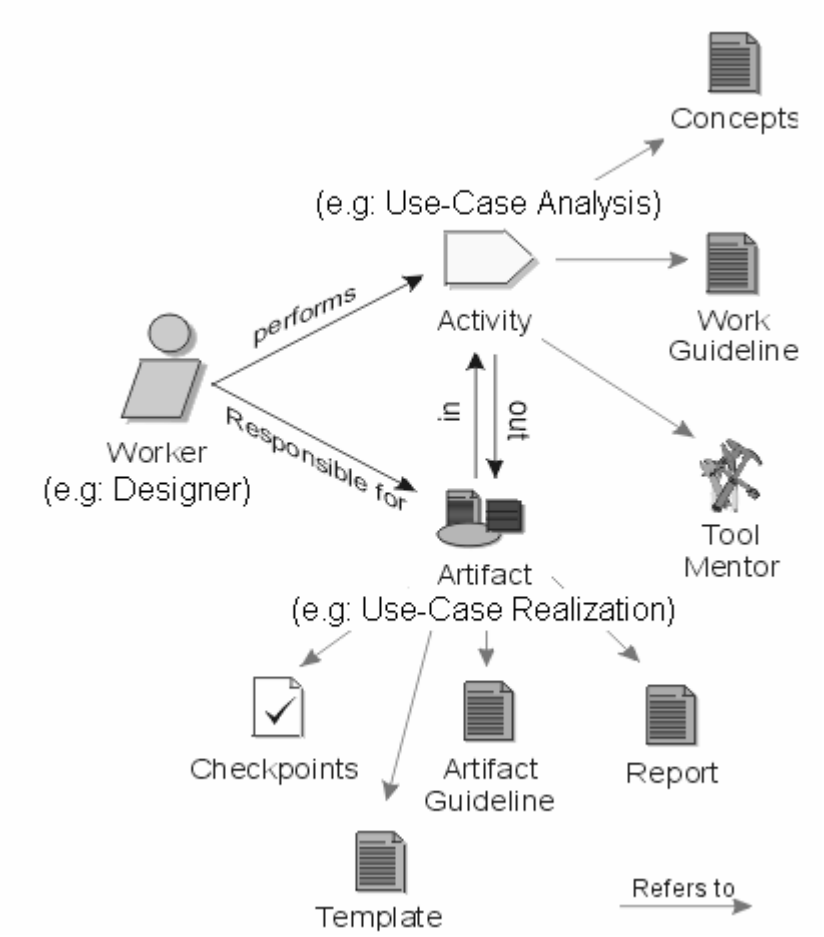
[1] USDP فرآیند اصلی که قابل Cutomize شدن می باشد. فرآیند Rational Unified Pocess (RUP) یک نمونه ای از فرآیندUSDP است(البته با اضافاتی چند). در این فصل این دو را یکسان فرض خواهیم نمود.

[2] Workers

[3] Activities

[4] Artifacts

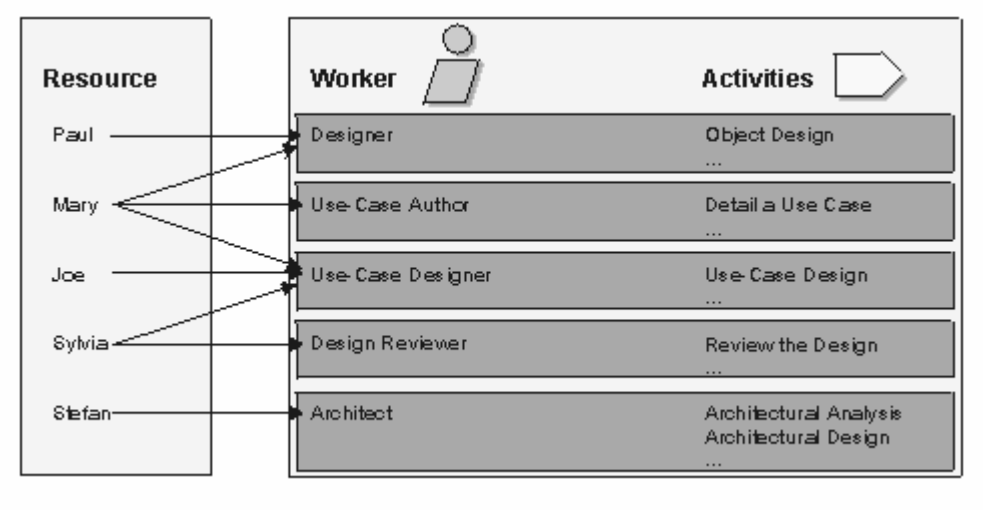
[5] Workflows




شکل ٦-١ مؤلفه های اساسی RUP

حال به توضیح این عناصر چهارگانه را می پردازیم:

§ **کارکنان**: این اصطلاح رفتار و مسئولیتهایی که یک نفر (یا یک تیم) در پروژه بعهده دارد را مشخص می نماید. این رفتار به صورت فعالیتهایی که باید این نفر انجام دهد، بیان می شود. همچنین مسوولیتها این نفر در رابطه با تولید/به روز رسانی/استفاده از فرآورده ها بیان می شوند. بعبارت ساده تر اصطلاح کارکن در RUP عبارآزمایش از نقشی که یک نفر (یا یک تیم) را در فرآیند تولید ایفا می کند. یک نفر می تواند چند نقش ایفا کند و چند نفر می توانند یک نقش را ایفا نمایند.(شکل ٦-٢)




شکل ٦-٢ رابطه بین کارکنان(نقش ها) و منابع(نیروی انسانی)

مثالهایی از کارکنان: سیستم آنالیست، طراح، طراح آزمایش.

توجه داشته باشید که برای هر کارکن دانستن مجموعه ای از مهارتها لازم است که بوسیله کسی که این نقش را بعهده دارد باید تامین شوند.

§ **فعالیتها**: کارهای که یک کارکن باید انجام دهد به صورت فعالیت بیان می شوند. پس یک فعالیت عبارت از واحد انجام کار است.هر فعالیت دارای هدف مشخصی بوده و معمولا به صورت تولید فرآورده های معینی بیان می شود. همچنین هر فعالیت به یک کارکن انتساب داده می شود.

در اصطلاح شئ گرایی کارکن یک شئ فعال[1] بوده و فعالیتهایی که بوسیله این کارکن انجام می شوند همان عملیات آن شئ خواهند بود.

مثالهایی از فعالیتها شامل :

§ پیدا کردن موارد کاربری و عامل ها بوسیله کارکن "سیستم آنالیست" انجام می شود.

§ بازبینی طراحی که بوسیله کارکن" بازبین کننده طراحی[2]" انجام می شود.

هر فعالیت از چند گام زیر تشکیل می شود:

§**گامهای اندیشیدن**[3]: در این گامها کارکن کار مورد نظر را درک کرده و فرآورده های ورودی را آماده می کند.

§**گامهای اجرا**[1]: کارکن یک فرآورده را تولید یا بهنگام سازی می نماید.

---

[1] Active Object

[2] Design Reviewer

[3] Thinking Steps

§**گامهای بازبینی**[۲]: کارکن نتایج را ارزیابی می نماید.

با یک مثال گامهای یک فعالیت را توضیح می نماییم.

فعالیت: پیدا نمودن موارد کاربری و عوامل مربوطه.

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 بسته بندی 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.

§**فرآورده ها**: فرآورده ها عبارتند از قطعه های اطلاعاتی(یا محصولاتی) که در طی فرآیند تولید نرم افزار، تولید، استفاده یا بهنگام سازی می شوند.(شکل ٦-۳)

فرآورده ها به عنوان ورودی و خروجی(نتیجه) فعالیت های عمل می کنند.

از دید شئ گرایی، همانطوری که فعالیتها به عنوان عملیاتی که یک شئ(کارکنان) انجام می نماید، عمل می کنند، فرآورده ها-نیز- به عنوان پارمترهای این فعالیتها به کار می روند. نمونه هایی از فراورده ها عبارتند از:

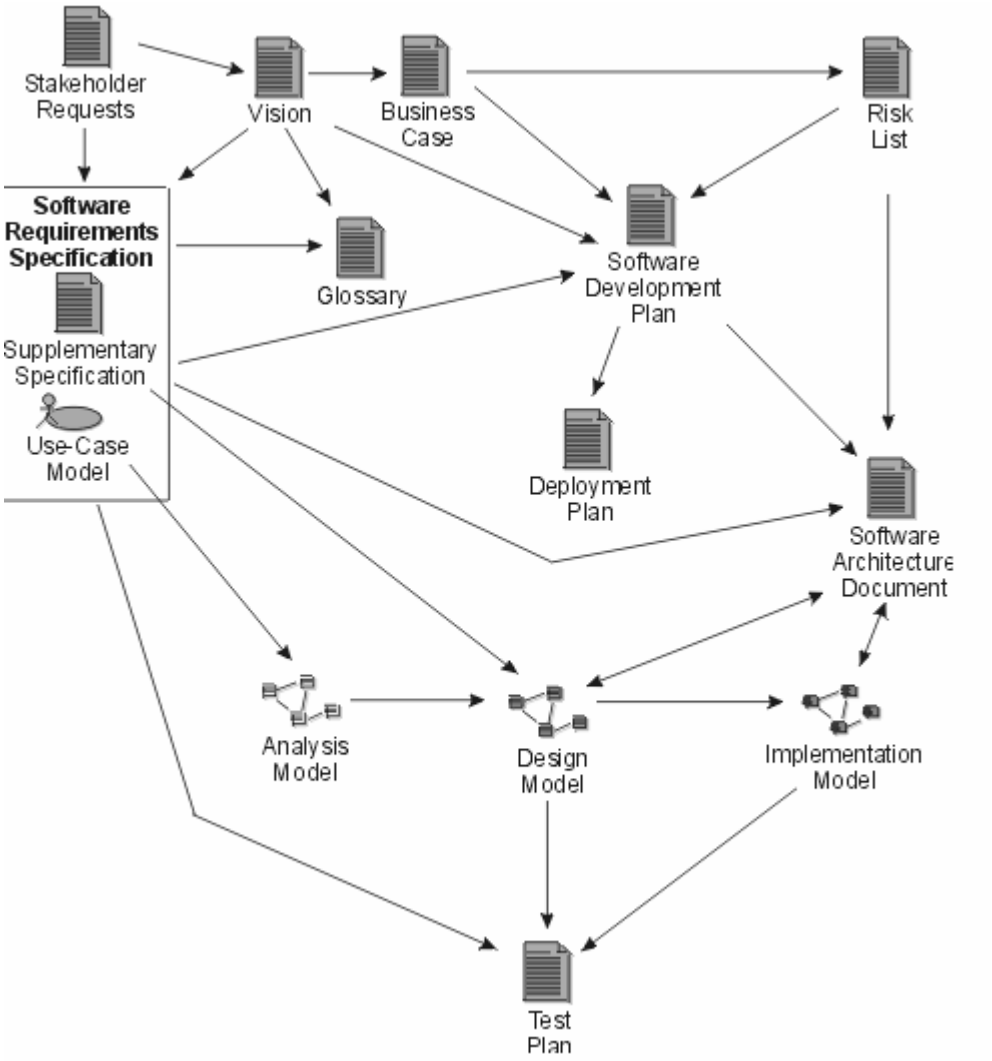
§مدل ها، مانند مدل موارد کاربری، مدل طراحی و ...
§عناصر مدل ها مانند کلاس ها، موارد کاربری، زیر سیستم ها
§مستندات، کد منبع[۳] و فایلهای اجرایی

---

[۱] Performing Steps

[۲] Reviewing Steps

[۳] Source Code




شکل۶-۳ فرآورده های اصلی در RUP

توجه داشته باشید که فرآورده ها یک مستند کاغذی نیستند. بسیاری از فرآیندهای تولید نرم افزار روی عمل مستند سازی و بالاخص مستند سازی کاغذی تاکید می ورزند و لی **RUP** چنین دیدگاهی ندارد بلکه استفاده از ابزارهای نرم افزاری مناسب برای تولید و ذخیره نمودن فرآورده ها را تشویق می نماید. هرگاه نیاز به مستند کاغذی وجود داشته باشد می توان از همان ابزارها برای تولید نسخه کاغذی فرآورده های مورد نظر را استفاده نمود.

در **RUP** فرآورده ها در ۵ مجموعه طبقه بندی می شوند:

۱. **مجموعه مدیریت**[۱]: این مجموعه شامل همه فرآورده هایی که وابسته به جنبه های مدیریت پروژه اند:
   - § فرآورده های برنامه ریزی مانند طرح توسعه نرم افزار و مورد کاری[۲].
   - § فرآورده های عملکردی مانند توصیف نشرها[۳]، تشخیص وضعیت پروژه، مستندات استقرار[٤].

۲. **مجموعه نیازمندیها**: این مجموعه شامل همه فرآورده هایی که تعریف و مشخصات سیستم نرم افزاری را در بردارند:
   - § مستند دورنما.
   - § نیازمندیها، به صورت نیازمندیهای سهامداران، مدل موارد کاربری و مشخصات تکمیلی.
   - § مدل کاری(در صورت نیاز) که برای درک فرآیندهای کاری که باید بوسیله سیستم پشتیبانی شوند، لازم است.

۳. **مجموعه طراحی**: این مجموعه شامل همه فرآورده هایی که سیستم مورد نظر را توصیف می نمایند:
   - § مدل طراحی
   - § توصیف معماری
   - § مدل آزمایش

٤. **مجموعه پیاده سازی**:
   - § کد منبع و فایلهای اجرائی
   - § فایلهای داده ای مورد نیاز

۵. **مجموعه استقرار**:
   - §اسکریپت های نصب[۵].
   - §مستندات کاربر
   - §مواد آموزشی[۱]

---

[۱] Management Set

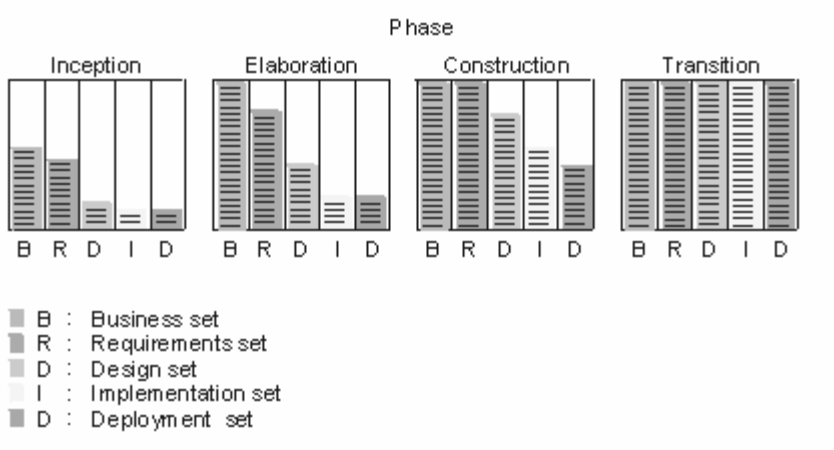
[۲] Business Case

[۳] Release Description

[٤] Deployment Documents

[۵] Installation Script

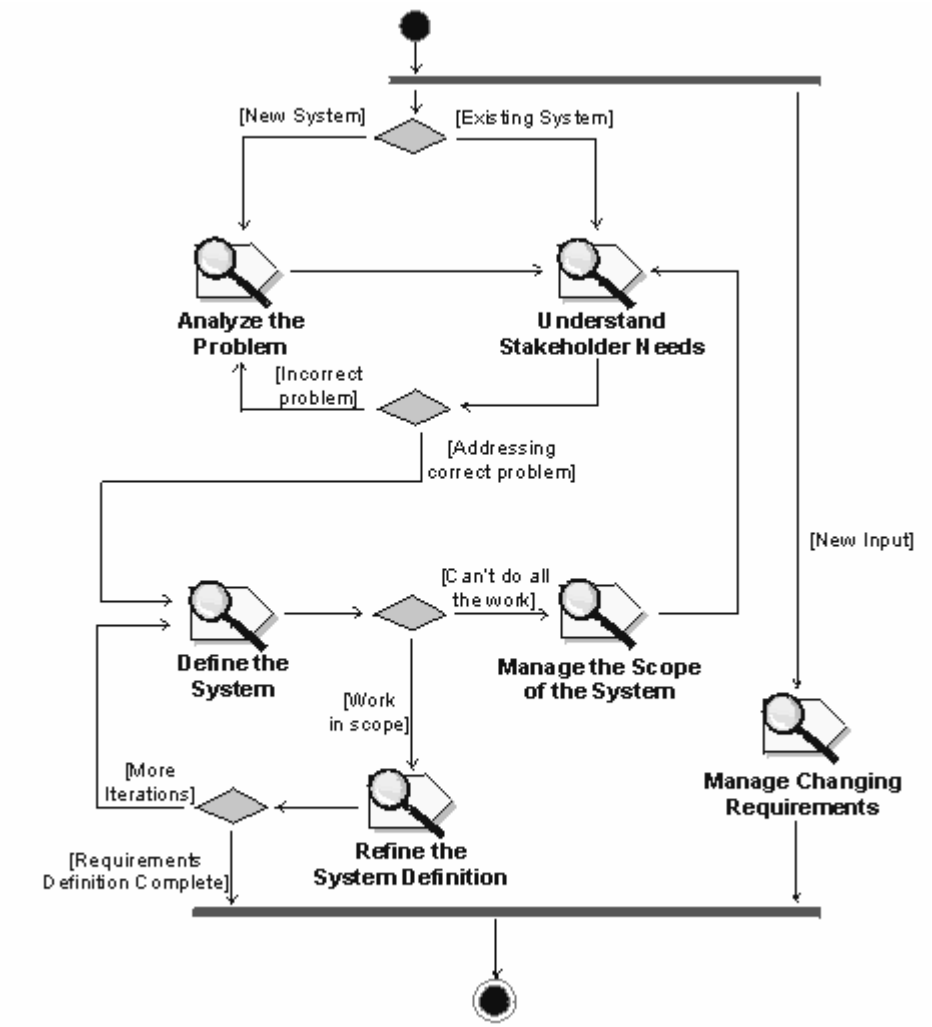
در روش توسعه افزایشی این پنج مجموعه در طی چرخه تولید نرم افزار تکامل می یابند. (شکل ٦-٤)




شکل ٦-٤ رشد مجموعه های فرآورده ها در طی چهار فاز تولید

§ **گردش کارها** : گردش کار عبارآزمایش از توالی مجموعه ای از فعالیت ها که نتیجه با ارزشی در پی دارند. در UML می توان گردش کارها را به صورت Sequence Diagram، Collaboration Diagram یا Activity Diagram نمایش نمود. در RUP معمولا از Activity Diagram استفاده می شود (شکل ٦-٥)

[1] Training Materials




شکل ٦-٥ نمونه ای از یک نمودار فعالیت که گردش کار
جمع آوری نیازمندیها را نمایش می دهد

می توان مجموعه همه فعالیت ها در گردش کارهای متعددی سازمان دهی و طبقه بندی نمـود. در RUP گردش کارها به سه طبقه تقسیم می گردند:

۱. **گردش کارهای پایه**[۱]: در RUP همه کارکنان و فعالیـت هـای موجـود در ۹ گـردش پایـه گروه بندی می شوند. گردش کارهای پایه به دو گروه تقسیم می شوند: گردش کارهـای فرآینـدی و گردش کارهای پشتیبانی.

گروه اول شامل ٦ گردش کار زیر است:

[۱] Core Workflows

۱. مدل سازی کاری[۱]
۲. جمع آوری نیازمندیها
۳. تحلیل و طراحی
٤. پیاده سازی
٥. آزمایش
٦. استقرار

گروه دوم شامل ۳ گردش کار زیر است:

۱. مدیریت پروژه
۲. مدیریت پیکربندی
۳. محیط[۲]

با وجود اینکه نام گردش کارهای فرآیندی ششگانه شبیه نام فازهای سنتی روش آبشاری بوده و لی چنانکه خواهیم دید فازهای فرآیند تکراری با فازهای سنتی تفاوت دارند. در RUP این گردش کارها بارها و بارها در طی چرخه تولید نرم افزار تکرار می شوند(البته در هر تکرار روی هر کدام از آنها با نسبت های متفاوت تاکید می گردد).

۲. **گردش کارهای تکرار**[۳]: گردش کارهای تکرار روشی دیگر برای بیان فرآیند تولید که روی روشن کردن آنچه در یک تکرار معمولی رخ می دهد، تاکید می نماید.
به عبارت دیگر، این گردش کارها عبارتند از یک نمونه[٤] از RUP برای یک تکرار.

۳. **جزئیات گردش کار**[٥]: با توجه به این است که گردش کارهای پایه شامل جزئیات زیادی بوده اند، لذا در RUP این گردش کارها به چند گروه تقسیم می شوند که هر گروه شامل فعالیتهایی که شدیدا بهم وابسته اند(مثلا با هم انجام می شوند یا نتایج میانی قابل توجهی را تولید می نمایند). هر گروه را *جزئیات گردش کار* گویند.

همچنین جزئیات گردش کار، جریان اطلاعات-یعنی فرآورده های ورودی و خروجی-و نحوه ارتباط فعالیت ها بوسیله فرآورده های متفاوت را نمایش می دهد.

---

[۱] Business Modeling Workflow

[۲] Environment Workflow

[۳] Iteration Workflow

[٤] Instance

[٥] Workflow Details

## § عناصر دیگر(ثانوی) RUP

کارکنان، فعالیتها(سازماندهی شده در گردش کار) و فرآورده ها، ستون فقرات ساختار ایستای RUP را تشکیل می دهند. اما مؤلفه های دیگری وجود دارند که با استفاده از آنها درک و به کارگیری RUP آسانتر می شود:

§**راهنماها**[۱]: فعالیتها و گامها معمولا به صورت مختصر بیان می شوند تا بتوان از آنها بعنوان مرجع استفاده نمود. همراه فعالیتها، گامها و فرآورده ها، راهنماها و جود دارند که عبارتند از قواعد پیشنهادی و شهودی[۲] که نحوه انجام این فعالیت ها و گامها را بیان می نمایند.

§**الگوها**[۳]: الگوها عبارتند از مدلهایی یا نمونه هایی از فرآورده ها که بوسیله آنها می توان فرآورده های مورد نظر را ساخت. RUP حاوی الگوهای زیر است:

§الگوهای MS-Word 97 / 2000 برای مستند سازی.

§الگوهای MS Frontpage برای بهنگام سازی RUP.

§الگوهای MS Project برای طرح و برنامه ریزی.

§**راهنمایی های ابزار**[٤]: راهنمایی های ابزار عبارتند از روش های انجام گام های یک فعالیت بوسیله یک نرم افزار معین.

در RUP راهنمایی های ابزار، روش انجام فعالیت های گوناگون بوسیله نرم افزارهای شرکت Rational مانند: Rose، RequisitePro، Clear Case و Test Studio توضیح می دهند.

§**مفاهیم**: بعضی از مفاهیم پایه مانند: ریسک، تکرار، فرسنگ شمار و ... در قسمتهای مختلف RUP توضیح داده شده اند.

§Procces Framework: مؤلفه های ذکر شده چهارچوب انعطاف پذیر RUP را بوجود می آورند زیرا این مؤلفه ها قابل اضافه، بهبود یا جایگزینی با توجه به نیازهای سازمان هستند.

---

[۱] Guidelines

[۲] Heuristics

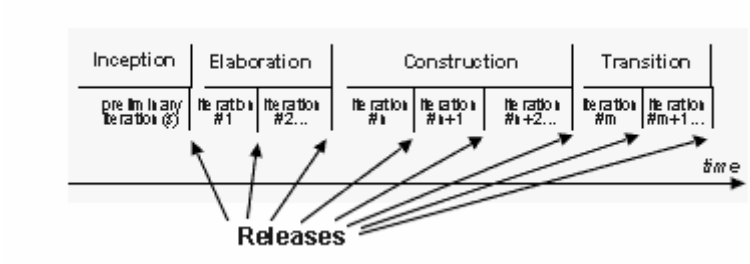
[۳] Templates

[٤] Tool Mentors

## ساختار پویا

این بعد چرخه تولید نرم افزار-از دیدگاه مدیریتی- را بررسی می کند. در دیدگاه مدیریتی چرخه حیات نرم افزار به تعدادی دوره[۱] تقسیم می شود که هر دوره با یک نسخه از محصول[۲] که به مشتریان تحویل داده می شود، خاتمه می یابد.

هر دوره شامل چهار فاز آغازین[۳]، تشریح[٤]، ساختن[٥] و انتقال[٦] است. هر فاز به نوبه خود به تعدادی تکرار[۷] تقسیم می شود.(شکل ٦-٦)




شکل٦-٦ رابطه فازها، تکرارها و نشرها

در پایان هر تکرار یک نشر[۸] تولید می شود که عبارآزمایش از زیر مجموعه ای قابل اجرا(نه ضرورتاً کد قابل اجرا بلکه می تواند-نیز- یک نمونه باشد) از محصول نهائی که می تواند داخلی[۹] یا خارجی[۱۰] باشد. نشر داخلی بوسیله تیم توسعه دهندگان برای ارزیابی موفقیت یک تکرار و برای اهداف نمایشی استفاده می شود. در مقابل، نشر خارجی(مانند Beta Release ) برای آزمایش بوسیله کاربران و مشتریان در اختیار آنها قرار می گیرد. سپس بر اساس نظرات آنها توسعه دهندگان اشکالات سیستم را برطرف می سازند.

توسعه دهندگان بوسیله مدلهای متعدد آنچه در این فازها رخ می دهد را مجسم می کنند.

---

[۱] Cycle

[۲] Software Generation

[۳] Inception

[٤] Elaboration

[٥] Construction

[٦] Transition

[۷] Iterations

[۸] Release

[۹] Internal Release

[۱۰] External Release

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-می رساند.

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* بزرگراه ها هستند که بوسیله آن مشخص می شود چه فاصله ای تا رسیدن به مقصد مانده است. از اینجا به این نقاط فرسنگ شمار[۱] گویند.

دو نوع فرسنگ شمارها داریم: اصلی[۲] و فرعی[۳]. از فرسنگ شمار اولی برای تعیین پایان یافتن یک فاز و از دومی برای تعیین پایان یافتن یک تکرار استفاده می شود.

---

[۱] Milestones

[۲] Major Milestone

[۳] Minor Milestone

# ۷- مدلسازی موارد کاربری[۱]

## مقدمه

مدتها در فرآیندهای شئ گرا و غیر شئ گرا از سناریوها برای کمک به فهم نیازمندیهای سیستم استفاده می شد. اما این سناریوها با اینکه همیشه تولید می شدند بندرت مستند سازی و نگهداری می شدند، تا زمانیکه Ivar Jacobson این وضعیت را با ارائه متدولوژی Objectory خود دگرگون نمود. او در متدولوژی خود نام این سناریوها را مورد کاربری[۲] گذاشت و آنرا محور اصلی کار خود قرار داد. بتدریج جامعه شئ گرا با استفاده از UC ها به اهمیت آن پی برد و آنرا بعنوان ابزار قوی معرفی نمود.

ویژگی بارز این روش این است که مسئله جمع آوری نیازمندیها[۳] از دیدگاه غیر تکنیکی بررسی می کرد. در واقع دو نگاه به یک سیستم وجود دارند: یکی نگاه به سیستم از بیرون آن(غیر تکنیکی) و دیگری نگاه فردی که می خواهد سیستم را بسازد (تکنیکی). این دو دیدگاه کاملاً با یکدیگر متفاوتند.

با این مقدمه می توان UC را به صورت زیر تعریف نمود:

UC دنباله ای از عملیات است که یک سیستم انجام می دهد 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" می تواند هر کدام (با توجه به تعریف فوق) یک مورد کاربری باشند.

در زبان مدلسازی UML[۴] مورد کاربری را با یک بیضی نمایش می دهند که نام UC در داخل بیضی(یا زیر آن)نوشته می شود.

مفهوم دیگری که اینجا مورد نیاز است مفهوم عامل[۵] می باشد. عامل در حقیقت شئ خارج از حیطه سیستم است که مستقیماً با آن در ارتباط است. کاربران و کلیه سیستم های که باسیستم مورد نظر در ارتباط هستند عاملهای آن هستند.

---

[۱] Use Case Modeling

[۲] Use Case(UC)

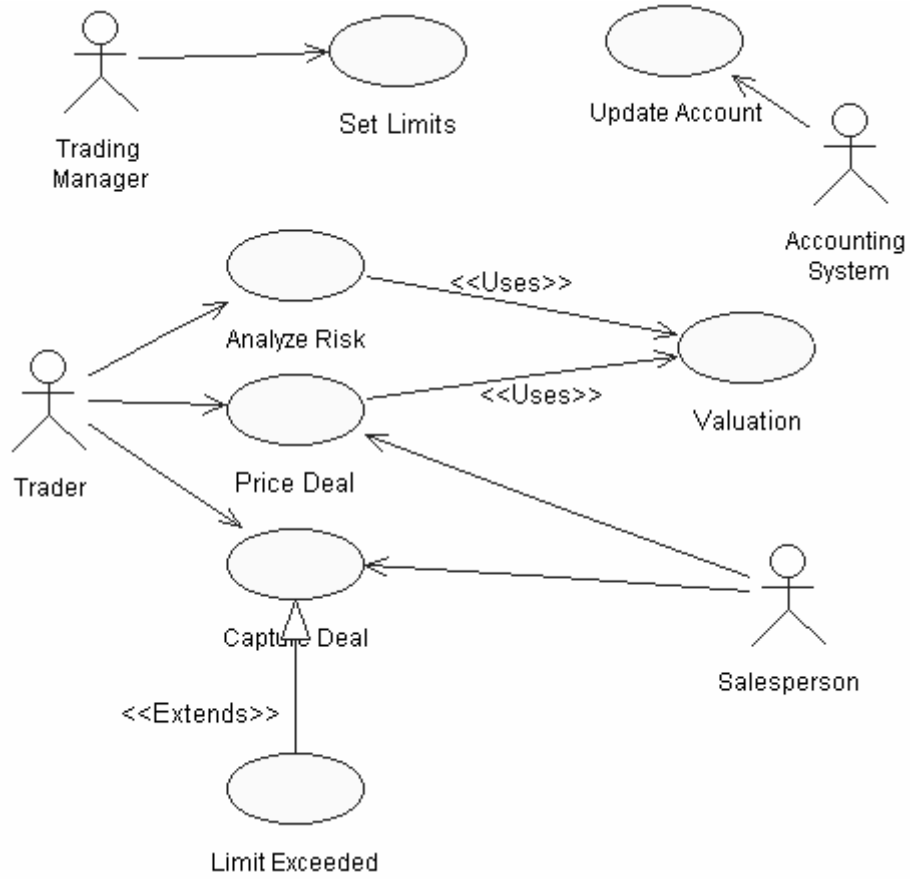
[۳] Requirements Gathering

[۴] Unified Modeling Language

[۵] Actor

مثال: در (شکل ۷-۱) چهار عامل دیده می شوند 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 می تواند نقشهای متعددی داشته باشد. به عنوان مثال شخصی که Trader است می تواند Trading Manager هم باشد یا یک Trader می تواند Salesperson هم باشد. بنابراین در بدست آوردن عاملهای یک سیستم بهترست به نقشهای افراد بجای تتیرهای شغلی آنها توجه شود.

رابطه موارد کاربری و عوامل m:n می باشد یعنی هر عامل می تواند اجرا کننده چندین UC باشد و هر UC هم می تواند بوسیله چند عامل اجرا گردد.




شکل ۷-۱

سیناریو عبارتست از یک مسیر در میان سیستم است. مجموعه سناریوهای متشابه یا مورد کاربری را می سازند. بعبارت دیگر سناریوها یک نمونه و موارد کاربری یک کلاس است.

## روابط توسعه به(Extends) و استفاده از (Uses)

در شکل ۷-۱ غیر از اتصالات معمولی بین عاملها و موارد کاربری، دو نوع اتصال دیگر دیده می شود که "استفاده از" و "توسعه به" نام دارند.

- اتصال "توسعه به" یک UC به یک UC مشابه اما اندکی محدودتر (همان رابطه وراثت) وصل می نماید. در شکل۷-۱ مورد کاربری"Capture Deal" کارهای روتین مربوط به قرار داد را انجام می دهد. اما اگر برای بستن قرار داد با یک مشتری شروطی موجود باشند، برای بستن قرار داد با مشتری که این شروط را ندارد نمی توان کارهای روتین را طی نمود و احتیاج به یک UC خاصی هست که این UC خاص با اتصال"توسعه به" به UC اصلی وصل می گردد.
- اتصال دیگری که در UC مطرح است "استفاده از" می باشد. وقتی رفتار مشترک بین دو UC وجود دارد برای جلوگیری از تکرار آن در UC آنرا به صورت یک UC جدا در نظر می گیرند و UC های دیگرکه می خواهند از أن استفاده نمایند با اتصال "استفاده از" به آن مرتبط می شوند. به عنوان مثال در شکل ۷-۱ موارد کاربری Analyze Risk و Price Deal هر دو احتیاج به رفتار Value The Deal دارند.

مدلسازی موارد کاربری شامل ساختن دیدهای خارجی[۱] از سیستم بصورت مجموعه ای از عوامل و موارد کاربری مرتبط می باشد. حال این سوال را مطرح می نماییم: چگونه یک مدل موارد کاربری ساخته می گردد؟

یکی از روشهای ایجاد مدل موارد کاربری، روش جعبه سیاه[۲] بوده که در آن سیستم مورد نظر بصورت یک جعبه سیاه که به رویدادهای گوناگون که آغاز کننده آنها عوامل بوده عکس العمل نشان می دهد، دیده می شود. موارد کاربری راه های مختلف استفاده از از جعبه سیاه بازای رویدادهای مختلف را نمایش می دهد. مراحل به کارگیری این روش بشرح ذیل می باشد:

- صورت مسئله یا هدف سیستم را مشخص نمایید: در واقع هدف سیستم با جواب دادن به این سوال که "چرا می خواهیم سیستم را بسازیم؟" مشخص می گردد[۳].
- شناسائی عوامل: از تعریف مسئله شروع کرده و سوالاتی درباره اینکه چه کسانی(یا سیستمهایی) متصدی انجام سرویسهایی که سیستم مورد نظر باید فراهم کند، را می پرسیم. پاسخ این سوالها منجر به تشخیص عوامل اولیه سیستم می گردد. معمولاً اولین سوال که باید پرسیده گردد"آیا

---

[۱] External View

[۲] Black-Box Approach

[۳] اصطلاحاً به آن Statement of Purpose گویند.

مشتری به صورت مستقیم با سیستم در تماس است یا خیر؟" جواب این سوال می تواند منجر به یافتن اولین عامل سیستم گردد. با ادامه زنجیر سوالها و متمرکز شدن روی نحوه برآوردن هدف های سیستم بقیه عوامل پیدا می شوند.

- شناسائی موارد کاربری: برای شناسائی UCها باید یک نگرش کاربر گرا(User-Oriented) به سیستم داشته باشیم و از خودمان بپرسیم نحوه ارتباط کاربر با سیستم چگونه خواهد بود و چه سرویسهایی مورد انتظار کاربر می باشد؟ در واقع هر UC باید خصوصیات زیر را داشته باشد:

۱- یک واحد مستقل از تعامل (Single Unit of Interaction)

۲- باید تنها بوسیله یک عامل و در یک مکان انجام گیرد.

۳- باید به یک نتیجه ارزشمندی از دید کاربر منتهی گردد و سیستم را از یک حالت شناخته شده به یک حالت شناخته شده دیگر منتقل نماید.

٤- ایجاد مدل موارد کاربری: با استفاده از Notation توصیف شده موارد کاربری را ایجاد می نماییم.

٥- تشریح موارد کاربری: برای تشریح موارد کاربری از مصاحباتی که با کاربران سیستم انجام داده أیم استفاده نموده و زنجیر فعالیتهای بوجود آمده بعلت رخ دادن رویداد متعلق به U.C را یاداشت می کنیم. بدین صورت درک U.C آسانتر خواهد بود. برای هر UC توصیف زیر را تهیه می کنیم:

- نام مورد کاربری
- هدف مورد کاربری: به صورت فشرده هدف از یان مورد کاربری را بیان می نماییم
- توصیف مورد کاربری: مجموعه فعالیتهای مورد نیاز برای اجرای UC

با یک مثال تفصیلی نحوه اعمال گامهای ذکر شده را بیان می نماییم

**مثال: تعمیرگاه**

- صورت مسئله:

"*هدف سیستم فراهم نمودن مدیریت کارا برای همه جنبه های چرخه سرویس دهی و تعمیر از تعریف کارهای[1] مورد نیاز مشتریان گرفته تا انتهای این کارها می باشد*" *سیستم باید تسهیلات زیر را ارائه نماید:*

- *رزرو کارها (شامل سرویس و تعمیر)*
- *شناسائی قطعات یدکی مورد نیاز و درخواست آنها*
- *زمانبندی کارها*

[1] اینجا مقصود از کارها همان Customer Jobs که شامل سرویس دهی و تعمیر می باشد.

§ *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 اولویت دار*

*استثناها: رویس دهی 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 می شویم در می یابیم 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 می نماید. بنابراین می توان گفت که عامل دوم همان مسئول مشتریان می باشد. در ادامه این سناریو می بینیم که کنترل کننده قطعات درخواست مشتری را بررسی نموده و مشخص می نماید 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) بالآخره 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 می باشند:

کنترل کننده قطعات، مکانیک ◄عوامل داخلی

تولید کننده ◄ عامل خارجی

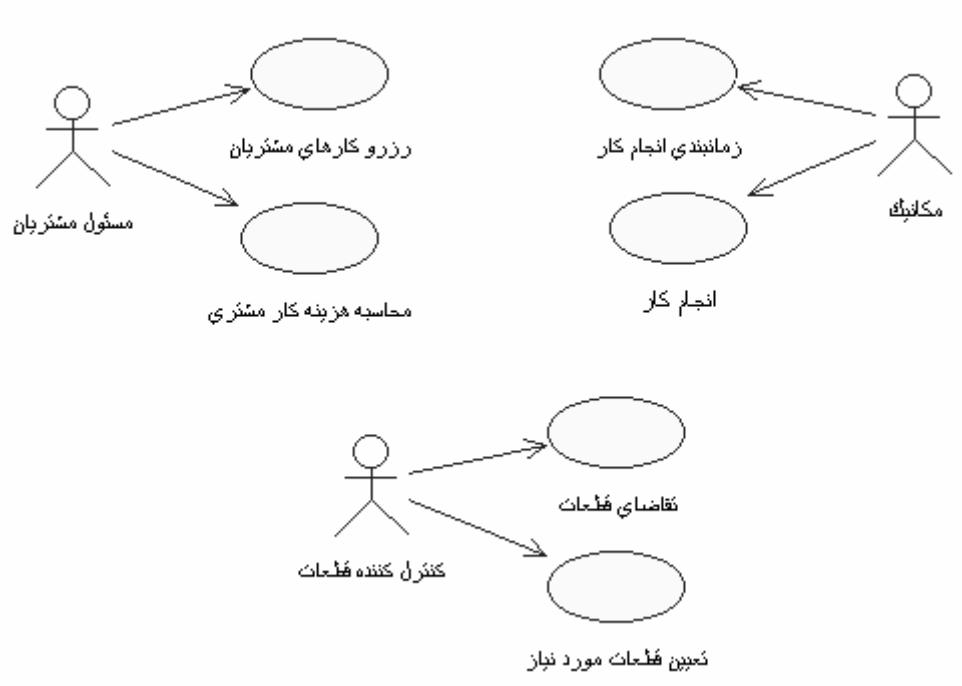
در جدول زیر عواملی که با سیستم در ارتباط مسقیم هستند لیست شده اند:

| 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 می توان از موارد کاربری زیر نام برد:

| 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 | مکانیک |
| مدیریت کار از ابتدا تا خاتمه، اطمینان از درستی انجام آن و ثبت جزئیات کار انجام شده. | فرارسیدن زمان انجام کار | مکانیک |
| اطمینان از رضایت مشتری، دریافت مزد کار و تحویل ماشین به مشتری | رسیدن مشتری برای تحویل ماشین | مسئول پذیرش مشتریان |

- ایجاد نمودار موارد کاربری: با توجه به آنچه بیان شد، نمودار در شکل ۷-۲ کشیده شده است.




شکل ۷-۲ نمودار موارد کاربری مثال تعمیرگاه

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

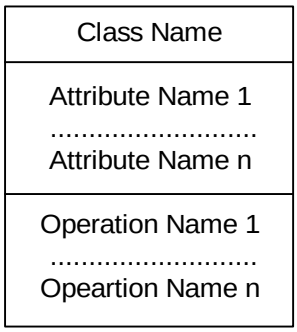
| نام مورد کاربری | رزرو کارهای مشتریان |
|---|---|
| هدف مورد کاربری | تعیین کار مناسب که نیازهای مشتریان را به نحو احسن برآوردکند |
| گامهای مورد کاربری | ۱- بدست آوردن جزئیات خودرو و مشتری:<br>۱-۱ برای مشتریان فعلی جزئیات مربوط به آنها استخراج کنید<br>۱-۲ برای مشتریان جدید مشخصات مورد نیاز(مانند: نام، آدرس، شماره خودرو، مدل آن، و سال ساخت) را ثبت نمایید.<br>۲- اگر کار مورد نیاز سرویس باشد، سرویسهای مناسب مدل خودرو را پیدا کنید.<br>۳- اگر کار مورد نیاز تعمیر باشد، هزینه آنرا پیش بینی نمایید.<br>٤- بر زمان و ساعت کار با مشتری به توافق برسید.<br>۵- مشخصات ذکر شده -بعد از تایید مشتری-راثبت نمایید. |

# ۸- مدلسازی کلاسها[1]

## نمایش مفاهیم اساسی مدلسازی کلاسها در UML

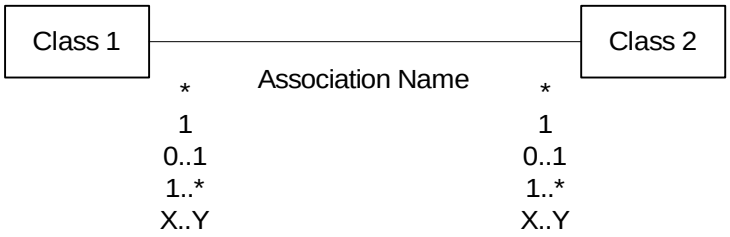
در قسمتهای قبل مفاهیم مرتبط به کلاس بعنوان مفهوم اصلی همه متدولوژیها شئ گرا را بیان کردیم. در این بخش به نحوه نمایش کلاس و مفاهیم دیگر در UML و گامهای لازم برای رسم نمودار کلاس را می پردازیم.

در UML کلاس به صورت زیر نمایش داده می شود:




شکل ۸-۱

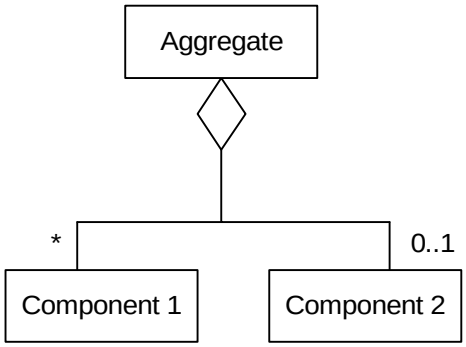
- **رابطه انجمنی**




شکل ۸-۲

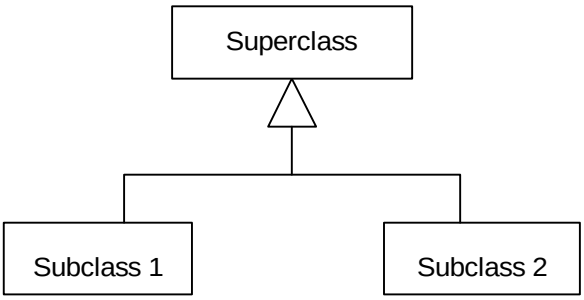
[1] Class Modeling

- **رابطه تجمعی**



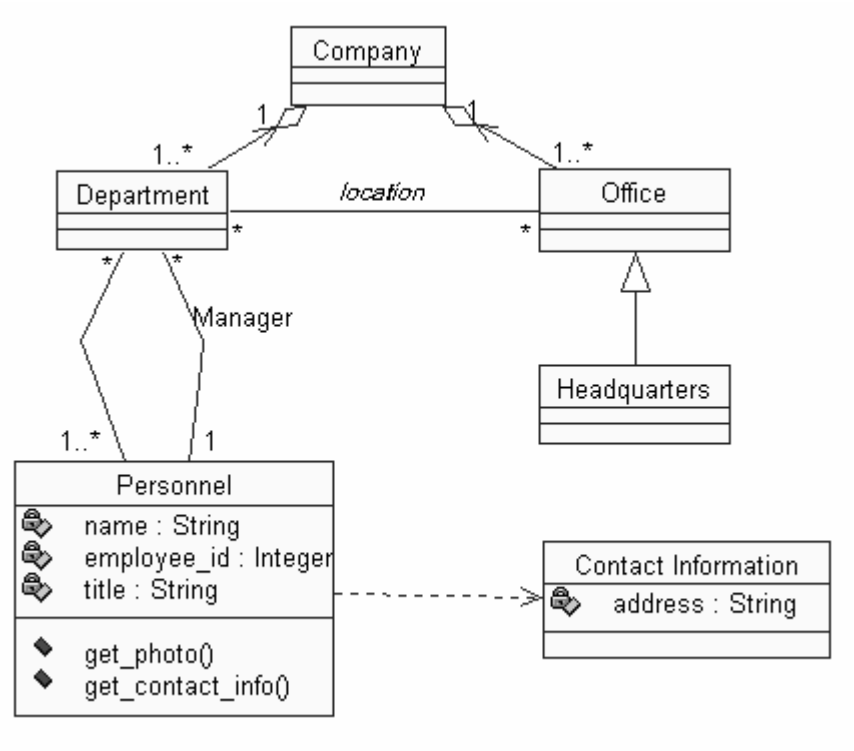

شکل ۸-۳

- **رابطه وراثت**




شکل ۸-٤

مثال: نمودار کلاس برای شرکت، دفتر و...(شکل ۸-۵)




شکل ۸-۵ : نمونه ای از نمودار کلاسها

## تکنیک مدلسازی

برای رسم نمودار کلاسها، مراحل زیر باید طی شوند:

۱- شناسائی کلاسها: برای شناسائ کلاسها می توان از روشهای زیر استفاده نمود:

- استفاده از کارتهای CRC
- صورت مسئله و نیازهای وظیفه مندی[۱] تحلیل می کنیم: نامها، کلاسها یا صفات کاندید و فعلها، اعمال یا Associationهای کاندید به حساب می آیند. این روش باید بدقت-وبا توجه به مفهوم کلاس و عدم اکتفا به تحلیل صرف واژه ها- بکاربرد و الا لیستی طولانی از کلاسهای بی معنی 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 می نماییم.

۳- تکمیل و توسعه نمودار کلاسها بوسیله افزودن صفات و اعمال مورد نظر.

٤- استفاده از الگوهای معروف برای آسانی طراحی و بهره برداری از استفاده مجدد.

[۱] Functional Requirement

۵- معمولاً، تا این مرحله نمودار پیچیدهای پدید خواهد آمد که با اِعمال مفهوم روابط تجمعی، وراثت، وابستگی می توان از پیچیدگی آن کاهش نمود.

## مثال: تعمیرگاه

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 شئ را در بردارند، پیدا می نماییم: مشتری، خودرو، سرویس، تعمیر، کار.

نام و آدرس از صفات مشتری محسوب می شوند.

شماره ماشین و مدل آن از صفات ماشین محسوب می گردند. نوع سرویس(معمولی یا با اولویت) صفت سرویس محسوب می شود.

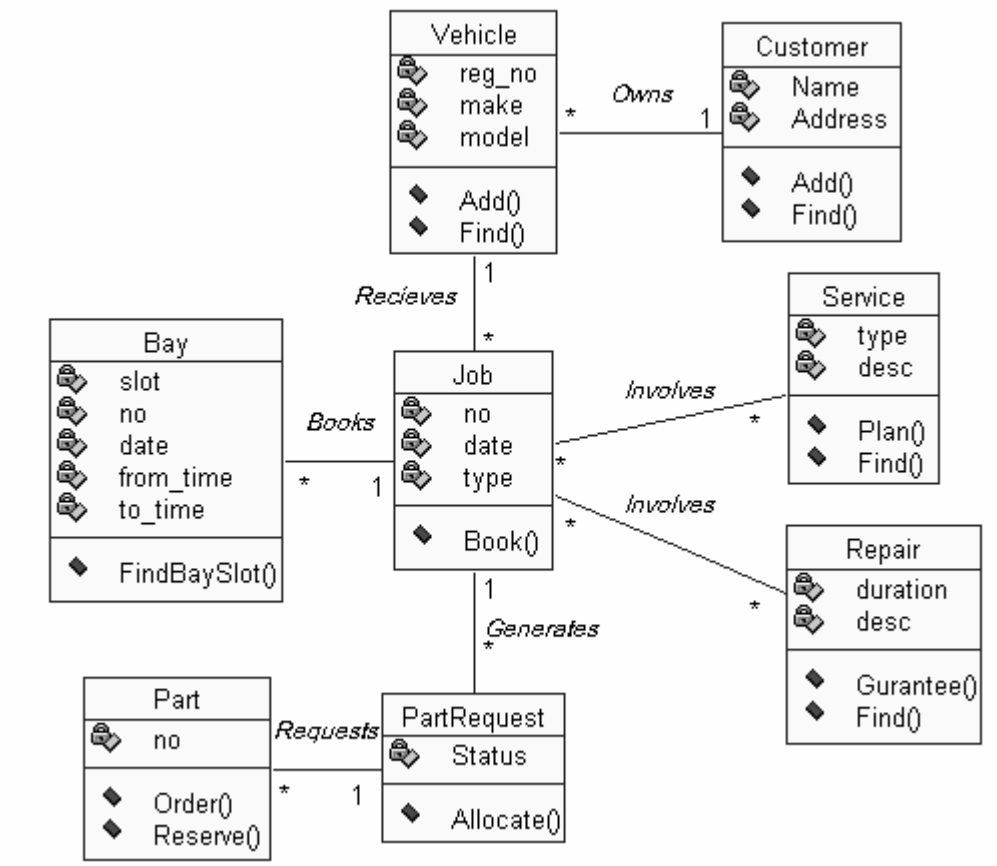
چون نباید به تحلیل گرامری اکتفاد نمود می توان روابط زیر را اضافه کرد: ماشین یک مالک دارد و آن مشتری است. "کار" در رابطه با ماشین انجام می گیرد. یک کار شامل سرویس، تعمیر یا هر دو می باشد.

- 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 زیر قید شده اند:

| 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

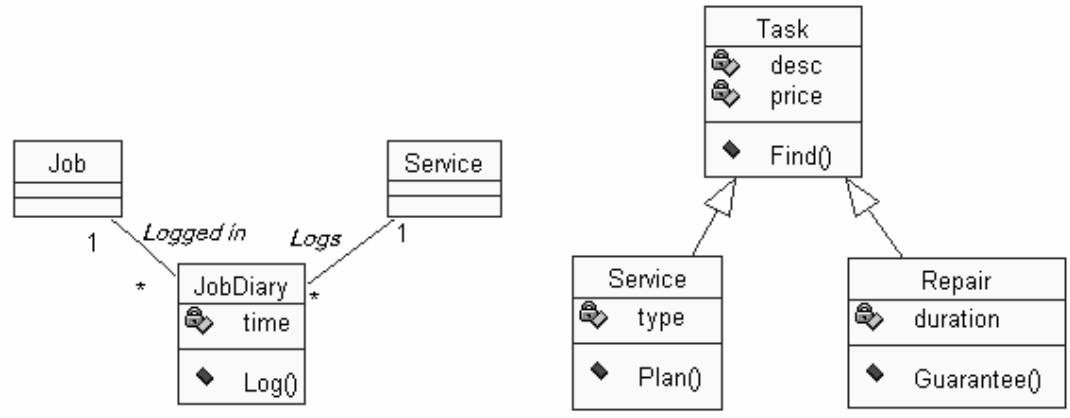
۳- تکمیل نمودار و اضافه صفات و اعمال مهم(شکل ۸-٦)




شکل ۸-٦ نمودار کلاسهای مثال تعمیرگاه

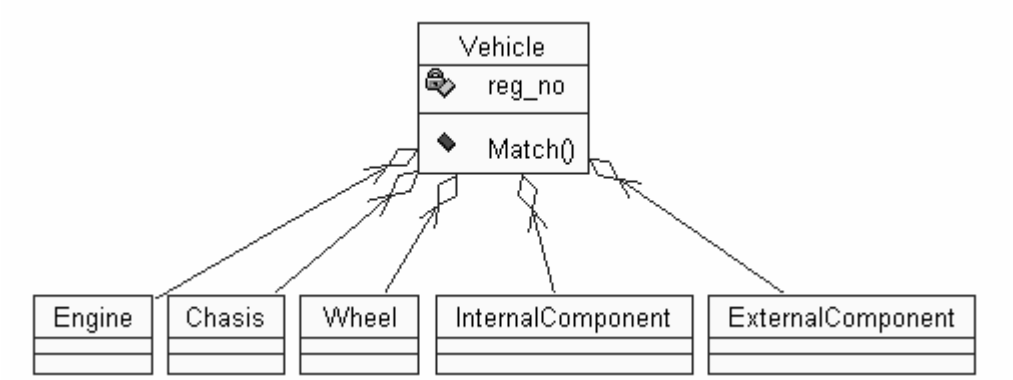
٤- استفاده از الگوها(از آن در این مثال مقدماتی صرف نظر می نماییم)

٥- اعمال مفاهیم روابط تجمعی، وراثت و وابستگی(شکل ۸-۷ و ۸-۸)




(ب) (آ)

شکل ۸-۷ اعمال رابطه وراثت و رابطه وابستگی به نمودار کلاسها




شکل ۸-۸ اعمال رابطه تجمعی به نمودار کلاسها

در (شکل ۸-۷ آ) از مفهوم وراثت استفاده شده است. چون هر دو کلاس Service و Repair صفات و اعمال مشترکی[۱] و متفاوتی دارند پس می توان یک Superclass مانند Task معرفی نمود که صفات و اعمال مشترک آنها را دربرگیرد.

مواردی وجود دارد که در آن می توان برای رابطه وابستگی صفتهایی یا اعمالی درنظر گرفت. در این موارد این رابطه به صورت کلاس مدلسازی می گردد. برای مثال(شکل ۸-۷ 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 می گیریم. این کلاس وابستگی بین دو کلاس اصلی را نمایش داده و با هرکدام رابطه یک به چند دارد.

در(شکل ۸-۸) از رابطه تجمعی استفاده شده است. چنانکه می دانیم یک خودرو از اجزائی مانند موتور(Engine)، شاسی(Chasis)، چرخها(Wheels) و غیره تشکیل می گردد.

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

# ۹- مدلسازی تعامل و همکاریها[۱]

همانطوریکه بیان کردیم می توان یک سیستم نرم افزاری 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 می نماید. در واقع نمودار کلاس ساختار ایستای سیستم را توصیف می نماید. 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 می باشد. یک خودرو از موتور، چرخ ها، شاسی و ... تشکیل می شود(ساختار ایستا) همچنین برای اینکه خودرو حرکت کند این اجزاء با ترتیب خاصی همکاری می کنند(رفتار پویا). این جنبه بوسیله نمودار تعامل و همکاریها مدلسازی می گردد. در مدلسازی تعامل، همکاری گروهی از اشیاء در انجام یک عمل معین نشان داده می گردد. بطور کلی منظور از یک عمل معین، مورد کاربری می باشد و هر نمودار تعامل رفتار یک مورد کاربری را با توجه به اشیاء دخیل در آن و ارتباطات ما بین این اشیاء نشان می دهد.
نمودار تعامل بر دو نوع است: یکی نمودار ترتیبی[۲] و دیگری نمودار همکاری[۳]که هر یک توضیح داده خواهد شد.

## نمودار ترتیبی

نمودار ترتیبی معمولاً در بیان مراحل اجرای یک مورد کاربری استفاده می گردد. در این نمودار اشیاء به صورت جعبه هایی در بالای خطوط منقطع عمودی نمایش داده می شود. به هر یک از این خطوط عمودی، خط عمر آن شئ گفته می شود. این اشیاء می توانند مجرد[۴](بدون نام[۵]) باشند که نشانگر یک نمونه از یک کلاس است و به صورتClassName:: نام گذاری می شوند. همچنین اشیاء می توانند نشان دهنده نمونه واقعی باشند و به صورت ClassName::ActualInstance نام گذاری می شوند. پیامی که از یک شئ به دیگری فرستاده می شود توسط خط جهت داری در میان خطوط عمر

---

[۱] Interaction and Collaboration Modeling

[۲] Sequence Diagram

[۳] Collaboration Diagram

[۴] Abstarct Objects

[۵] Anynomous

دو شئ کشیده می شود. هر پیام نشان دهنده درخواست اجرای یک عمل یا اعلام رخ دادن یک رویداد بوده و می تواند شامل مجموعه ای 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 می شود(بین دو علامت[] نوشته می شود). بخش دوم علامت تکرار[1] که با × نمایش داده می شود و معنای آن ارسال این پیام به چندین شئ گیرنده می باشد.

برای نمایش بازگشت پیام از فلشی در جهت مخالف استفاده می گردد(⟵ ). این فلش بمعنی بازگشت یک پیام است و بهتراست مواقعی که بازگشت از یک پیام مفهوم خاصی را منتقل نمی کند این فلش نمایش داده نشود. در سمت چپ نمودار تعامل اشیاء می توان-به صورت اختیاری-شرحی با فرم ساخت یافته برای مورد کاربری مورد نظر را نوشت. هدف این شرح تسهیل درک نمودار تعامل اشیاء می باشد. نمودارهای ترتیبی بسیار ساده و براحتی قابل درک می باشند و بعلاوه این نمودارها قبالیت نمایش فرآیندهای همزمان را دارند.

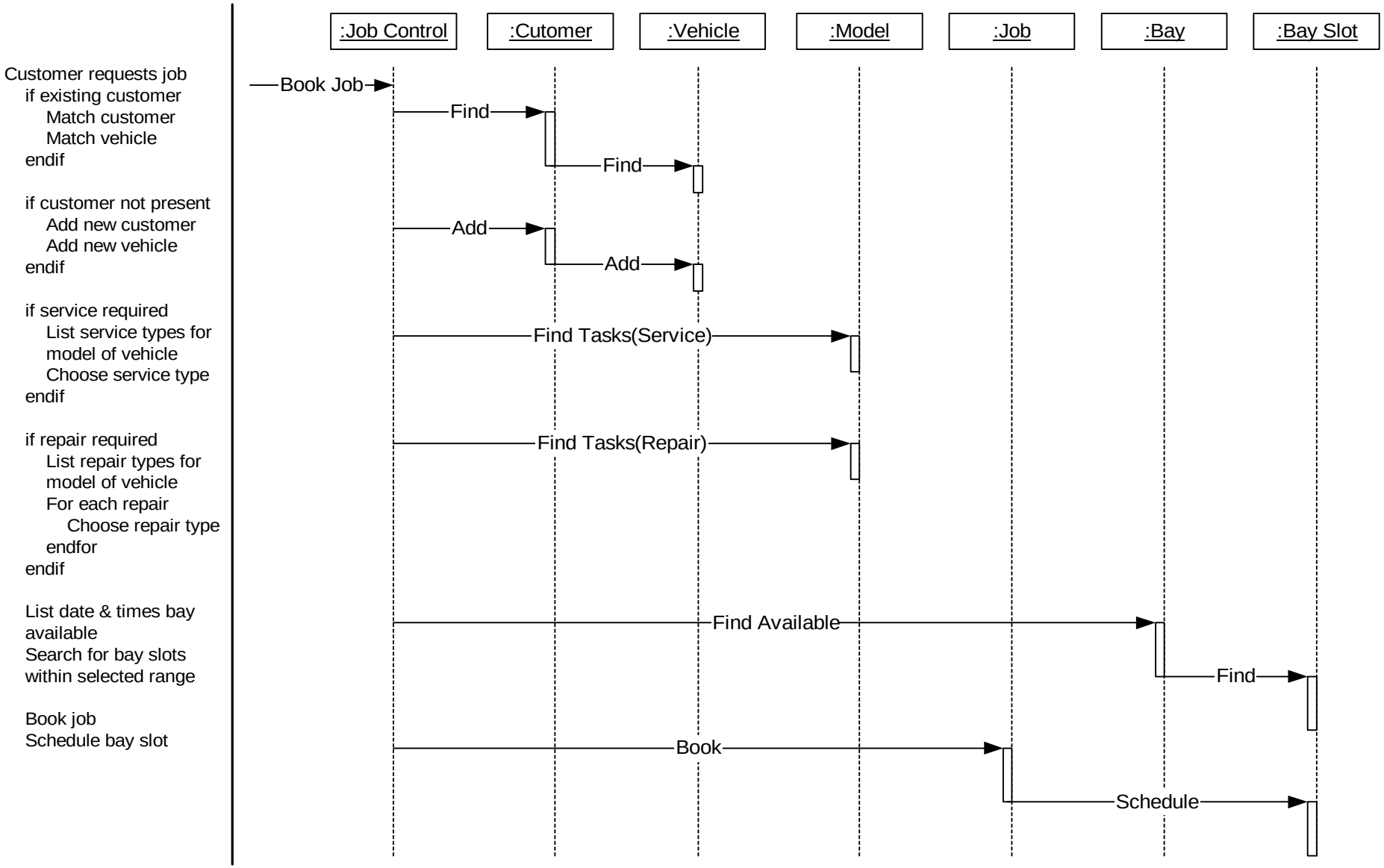
## مثال: تعمیرگاه

ترسیم نمودار تعامل اشیاء برای مورد کاری "ثبت کار استاندارد برای مشتری"[2].

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 ۹-۱ نمایش داده شده است.

[1] Iteration Marker

[2] Book Standard Job for Customer

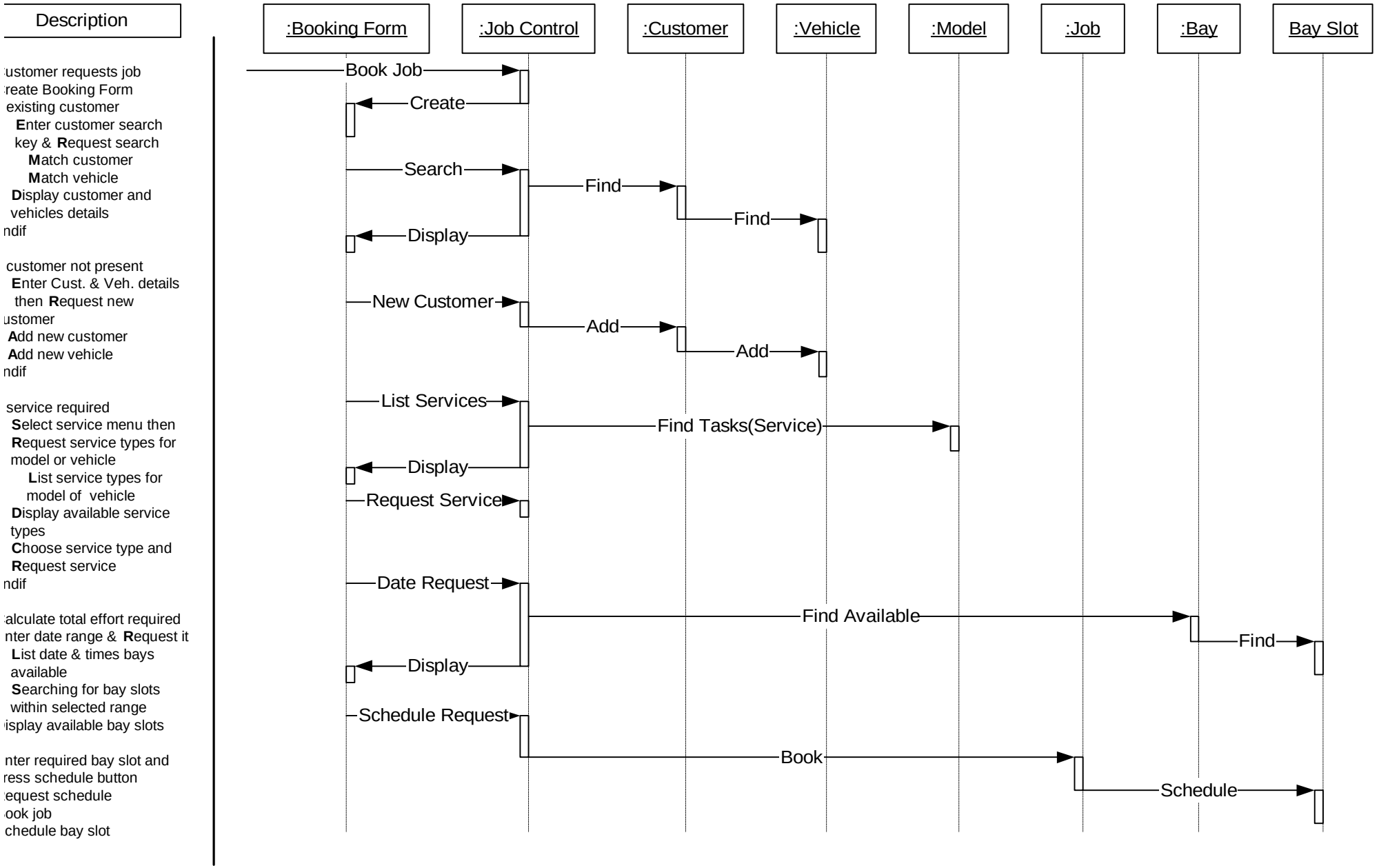



شکل ۹-۱ نمودار اولیه تعامل اشیاء برای مورد کاربری"ثبت کار استاندارد برای مشتری"

همانطوریکه در نمودار ملاحظه می کنیم یک شئ جدید -که در نمودار کلاس وجود نداشته- بنام Job Control اضافه گردیده است. هدف از اضافه این شئ جدا نمودن اشیاء کاری از تغییرات در واسط کاربر می باشد. بدین صورت سطح استفاده مجدد بالا خواهد رفت. در واقع اضافه این شئ نمونه ای از مواردی که در آن برای طراحی بهتر سیستم یک سری کلاسهای کمکی به آن اضافه می نماییم. با رسیدن پیام Book Job شئ Job Control فعال شده و کنترل بقیه اشیاء را بعهده می گیرد.

در شکل ۹-۱ واسط کاربر نادیده گرفته شده است. برای اضافه آن باید شئ Booking Form اضافه گردد(شکل ۹-۲).

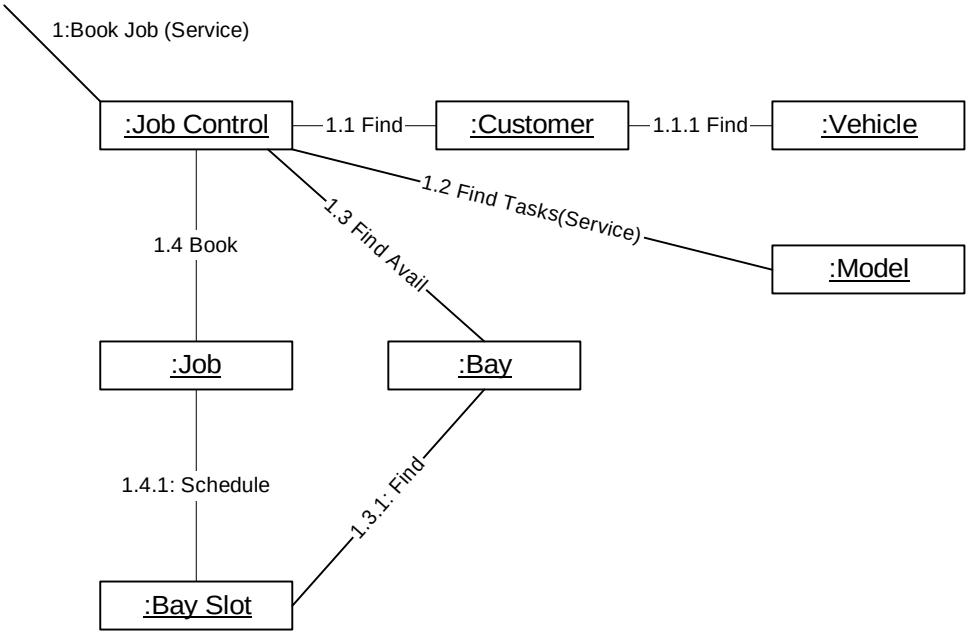
در شکل ۹-۲ بر اثر درخواست ثبت کار مشتری پیام Book Job به شئ Job Control فرستاده شده که به دنبال آن این شئ پیام Create به شئ Bookinf Form می فرستد و بدین صورت اجرای مورد کاربری آغاز می شود.




شکل ۹-۲ نمودار تعامل مورد کاربری "ثبت کار استاندارد مشتری" که واسط کاربر در آن در نظر گرفته شده است

## نمودار همکاری

در این نمودار نیز اشیاء در مستطیلها نمایش داده می شوند و فلشها نیـز نـشان دهنـده پیامهـای رد وبدل شده بین اشیاء هستند. اما در این نمودارها ترتیب انجام کار با شماره بندی پیامها دنبال مـی گـردد و شماره بندی بگونه ای انجام می گردد که هر شماره نشان می دهد این عمل از کجا آغاز شده است.
در شکل ۹-۳ مثالی از یک نموار همکاری دیده می شود.
به طور خلاصه می توان گفت که از نموداز تعامل در نمایش رفتار اشـیاء مختلـف در درون یـک مـورد کاربری استفاده می شود. این نمودار همکاری و ارتباط بین اشیاء را بخوبی به تصویر می کـشد امـا وارد جزئیات رفتار اشیاء نمی گردد. برای نمـایش جزئیـات رفتـار یـک شـئ در یـک مـورد کـاررری بایـد از مدلسازی حالت استفاده شود.




شکل ۹-۳ نمونه ای از یک نمودار همکاری

# ۱۰- مدلسازی حالت[۱]

تا اینجا بیشتر تکنیکهای ارائه شده روی بیان ویژگی های بیرونی اشیاء و روابط آنها متمرکز بوده و لی به ویژگیهای داخلی آنها کمتر پرداخته شده است. مدلسازی حالت روی این جنبه ها تاکید می نماید.

مفهوم کلاس جنبه ایستای یک موجودیت را دربر می گیرد در حالیکه مفهوم حالت روی رفتار آن موجودیت تمرکز می یابد. در حقیقت می توان اشیاء را به صورت ماشین های متناهی[۲] که در هر آن در یک حالت بخصوص بسر می برند، تصور کرد. تعداد این حالتها متناهی می باشد.

بنابراین برای هر شئ یک نمودار حالت کشیده می شود که بیانگر تمام وضعیتهایی را که آن شئ می تواند بر اثر بروز رخدادهای مختلف به خود بگیرد. برای مثال شئ کتاب را در نظر بگیرید اگر به کتاب بعنوان یک FSM نگاه کنیم در می یابیم که کتاب حالتهای گوناگونی دارد: حالت اولیه: وجود کتاب درکتابخانه، حال این کتاب می تواند -بعنوان مثال- حالتهای "قرض داده شده" یا "گم شده" را بخود بگیرد. به رفتار یک شئ در زمانهای متفاوت در یک سیستم چرخه حیات شئ[۳] در آن سیستم گفته می شود.

## مفاهیم کلیدی

قبل از بیان تکنیکهای مدلسازی حالت به بیان مفاهیم کلیدی آن می پردازیم:

- حالت[۴]: برای هر شئ تعدادی متناهی از حالتها و جود داشته که از صفات شئ و روابط شئ با اشیاء دیگر برای نمایش حالت آن استفاده می گردد. بنابر این مقادیر فعلی صفات و روابط شئ بیانگر حالت فعلی آن می باشد. نکته جالب این است که شئ اشیاء اصولاً به تاریخچه خود حساسیت[۵] دارند بدین معنی که اگر یک شئ را ایجاد نمایید و روی آن اعمالی انجام دهید عمل بعدی از نتیجه اعمال گذشته -که در متغیرهای حالت ظاهر می شود- متاثر خواهد بود.

---

[۱] State Modeling

[۲] Finite State Machines(FSM)

[۳] Object Life Cycle

[۴] State

[۵] History Sensitive

§ انتقال[1]: یک انتقال عبارت از تغییری در حالت شئ که بوسیله یک موثر 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 می شود. این عامل(موثر) می تواند درخواست انجام یک عمل، یک رویداد، یک شرط نگهبان و یا ترکیبی از اینها باشد.

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

§ متغیرهای حالت[3]: مجموعه متغیرهایی که یک حالت را نشان می دهند.

## تکنیکهای مدلسازی حالت

تکنیک کلی این است که نخست دنبال اشیاء کلیدی رفته و تلاش می نماییم حالات گوناگون شئ را پیدا کنیم. باید دنبال الگوهای ساده از رویدادها جستجو نماییم سپس-به صورت تدریجی- الگوهای جایگزین مانند خطاها و استثناها را مطرح می کنیم. توجه داشته باشید که یک شئ در طول چرخه حیات خود می تواند 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 می کنیم.

## مثال: تعمیرگاه

در این مثال کلاس کلیدی همان کلاس کار می باشد. با توجه به عملکرد سیستم رویدادهای خارجی و موارد کاربری مربوط به آنها را شناسائی می نماییم (شکل ۱۰-۱)

| 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 ۱۰-۱

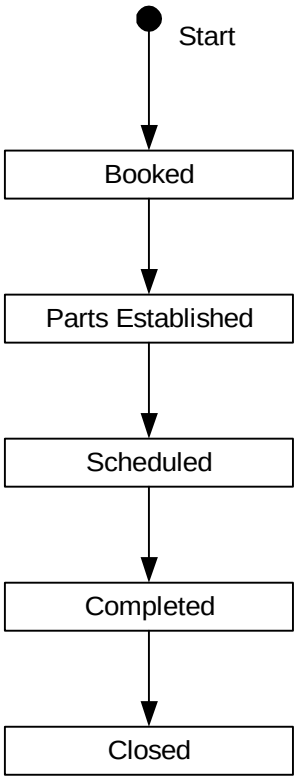
توجه داشته باشید که رویداد Parts Requested هنگامی رخ می دهد که کنترل کننده قطعات اعلام نیاز نماید(زیرا قطعات در تعمیرگاه وجود ندارند). نمودار اولیه حالت در شکل ۱۰-۲ نمایش شده

[1] Transition

[2] Guarded Conditions

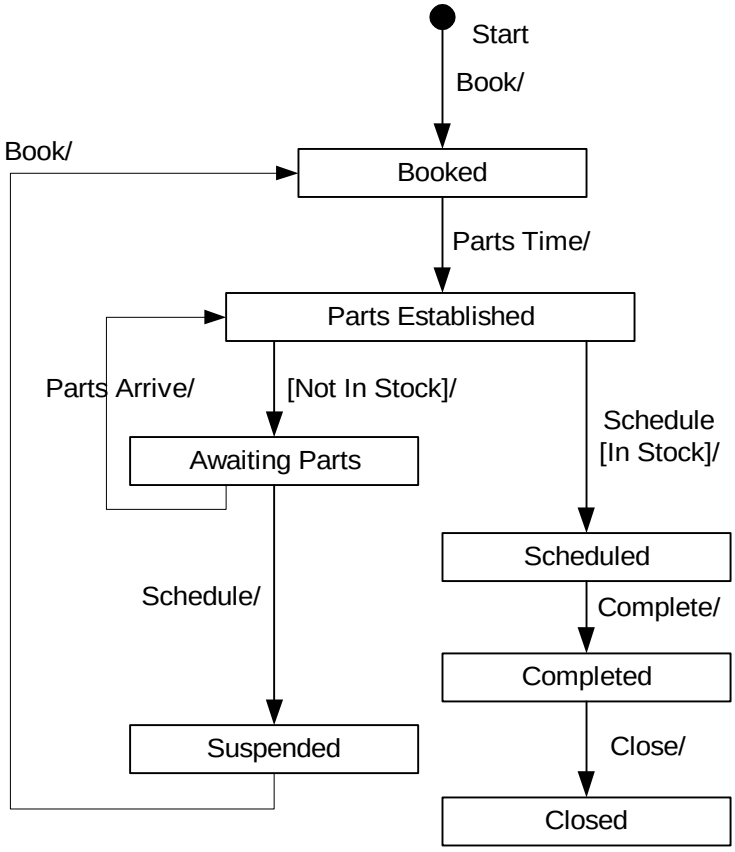
[3] State Variables

است. در این سیستم فرض بر این است که "کار" هنگامی زمانبندی می گردد 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 می نماییم (بین Parts Established و Scheduled). همچنین از یک شرط نگهبان دیگر [Not in Stock] استفاده می نماییم. (بین Parts Established و Awating Parts) (شکل ۱۰-۳) نکته آخر این است که رویداد Parts Requested روی حالت شئ Job اثر نمی گذارد. در واقع این شئ به رویداد دیگری علاقمند می باشد 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 می گردد.




شکل ۱۰-۲ نمودار اولیه حالت برای شئ 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.

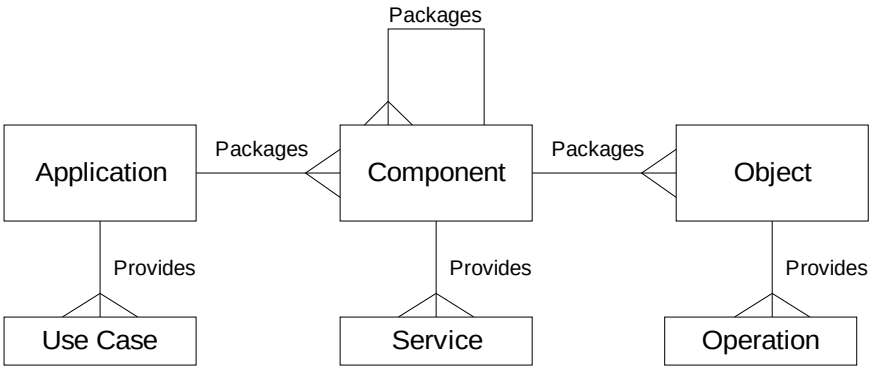



شکل ۱۰-۳ نمودار نهائی حالت برای شئ کار

# ۱۱- مدلسازی مولفه ها[۱]

یک مولفه عبارتست از قطعه ای-معمولا دودویی[۲]- از کد که پیاده سازی آن مخفی بوده و می توان از آن در نرم افزارهایی متفاوت-بوسیله واسط آن- استفاده نمود. بعبارت دیگر مولفه در نرم فزار متناظر با IC در سخت افزار می باشد.

شکل ۱۱-۱ رابطه بین یک برنامه کاربردی،و موارد کاربری، مولفهای و اشیاء آن ر ا نمایش می دهد.




شکل ۱۱-۱

همانطوریکه گفتیم روش موارد کاربری برای استخراج و نمایش نیازهای وظیفه مندی یک سیستم به کار می رود. بنابراین یک برنامه کاربردی از تعدادی موارد کاربری-که در مدل موارد کاربری دیده می شوند- تشکیل می شود. حال، بهترین شیوه پیاده سازی این موارد کاربری چیست؟

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

متدولوژیهای شئ گرا با مطرح کردن مفهوم کلاس تلا ش نمودند به این سوال پاسخ دهند. همچنین به نظر سردمداران تکنولوژی شئ گرائی کلاس واحد استفاده مجدد-که بزرگترین مزیت این روش حساب می شد-نیز هست. به نظر می رسد که در سیستم های نرم افزاری 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 پیاده سازی نیاز است. نکته دوم این است که مکانیزم های استفاده مجدد که در

[۱] Components Modeling

[۲] Binary

زبانهای OOP مطرح است محدود به برنامهای خود زبان می باشند. بدین علت OOP در مجسم کردن ایده IC نرم افزاری ناموفق عمل کرده است(البته تا حدودی).

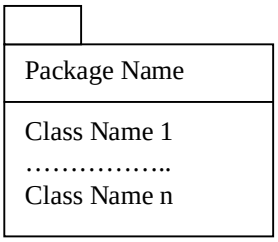
با توجه به این مقدمه ارزش مولفهای معلوم می گردد چراکه مولفه ها با گروه بندی کلاسهای منطقاً مرتبط در سطح طراحی(Packages) و پیاده سازی(Components) ساختار حجیمتر مورد نیاز را فراهم کرده و با پیروی از یک استاندارد خاص[1] در تعریف واسطهای خود به ایده IC نرم افزاری جامه عمل پوشانده است.

بدین صورت می توان گفت که بهتر است موارد کاربری به صورت مولفه های قابل استفاده مجدد(که فراهم کننده سرویس هستند) پیاده سازی شوند. هر مولفه از تعدادی کلاس منطقا مرتبط[2] تشکیل می شوند. کلاس فراهم کننده اعمال می باشند.

## نمودار بسته ها

در UML مولفه ها بوسیله مفهوم بسته ها(Packages) ابل نمایش می باشند. ایده بسته بندی را می توان بر روی همه عناصر UML بکار برد. اما بیشترین کاربرد آن بر روی کلاسها می باشد.

نماد Packageها در UML به صورت زیر می باشد:




شکل ۱۱-۲

مفهوم وابستگی[3] بین بسته ها: اگر بسته ای از سرویس های بسته دیگری استفاده نماید، گویند بسته اولی به بسته دوم وابسته است. این وابستگی با یک فلش ( ◄┈┈┈┈ ) جهت دار نمایش داده می شود.

مدلسازی مولفه ها شامل کشیدن نمودار بسته ها و نمودار استقرار مولفه ها می باشد.

مثال: تعمیرگاه: سرویس کاری نگهداری ماشین(Vehicle Maintenance or VM ) را در نظر بگیرید:

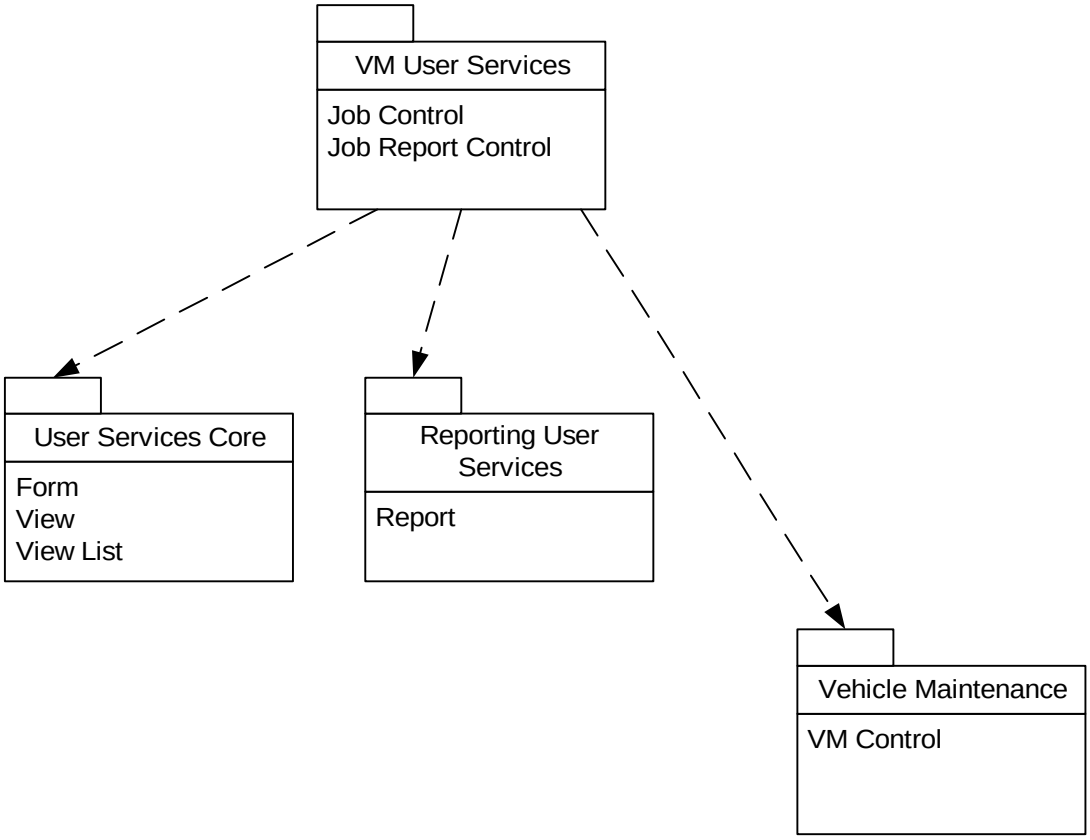
أین سرویس بوسیله یک بسته فراهم می گردد. در شکل زیر سرویس کاربر مربوط به این سرویس کاری به صورت مجموعه ای از بسته ها نمایش داده شده است.

---

[1] مانند CORBA و COM

[2] Logical Related (Cohesive) Classes

[3] dependency



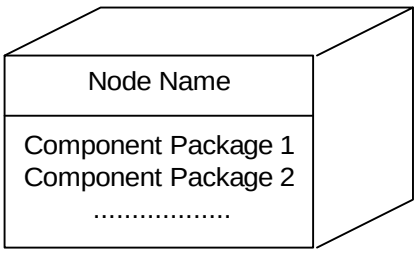

شکل ۱۱-۳

در شکل ۱۱-۳ دیده می شود که بسته VM User Services به سه بسته User Services Core، Reporting User Services و Vehicle Maintenance وابستگی دارد. هرگاه بین دو کلاس از دو بسته، وابستگی وجود داشته باشد. بین آن دو بسته وابستگی وجود خواهد داشت.

## نمودار استقرار

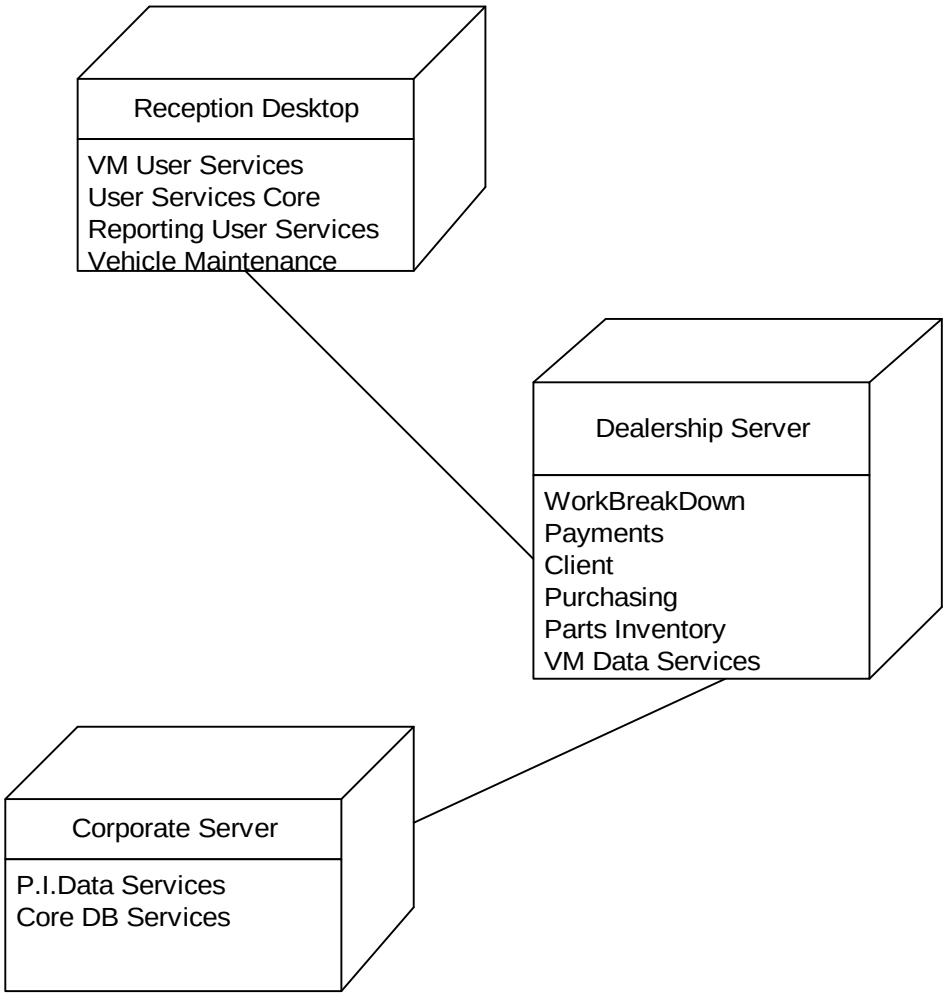
این نمودار ارتباطات فیزیکی ما بین اجزاء سخت افزار و نرم افزاری یک سیستم را نشان می دهد. یک نمودار استقرار برای نشان دادن چگونگی پراکندن اشیاء روی گره های یک سیتسم توزیع شده بکار می رود.

در UML گره ها با نماد زیر نشان داده می شوند:




شکل ۱۱-٤ نمایش یک گره در UML

مثال: بسته های مثال قبلی در نمودار استقرار زیر نمایش می شوند:




شکل ۱۱-۵ نمودار استقرار سیستم نگهداری خودرو(شامل سرویس و تعمیر و ...)