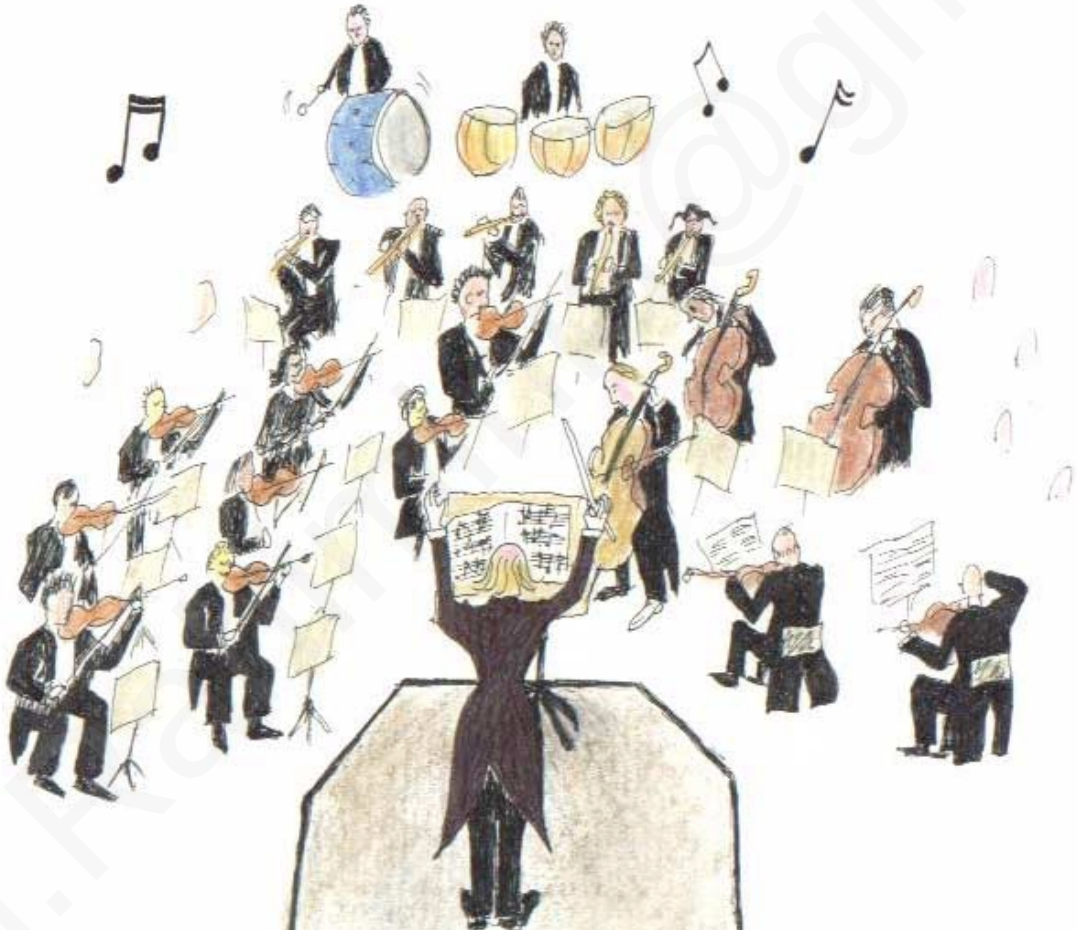
کاربرد UML

تحلیل و طراحی شیء گرا با استفاده از UML



بسمه تعالی

مقدمه مترجم

تعداد کمی از کتابهای مهندسی نرم افزار به زبان فارسی موجود است و اکثر آن تعدادی هم که هستند بسیار کلی و غیر کاربردی هستند و بیشتر آنها توسط مترجمین فاقد تخصص در زمینه مهندسی نرم افزار فقط ترجمه شده اند، بنابراین تصمیم گرفتم یکی از کتابهایی که خودم در این زمینه مطالعه کردم که همچنین برای شروع نیز بسیار عالی است را به فارسی روان ترجمه کنم.

لازم به ذکر است که من نه مترجم هستم نه نویسنده و فقط در زمینه کامپیوتر و برنامه نویسی تجربه دارم بنابراین این اثر حتما دارای مشکلات نگارشی است، به همین دلیل از شما خواهش دارم که نظرات و انتقادات و اشکالات این کتاب را برای من ارسال کنید.

چه کسانی این کتاب را بخوانند؟

- کسانی که قصد آشنایی با زبان مدل سازی **UML** را دارند.
- کسانی که قصد آشنایی با جدید ترین فرایند تولید نرم افزار (**RUP**) را دارند.
- کسانی که قصد آشنایی با نحوه تحلیل و طراحی نرم افزارهای شیءگرا را دارند.
- کسانی که قصد کوچ از برنامه نویسی تابعی به برنامه نویسی شیءگرا را دارند.
- برنامه نویسانی که بدون تحلیل و طراحی وارد مرحله کدنویسی می شوند.
- و ....

نحوه مطالعه کتاب

پیشنهاد میکنم هرگاه در جلوی کلمات فارسی معادل انگلیسی آنها را در پرانتز آوردم شما همان کلمات انگلیسی را برای آن مفهوم یاد بگیرید به طوری که با دیدن کلمه فارسی معادل انگلیسی آن در ذهن شما نقش ببندد. به دو دلیل حتماً این کار را انجام دهید : اول اینکه من کتاب را برای درک بهتر شما به صورت کلمه به کلمه عیناً ترجمه نکردم و ممکن است مفهوم کلی کلمات انگلیسی داخل پرانتز را به فارسی گفته باشم، و دوم اینکه معادل فارسی **مشخص و واحدی برای آن کلمات وجود ندارد. این کلمات انگلیسی در اوایل کتاب هرجا که لازم بود آمده اند ولی در اواسط کتاب فرض بر این است که شما با همه آنها آشنا هستید و دیگر در آنجا معادل انگلیسی را برای حفظ خوانایی کتاب تکرار نکرده ام.**

این کتاب در حجم کمی تمام فرایند تولید نرم افزار را همراه با مفاهیم شیءگرایی پوشش داده است. از این رو مطالعه دقیق و جمله به جمله کتاب بسیار مهم است.

این کتاب فقط کلیات مربوط به هر قسمت از فرایند توسعه نرم افزار را گفته است. اطلاع از این کلیات برای هر شخصی که قصد شرکت در گروه های مهندسی نرم افزار را دارد لازم است، اما به هیچ عنوان کافی نیست. شما برای شرکت در یک گروه تولید نرم افزار ضمن اینکه باید کلیات هر قسمت را بدانید باید در یک قسمت نیز متخصص باشید.

فهرست

- استراتژی شیء گرا
- خلاصه

درس چهارم : خلاصه ای از UML

- نمودار مورد کاربرد (Use Case Diagram)
- نمودار کلاس (Class Diagram)
- نمودار همکاری (Collaboration Diagram)
- نمودار ترتیب (Sequence Diagram)
- نمودار وضعیت (State Diagram)
- نمودار بسته یا بسته بندی (Package Diagram)
- نمودار اجزاء (Component Diagram)
- نمودار گسترش (Deployment Diagram)
- خلاصه

درس پنجم : فاز دریافت (INCEPTION PHASE)

درس ششم : فاز جزئیات (ELABORATION PHASE)

- **موارد قابل تحویل**
- **خلاصه**

درس هفتم : ساخت نمودار مورد کاربرد

- بازیگران (Actors)
- هدف موارد کاربرد (Use Case)
- تکه تکه بودن مورد کاربرد (Use Case Granularity)
- توصیف موارد کاربرد (Use Case Descriptions)
- موارد کاربرد (Use Case) در فاز جزئیات (Elaboration)
- کشف موارد کاربرد (Use Case)

- کارگاه طرح ریزی نیازمندی ها (Joint Requirements Planning Workshops (JRP))
- مشورت و تبادل اندیشه
- خلاصه

باقی کتاب در دست ترجمه است.

درس اول : مقدمه ای بر UML

UML چیست؟

زبان مدل سازی UML (**Unified Modelling Language**) زبانی گرافیکی است، و قواعدی دارد که ما میتوانیم به وسیله این قواعد، عناصر بزرگ سیستم های نرم افزاری را توصیف کنیم (در **UML** به این توصیفات محصول (**Artifacts**) میگویند). در این کتاب، مفاهیم اصلی **UML** را شرح میدهیم، و همچنین خواهیم دیدکه در پروژه های نرم افزاری **UML** چگونه به کار میرود.
UML کاملاً بر اساس توسعه نرم افزار به صورت شیء گرا است، بنابر این ما در این کتاب بعضی از قوانین مهم شیء گرایی را نیز برسی خواهیم کرد.
در این درس نگاهی کوتاه به تاریخچه **UML** خواهیم داشت، و همچنین در باره نیاز امروز به وجود یک زبان مشترک در صنعت نرم افزار بحث خواهیم کرد. و بعد خواهیم دید که **UML** در پروژه های نرم افزاری چگونه نقش بازی میکند.

یک زبان مشترک

تمام صنعت ها برای خودشان یک زبان مشخصی دارند، مثلاً یک انتگرال در ریاضیات شکل خاصی دارد.

$$\int_0^{\infty} 1/x^2 \, dx$$

انتگرال در ریاضی

تمام ریاضی دانان جهان با دیدن این شکل فورا متوجه میشوند که من یک انتگرال را نشان داده ام. اگرچه این شکل بسیار ساده است ولی مفاهم عمیق و پیچیده ای دارد. بنابر این نمادسازی بسیار ساده است و تمام ریاضی دانان جهان میتوانند به وضوح و بدون هیچ گونه ابهامی با این شکل رابطه برقرار کنند. ریاضی دانان زبان مشترکی برای انتقال مفاهیم دارند درست مثل موسیقی دانان، مهندسان الکترونیک، مهندسین ساختمان و تعداد زیادی از دیگر حرفه ها.

تا امروز زبان مشترک بین مهندسین نرم افزار ناقص است. از سال 1989 تا 1994 دوره ای بود که به آن جنگ متد ها می گفتند، بیشتر از 50 زبان مدل سازی نرم افزار وجود داشت که هر کدام شکل و شمایل مخصوص به خود داشتند و هیچکدام از آنها کامل نبودند. در اواسط سال 1990 سه متد مدل سازی نرم افزار به عنوان قویترین متد ها معرفی شدند. که هر کدام از این متد ها مزیت متفاوتی نسبت به دو متد دیگر داشت. این سه متد و مزیت هایشان در زیر آمده است :

- **Booch** : برای طراحی و مدل سازی عالی بود. ابداء کننده این متد یعنی **Grady Booch** با زبان **Ada** کار میکرد که باعث شد این متد را ابداء کند. اگرچه این متد قوی بود ولی علامتهای نشانه گذاری بسیار ضعیفی داشت (تعداد زیادی از شکلهایی شبیه به توده های ابر – که خیلی زیبا نبود)
- **OMT (Object Modelling Technique)** : برای تحلیل و سیستم های اطلاعاتی بهترین بود.
- **OOSE (Object Oriented Software Engineering)** : یک مدل شبیه به مدل مورد کاربرد (**Use Case**) بود. موارد کاربرد (**Use Cases**) تکنیک قدرتمندی برای فهم رفتار کل سیستم است.

در سال 1994 سازنده متد **OMT** یعنی **Jim Rumbaugh** و قتی الکترونیک را رها کرد و به **Grady Booch** در شرکت **Rational** پیوست جامعه مهندسین نرم افزار حیرت زده شدند. هدف مشارکت این دو نفر این بود که ایده هایشان را ترکیب کنند و در قالب یک متد متحد ارائه دهند، که در آن زمان به متدی که این دو نفر ابداء کردند نام **Unified Method** را اختصاص دادند.

در سال 1995 سازنده **OOSE** یعنی **Ivar Jacobson** نیز به **Rational** پیوست و ایده های او نیز به خورد متد **Unified Method** داده شد و نهایتاً متد به وجود آمده **Unified Modelling Language** نام گرفت. (این گروه سه نفره که بنیان گذاران **UML** هستند به سه تفنگ دار معروف شدند).

رفته رفته این متد جدید در صنعت نرم افزار محبوبیت شد و کنسرسیوم **UML** رسمیت پیدا کرد که شرکتهای **Hewlett-Packard** و **Microsoft** و **Oracle** عضو این کنسرسیوم هستند.

خلاصه

UML یک زبان گرافیکی برای جمع آوری محصولات توسعه نرم افزار است.

این زبان نشانه ها را برای ما فراهم کرده تا مدل ها را تولید کنیم.

UML به عنوان تنها زبان صنعت نرم افزار تائید شده است.

UML در اصل توسط سه تفنگ دار، در شرکت **Rational** طراحی شده است.

این زبان بسیار توانمند است و تمام مفاهیم مهندسی نرم افزار را پشتیبانی میکند.

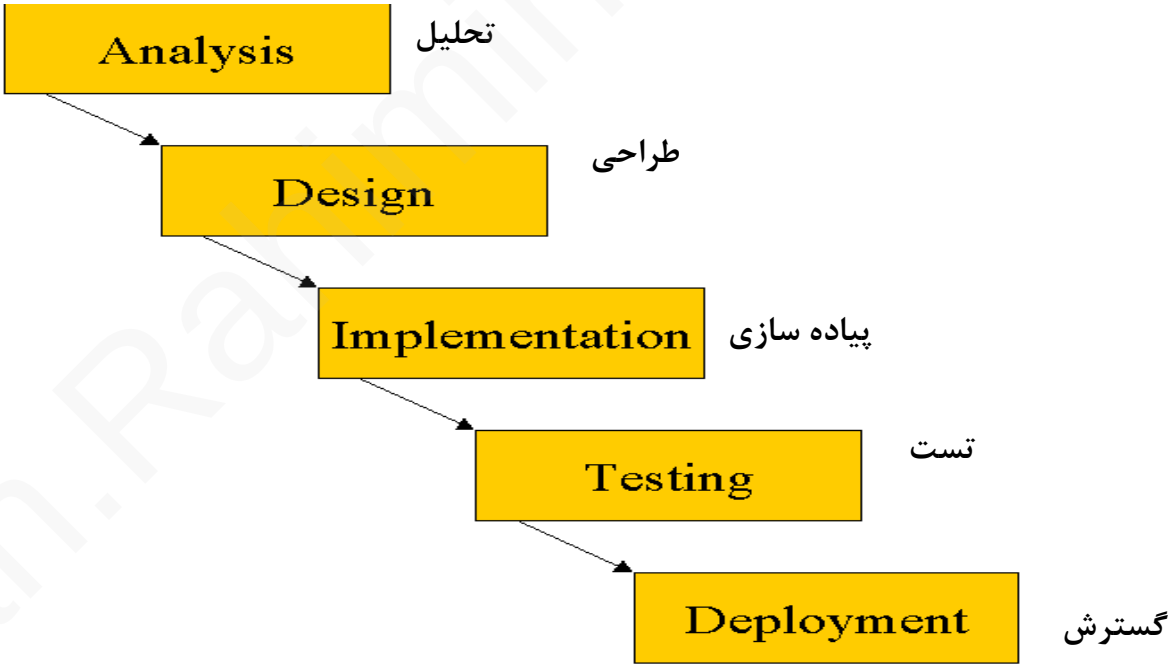
درس دوم : UML همراه با فرایند توسعه

UML برای نمادسازی

سه تفنگ دار وقتی در حال توسعه UML بودند تصمیم قاطعی گرفتند که تمام موارد مربوط به فرایند توسعه نرم افزار را از این زبان حذف کنند. چون فرایند ها بسیار ستیزگر هستند – کسی که برای کمپانی A کار میکند مسلماً از کمپانی B بد میگوید. بنابر این UML زبانی عمومی است که ما را قادر میسازد تا بتوانیم وضعیت توسعه نرم افزار را بر روی کاغذ بیاوریم.

در جاهای دیگر ممکن است به UML عناوینی دیگر از جمله : زبان ساده، نماد ساز، دستور زبان و ... اختصاص دهند، شما هرچه دوست دارید آن را صدا کنید، به هر حال UML هرگز به شما نمیگوید یک نرم افزار را چگونه توسعه دهید.

برای اینکه UML را به طور موثر یاد بگیرید، ما در این کتاب از یک فرایند توسعه ساده پیروی خواهیم کرد و سعی میکنیم به شما بفهمانیم که UML چگونه در هر مرحله از فرایند توسعه نرم افزار به شما کمک خواهد کرد. برای شروع بیایید به یک فرایند معمولی توسعه نرم افزار نگاهی داشته باشیم.

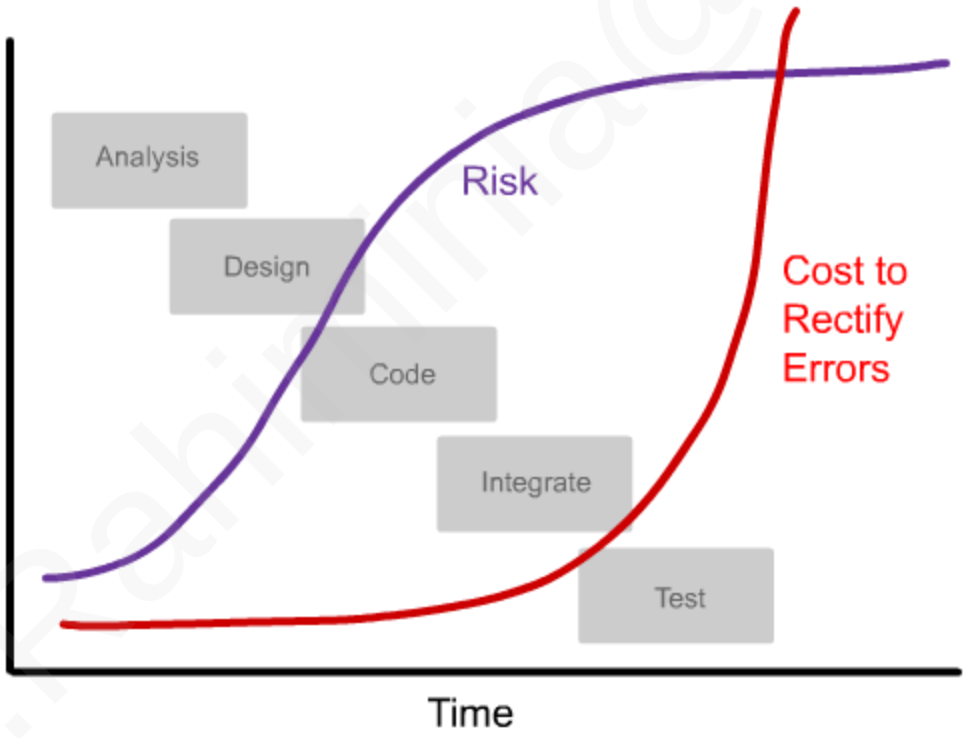


مدل سنتی آبشاری (Waterfall)

در مدل آبشاری (**Waterfall**) هر مرحله از توسعه نرم افزار باید به پایان برسد تا بتوان مرحله بعدی را شروع کرد.

این فرایند ساده (و آسان برای مدیریت) پیچیدگی و بزرگی یک پروژه بزرگ را به پنج قسمت تقسیم میکند، مشکلات اصلی این فرایند به شرح زیر است :

- حتی سیستم های بزرگ هم قبل از اینکه به مرحله طراحی (**Design**) برسند باید کاملا درک شده و تحلیل شوند، که این مسئله پیچیدگی را افزایش داده و موجب سردرگمی و فشار آمدن به توسعه گران میشود.
- نمایان شدن ریسکها به تاخیر می افتد. مشکلات بزرگ غالباً در مراحل آخر توسعه بویژه در هنگام مجتمع سازی (**integration**) سیستم پدیدار میشوند، و ممکن است هزینه برطرف کردن خطاهای طراحی بیشتر از هزینه خود طراحی شود.
- در پروژه های بزرگ هر مرحله از فرایند بسیار طولانی میشود. مثلا یک مرحله دو ساله نگهداری برای کارمندان دستورالعمل خوبی نیست.



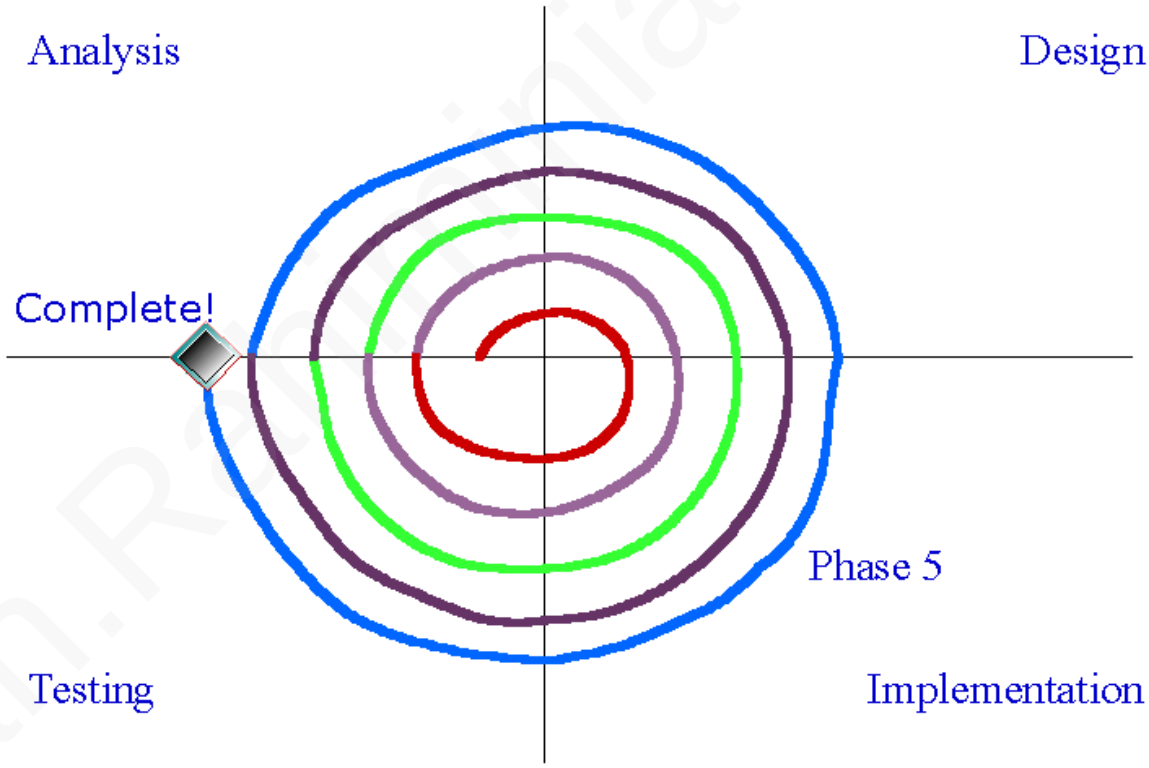
در مدل آبشاری هرچه زمان بیشتر باشد، هزینه و ریسک نیز بیشتر است.

گذشته از این، فاز تحلیل در آغاز پروژه ناقص انجام میشود و همین امر موجب پدیدار شدن ریسکهای خطرناک و عدم درک نیازمندیهای (**requirements**) کاربران نهایی پروژه میشود. در این صورت حتی اگر مراحلی طراحی (**Design**) پیاده سازی (**Implementation**) تست (**Test**) و

گسترش (Deployment) نیز به خوبی انجام شود، در نهایت محصول تولید شده آن چیزی که مشتری یا کاربران نهایی میخواستند نخواهد بود.
با تامل در پاراگراف قبل به این نتیجه میرسیم که برای تولید پروژه های کوچک هیچ چیزی در مدل آبشاری اشتباه نیست، به شرط اینکه پروژه به اندازه کافی کوچک باشد. یعنی تمام افراد گروه توسعه قادر باشند تمام جهات پروژه را بفهمند و همچنین چرخه زندگی (lifecycle) پروژه کوتاه باشد (مثلاً چند ماه)، در این صورت است که مدل آبشاری (Waterfal) یک فرایند با ارزش می شود، حد اقل بهتر از بی نظمی است.

مدل حلزونی یا مارپیچی (Spiral)

یک روند دیگر برای توسعه نرم افزار مدل حلزونی است. در این مدل ما با یک سری از چرخه های زندگی (Lifecycle) کوتا پروژه را تولید میکنیم، که در پایان هر چرخه یک نرم افزار قابل اجرا منتشر میشود.



یک فرایند حلزونی – در اینجا پروژه به پنج فاز تقسیم شده است، هر فاز بر اساس فاز قبل به همراه یک نسخه اجرایی از نرم افزار ساخته میشود.

مزیت های این روند به شرح زیر است :

- **تیم توسعه به جای اینکه چندین سال را صرف یک فعالیت کنند، میتوانند روی تمام چرخه زندگی نرم افزار یعنی تحلیل، طراحی، پیاده سازی و تست کار کنند.**
- **ما خیلی زود نظرات مشتری را راجع به پروژه دریافت میکنیم و میتوانیم مشکلات را خیلی زود تر از اینکه به مرحله پیاده سازی برسند کشف و رفع کنیم.**
- **می توانیم ریسک ها و بویژه ریسک دوباره کاری را خیلی زود تر برطرف کنیم.**
- **میتوانیم مقیاس و پیچیدگی پروژه را در اوایل کار کشف کنیم.**
- **تغییرات تکنولوژی آسانتر در پروژه جا داده میشود.**
- **نشر منظم نرم افزار موجب دلگرمی میشود.**
- **وضیعت پروژه (برای مثال – چقدر از پروژه تکمیل شده است؟) دقیق تر تشخیص داده میشود.**

معایب فرایند حلزونی به شرح زیر است :

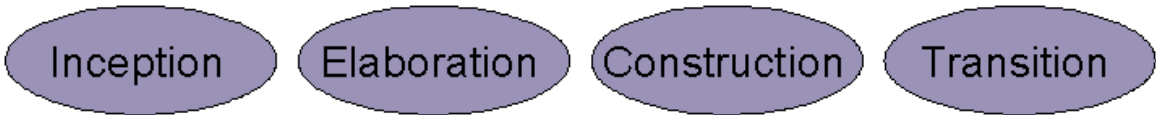
- **فرایند توسعه به طور عادی بر اساس توسعه سریع نرم افزار است یعنی در همان مراحل اول باید یک نرم افزار اجرایی تولید گردد.**
- **مدیریت فرایند توسعه خیلی مشکل است.**

برای مقابله با معایب مدل حلزونی مدل دیگری ابداء شده است که فرایند رشد تکراری (Iterative, Incremental Framework) نام دارد.

فرایند رشد تکراری (Iterative, Incremental Framework)

این مدل تغییر یافته مدل حلزونی است، اما رسمی تر از مدل حلزونی است. ما هم در باقی مانده این کتاب برای توسعه نرم افزار از این مدل پیروی خواهیم کرد.

این فرایند به چهار فاز بزرگ تقسیم می شود : فاز دریافت (Inception)، فاز جزئیات (Elaboration)، فاز ساخت (Construction) و فاز انتقال (Transition). این فازها به ترتیب انجام میگیرند، البته این فازها را با مراحل فرایند آبشاری که یکی پس از دیگری انجام میگیرد اشتباه نگیرید، در زیر هریک از این فازها و فعالیت هایی که در هر کدام انجام میگیرد را شرح داده ام.



چهار فاز فرایند رشد تکراری (Iterative, Incremental Framework)

فاز دریافت (Inception)

فاز دریافت (Inception) مربوط به تعین هدف و تعریف یک دیدگاه برای پروژه است. برای پروژه های کوچک این فاز میتواند فقط یک گفتگو همراه با نویشیدن قهوه باشد. اما برای پروژه های بزرگ کارهای زیادی لازم است. محصولاتی که در این فاز میتوانند تولید شوند به شرح زیر است :

- **مستندات مربوط به دیدگاه پروژه**
- **کشف اولیه نیازمندیهای مشتری**
- **اولین قسمت از واژه نامه پروژه (توضیحات بیشتر را خواهم داد)**
- **موارد تجاری (Business Case) (شامل معیارهای موفقیت، پیش بینی های مالی، تخمین بازگشت سرمایه و .....)**
- **تشخیص ابتدائی ریسک**
- **طرح کلی پروژه**

جزئیات فاز دریافت را در درس چهارم توضیح خواهم داد.

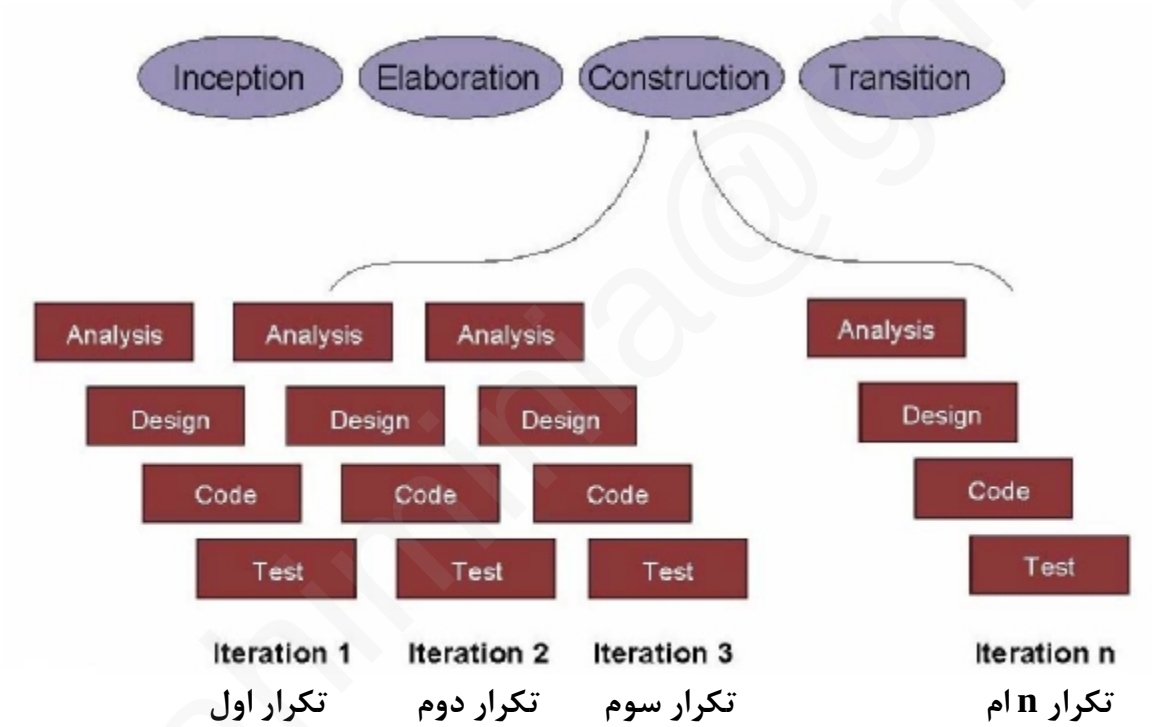
فاز جزئیات (Elaboration)

هدف فاز جزئیات (Elaboration) تحلیل پروژه، توسعه بیشتر طرح کلی پروژه و رفع ریسکهای قسمتهایی از پروژه که ریسک بالایی دارند. در پایان فاز جزئیات ما باید یک درک کلی از کل پروژه داشته باشیم.

در این مرحله دو نمودار از نمودارهای موجود در UML بسیار با ارزش هستند. نمودار مورد کاربر (Use Case) در فهم نیازمندیهای مشتری به ما کمک میکند، و همچنین برای شرح مفهامی که مشتری ما آنها را بفهمد از نمودار کلاس (Class Diagram) استفاده میکنیم. در مورد این دو نمودار توضیحات بیشتر را بعداً خواهم داد.

فاز ساخت (Construction)

در این فاز محصول مورد نظر ساخته میشود. کارهایی که در این فاز از پروژه انجام میشود به جای اینکه به شکل مدل خطی انجام گیرد، به ترتیب مدل حلزونی بر اساس یکسری از تکرارها (Iteration) انجام میگیرد. روش انجام هر تکرار (Iteration) به صورت یک مدل ساده آبشاری است. با حفظ کوچکی و امکان پذیر بودن انجام هر تکرار مشکلات مدل آبشاری دیگر نادیده گرفته میشود، چون همانطور که قبلاً گفتم مدل آبشاری برای توسعه پروژه های کوچک خوب است، در مدل تکراری ما یک پروژه را به تکرارهایی (Iteration) تقسیم میکنم و هر تکرار را بر اساس مدل آبشاری توسعه میدهیم. برای درک بیشتر این مفاهیم به تصویر زیر دقت کنید :



فاز ساخت (Construction) شامل یکسری از مدلهای آبشاری کوچک (Mini Waterfall) است. توجه داشته باشید که در فاز دریافت (Inception) و فاز جزئیات (Elaboration) فقط نمونه های اولی ساخته میشوند، این نمونهای اولیه نیز میتوانند درست مثل فاز ساخت توسط تکرارهایی از مدلهای آبشاری کوچک (Mini Waterfall) ساخته شوند، اما ما در این کتاب فاز دریافت و جزئیات را ساده کردیم و فقط در فاز ساخت از مدل آبشاری استفاده کردیم.
هدف ما در پایان هر مجموعه از تکرارها رسیدن به یک سیستم اجرایی است. (اگرچه در مراحل اول فقط یک سیستم خیلی محدود). به این تکرار ها رشد (Increments) یا استخوان بندی (Framework) هم میگویند.

فاز انتقال (Transition)

در این فاز محصول تولید شده به مشتری انتقال میابد. فعالیت هایی که در این فاز انجام میگیرد به شرح زیر است :

- **انتشار نسخه آزمایشی (Beta) برای تست محصول تولیدی توسط تمام کاربران سیستم.**
- **تست کارخانه ای، یا تست کردن سیستم به صورت همزمان با سیستمی که قرار است محصول تولیدی جایگزین آن شود.**
- **گرفتن داده ها (مثلاً تبدیل پایگاه داده موجود به فرمت جدید، وارد کردن داده ها در پایگاه داده و ...)**
- **آموزش کاربران جدید سیستم.**
- **بازاریابی، پخش و فروش**

فاز انتقال را با فاز تست در مدل آبشاری اشتباه نگیرید. در ابتدای فاز انتقال یک محصول کامل و تست شده و قابل اجرا برای کاربران وجود دارد. همانطور که در لیست بالا می بینید بعضی از پروژه ها ممکن است به تست بتا نیاز داشته باشند، اما قبل از رسیدن به این مرحله محصول تولیدی باید تماماً کامل باشد.

چند تکرار؟ و هرکدام از آنها چقدر باید طول بکشد؟

یک تکرار عموماً باید بین دو هفته و دو ماه باشد. بیشتر از دو ماه موجب افزایش پیچیدگی در تکرار میشود. یک پروژه بزرگ و پیچیده لزوماً نباید تکرار های طولانی تری داشته باشد، در عوض میتواند تعداد تکرار های بیشتری داشته باشد، چون تکرار های طولانی مقدار پیچیدگی که توسعه گران مجبورند در یک زمان بر عهده بگیرند افزایش می دهد.

عواملی که در افزایش مدت زمان تکرار موثر هستند به شرح زیر است :

- صرف زمان کم برای مراحل اول توسعه. صرف زمان بیشتر در مراحل اول توسعه به توسعه گران این فرصت را می دهد که بر روی تکنولوژی های جدید کار کنند، یا زیرساخت های پروژه را تعریف کنند.
- توسعه گران تازه کار
- گروه های توسعه همزمان
- گروه های غیر متمرکز

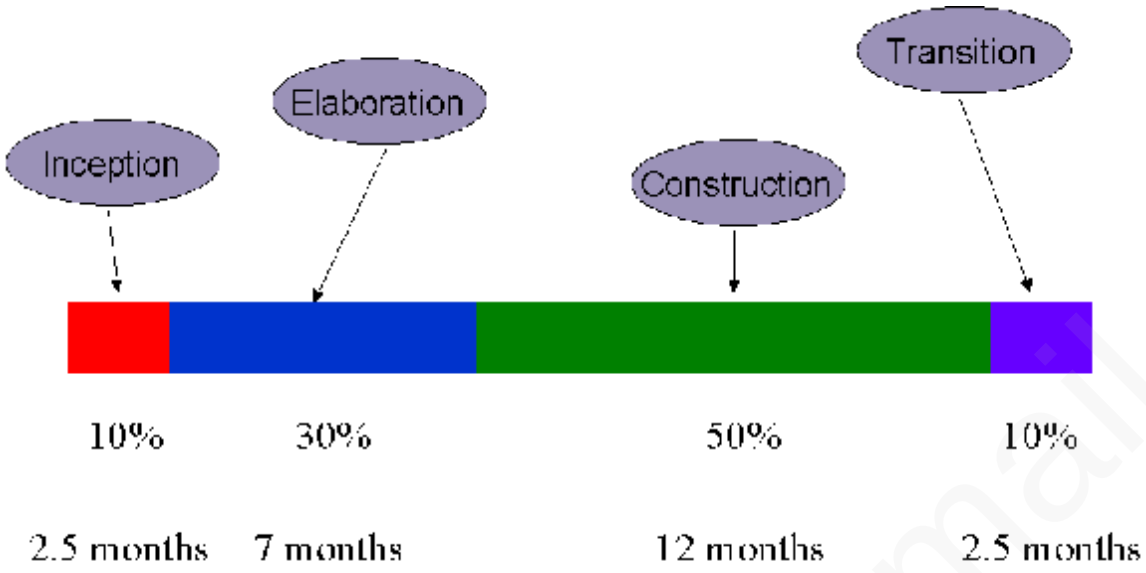
زمان بندی پروژه (Time Boxing)

یک روند برای مدیریت رشد تکراری، زمان بندی است. این روند یک روند سخت گیر است که برای هر تکرار، دوره زمانی ثابتی معین میکند که در این دوره زمان آن تکرار باید کامل شود.
چه یک تکرار در دوره زمانی خودش کامل شود چه کامل نشود، در هر صورت آن تکرار به پایان رسیده است. در پایان هر تکرار دلایلی که باعث تاخیر در تکامل تکرار شده است باید بررسی شود و برای تکمیل شدن تکرار دوباره زمان بندی کرد.
پیاده سازی زمان بندی مشکل است، و نظم و ترتیب زیادی را در کل پروژه نیاز دارد. وقتی یک تکرار 99% کامل میشود خیلی وسوسه انگیز است که از حد و مرز زمان بندی تجاوز کنیم. حتی اگر یک بار از این وسوسه بر ما قلبه کند و از مرز زمان بندی تجاوز کنیم دیگر در کل پروژه نظم و ترتیب رعایت نمی شود.
بعضی از مدیران تصور می کنند که زمان بندی انعطاف را از بین می برد، اما این طور نیست. اگر یک تکرار در دوره زمانی خودش تکمیل نشود باید آن را در تکرار های بعد تکمیل کرد و طرح کلی تکرار ها را دوباره طرح ریزی کرد، که خود این کار ممکن است موجب تغییر در زمان تحویل یا حتی اضافه کردن تکرارهای بیشتر شود. به هر حال زمان بندی مزیت هایی نیز دارد که به شرح زیر است :

- برنامه ریزی سخت وادار میکند که پروژه به موقع پیشرفت کند. زمان بندی رها نمیشوند و پروژه از مسیر خود منحرف نمی شود.
- اگر توسعه گران از عواقب عدم رعایت زمان بندی که ممکن است باعث شکست پروژه شود، ترس داشته باشند کارشان را به موقع انجام می دهند. البته به شرط اینکه زمان بندی درست انجام شده باشد.

الگوی زمان بندی پروژه

هر کدام از چهار فاز توسعه نرم افزار چقدر باید طول بکشد؟ این کاملا به پروژه وابسته است، البته میشود یک نمونه کلی ارائه کرد، مثلاً 10% فاز دریافت، 30% فاز جزئیات، 50% فاز ساخت و 10% فاز انتقال.



زمان بندی برای هر فاز – این مثال طول هر فاز را برای یک پروژه دو ساله نشان می دهد.

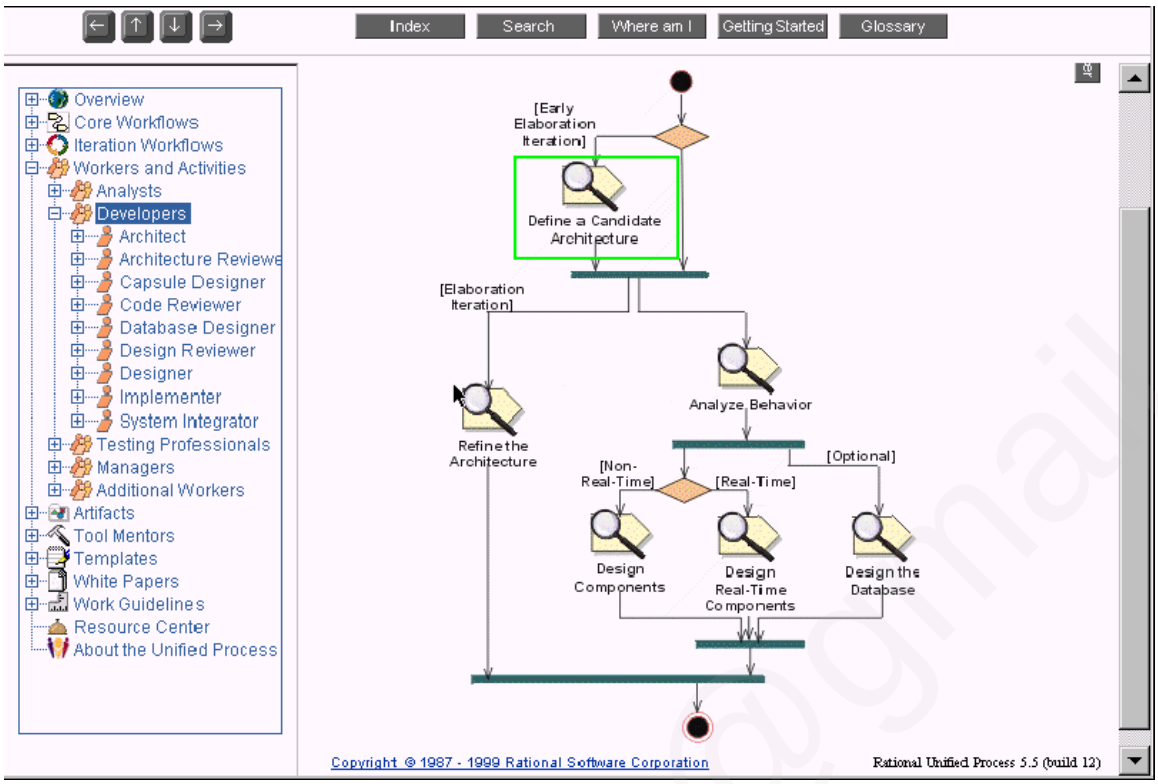
فرایند عقلانی توسعه نرم افزار (Rational Unified Process)

این فرایند که به اختصار به آن RUP می گویند. مشهور ترین نمونه از چرخه رشد تکراری است که همکنون به کار برده می شود. RUP توسط همان سه تفنگ دار که UML را توسعه دادند، توسعه داده شده است. بنابر این RUP و UML مکمل یکدیگر هستند.

اصلاً از نظر شرکت Rational تمام پروژه ها با هم متفاوت هستند، و هر کدام نیاز های متفاوتی دارند. مثلاً یک پروژه فاز دریافت کوتاهی دارد ولی ممکن است فاز دریافت پروژه دیگر حتی چند سال طول بکشد.

خلاصه RUP بسیار انعطاف پذیر است و قادر است هر فاز از فرایند را سفارشی کند. RUP همچنین نقش هر فرد در پروژه را به دقت در اشکالی که کارمندان (Workers) نام دارند تعیین می کند.

درضمن شرکت Rational محصولی را تولید کرده است که به توسعه پروژه ها بر مبنای RUP کمک میکند، برای اطلاعات بیشتر می توانید به سایت www.Rational.com مراجعه کنید.



تصویری از RUP 2000

مزیت ها و معایب RUP از حوزه این کتاب خارج است، هرچند در تمام این کتاب از هسته RUP یعنی چرخه رشد تکراری (Iterative, Incremental Lifecycle) برای کاربرد نمودارهای UML پیروی میشود.

خلاصه :

فرایند رشد تکراری مزیت های زیادی نسبت به فرایند های سنتی دارد.
این فرایند به چهار قسمت تقسیم شده است دریافت – جزئیات – ساخت – انتقال
رشد تکرای به این معنی است که هدف، بعد از چند تکرار، تولید کد اجرائی است.
تکرارها می توانند زمان بندی شوند.
باقیمانده این کتاب متمرکز است بر فرایند رشد تکراری و اینکه UML چگونه موارد قابل تحویل هر فاز از فرایند را تولید میکند.

درس سوم : شیء گرایی

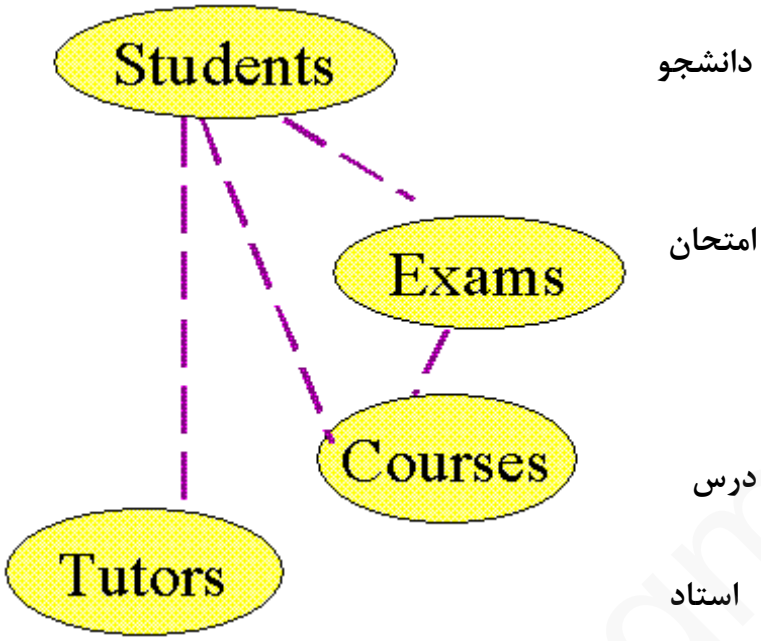
در این درس نگاهی به مفاهیم شیء گرایی (**Object Orientation (OO**) خواهیم انداخت. **UML** برای پشتیبانی از شیء گرایی طراحی شده است، و ما هم قبل از انیکه به سراغ خود **UML** برویم، ابتدا در این درس مفاهیم شیء گرایی را توضیح خواهیم داد.

برنامه نویسی ساخت یافته (Structured Programming)

اول از همه بیایید ببینیم سیستم های نرم افزاری توسط ساختار ها (**Structure**) طراحی میشوند (بعضی مواقع تابعی (**Functional**) به آن تابعی هم می گویند).
روش متداول در برنامه نویسی ساخت یافته این بود که به مسئله نگاه می کردند و مجموعه ای از توابع، که نیازمندیهای مسئله را انجام میدادند طراحی می شد. اگر این توابع بیش از حد بزرگ میشد، آنها را به قسمت های کوچکتر تقسیم میکردند. به این فرایند تجزیه تابعی (**Functional decomposition**) میگویند.
بیشتر تابع ها برای کار کردن به انواعی از داده ها نیاز دارند. در سیستمهای تابعی این داده ها یا در یک پایگاه داده یا در متغیر های سراسری (**global variables**) نگهداری میشوند.
برای مثال یک سیستم مدیریت دانشگاه را در نظر بگیرید، این سیستم جزئیات هر دانشجو و هر استاد و همچنین دروس ارائه شده در دانشگاه و دروس انتخاب شده توسط دانشجویان را نگهداری میکند.
یک طراحی تابعی ممکن است موجب به وجود آمدن این توابع شود :

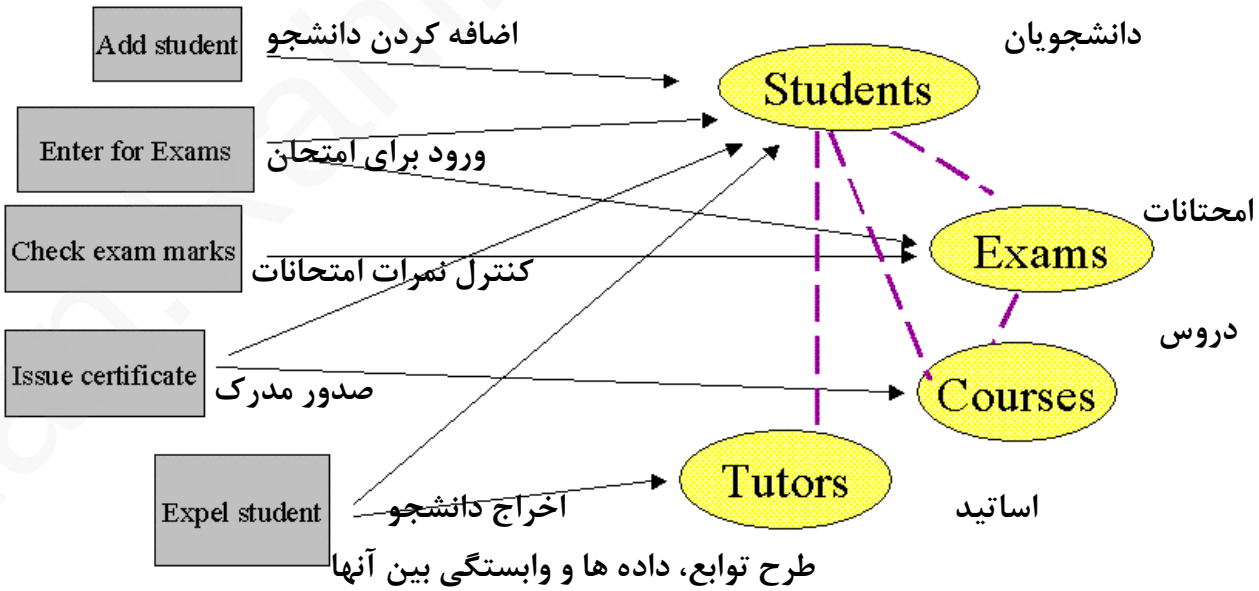
- اضافه کردن دانشجو
- ورود برای امتحانات
- کنترل نمرات امتحانات
- صدرو مدرک
- اخراج دانشجو

ما برای پشتیبانی از این توابع به یک مدل داده ای نیز نیازمندیم. ما باید اطلاعات مربوط به دانشجویان، اساتید، امتحانات و دروس را نگهداری کنیم، بنابر این یک طرح برای پایگاه داده ایجاد میکنیم تا این اطلاعات را برای ما نگهداری کند.



الگوی ساده ای از پایگاه داده – خطچین ها داده های مربوط به هم را نشان می دهد – مثلا هر دانشجو با یک یا چند درس مرتبط است

توابعی که در بالا تعریف کردیم کاملا به این توابع وابسته هستند، مثلا تابع اضافه کردن دانشجو باید لیست دانشجویان را تغییر دهد، تابع صدور مدرک باید به داده های دانشجویان و نتایج امتحانات دسترسی داشته باشد،
شکل زیر طرح توابع و داده ها و ارتباطات بین آنها را نشان می دهد :



طرح توابع، داده ها و وابستگی بین آنها

مشکل برنامه نویسی ساخت یافته این است که اگر مسئله پیچیده باشد نگهداری سیستم هم سخت تر و سخت تر میشود. در مثال بالا اگر یکی از نیازمندیها تغییر کند و باعث تغییر در نحوه ذخیره شدن اطلاعات دانشجویان شود، چه اتفاقی می افتد؟

به عنوان مثال فرض کنید سیستم ما به خوبی کار میکند، اما ما به این نتیجه میرسیم که ذخیره کردن تاریخ تولد دانشجویان در دو رقم ایده خوبی نیست. بدیهی است که برای رفع این مشکل باید فیلد تاریخ تولد از جدول دانشجویان را از دو رقم به چهار رقم تغییر دهیم.

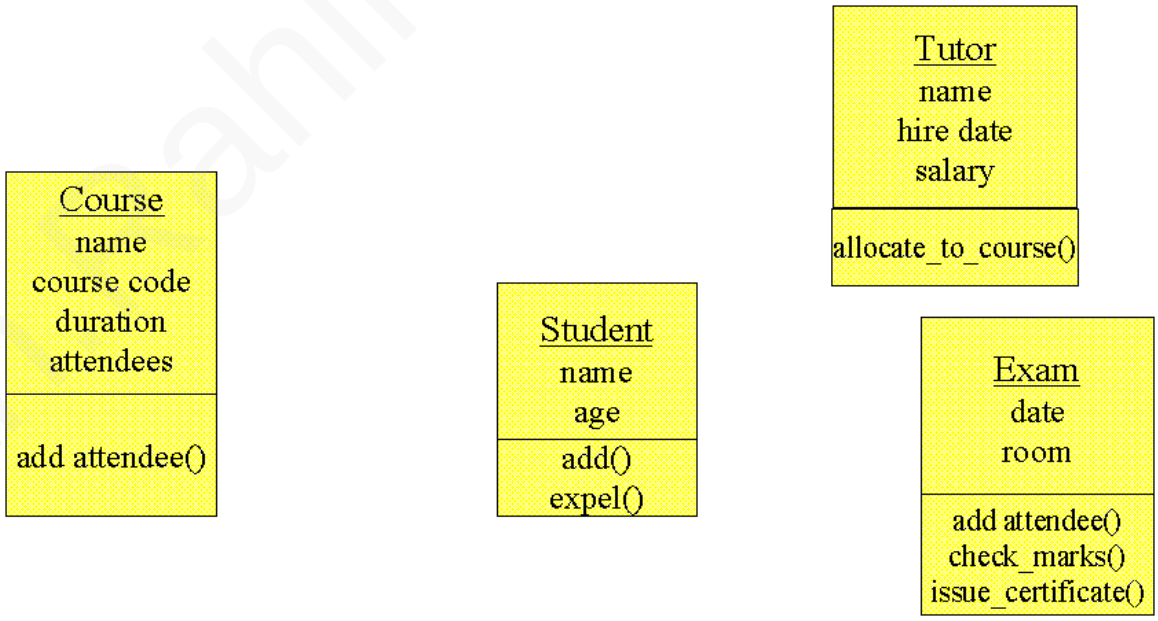
مشکلی که با این تغییر به وجود می آید این است که این تغییر بر کل پروژه اثر میگذارد، داده های امتحانات و دروس و اساتید به داده های دانشجویان وابسته هستند، بنابر این همین تغییر کوچک موجب از کار افتادن یا درست عمل نکردن تمام توابع می شود.

بنابر این ما با یک مشکل اساسی رو به رو میشویم و بدتر از آن اینکه در کد برنامه به آسانی نمی توانیم تمام این وابستگی ها را پیدا کنیم.

روند شیء گرا (Object oriented approach)

روند شیء گرا، با قرار دادن توابع و داده های وابسته در یک مکان واحد (**module**) مشکلات برنامه نویسی ساخت یافته را کمتر کرده است.

به شکل زیر دقت کنید، توابع و داده های مربوط به هم را نشان می دهد. برای مثال توابع اضافه کردن دانشجو و اخراج دانشجو کاملا به دادهای دانشجویان وابسته است.



توابع و داده های وابسته در یک مکان واحد قرار داده شده اند.

دو نکته درباره برنامه نویسی ماژولار :

- **وقتی برنامه در حال اجرا است بیشتر از یک نمونه از یک ماژول میتواند وجود داشته باشد. در سیستم دانشگاه برای هر دانشجو یک نمونه از ماژول دانشجو وجود دارد و هر نمونه داده ها و نام مخصوص به خود را دارد.**
- **ماژولها با فراخانی توابع یکدیگر میتوانند با هم ارتباط بر قرار کنند.**

به صورت کپسول در آوردن (Encapsulation)

یک قانون وجود دارد که : هر نمونه از ماژول فقط می تواند داده های خودش بخواند یا تغییر دهد. برای مثال یک نمونه از ماژول استاد نمی تواند مقدار معدل یک نمونه از ماژول دانشجو را تغییر دهد یا بخواند.

به این مفهوم به صورت کپسول در آوردن (Encapsulation) می گویند، این کار موجب بهبود ساختار سیستم می شود، و مشکلات برنامه نویسی ساخت یافته از بین می رود. به طوری که تغییر در داده ها موجب تغییرات کمی در دیگر قسمتها می شود .

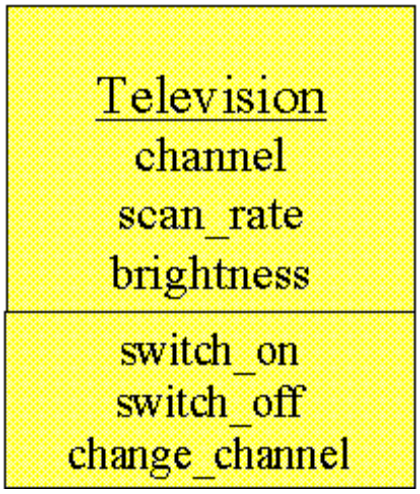
با کپسول سازی، برنامه نویس می تواند مطمئن باشد که مثلاً فقط ماژول دانشجو میتواند داده های ماژول دانشجو را تغییر دهد، و اگر در نحوه ذخیره سازی داده های دانشجو تغییری صورت گیرد برنامه نویس مطئمن است که فقط باید ماژول دانشجو را تغییر دهد.

اشیاء (Objects)

دراین درس من به مجموعه ای از توابع و داده های وابسته بع عنوان ماژول اشاره کردم، اما اگر به مشخصات این ماژول ها بنگریم مشابهات زیادی با جهان واقعی خواهیم دید.

اشیاء در دنیای واقعی با دو چیز شناخته می شوند: هر شیء واقعی داده ها و رفتار هایی دارد، برای مثال تلوزیون یک شیء است که داده های مخصوص به خود از جمله میزان روشنایی، میزان صدا و ... را دارد. و همچنین یک تلویزیون می تواند کارهایی انجام دهد، برای مثال خاموش و روشن شود، شبکه ها را تغییر دهد و ..... .

ما می توانیم این اطلاعات را در قالب ماژولهای نرم افزاری همانند شکل زیر نشان دهیم.



داده ها و رفتار یک تلویزیون

به این دلیل است که به ماژولها، شیء میگوییم و کتابهای تحلیل، طراحی و برنامه نویسی شیء گرا داریم.

از آنجایی که سیستم های نرم افزاری مشکلات جهان واقعی را حل می کنند (مدیریت دانشگاه، کتابخانه، انبار، اتوماسیون اداری و ....) پس ما می توانیم اشیاء موجود در مسئله را شناسایی کرده و این اشیاء را به اشیاء نرم افزاری تبدیل کنیم.

مفاهیم شیء گرایی تجریدی از مفاهیم واقعیت است، به این معنی که اگر مسئله تغییر کند (مثلا نیازمندیها تغییر کنند) تغییر آن در سیستم بسیار آسان است.

اصطلاحات فنی

داده های یک شیء را صفات (Attributes) آن شیء می گویند. رفتارهای یک شیء را متدهای (Methods) آن شیء می گویند. متدها درست مانند توابع در زبانهای برنامه نویسی هستند. اصطلاح دیگری که وجود دارد واژه کلاس است، به طور ساده یک کلاس الگوی یک شیء است. یک کلاس تعیین میکند که هر شیء هم نوعش چه صفات و چه متدهایی را دارد. مثلا در سیستم دانشگاه یک کلاس به نام دانشجو وجود دارد.
صفات کلاس دانشجو نام، معدل و ... و متدهای این کلاس اضافه کردن و اخراج هستند. در قسمت کد سیستم ما فقط یک بار این کلاس و صفات و متدهایش را تعریف می کنیم، و بعد در حالی که برنامه در حال اجراست می توانیم هر تعداد از کلاس مذکور را بسازیم. به نمونه هایی که از کلاس در هنگام اجرای برنامه ساخته می شوند شیء گفته می شود، و هر شیء داده های مخصوص به خودش را دارد.

استراتژی شیء گرا

این درس فقط خلاصه ای از مزیتهای شیء گرا را گفت، اما سوالهایی مثل : چگونه در هنگام طراحی سیستم، اشیاء لازم را شناسایی کنیم؟ و اینکه این اشیاء چه صفت ها و چه متدهایی داردند؟ و یا اینکه یک کلاس چقدر باید بزرگ باشد؟ جواب داده نشده است. در ادامه این کتاب ما به سراغ نحوه توسعه نرم افزارهای شیءگرا با استفاده از UML می رویم و جواب سوالات بالا را خواهیم گرفت.

درست است که شیءگرایی در طراحی سیستم بسیار قدرتمند است ولی در گذشته شیء گرایی یک ضعفی که داشت، این بود که در بیان عملکرد کل سیستم ضعیف بود. یعنی با نگاه کردن به کلاسها نمی توان عملکرد کل سیستم را شرح داد، چون کلاسها در سطح خیلی پایینی به عنوان زیرساخت یک پروژه کار می کنند. فرض کنید که شما برای اینکه بفهمید یک کامپیوتر چگونه کار می کند به سراغ نحوه عملکرد ترانزیستورها بروید. مسلم است که خیلی سخت و خیلی دیر خواهید فهمید که کامپیوتر چگونه کار میکند، یا اصلاً ممکن است موفق نشوید، یا شاید عمرتان قد ندهد ! ! !

روند مدرنی که توسط زبان UML پشتیبانی می شود این است که در مراحل اولیه ساخت یک پروژه همه چیز درباره کلاس و شیء فراموش می شود و بجای آن تمرکز می رود بر روی اینکه سیستم چه کاری را باید انجام دهد. سپس کلاسها رفته رفته ساخته می شوند. در این کتاب ما تمام مراحل ساخت پروژه از تحلیل اولیه تا طراحی کلاس را توضیح می دهیم.

خلاصه

- شیءگرایی تفاوت اندکی با روند ساختاری دارد.
- داده ها و رفتارهای مرتبط را در یک کلاس قرار می دهیم.
- در طول برنامه نمونه هایی از کلاس را به عنوان شیء می سازیم.
- اشیاء با فراخانی متدهای یکدیگر می توانند با هم همکاری کنند.
- در یک شیء داده ها کپسول شده است و فقط خود شیء می تواند داده هایش را تغییر دهد.

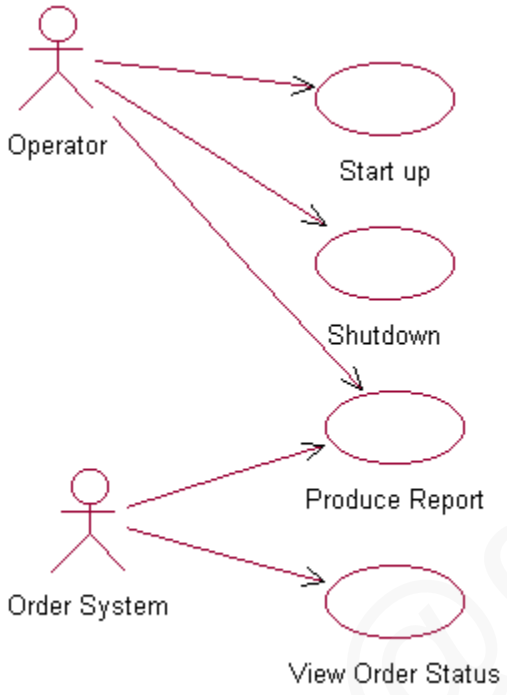
درس چهارم : خلاصه ای از کل زبان UML

قبل از ایکه به سراغ خود زبان UML برویم. بعضی از مفاهیم کلی این زبان را شرح می دهیم.
اولین چیزی که راجع به UML باید بگویم این است که اصلاً زبان UML از چه قسمت هایی تشکیل شده است؟ زبان UML فقط از چند نمودار (مدل) تشکیل شده است. به واسطه این نمودار ها ما قادر خوهیم بود از چند جهت یک سیستم را ببینیم یا بسازیم. در توسعه یک نرم افزار چندین نقش وجود دارد. برای مثال :

- تحلیلگران
- طراحان
- کد کنندگان (Coders)
- مشتری
- پشتیبانان فنی

هر کدام از این نقشها وابسته به یک جنبه از سیستم هستند، و هر کدام سطح متفاوتی از جزئیات را نیاز دارند. برای مثال کد کنندگان سیستم باید طراحی سیستم را بفهمند تا قادر باشند آن را به کد تبدیل کنند. ولی مشتری به همچین جزئیاتی نیاز ندارد، پس باید یک دید دیگر از سیستم به مشتری نشان داد تا بفهمد کل سیستم چه کاری انجام می دهد. شما انتظار ندارید که طراحی کلاسهای سیستم را برای مشتریی که هیچ اطلاعاتی از برنامه نویسی ندارد بفرستید و بگویید سیستم این کارها را انجام می دهد. UML زبانی است که به واسطه نموداری مختلف خود، فهم سیستم را برای تمام نقشهای یک پروژه مقدور کرده است.
در ادامه این درس خلاصه ای از نمودارهای مهم این زبان آمده است. البته در دروس بعدی جزئیات تمام آنها شرح داده می شود.

نمودار مورد کاربرد (Use Case Diagram)



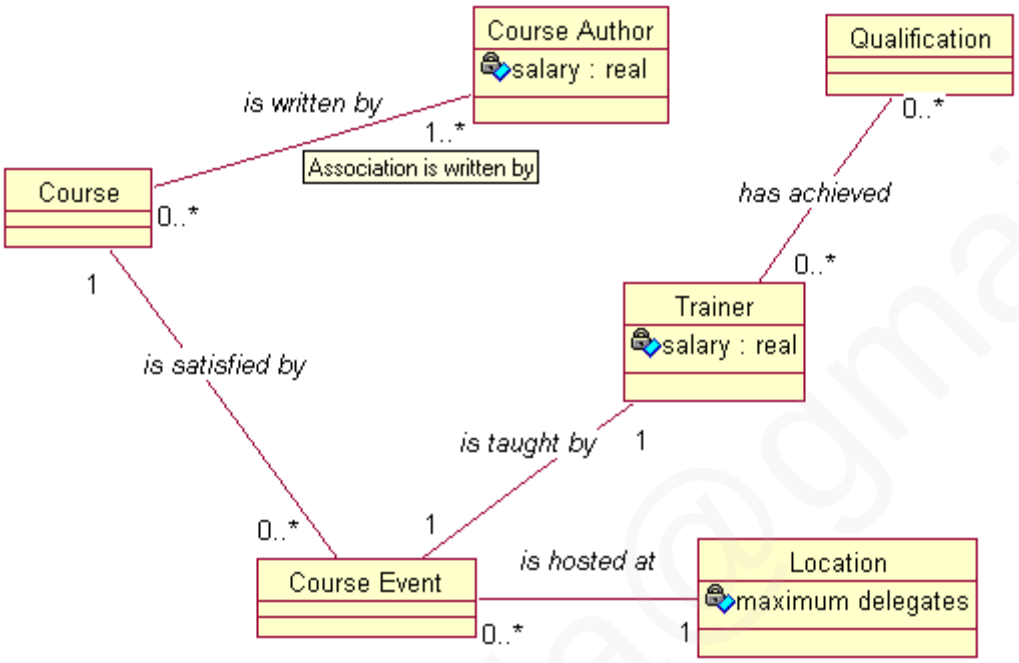
نمودار مورد کاربرد.

نمودار مورد کاربر رفتار سیستم را از نظر کاربر شرح می دهد. این نمودار در هنگام تحلیل سیستم بسیار با ارزش است – توسعه موارد کاربرد به ما کمک می کند که نیازمندیهای سیستم را درک کنیم.

این نمودار به راحتی قابل فهم است و همین موضوع موجب می شود که هم توسعه دهندگان سیستم (تحلیل گر – طراح – کد نویس - امتحانگر) و هم مشتری بتوانند با این نمودار کار کنند.

درست است که نمودار مورد کاربر بسیار ساده است ولی نباید از آن چشم پوشی کرد، چون به واسطه این نمودار است که فرایند توسعه سیتم از فاز دریافت به فاز جزئیات می رود.

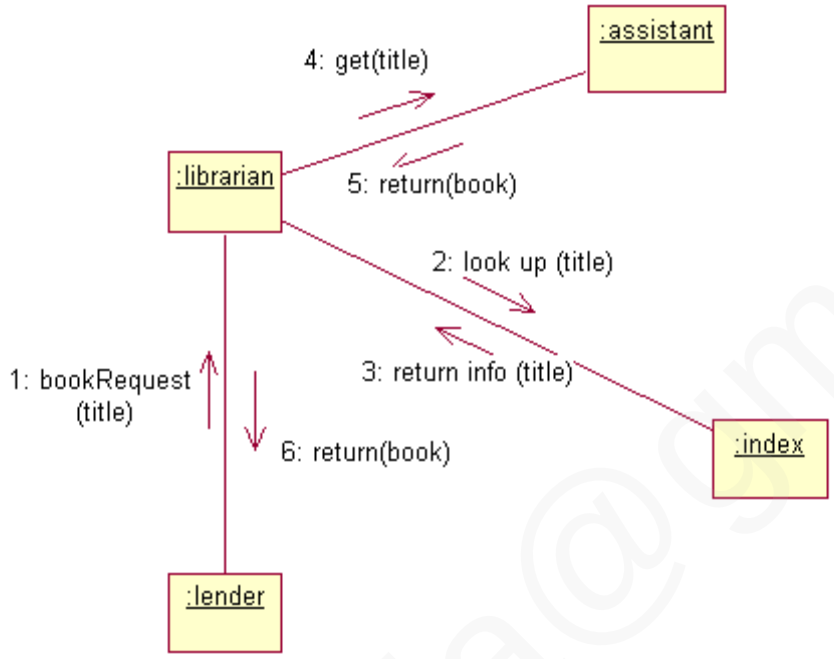
نمودار کلاس (Class Diagram)



نمودار کلاس در UML

رسم نمودار کلاس، یک جزء ضروری از متد طراحی شیءگرا است. در دروس بعدی خواهیم دید که از نمودار کلاس هم در مرحله تحلیل و هم در مرحله طراحی استفاده می شود. از نمودار کلاس برای طرح ریزی مفاهیم عمده ای که مشتری آنها را می فهمد استفاده می شود (و به همین دلیل است که به آن مدل مفهومی نیز میگوییم). مدل مفهومی یک تکنیک بسیار قدرتمند در تحلیل نیازمندیها است.

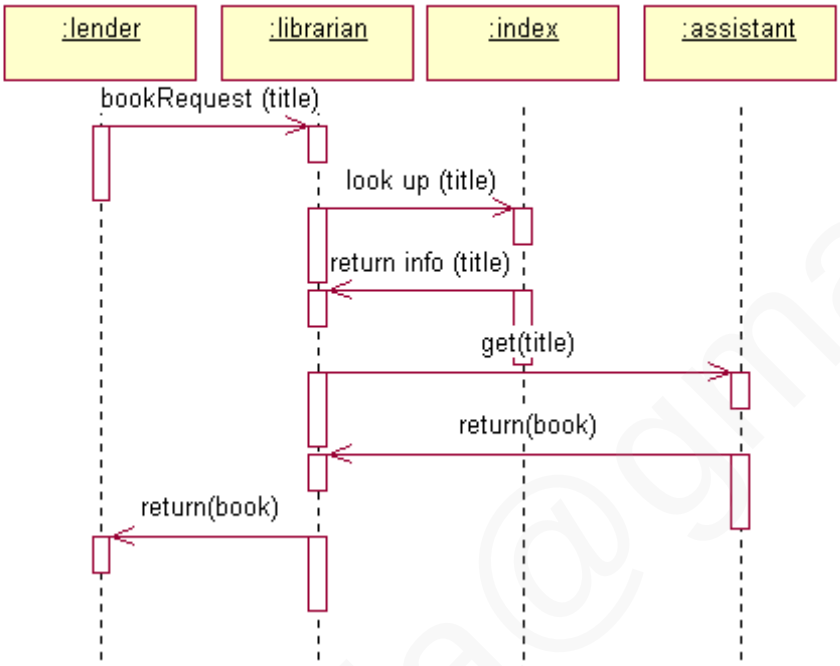
نمودار همکاری (Collaboration Diagram)



نمودار همکاری در UML

در نرم افزارهای شیءگرا تمام کاری که برنامه باید انجام دهد، توسط همکاری اشیاء برنامه صورت می گیرد. در زبان UML ما می توانیم نمودار همکاری اشیاء را رسم کنیم که نحوه همکاری اشیاء برنامه را شرح دهد. در اینجا مثال خوبی می توان گفت که چرا UML فقط یک زبان است و یک فرایند توسعه نرم افزار نیست؟ UML را مثل حروف الفبا فرض کنید حروف الفبا به شما فقط می گویند که چطور کلمات و جملات را بنویسید، ولی هرگز به شما نمی گویند که چطور یک کتاب بنویسید. درست است که درک نشانه های UML آسان است، مثلا برای معرفی یک کلاس یا یک شیء از یک مربع که به سه قسمت تقسیم شده است استفاده می شود. ولی طراحی همکاری و ارتباطات بین اشیاء سخت است. که به این قسمت طراحی نرم افزار می گویند که باید قوی و محکم باشد تا نگهداری و توسعه نرم افزار ساده باشد. برای نحوه طراحی نرم افزار باید یک کتاب جداگانه بنویسیم ولی برای حرفه ای شدن در این قسمت بیشتر به تجربه نیاز است. درست مثل رانندگی، شما هرچقدر که کتاب آموزش رانندگی را بخوانید به خوبی شخصی که چندین سال است رانندگی می کند نمی توانید رانندگی کنید.

نمودار ترتیب (Sequence Diagram)

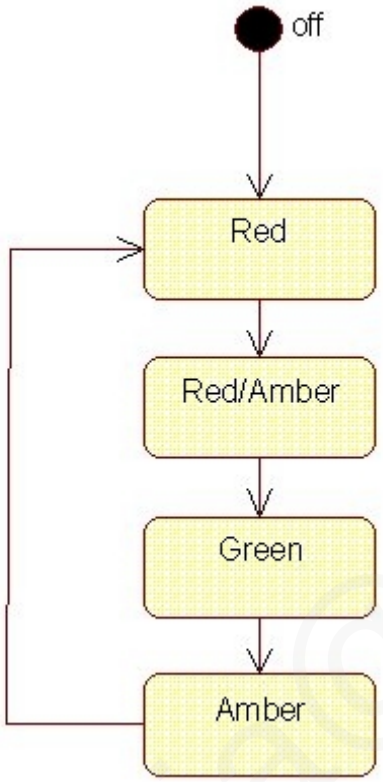


نمودار ترتیب در UML

نمودار ترتیب کاملاً به نمودار همکاری وابسته است. و هر دوی آنها نیز یک اطلاعات را نشان می دهند، ولی در شکلهای متفاوت. خط چین ها در نمودار، زمان را نشان می دهند، بنابر این چیزی که ما اینجا می توانیم به دست آوریم این است که در قالب زمان اشیاء چگونه با یکدیگر فعل و انفعال دارند.

بعضی از ابزارهای زبان UML مثل Rational Rose به صورت خودکار می توانند نمودار ترتیب را از نمودار همکاری تولید کنند، و نمودار ترتیبی که در بالا مشاهده می کنید درست از همین راه بدست آمده است.

نمودار وضعیت (State Diagram)



نمودار وضعیت چراغ راهنمایی و رانندگی

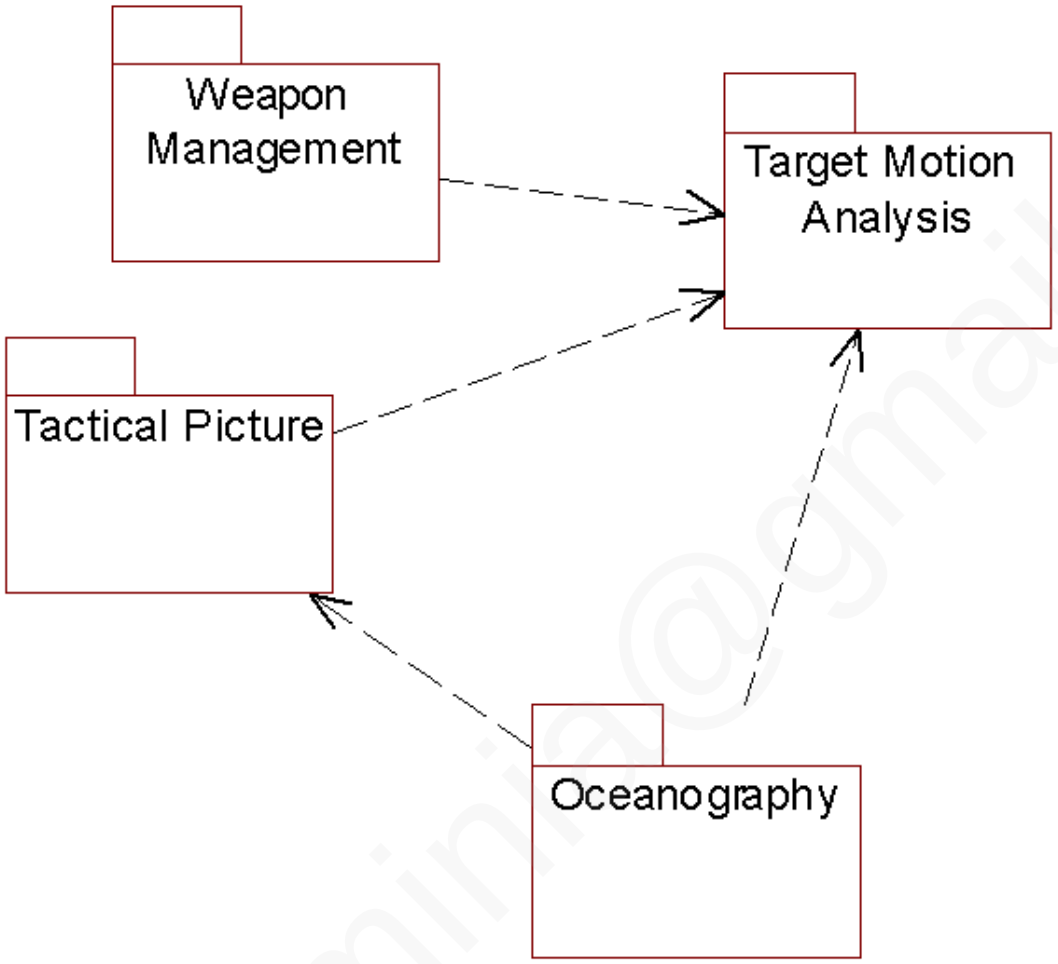
بعضی از اشیاء در هر زمان میتوانند در یک وضیت خاصی باشند. مثلاً یک چراغ راهنما می تواند یکی از وضعیت های خاموش، قرمز، زرد و یا سبز را داشته باشد. بعضی وقتها ترتیب انتقال بین وضعیتها خیلی پیچیده است. برای مثال در مثال بالا ما نمی خواهیم از وضعیت قرمز به وضعیت سبز برویم (که موجب تصادف شود!!!).

با اینکه چراغ راهنما یک مثال ساده است ولی عدم رعایت همین نکات ساده و نا منظم بودن تغییر وضعیت ها موجب به وجود آمدن مشکلات اساسی در یک سیستم نرم افزاری می شود.

برای مثال، در شهری یک صورت حساب گاز برای یک شخصی که چهار سال قبل درگذشته بود فرستاده شده بود – این اتفاق واقعاً افتاده است و فقط به این دلیل که یک برنامه نویسی در یک جا دقت کافی در تغییر وضعیت ها نداشته و موجب ارسال این صورت حساب شده است.

پس چون انتقال وضعیت ها کاملاً پیچیده هستند، زبان UML نمودار وضعیت را به وجود آورده است تا به واسطه این نمودار نحوه تغییر وضیت اشیاء را به دقت مدل کنیم.

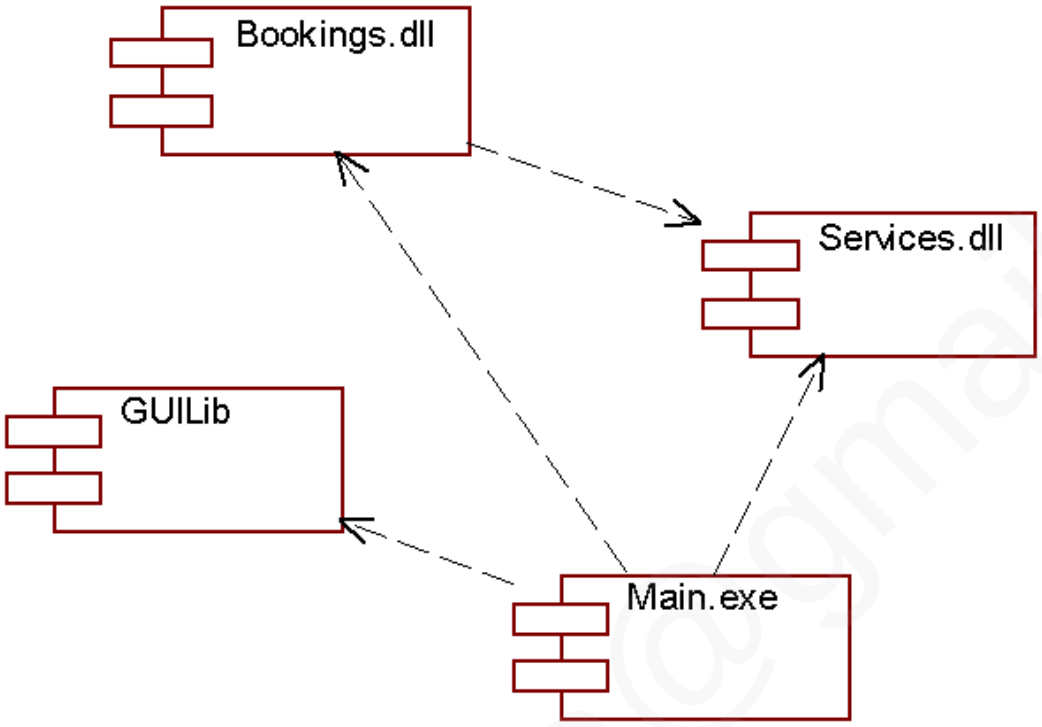
نمودار بسته یا بسته بندی (Package Diagram)



نمودار بسته بندی در UML

سیستم های بزرگ برای انیکه آسانتر درک شوند، باید به قسمتهای کوچکتری تقسم شوند، و نمودار بسته بندی در UML ما را قادر میسازد تا این تقسیم را به طور کارامدی مدل کنیم. جزئیات این مدل را در درس معماری سیستم های بزرگ توضیح داده ام.

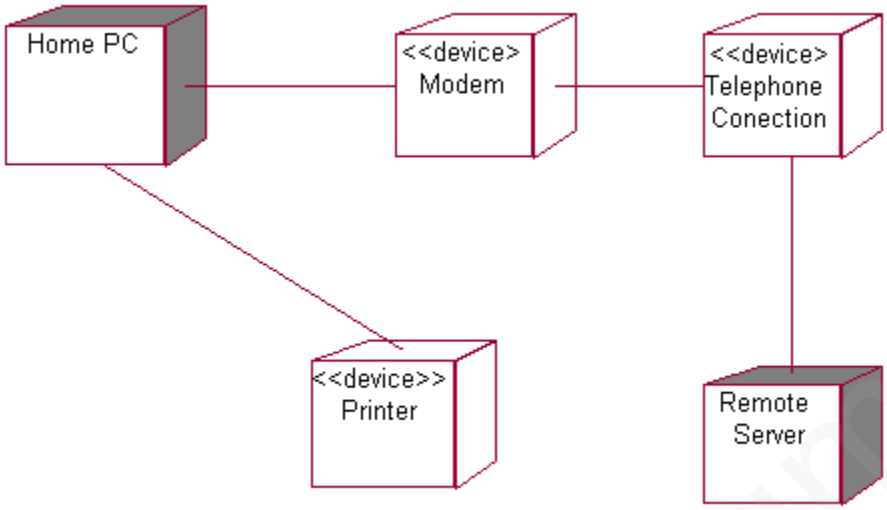
نمودار اجزاء (Component Diagram)



نمودار اجزاء

نمودار اجزاء شبیه به نمودار بسته بندی است، و مشخص می کند که سیستم ما از چه بخشهایی تشکیل شده است، و همچنین وابستگیهای بین هر ماژول را معین می کند. برخلاف نمودار بسته بندی که تاکید آن بر قسمت بندی نرم افزار است، تاکید نموداراجزاء بر اجزای فیزیکی تشکیل دهنده نرم افزار (فایلها، هدرها، فایلهای اجرایی و ....) است. جزئیات این نمودار را در درس معماری سیستمها توضیح داده ام.

نمودار گسترش (Deployment Diagram)



نمودار گسترش

زبان UML نموداری را برای مشخص کردن اینکه نرم افزار چگونه گسترش داده می شود به وجود آورده است. نمودار بالا وضعیت یک کامپیوتر شخصی را نشان می دهد.

خلاصه

زبان UML چندین دیدگاه از یک سیستم را نمایش می دهد، که لیست همه آنها با یک توضیح کوچک در زیر آمده است :

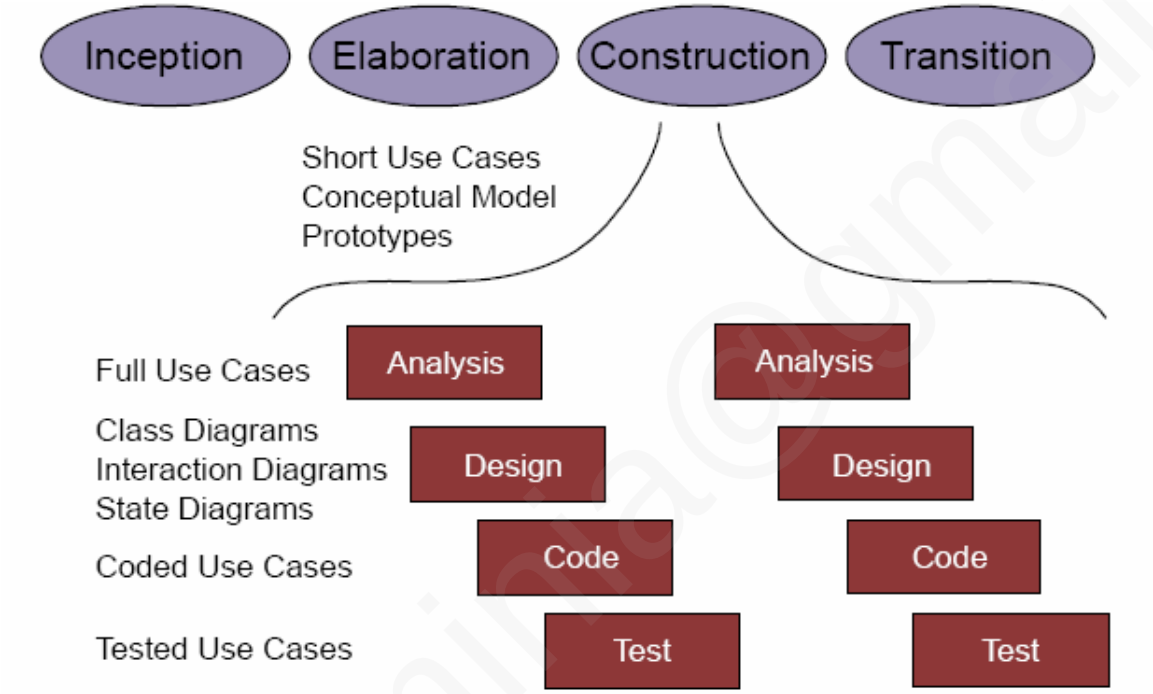
- **نمودار موارد کاربرد : سیستم ما چگونه با جهان خارج از خود ارتباط بر قرار می کند؟**
- **نمودار کلاس : چه اشیائی نیاز داریم؟ و این اشیاء چگونه به هم وابسته هستند؟**
- **نمودار همکاری : اشیاء چگونه با هم فعل و انفعال دارند؟**
- **نمودار ترتیب : اشیاء چگونه با هم فعل و انفعال دارند؟**
- **نمودار وضعیت : اشیاء ما باید در چه وضعیتهایی باشند؟**
- **نمودار بسته بندی : توسعه نرم افزار چگونه ماژولار شده است؟**
- **نمودار اجزاء : اجزاء نرم افزار چگونه به هم وابسته هستند؟**
- **نمودار گسترش** : نرم افزار ما چگونه گسترش داده خواهد شد؟

در دروس بعد عملاً وارد جزئیات تک تک فازهای توسعه نرم افزار و فعالیتهای قابل انجام در آنها می شویم و هر فاز را در قالب یک درس شرح می دهیم. و کل فرایند نوسعه نرم افزار را به طور خلاصه

پیگری می کنیم. بدیهی است که هر کدام از این فازها نیاز به تخصصهای خاصی دارد و برای هر کدام می توان چندین کتاب نوشت، ما در این کتاب فقط فعالیتهایی که در این فازها انجام می شود را با توضیحات مختصر و چند مثال در اختیار شما می گذاریم، تا یک دیدگاه کلی از این فازها داشته باشید و به سمت هرکدام که علاقه دارید بروید و در آن متخصص شوید، و برای متخصص شدن نیز صرفاً مطالعه کافی نیست بلکه باید در یک تیم به صورت عملی کار کنید. البته از آنجایی که فازهای طراحی و ساخت شامل مباحث تحلیل و طراحی شیء گرا می شود، این دو فاز را با جزئیات بیشتری برسی خواهیم کرد.

درس پنجم : فاز دریافت[1]

در باقیمانده این کتاب، چگونگی کاربرد زبان UML در پروژه های واقعی را شرح می دهیم. ما از فرایندی که در درس اول معرفی شد و شکل آن در زیر آمده است پیروی می کنیم :



فرایندی که در این کتاب از آن پیروی خواهیم کرد.

نام هر مدلی از زبان UML که در هر مرحله تولید خواهد شد را در این تصویر آورده ام. برای مثال در مرحله طراحی نمودار های کلاس، تعامل و وضعیت تولید می شوند. جزئیات هر یک از این نمودار ها را در ادامه کتاب توضیح خواهم داد.

فعالیتهایی که در این فاز انجام می شود به شرح زیر است :

- مشخص کردن دیدگاه محصول
- تولید یک مورد تجاری
- تعیین حوزه پروژه
- تخمین هزینه کل پروژه

[1] این فاز در بعضی کتب فاز شروع هم معنی شده است. اما به نظر من دریافت برای آن بهتر و گویا تر است.

طول مدت این فاز کاملاً وابسته به پروژه است. فعالیتهایی که در این فاز برای یک پروژه تجاری انجام می شود، ممکن است فقط شامل تعریف یک دیدگاه و گرفتن سرمایه از بانک باشد، ولی برای یک پروژه بزرگ ممکن است نیاز به تحلیل نیازمندیها، پیش مطالعات، مناقصه و ... داشته باشیم. همه اینها وابسته به پروژه است.

در این کتاب فرض ما بر این است که فاز دریافت انجام شده است، و یک مطالعه تجاری انجام شده و نیازهای اولیه مشتری مشخص شده است.

درس ششم : فاز جزئیات

در فاز جزئیات وارد جزئیات مسئله می شویم. نیازمندیها و تجارت مشتری را درک می کنیم، و طرح را توسعه می دهیم.
در این فاز نباید بیش از حد وارد جزئیات شویم، بویژه جزئیات مربوط به پیاده سازی سیستم.
ما نیاز به یک دید کلی از سیستم و درک عملکرد آن داریم.

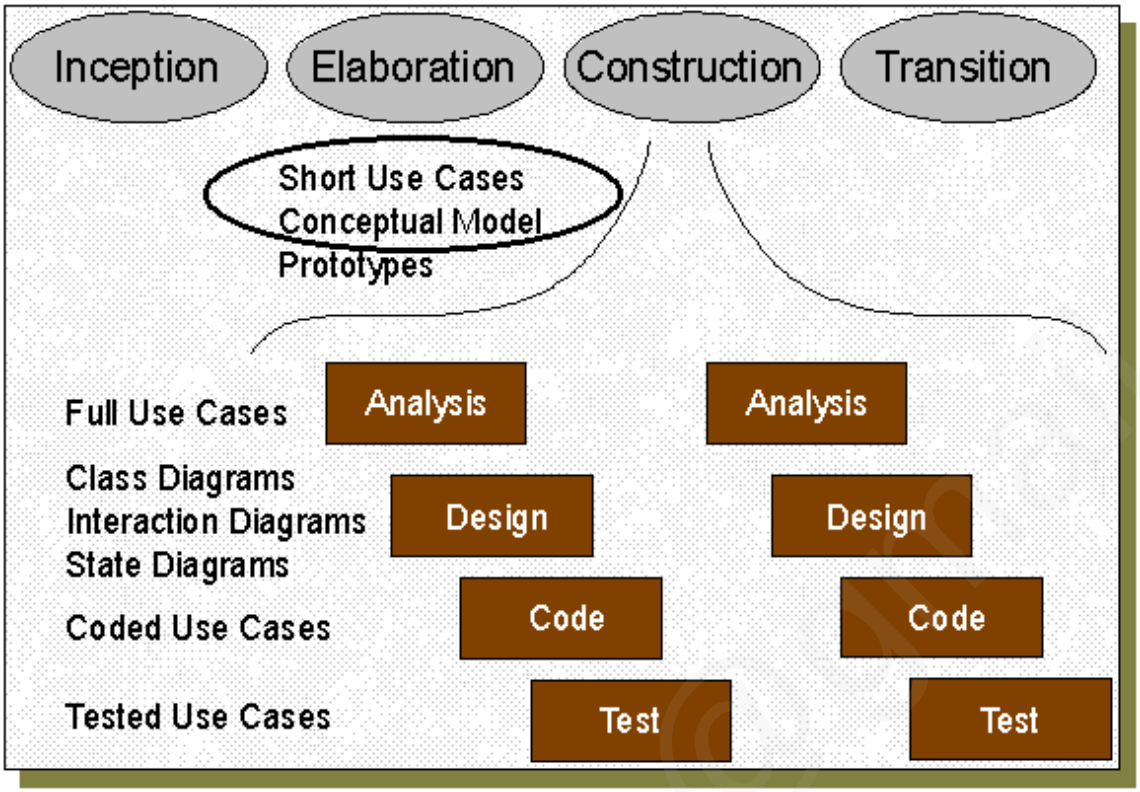
نمونه سازی (Prototyping)

یک فعالیت کلیدی در فاز جزئیات کاهش دادن ریسکها است. ریسکهای اولیه شناسایی و کم رنگ می شوند، که این امر موجب می شود مشکلات کمتری را برای پروژه به وجود آید.
نمونه سازی اولیه قسمتهای مشکل پروژه کمک بزرگی در کاهش ریسکها است. مسلم است که ما در این فاز نمی خواهیم وارد طرحی و پیاده سازی نمونه های اولیه شویم، بلکه در این فاز فقط شرحی از قسمتهای مهم و مشکل پروژه تولید می شود.

موارد قابل تحویل در این فاز

موارد قابل تحویل به محصولاتی که در یک فاز تولید می شود می گویند. برای مثال موارد قابل تحویل در این فاز می تواند دو نمودار از سیستم باشد.
اولین نمودار، نمودار مورد کاربرد می باشد. این نمودار به ما کمک می کند تا بفهمیم سیستم چه کارهایی باید انجام دهد و سیستم چگونه با جهان خارج از خود (مثلاً کاربران) ارتباط برقرار می کند.
دومین نمودار نمودار ادراک[1] است. این نمودار به ما کمک می کند تا مسئله مشتری را در قالب یک نمودار زبان UML مشخص کنیم. این نمودار تمام مفاهیم کلی مسئله مشتری و چگونگی ارتباطات بین آنها را شرح می دهد. برای ساخت این نمودار از نمودار کلاس زبان UML استفاده می کنیم. ما از مدل ادراک در فاز ساخت برای ساخت کلاسها و اشیاء نرم افزار استفاده می کنیم.
جزئیات دو نموداری که در بالا معرفی شد در دو درس بعدی به ترتیب آمده است.

[1] در بعضی موارد به نمودار کلاس نمودار ادراک هم می گویند.



ساخت دو نمودار از زبان UML در فاز جزئیات

خلاصه

فاز جزئیات مربوط است به توسعه و درک مسئله، بدون نگرانی در مورد جزئیات طراحی (البته به جز شناسایی ریسکها و نمونه های اولیه ای که نیاز است).

دو نمودار از زبان UML که در این فاز به ما کمک می کند : نمودار مورد کاربرد و نمودار ادراک.

درس هفتم : ساخت نمودار مورد کاربرد

یکی از نمودارهای قدرتمند زبان UML نمودار مورد کاربرد است. به طور ساده، مورد کاربرد شرحی از تعاملات بین کاربر و سیستم است. با ساخت مجموعه ای از نمودار های مورد کاربرد می توانیم به صورت مختصر و مفید، تمام سیستمی که می خواهیم بسازیم را شرح دهیم.

موارد کاربرد معمولاً با ترکیب فعل/اسم وجود خود را شرح می دهند. برای مثال، پرداخت صورت حساب یا ساخت اعتبار و .... .

به عنوان مثال اگر در حال ساخت سیستم پرتاب موشک هستیم، یک نمودار مورد کاربرد ممکن است شلیک موشک باشد.

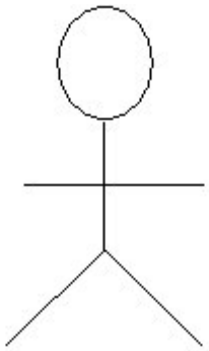
در کنار نام مورد کاربرد، توضیحی کاملاً متنی در باره تعاملات بین کاربر و سیستم قرار می دهیم. این توضیحات متنی عموماً پیچیده هستند. اما زبان UML یک نمایش کاملاً ساده از مورد کاربرد را همانند شکل زیر ارائه کرده است :

نمادسازی مورد کاربرد

بازیگران (Actors)

یک مورد کاربرد نمی تواند خودش را به کار بیندازد، بلکه بازیگر است که می تواند یک مورد کاربرد را به کار بیندازد. برای مثال اگر در حال توسعه سیستم یک بانک هستیم و یک مورد کاربرد به نام دریافت پول داشته باشیم، در این صورت به یک مشتری نیز نیازمندیم تا پول را دریافت کند. از اینرو در اینجا مشتری یکی از بازیگران خواهد بود.

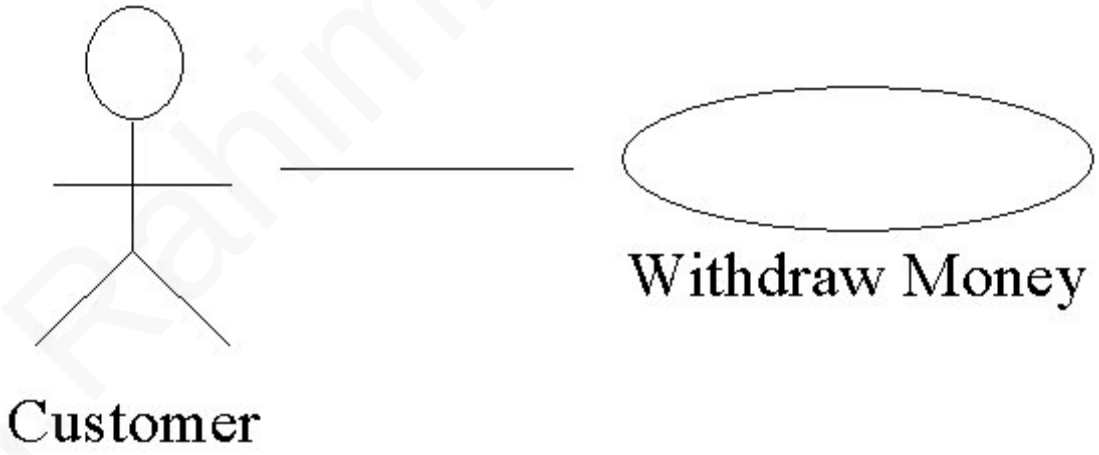
نماد یک بازیگر به صورت زیر است :



نماد زبان UML برای یک بازیگر

بازیگران فقط انسانها نیستند. هر چیزی که یک مورد کاربر را به کار بیندازد یک بازیگر است. مثلاً یک کامپیوتر دیگر. حتی مفاهیمی که وجود خارجی ندارند هم می توانند به نقش یک بازیگر را ایفا کنند، مثلاً زمان یا یک تاریخ خاص. برای مثال ممکن است یک مورد کاربرد به نام پاکسازی سفارشهای قدیمی در یک سیستم سفارش دهی داشته باشیم و این مورد کاربرد یک بازیگر به نام آخرین روز کاری داشته باشد.

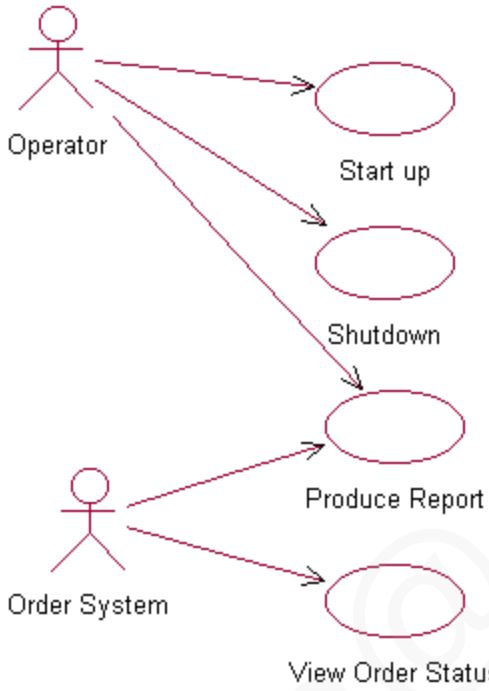
همانطور که گفتم بازیگرها به موارد کاربرد وابسته هستند، و این مفهوم را در زبان UML می توانیم با یک که بازیگر را به مورد کاربرد متصل می کند نشان دهیم :



ارتباط بین یک بازیگر و مورد کاربرد

در اکثر سیستمها یک بازیگر مشخص می تواند با چند مورد کاربرد متفاوت تعامل داشته باشد. و همچنین یک مورد کاربرد مشخص می تواند توسط چند بازیگر مختلف راه اندازی شود. و همین امر

موجب به وجود آمدن نمودارهای مورد کاربرد می شود، یک مثال از نمودار مورد کاربرد در زیر آمده است.



شرح کامل یک سیستم توسط بازیگران و موارد کاربرد

هدف موارد کاربرد (Use Case)

تعاریف و مفاهیم مربوط به موارد کاربرد و همچنین نشان دادن و نماد سازی آنها در زبان **UML** بسیار ساده است. ولی این دلیل نمی شود که در پروژه ها این نمودار را حذف کنیم. اگر با خودتان فکر می کنید که این نمودار ها خیلی ساده هستند و اصلا ارزش روی کاغذ آمدن و فکر کردن هم ندارند، در اشتباه هستید، چون موارد کاربرد بسیار قدرتمند هستند.

فوایدی که موارد کاربرد دارند به شرح زیر است :

- حوزه یا هدف پروژه را مشخص می کنند. و قادرند اندازه و هدف کل توسعه را نمایش دهند.
- موارد کاربرد شباهت زیادی به نیازمندیهای سیستم دارند. نیازمندیهایی که به صورت نا مرغوب نوشته شده اند، گیج کننده و مبهم هستند، در حالی که نمودارهای موارد کاربرد نظم و ترتیب بیشتری دارند، و کاملاً واضح هستند.
- مجموع تمام نمودارهای موارد کاربرد، کل سیستم را تشکیل می دهند. به این معنی که اگر یک چیزی در موارد کاربرد نیامده باشد، آن چیز خارج از حوزه و هدف سیستمی است که ما

می خواهیم توسعه دهیم. بنابر این موارد کاربرد تمام پروژه را بدون هیچ ابهامی شرح می دهند.

- **نمودارهای موارد کاربرد ارتباط بین توسعه گران و مشتریان را برقرار می کنند (چون این نمودارها بسیار ساده هستند، و هرکسی می تواند آنها را بفهمد).**
- **موارد کاربرد در کل فرایند توسعه راهنمایی برای تیم توسعه هستند. موارد کاربرد ستون فقرات توسعه سیستم هستند، و هر کاری را که می خواهیم انجام دهیم باید به آنها نگاه کنیم.**
- **در دروس بعدی خواهیم دید که چگونه به وسیله موارد کاربرد کار توسعه سیستم را طرح ریزی میکنیم و زمان لازم برای توسعه سیستم را تخمین می زنیم.**
- **موارد کاربرد معیار و مبنای آزمایش سیستم هستند.**
- **موارد کاربرد در ساخت راهنما برای کاربران به ما کمک می کنند.**

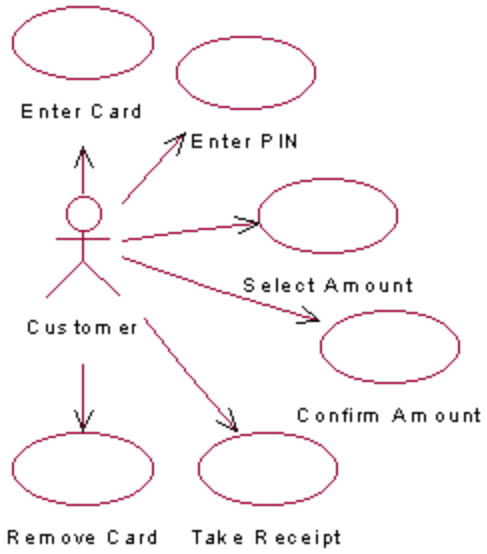
بعضاً ادعا می شود که موارد کاربرد تعریف ساده ای از نیازمندیهای سیستم است، ولی هرکسی که این ادعا را دارد کاملاً در اشتباه است و هدف موارد کاربرد را درک نکرده است.

تکه تکه بودن مورد کاربرد (Use Case Granularity)

تصمیم گرفتن درباره تکه های یک مورد کاربرد سخت است – مثلاً، آیا هر کاربر سیستم باید با یک مورد کاربرد تعامل داشته باشد؟ یا، آیا موارد کاربرد باید تمام تعاملات را به صورت کپسول شده درآورند؟ برای مثال یک سیستم خود پرداز را در نظر بگیرید. ما باید سیستمی بسازیم که کاربر قادر باشد از آن پول دریافت کند. در این کار با یک سری از تعاملات معمولی به شرح زیر رو به رو می شویم :

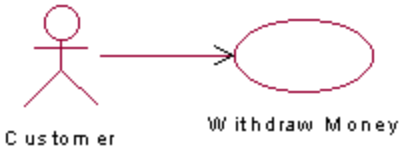
- **کارت را وارد کن**
- **شماره رمز را وارد کن**
- **مقدار پول مورد نظر را انتخاب کن**
- **مقدار پول مورد نظر را دریافت کن**
- **کارت را بردار**
- **رسید را بگیر**

آیا هر کدام از این مراحل (مثلاً کارت را وارد کن) یک مورد کاربرد است؟



به نظر شما، آیا این یک مورد کاربرد خوب و کارامد است؟

این یک اشتباه عمومی در ساخت موارد کاربرد است. در اینجا تعداد زیادی از موارد کاربرد کوچک و بی اهمیت را به وجود آوردیم. اگر این اشتباه را در سیستم های بزرگ مرتکب شویم، در نهایت تعداد زیادی مورد کاربرد خواهیم داشت که همین امر موجب افزایش شدید پیچیدگی می شود.
برای کنترل این پیچیدگی ها باید موارد کاربرد را تا جایی که امکان دارد در سطوح بالا نگاه داریم.
برای تولید یک مورد کاربرد خوب باید یک قانون کلی را در ذهن داشته باشیم، و آن این است که :
یک مورد کاربرد باین هدف را برای بازیگر مشخص کند.
با به کار بستن این قانون ساده در مثال بالا ما باید از خودمان سوال کنیم که آیا هدف بازیگر در نمودار مورد کاربرد بالا گرفتن رسید است؟ مسلم است که نه.
با بکار بردن این قانون برای تمام موارد کاربردی که در نمودار بالا است متوجه می شویم که هیچ کدام از آنها هدف کاربر نیستند. هدف کاربر دریافت پول است. و این باید مورد کاربرد باشد.



یک مورد کاربرد کارآمد

چون ما تا به حال برای توسعه سیستم ها از روند تجزیه تابعی که وظایف پیچیده را به وظایف کوچکتر تقسیم می کرد استفاده کردیم، در اوایل تولید موارد کاربرد کارآمد برایمان مشکل است.
در دروس بعدی می بینید که موارد کاربرد نیز می توانند تجزیه شوند، اما در این فاز از توسعه که فعلاً در آن هستیم (یعنی فاز جزئیات) باید از این کار دوری کنیم و تا قبل از رسیدن به فاز ساخت به آن فکر نکنیم.

توصیف موارد کاربرد (Use Case Descriptions)

هر مورد کاربرد دارای یکسری توضیحات متنی در مورد تعاملات خود است.
زبان UML چگونگی ساختار و محتوای این توضیحات را دقیقاً مشخص نکرده است. ولی ما باید از الگوی زیر برای این توضیحات استفاده کنیم :

مورد کاربرد :	نام مورد کاربرد
توضیح کوتاه :	توضیح مختصری از مورد کاربرد
پیش شرط ها :	شرحی از شروطی که باید قبل از اینکه این مورد کاربرد احضار شود بر قرار باشند
پس شرط ها :	شرحی از پیشامدهایی که در پایان این مورد کاربرد رخ می دهد.
جریان اصلی :	لیستی از تعاملات سیستم که اکثراً در حالت عادی وجود دارند برای مثال در سیستم دریافت پول می تواند به این ترتیب باشد : کارت را وارد کن، شماره رمز را وارد کن و....
جریان فرعی :	شرحی از تعاملات فرعی
جریان استثنا :	شرحی از وقایع غیر منتظره یا رخدادهای غیر قابل پیش بینی

الگوی شرح مورد کاربرد

موارد کاربرد در فاز جزئیات

وظیفه اصلی ما در فاز جزئیات، شناسایی تمام موارد کاربرد است. در کل این فاز این قانون را در ذهن خود داشته باشید : طول کیلومتری ولی عمق سانتیمتری. یعنی در این فاز تا انجا که امکان

دارد تمام موارد را پوشش دهید ولی با جزئیات و عمق کم. این قانون کمک می کند تا پیچیدگی کاهش یابد.
در فاز جزئیات فقط نمودار مورد کاربرد و توضیح مختصری از هر نمودار کافی است. جزئیات کامل هر نمودار مورد کاربرد را در فاز ساخت تولید خواهیم کرد. هر وقت که یک مورد کاربرد شناسایی شد، ما آن را با نیازمندیها مقایسه می کنیم تا مطمئن شویم که تمام نیازمندیها را پوشش دهد.
اگر در این مرحله موارد کاربردی که دارای ریسک بسیار بالا هستند شناسایی شد، شرح جزئیات این قبیل موارد کاربرد لازم است. همچنین تولید نمونه های اولیه در این مرحله ریسک را بسیار کاهش می دهد.

کشف موارد کاربرد

یک راه برای کشف موارد کاربرد، مذاکره و مصاحبه با کاربران سیستم است. این کار سختی است، چون ممکن است دو نفر دید کاملاً متفاوتی از اینکه سیستم قرار است چه کاری را انجام دهد، داشته باشند. در این کار بعضی از توسعه گران مصاحبه تک به تک و بعضی دیگر مذاکره گروهی را ترجیح می دهند. راه دیگری که برای کشف موارد کاربرد وجود دارد و همچنین محبوبتر هم است، کارگاه (workshop) است.

کارگاه طرح ریزی نیازمندی ها ((Joint Requirements Planning Workshops (JRP)

کارگاه تعدادی از افرادی که می خواهند سیستم را توسعه دهند را به دور یکدیگر می نشاند. در این گروه همه افراد دیدگاه خودشان را درباره کارهایی که سیستم باید انجام دهد را ارائه می کنند.
کلید موفقیت این کارگاه ساده انگاری است. همه افرادی که حاضر هستند را تشویق می شوند تا دیدگاه خود را راجع به سیستم ارائه دهند، و مطمئن هستند که تمام دیدگاههای آنها گرفته خواهد شد.
در جلسه یک منشی نیز حاضر است و همه چیز را مستند می کند. منشی ممکن است روی کاغذ کار کند ولی بهتر است که نمودارها بر روی پروژکتور باشد.
در اینجا سادگی نمودارهای مورد کاربرد اجتناب ناپذیر است، چون همه افراد حاضر، حتی کسانی که در عمل کامپیوتر نیز تجربه ای ندارند باید مفهوم این نمودارها را به آسانی درک کنند.
روند ساده اجرای یک کارگاه می تواند به ترتیب زیر باشد :

1. **در ابتدا همفکری در باره تمام بازیگران**
2. **همفکری در باره تمام موارد کاربرد**

3. توجیه تک تک موارد کاربرد به وسیله شرح یک خطی یا یک پاراگرافی برای هرمورد کاربرد.
4. نمادسازی موارد کاربرد.

چند نکته که در کارگاه باید رعایت شود :

- در مورد کشف تمام موارد کاربرد و تمام بازیگران سخت گیری نکنید. این طبیعی است که بعضی از موارد کاربرد بعداً در طول فرایند پدیدار شوند.
- اگر در مرحله سوم نتوانستید یک مورد کاربرد را توجیه کنید، ممکن است اصلاً آن یک مورد کاربرد نباشد، بنابراین آن را پاک کنید (در این کار تردید نداشته باشید و هر مورد کاربردی را که احساس کردید لازم نیست به راحتی حذف کنید چون اگر لازم باشند بعداً می توان آنها را برگرداند).

باید مراقب باشید که نصایح بالا موجب ناکارامدی و بی نظمی در نمودار ها نشود، اما به خاطر داشته باشید که منفعت فرایند تکراری این است که در ابتدا هیچ چیز 100% کامل نیست و رفته رفته در تکرارهای بعدی کامل می شود.

چند نکته که در مشورت باید رعایت شود :

- همه ایده ها را مستند کنید، حتی ایده هایی که به نظرتان مزخرف می آید را هم مستند کنید چون ممکن است این ایده های نا کارامد ایده های بسیار خوبی را در ذهن دیگر شرکت کنندگان به وجود آورد.
- هیچ ایده ای را ارزیابی یا نقد نکنید.

خلاصه

موارد کاربرد وسیله قدرتمندی برای مدل سازی وظایفی است که سیستم باید انجام دهد.
موارد کاربرد بهترین وسیله برای مشخص کردن حوزه و هدف سیستم است.
باید مراقب باشیم که موارد کاربرد بیش از حد خرد نشوند و جزئیات را نمایش ندهند. در غیر این صورت پیچیدگی افزایش می یابد.
بهترین راه برای ساخت موارد کاربرد تشکیل یک کارگاه همراه با مشتری است.

ادامه کتاب در دست ترجمه است.
منتظر ادمه کتاب باشید.