به نام یزدان پاک

طراحی سیستم‌های شیءگرا

نوشته: پروفسور گریدی بوچ

ترجمه: دکتر محسن صدیقی مشکنانی

فهرست مطالب

فهرست شکل‌ها

فهرست جدول‌ها

فصل ۰

مقدمه

پروفسور گریدی بوچ (Grady Booch) یکی از معروف‌ترین و شناخته شده‌ترین صاحبنظران در مورد مدل شیء گرا در جهان است. کتاب طراحی شیء گرا و کاربردها[1] که در سال ۱۹۹۱ میلادی توسط ایشان نوشته شده است، همچنان یکی از مشخص‌ترین متون درسی و یک مرجع عمومی برای تحلیل، طراحی و برنامه سازی شیء گرا می‌باشد.

فصل اول این کتاب به موضوع پیچیدگی (complexity) در کل و پیچیدگی نرم‌افزار و نقش تفکیک (decomposition)، تجرید (abstraction) و سلسله مراتب (hierarchy) در برخورد با آن، می‌پردازد.

در **فصل دوم** مدل شیء گرا (object model) و ارکان اصلی شامل تجرید (abstraction)، دربرگیری (encapsulation)، مؤلفه‌ای بودن (modularity) و سلسله‌مراتب (hierarchy) و همچنین ارکان فرعی آن شامل نوع بندی (typing)، توازن یا هموجودی (concurrency) و ماندگاری (persistence) تبیین شده است.

فصل سوم به شیءها (object)، کلاس‌ها (classe) و ویژگی‌های آنها و همچنین به روابط مختلف بین شیءها، بین کلاس‌ها و بین شیءها و کلاس‌ها می‌پردازد.

فصل چهارم حاوی موضوع طبقه بندی یا کلاس بندی (classification)، ارتباط آن با مدل شیء، مشکلات کلاس‌بندی، روش‌هایی برای شناسایی و تعیین شیءها، کلاس‌ها و همچنین ارتباط و رفتار آنها با یکدیگر (مکانیزم mechanism) می‌باشد.

فصل پنجم قواعد علامت‌گذاری برای چهار نمودار اصلی و دو نمودار کمکی را ارایه می‌کند. نمودارهای اصلی شامل نمودار کلاس (class diagram)، نمودار شیء (object diagram)، نمودار مؤلفه (module diagram) و نمودار فرایند (process diagram) می‌باشد. نمودارهای کمکی شامل دو نمودار تغییر حالت (state transition diagram) و نمودار زمانی (timing diagram) است.

در **فصل ششم** فرایند طراحی شیء‌گرا به‌عنوان یک فرایند فزاینده (incremental) و تکرار شونده (iterative) مورد بحث قرار گرفته و قدم‌های اصلی در این راه مطرح شده است: شناسایی و تعیین شیءها و کلاس‌ها در سطحی از تجرید، شناسایی و تعیین معنی (semantics) این کلاس‌ها و شیءها، شناسایی و تعیین روابط بین این کلاس‌ها و شیءها، و بالاخره پیاده‌سازی این کلاس‌ها و شیءها.

در **فصل هفتم** نکات عملی (pragmatics) در طراحی شیء‌گرا، نقش این روش در چرخه‌ی عمر نرم‌افزار و تأثیر آن در مدیریت نرم‌افزار مورد بررسی قرار می‌گیرد.

---

[1] *Booch, G. (1991). "Object Oriented Design with Applications", Benjamin.*

بقیه فصول کتاب بوچ (فصول هشتم تا سیزدهم) حاوی پنج کاربرد در زمینه‌های مختلف است، که هر یک طی یک فصل توسط یک زبان شیءگرا ارایه شده است.

این نوشتار ترجمه‌ی منتخبی از مطالب فصول اول تا هفتم کتاب بوچ است. منتخبی که سعی شده است ساختار و اهداف اصلی بوچ را دربر داشته باشد. بخش "خلاصه" (summary) از هر فصل به‌طور کامل ترجمه شده است. در عین حال در حاشیه متن عناوینی نیز قید شده است، به این امید که متن را برای خواننده گویاتر نماید. قواعد علامت‌گذاری UML[1] نیز که بعدها توسط بوچ و همکارانش در مؤسسه‌ی غیر انتفاعی OMG[2] به‌جای علامت‌گذاری فصل پنج ارائه گردید و اکنون عملاً به‌صورت یک استاندارد برای مدل‌سازی شیءگرا درآمده است، در قالب ۷ فصل اضافه شده است.

الحمدلله ربّ العالمین

محسن صدیقی مشکنانی

[1] *Unified Modeling Language*

[2] *Object Management Group (www.omg.org)*

فصل ۱

پیچیدگی (Complexity)

۱- ۱ پنج ویژگی سیستم‌های پیچیده *(The Five Attributes of Complex System)*

غالباً پیچیدگی فرم سلسله مراتب (hierarchy) را می‌گیرد، که در آن سیستم پیچیده از زیر سیستم‌های مرتبط به هم تشکیل می‌شود. و هر زیر سیستم نیز به نوبه‌ی خود دارای زیر سیستم‌هایی است و الی آخر. تا اینکه به عناصر اولیه (elementary) برسیم.

انتخاب اینکه در یک زیر سیستم چه اجزایی، اولیه (primitive) است، نسبتاً دلخواه بوده و بیشتر به دیدگاه ناظر بستگی دارد. چیزی که برای یک ناظر، اولیه محسوب می‌شود، ممکن است برای ناظری دیگر، از سطح بالاتری از تجرید (abstraction) برخوردار باشد.

پیوندهای درون مؤلفه‌ای (intra component linkages) عموماً قوی‌تر از پیوندهای برون مؤلفه‌ای (inter component linkage) که بین یک مؤلفه با مؤلفه‌های دیگر است، می‌باشند [8]. این تفاوت بین پیوندی‌های درون مؤلفه‌ای و برون مؤلفه‌ای موجب جدایی قسمت‌های مختلف سیستم می‌شود. به‌گونه‌ای که مطالعه‌ی مجزای هر قسمت را میسر می‌سازد.

سیستم‌های سلسله مراتبی معمولاً ترکیبی تنها از چند نوع مختلف از زیر سیستم‌ها با ترکیب‌ها و ترتیب‌های مختلف هستند[9].

> سیستم‌های پیچیده‌ای که کار می‌کنند بایستی از سیستم ساده شروع شده و به تدریج کامل شده باشند.
>
> ... سیستم پیچیده‌ای که از هیچی (scratch) طراحی شده باشد، هرگز کار نمی‌کند. باید با سیستم ساده‌ی در حال کار شروع کنید [11].

در روند تکامل سیستم‌ها، شیءهایی که زمانی پیچیده بودند، با ساخته شدن سیستم‌های بزرگتر، شیءهای ساده‌تر تلقی می‌شوند.

۱- ۲ فرم متعارف یک سیستم پیچیده *(The Canonical Form of a Complex System)*

یک مثال (خلبان)

کشف تجرید و مکانیزم‌های عمومی، فهم ما از سیستم‌های پیچیده را ساده کرده است. به عنوان مثال تنها با یک توجیه چند دقیقه‌ای یک خلبان مجرب (با درک خواص مشترک هواپیماهای مشابه و ویژگی‌های منحصر به‌فرد یک هواپیمای بخصوص) قدم به هواپیمایی می‌گذارد که قبلاً با آن پرواز نکرده است و آن را به‌سلامت هدایت می‌کند. این مثال بیانگر آن است که ما از واژه‌ی سلسله مراتب (hierarchy) استفاده می‌کرده‌ایم (گرچه سست). اما در اکثر سیستم‌های جالب توجه، سلسله مراتب‌های بسیاری حضور دارند.

ما دریافته‌ایم که سلسله مراتب‌های نوعی از (kind of) و جزئی از (part of) در هر سیستم مهم است. با دلائلی که در فصل دوم مطرح خواهد شد، سلسله مراتب نوعی از، را ساختار کلاسی (class structure) و سلسله مراتب جزئی از، را ساختار شیء (object structure) می‌نامیم.

شکل متعارف سیستم‌های پیچیده.

اگر مفاهیم ساختار کلاسی و ساختار شیء و پنج مشخصه‌ی سیستم‌های پیچیده را با هم در نظر بگیریم، درمی‌یابیم که تمام سیستم‌های پیچیده مجازاً به یک شکل متعارف (canonize form) هستند، (شکل ۱-۱).

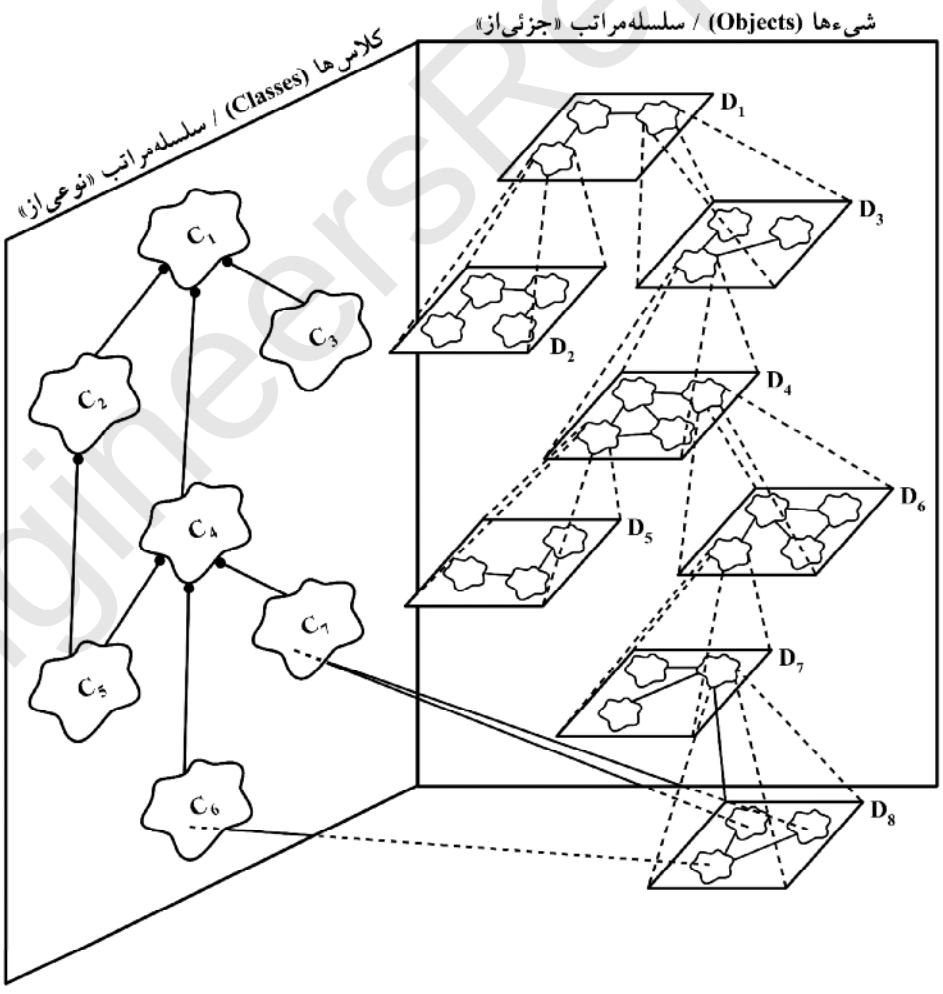
در این‌جا ما دو سلسله مراتب متعامد (or trogon) از یک سیستم را می‌بینیم: ساختار کلاسی و ساختار شیء.

عدم استقلال ساختار شیء و ساختار کلاسی از یکدیگر.

هر سلسله مراتب دارای سطوح متعددی است. کلاس‌ها و شیءهای مجردتر، از موارد ابتدایی‌تر ساخته شده‌اند. به‌ویژه در بین قسمت‌های ساختار شیء روابط کاملاً تعریف شده‌ای بین شیءهای آن سطح وجود دارد. با نگاه به داخل هر سطح، سطح دیگری از پیچیدگی معین می‌شود. توجه کنید که ساختارهای کلاس و شیء کاملاً مستقل از یکدیگر نیستند. بلکه هر شیء از ساختار شیء بیانگر یک رویداد (instance) خاص از همان کلاس است. همچنان‌که در شکل نمایش داده شده است. در داخل یک سیستم، تعداد شیءها خیلی بیشتر از تعداد کلاس‌هاست. بنابراین با نشان دادن سلسله مراتب‌های "جزئی از" و "نوعی از" ما صراحتاً افزونگی یا زوائد (redundancy) سیستم مورد نظر را نشان می‌دهیم. اگر ساختار کلاسی از سیستم را معلوم نمی‌کردیم، ناچار بودیم دانش خود را در مورد ویژگی‌های هر جزء را به‌طور جداگانه تکرار کنیم. با منظور کردن ساختار کلاسی، تمام این ویژگی‌های مشترک را جمع می‌کنیم.

تعداد شیءها و کلاس‌ها.

رفع افزونگی.



شکل ۱-۱: فرم متعارف یک سیستم پیچیده

سیستم پیچیده‌ی موفق

بر اساس تجربه‌ی ما سیستم‌های پیچیده‌ی نرم‌افزاریِ موفق، آنهایی هستند که طرحشان صراحتاً دارای ساختارهای کلاسی، و شیء خوب (well engineered) بوده و به‌علاوه پنج ویژگی سیستم‌های پیچیده را دربر داشته باشد. به‌عبارت

دیگر به ندرت سیستم نرم‌افزاری را دیده‌ایم که بدون این نکات، به موقع، در محدوده‌ی بودجه‌ی پیش بینی شده و مطابق با خواسته‌های تعیین شده، ایجاد شده باشد.

۱- ۳ محدودیت‌های انسان در برخورد با پیچیدگی‌ها

(The Limitation of Human Capacity for Dealing with Complexity)

این سؤال مطرح است که اگر می‌دانیم طرح سیستم‌های نرم‌افزاری چگونه باید باشد، پس چرا در ایجادِ موفقِ چنین سیستم‌هایی با مسائل جدی روبرو هستیم؟ همچنان‌که در فصل دوم بحث خواهیم کرد، این مفهوم پیچیدگی سازمان یافته‌ی (organized complexity) نرم‌افزار، که ما مجموعه‌ی اصول اولیه‌اش را مدل شیء (object model) می‌نامیم، نسبتاً جدید است. با این‌حال عامل دیگری که حکم‌فرمایی می‌کند محدودیت بنیادی انسان در برخورد با پیچیدگی است.

معمای بنیادی.

متأسفانه این غیر ممکن است که یک فرد تنها تمام جزئیات را با هم بتواند در نظر بگیرد [12] , [13]. ما با یک معمای بنیادی روبرو هستیم:

> پیچیدگی نرم‌افزارهای مورد تقاضا هر روز بیشتر می‌شود. در حالی‌که ما در برخورد با این پیچیدگی‌ها بطور بنیادی محدود هستیم. ما چه خواهیم کرد؟

۱- ۴ به نظم آوردن آشفتگی *(Bringing order to Chaos)*

همچنانکه Dijkstre پیشنهاد می‌کند:

> "روش چیره شدن بر پیچیدگی از زمان‌های قدیم شناخته شده است: فاصله بیانداز و پیروز شو" [14]. در موقع طراحی یک سیستم نرم‌افزاری پیچیده، تفکیک آن به قسمت‌های کوچک و کوچکتر بسیار ضروری است. به‌گونه‌ای که بتوانیم هر یک را به‌طور مستقل مورد توجه قرار دهیم.

تفکیک الگوریتمیک در مقابل تفکیک شیءگرا

راه درست تفکیک یک سیستم بزرگ کدامست؟ به‌وسیله‌ی الگوریتم‌ها یا بوسیله‌ی شیءها؟. جواب صحیح اینست که هر دو دیدگاه مهم هستند: دیدگاه الگوریتمی، دیدگاه حوادث را برجسته می‌کند، و دیدگاه شیءگرا روی عامل‌ها (agent) تأکید می‌کند. (خواه عامل‌هایی که موجب یک عمل می‌شوند و خواه آنهایی که عملی رویشان انجام می‌شود). با این‌حال این حقیقت وجود دارد که ما نمی‌توانیم یک سیستم پیچیده را همزمان با هر دو راه بسازیم، بخاطر این‌که این دو دیدگاه عمود به یکدیگرند. ما باید تفکیک سیستم را با یکی از این دو دیدگاه (الگوریتمی یا شیءگرا) شروع کنیم و سپس از ساختار حاصل به‌عنوان چهارچوبی برای بیان دیدگاه دیگر استفاده کنیم.

دیدگاه الگوریتمی دیدگاه شیءگرا.

عدم امکان استفاده‌ی همزمان از دو دیدگاه

تجربه (ی نویسنده) ما را به این هدایت می‌کند که ابتدا دیدگاه شیءگرا را به‌کار بریم. به‌دلیل این‌که این دیدگاه کمک بیشتری برای سازمان دادن به پیچیدگی ذاتی سیستم‌های نرم‌افزاری می‌کند همچنان‌که این دو دیدگاه به ما کمک کرده است تا بتوانیم پیچیدگی سازمان یافته‌ی سیستم‌های پیچیده و متنوعی مثل کامپیوترها، گیاهان، کهکشان‌ها و موضوعات بزرگ اجتماعی را تبیین کنیم.

دلایل مدل شیء.

سازمان دادن به پیچیدگی.

تفکیک شیءگرا، سیستم‌های کوچکتری را از طریق استفاده‌ی مجدد از مکانیزم مشترک ارایه می‌کند، بنابراین موجب صرفه‌جویی مهم (از expression) می‌شود.

صرفه جویی در بیان.

سیستم‌های شیءگرا انعطاف بیشتری در مقابل تغییرات دارند و لذا بهتر می‌توانند در طول زمان تغییر کنند به‌دلیل این‌که طراحی آنها بر اساس فرم‌های پایدار میانی است.

انعطاف در مقابل تغییر.

تفکیک شیءگرا به‌طور قابل ملاحظه ریسک ساختن سیستم‌های نرم‌افزاری بزرگ را کم می‌کند، به‌دلیل این‌که طراحی آنها به‌گونه‌ای است که بتوانند از سیستم‌های کوچکتر که ما در مورد آنها مطمئن هستیم به سیستم‌های بزرگتر رشد یابند.

تقلیل ریسک.

به‌علاوه تفکیک شیءگرا مستقیماً پیچیدگی ذاتی نرم‌افزار را هدف قرار می‌دهد (با کمک به ما برای تصمیم‌گیری‌های هوشمندانه در مورد جدا کردن امورِ مورد توجه ما در یک فضای بزرگ از حالت‌ها).

برخورد با پیچیدگی.

معنی طراحی *(The Meaning of Design)*

موستوو ([37] Mostove) هدف از طراحی را ساخت سیستمی با خصوصیات زیر می‌داند:

- سیستمی که مشخصاتِ وظیفه‌ی خاصی را جواب‌گویی می‌کند؛
- جواب‌گوی محدودیت‌های محیط نهایی است؛
- جواب‌گوی خواسته‌های صریح و ضمنی در مورد عملکرد و استفاده از منابع است؛
- معیارهای طراحی صریح یا ضمنی را جواب‌گوست؛
- محدودیت‌های موجود روی خود فرایند طراحی، مثل زمان یا هزینه، یا ابزارهای موجود برای طراحی را ارضاء می‌کند.

اهمیت ساختن مدل *(The Importance of Model Building)*

ساختن مدل در بین نظام‌های مختلف مهندسی به‌طور وسیعی مورد قبول قرار گرفته است؛ عمدتاً به‌خاطر این‌که ساخت مدل به اصول تفکیک (decomposition)، تجرید (abstraction)، و سلسله مراتب (hierarchy) متوسل می‌شود [38].

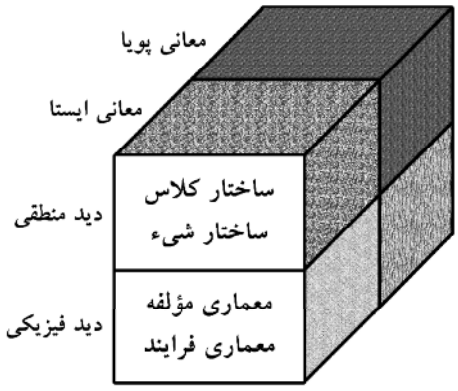
عناصر روش‌های طراحی نرم‌افزار *(The Elements of Software Design Methods)*

علی‌رغم وجود تفاوت‌ها در روش‌های مختلف طراحی نرم‌افزار (طراحی بالا به پایین، ساخت‌یافته؛ طراحی مبتنی بر داده‌ها - data driven-؛ طراحی شیءگرا)، همگی عناصر مشترکی دارند:

- علامت‌گذاری (notation): زبان بیان هر مدل؛
- فرایند (process): رهنمودهایی برای ترتیب ساخت مدل‌ها؛
- ابزار (tools): وسیله‌ای که کار یکنواخت ساختن مدل را حذف، و قواعدی را در مورد خود مدل‌ها تحمیل می‌کند، به‌طوریکه خطاها و ناسازگاری نمایان گردند.

۱- ۵ مدل‌های طراحی شیءگرا *(The Models of Object-Oriented Design)*

به‌دلیل اهمیت ساختمان مدل در ساختن سیستم‌های پیچیده، طراحی شیءگرا مجموعه‌ای غنی از مدل‌ها را ارائه می‌کند. مدل‌های طراحی شیءگرا در شکل زیر خلاصه شده است:



شکل ۱-۲: مدهای طراحی شیءگرا

۱- ۶ خلاصه‌ی فصل اول از کتاب طراحی شیءگرا / پیچیدگی *(Complexity)*

- نرم‌افزار ذاتاً پیچیده است؛ پیچیدگی سیستم‌های نرم‌افزاری غالباً فراتر از ظرفیت عقلانی انسان است؛
- کار گروه ایجاد و توسعه‌ی نرم‌افزار، مهندسیِ ارائه‌ی (illusion) سادگی است؛

- پیچیدگی غالباً فرم سلسله مراتب را می‌گیرد؛ درست کردن مدل‌های سلسله مراتبی "نوعی از" (kind of) و "جزئی از" (part of) برای سیستم‌های پیچیده مفید است؛
- سیستم‌های پیچیده عموماً حاصل فرم‌های پایدار میانی هستند؛
- محدودیت‌های اساسی برای درک انسان وجود دارد؛ با استفاده از تفکیک (decomposition)، تجرید (abstraction)، و سلسله مراتب (hierarchy) می‌توان این محدودیت را نشانه گرفت؛
- به سیستم‌های پیچیده هم با تأکید بر چیزها (things) و هم با تأکید بر فرایندها (processes) می‌توان نگاه کرد؛ دلایل قطعی برای تفکیک شیءگرا (object-orient decomposition) وجود دارد، که در آن ما به جهان به‌صورت مجموعه‌ای با معنی از شیءها نگاه می‌کنیم، مجموعه‌ای که با یکدیگر همکاری می‌کنند تا رفتارهای برتر (higher level behavior) حاصل شود؛
- طراحی شیءگرا روشی است که ما را به تفکیک شیءگرا هدایت می‌کند؛ طراحی شیءگرا علامت‌گذاری (notation) و همچنین فرایندی (process) را برای ساختن سیستم‌های نرم‌افزاری بزرگ تعریف می‌کند، و مجموعه‌ای از مدل‌های منطقی و فیزیکی غنی را ارائه می‌کند، که توسط آن ما می‌توانیم درباره‌ی جنبه‌های مختلف سیستم مورد نظر، استدلال کنیم.

فصل ۲

مدل شیء (The Object Model)

اصول مدل شیء.

طراحی شیءگرا بر یک پایه‌ی مهندسی محکم (soundly)، که مجموعه‌ی عناصرش را مدل شیء می‌نامیم (object model) بنا شده است. مدل شیء دربر گیرنده‌ی اصول تجرید (abstraction)، دربرگیری (encapsulation)، پیمانه‌ای یا مؤلفه‌ای بودن (modularity)، سلسله‌مراتب (hierarchy)، نوع‌بندی (typing)، هم‌وجودی (concurrency)، و ماندگاری (persistence) می‌باشد. هیچ یک از اصول فوق به خودی خود جدید نیستند. نکته‌ی مهم در مورد مدل شیء اینست که این عناصر به‌صورت همکاری کننده در کنارهم آورده شده‌اند.

طراحی شیءگرا اصولاً با دیدگاه سنتی طراحی ساخت‌یافته متفاوت است، طراحی شیءگرا نیاز به تفکر دیگری در مورد تفکیک (decomposition) دارد؛ و معماری‌هایی از نرم‌افزار را ارائه می‌کند که از فرهنگ طراحی ساخت‌یافته بسیار فراتر است. این تفاوت ناشی از این حقیقت است که متد طراحی ساخت‌یافته بر اساس برنامه‌سازی ساخت‌یافته درست شده است؛ در حالی‌که طراحی شیءگرا بر اساس برنامه‌سازی شیءگرا. متأسفانه برنامه‌سازی شیءگرا معنی‌های متفاوتی برای افراد متفاوت دارد. در اینجا نشان می‌دهیم که طراحی شیءگرا چی هست و چی نیست. و چگونه با متدهای دیگر طراحی فرق می‌کند (به‌وسیله‌ی استفاده‌اش از عناصر هفتگانه‌ی مدل شیء).

۲- ۱ عوامل شکل‌گیری مدل شیء *(Foundations of the Object Model)*

عواملی که مدل شیء را موجب شدند.

- پیشرفت در معماری کامپیوتر، از جمله سیستم‌های قابلیت (capability system) و حمایت سخت‌افزار از مفاهیم سیستم عامل؛
- پیشرفت در زبان‌های برنامه سازی، آنچنان‌که در Ada, CLU, smalltalk, simule ارائه شد؛
- پیشرفت در روش‌های برنامه‌سازی، از جمله عملکرد پیمانه‌ای (مؤلفه‌ای) (modularization) و پنهان کردن اطلاعات (information hiding)؛
- پیشرفت در مدل‌های بانک‌های اطلاعاتی؛
- تحقیق در هوش مصنوعی؛
- پیشرفت در فلسفه و علوم ادراکی (cognitive science).

۲- ۲ برنامه‌سازی، طراحی و تحلیل شیءگرا *(OOP , OOD , OOA)*

برنامه‌سازی شیءگرا یک روش پیاده‌سازی است که در آن برنامه‌ها به‌صورت مجموعه‌ای از شیءها که با یکدیگر همکاری می‌کنند، سازمان داده شده‌اند. هر یک از مجموعه‌ی شیءها نمایشگر یک رویداد (instance) از رده (class)

هستند، و رده یا کلاس‌ها اعضاء سلسله مراتبی از کلاس‌ها هستند که توسط روابط وراثت (inheritance relationship) متحد شده‌اند.

طراحی شیءگرا (object–oriented design) یک روش طراحی است که فرایند تفکیک شیءگرا (object oriented decomposition) و همین یک علامت‌گذاری برای رسم مدل‌های منطقی و فیزیکی و همینطور مدل‌های ایستا و پویا از سیستم مورد نظر را دربر می‌گیرد.

> "Object–oriented design is a method of design encompassing the process of object–oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic model of the system under design."

تحلیل شیءگرا (object–oriented analysis) یک روش تحلیل است که خواسته‌ها (requirements) را از نقطه نظر کلاس‌ها و شیءهایی که در لغت‌نامه‌ی زمینه‌ی مسأله یافت می‌شود، مورد بررسی قرار می‌دهد.

نحوه‌ی ارتباط این‌ها با یکدیگر: اصولاً محصول تحلیل شیءگرا به‌عنوان مدلی که از آن می‌توانیم طراحی شیءگرا را آغاز کنیم می‌تواند عمل کند؛ سپس محصول طراحی شیءگرا می‌تواند برای پیاده‌سازی کامل سیستم، با استفاده از روش‌های برنامه‌سازی شیءگرا مورد استفاده قرار گیرد.

۲- ۳ ارکان مدل شیء *(Elements of Object Model)*

مدل شیء، چهارچوب مفهومی (conceptual framework) هر چیز شیءگرا ست. این مدل شیء چهار عنصر یا رکن اصلی دارد:

- تجرید (abstraction)؛
- دربرگیری (encapsulation)؛
- مؤلفه‌ای یا پیمانه‌ای بودن (modularity)؛
- سلسله مراتب (hierarchy).

منظور از "اصلی" اینست که مدلی که هر یک از ارکان فوق را نداشته باشد شیءگرا نیست.

مدل شیء دارای سه رکن یا سه عنصر فرعی نیز می‌باشد:

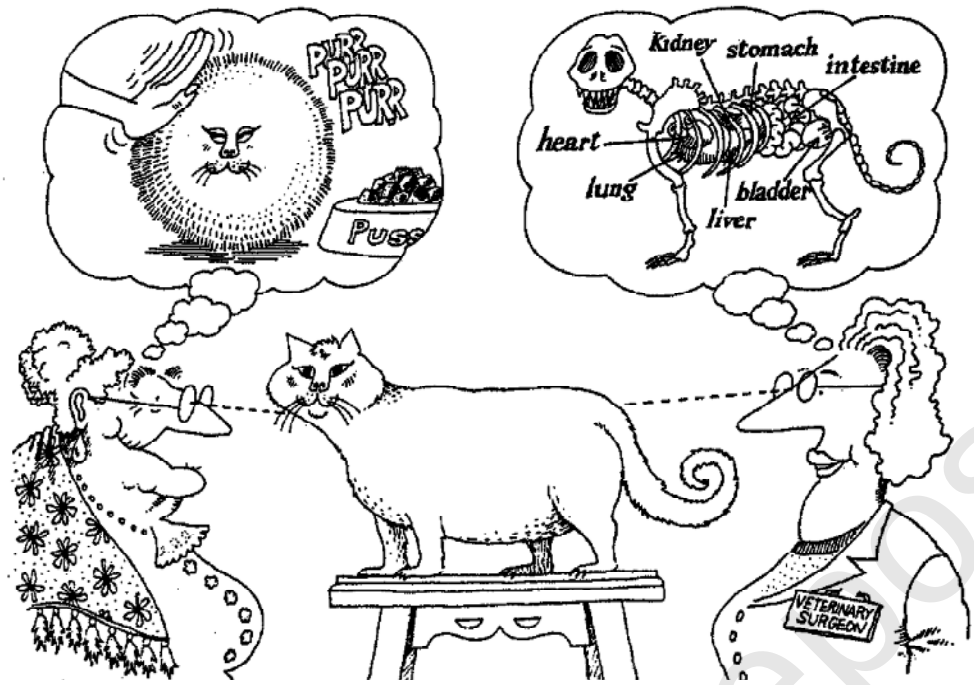
- نوع بندی (typing)؛
- توازی / هم‌وجودی (concurrency)؛
- ماندگاری (persistence).

منظور از "فرعی" اینکه هریک از اینها برای مدل شیء مفید هستند، اما ضروری نیستند.

بدون این چهارچوب مفهومی، شما ممکن است با زبانی مثل Smalltalk, Object Pascal, C++, CLOS یا Ada برنامه‌نویسی کنید، ولی طراحی شما بوی طراحی شبیه FORTRAN, Pascal ویا C را داشته باشد. در اینصورت شما قدرت بیان زبان شیءگرا یا زبان مبتنی بر شیء (object–based) را استفاده نکرده‌اید یا بد استفاده کرده‌اید. و احتمالاً بر پیچیدگی مسأله‌ی مورد نظرتان مسلط نشده‌اید.

تجرید *(Abstraction)*

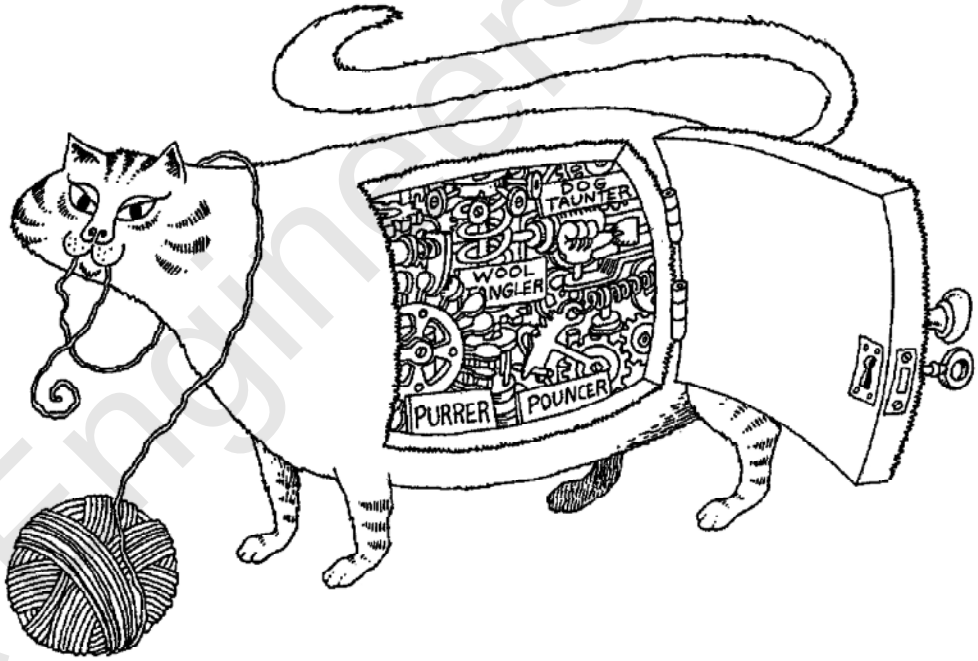
بسته به دیدگاه ناظر، یک تجرید مبین ویژگی‌های اساسی یک شیء است. که آن شیء را از انواع دیگر شیء متمایز می‌کند؛ و لذا حدود مفهومی (conceptual boundaries) تعریف شده‌ای را فراهم می‌کند.



شکل ۲-۱: تجرید روی ویژگی‌های اصلی شیء با توجه به دیدگاه ناظر، تکیه می‌کند

دربرگیری *(Encapsulation)*

دربرگیری، فرایند مخفی نگاه داشتن تمام جزئیات از یک شیء است که مشارکتی در ویژگی‌های اصلی شیء نداشته باشد.



شکل ۲-۲: دربرگیری جزئیات پیاده‌سازی شیء را پنهان می‌کند

پیمانه‌ای (مؤلفه‌ای) بودن *(Modularity)*

پیمانه‌ای بودن خصوصیتی است که موجب تفکیک سیستم به مجموعه‌ای از مؤلفه‌های (modules) منسجم (cohesive) و با وابستگی اندک (loosely coupled) می‌گردد.

شکل ۲-۳: مؤلفه‌ای بودن، تجریدها را در واحدهای مجزا تفکیک می‌کند

سلسله مراتب *(Hierarchy)*

تجریدهای متعدد.

تجرید چیز خوبی است، اما به‌جز در مورد کاربردهای بسیار ابتدایی، ممکن است تجریدهای متعددی را پیدا کنیم که در یک زمان نتوانیم آنها را مورد شمول قرار دهیم (comprehend). در برگیری کمک می‌کند تا با پنهان کردن دید داخلی تجریدها، بتوانیم با پیچیدگی این تجریدهای متعدد برخورد کنیم. پیمانه‌ای بودن با تفکیک و خوشه‌بندی (clustering) تجریدهایی که منطقاً به یکدیگر مربوط هستند، کمک می‌کند، اما اینها هنوز کافی نیست. مجموعه‌ای از تجریدها معمولاً یک سلسله مراتب را تشکیل می‌دهند، و ما به‌وسیله‌ی تشخیص این سلسله مراتب‌ها در طراحی خود، به‌طور قابل ملاحظه‌ای فهم خود از مسأله را ساده می‌کنیم.

سلسله مراتب.

سلسله مراتب یک رتبه‌بندی و مرتب کردن تجریدها ست.

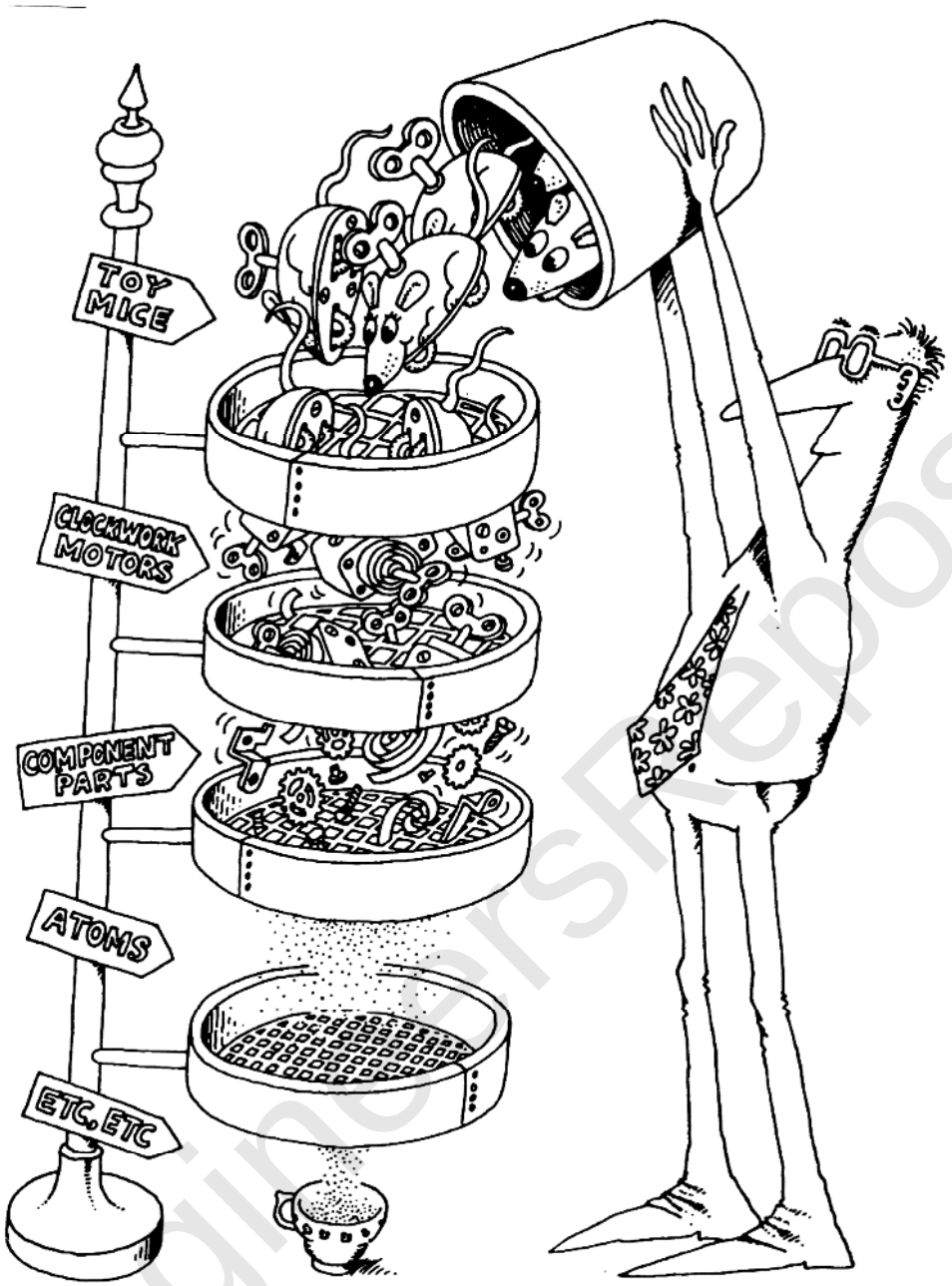
مهمترین سلسله مراتب‌ها.

در یک سیستم پیچیده مهمترین سلسله مراتب‌ها عبارتند از: ساختار کلاسِ آن سیستم (class structure) یا به عبارت دیگر سلسله مراتبِ "نوعی از" (the "kind of" hierarchy) و ساختار شیء (object structure) یا به‌عبارت دیگر سلسله مراتبِ "جزئی از" (the "part of" hierarchy) از آن سیستم.

وراثت مهمترین سلسله مراتبِ "نوعی از"، و یک عنصر اصلی از سیستم‌های شیءگرا می‌باشد.

مثال از تسلسل: وراثت یگانه.

اساساً وراثت یک رابطه در بین کلاس‌ها تعریف می‌کند، که در آن یک کلاس در ساختار یا رفتارِ تعریف شده در یک یا چند کلاس (که سلسله مراتب یگانه single inheritance و سلسله مراتب چندگانه multiple inheritance نامیده می‌شود) شریک می‌شود. بنابراین وراثت نمایش دهنده‌ی سلسله مراتبی از تجریدها ست.

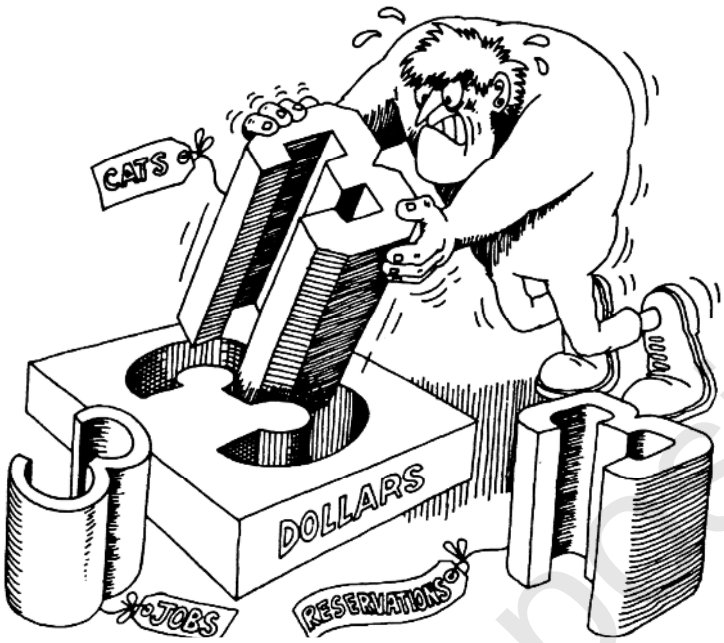


شکل ۲-۴: تجریدها یک سلسله‌مراتب را تشکیل می‌دهند

نوع‌بندی *(Typing)*

همچنان‌که Deutch پیشنهاد می‌کند، "نوع، توصیفی دقیق از خصوصیات ساختاری یا رفتاری است که در مجموعه‌ای از موجودیت‌ها مشترک است" [65]. در اینجا ما از واژه‌های نوع و کلاس را به‌صورت معادلِ هم استفاده می‌کنیم.

نوع‌بندی تحمیل (enforcement) کلاسی از شیء‌هاست، به ترتیبی که تعویض شیء‌ها از انواع مختلف با یکدیگر ممکن نباشد، یا این‌که حداکثر به‌صورت‌های کاملاً محدودی با یکدیگر تعویض گردند.



شکل ۲-۵: نوع‌بندی قوی[1] از تداخل تجریدها جلوگیری می‌کند

توازی یا هم‌وجودی *(Concurrency)*

برای بعضی از مسأله‌ها، ممکن است یک سیستم خودکار، حوادث متعددی را به‌طور همزمان دنبال نماید. در بعضی از مسائل حجم محاسبات ممکن است بیشتر از توان اجرایی یک پردازنده باشد. در هر دو حالت استفاده از مجموعه‌ای از کامپیوترهای توزیع شده (distributed) یا ریزپردازنده‌ها، طبیعی به‌نظر می‌رسد. (در سیستم‌هایی که تنها روی یک پردازنده اجرا می‌شوند، داشتن چندین ریسمان کنترل موازی (concurrent threads of control) به‌وسیله‌ی الگوریتم‌های برهه‌بندیِ زمان پردازنده (time sliding) نمود می‌یابد.

در بالاترین سطح تجرید، برنامه‌سازی شیءگرا می‌تواند مسأله توازی (هم‌وجودی) را با پنهان کردن توازی در داخل تجریدهای با قابلیتِ استفاده‌ی مجدد (reusable abstraction)، برای اغلب برنامه‌سازان سبکتر کند [72].

تأکید برنامه‌سازی شیءگرا، تأکید توازی (هم‌وجودی).

در حالی‌که برنامه‌سازی شیءگرا روی تجرید داده‌ها، دربرگیری و وراثت تکیه می‌کند، توازی (هم‌وجودی) روی "تجرید فرارونه‌ها" (process abstraction) و هماهنگی آنها (synchronization) تأکید می‌کند [74].

شیء فعال.

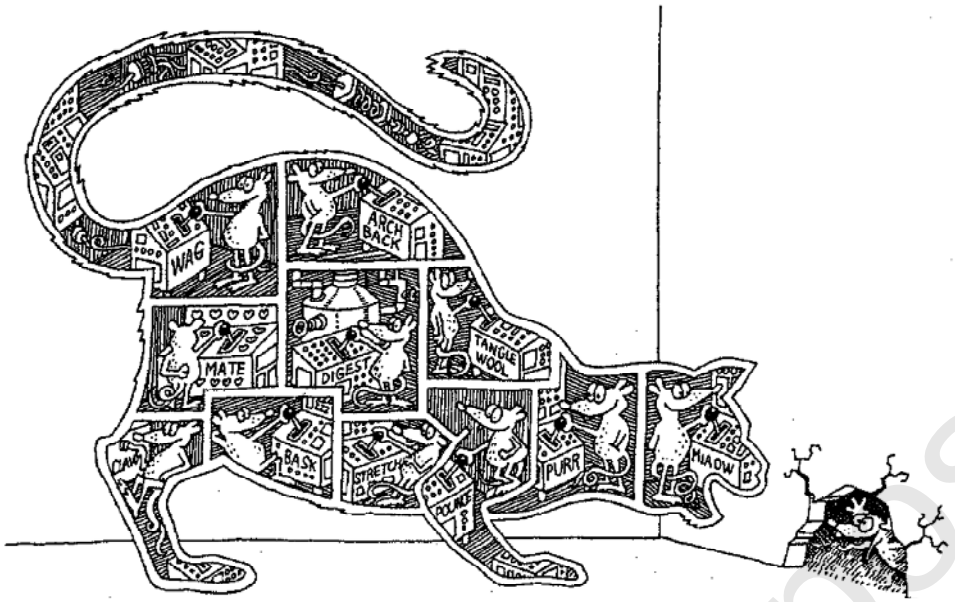
شیء مفهومی است که این دو دیدگاه مختلف را یکی می‌کند: هرشیء (که از تجرید دنیای واقع به‌دست آمده است) ممکن است یک ریسمان کنترل جداگانه (یک تجرید فرارونه) را نمایش دهد. چنین شیءهایی فعال (active) نامیده می‌شوند. در سیستمی که بر اساس طراحی شیءگرا می‌باشد، ما می‌توانیم دنیا را به‌صورت مجموعه‌ای از شیءها که با هم همکاری می‌کنند، تصور کنیم؛ شیءهایی که برخی از آنها فعال هستند و لذا به‌عنوان مراکزی از فعالیت‌های مستقل عمل کنند. با توجه به این مطالب، توازی را به‌صورت زیر تعریف می‌کنیم:

توازی (هم‌وجودی)

توازی (هم‌وجودی) خاصیتی است که یک شیء فعال را از یک شیء غیر فعال تمیز می‌دهد.

---

[1] *strong typing*

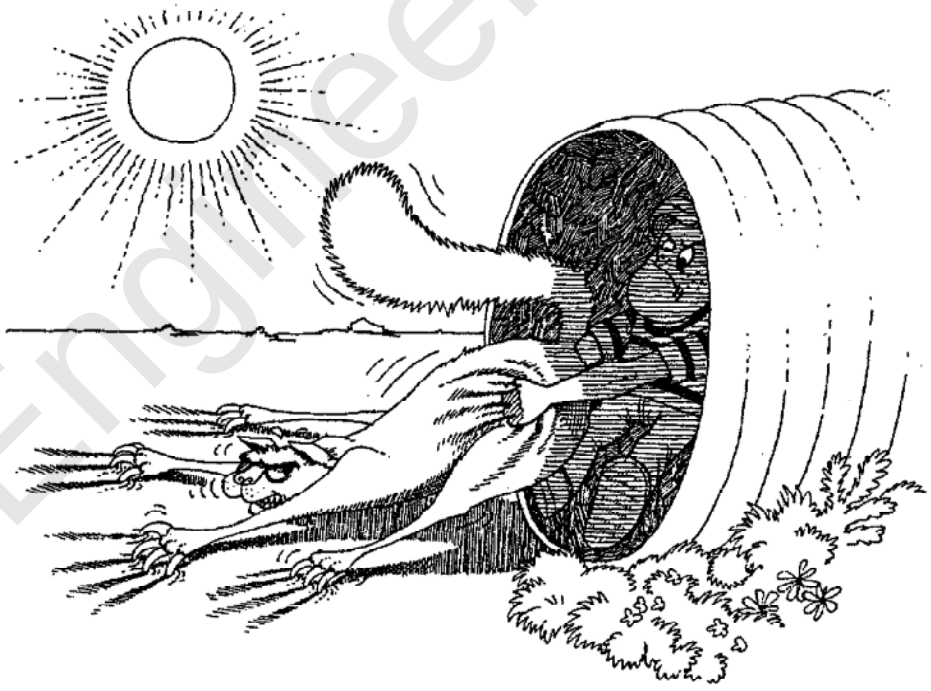


شکل ۲-۶: توازی اجازه می‌دهد تا شیء‌های مختلف در یک زمان عمل کنند

ماندگاری *(Persistence)*

در نرم‌افزار، یک شیء مقداری از فضا را اشغال کرده و برای مقدار مشخصی از زمان وجود دارد.

ماندگاری خصوصیتی از شیء است که به‌واسطه‌ی آن وجود شیء بر زمان غلبه می‌کند (یعنی شیء بعد از توقف حیات به‌وجودآورنده‌اش به حیات خود ادامه می‌دهد) و/یا وجود شیء بر فضا غلبه می‌کند (یعنی محل شیء از فضای آدرس‌هایی که در ابتدا به‌وجود آمده بود، حرکت می‌کند).



شکل ۲-۷: ماندگاری حالت[1] و کلاس شیء را در رابطه با زمان و مکان حفظ می‌کند

[1] *state*

کاربردهای مدل شیء *(Applications of the Object Model)*

مدل شیء ثابت کرده است که در زمینه‌های متعددی قابل به‌کارگیری است (جدول زیر). طراحی شیءگرا شاید تنها روشی باشد که امروز برای مقابله با پیچیدگی سیستم‌های خیلی بزرگ در اختیار ماست. ممکن است طراحی شیءگرا برای بعضی زمینه‌ها خیلی قابل توجیه نباشد. البته نه به دلایل تکنیکی، بلکه به دلایل غیر تکنیکی مثل عدم وجود افراد آموزش دیده‌ی مناسب، یا عدم وجود محیط ایجاد و توسعه‌ی (development) مناسب.

جدول ۲-۱: کاربردهای مدل شیء

Air traffic control	Document preparation	Petroleum engineering
Animation	Expert system	Reusable software components
Avionics	Film and stage storyboarding	Robotics
Banking and insurance software	Hypermedia	Software development environments
Business data processing	Inane recognition	Space station software
Chemical process control	Investment strategies	Spacecraft and aircraft simulation
Command and control systems	Mathematical analysis	Telecommunications
Computer aided design	Medical electronics	Telemetry systems
Computer aided education	Office automation	User interface design
Computer integrated manufacturing	Operating system	VLSI design
Database		

۲- ۴ خلاصه‌ی فصل دوم ازکتاب طراحی شیءگرا / مدل شیء *(The Object Model)*

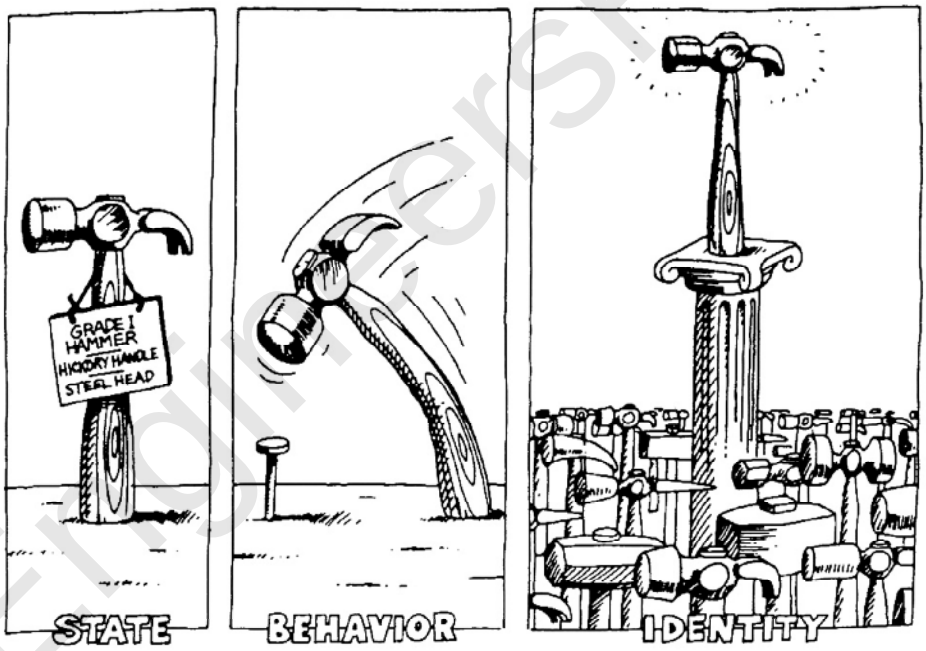
- بلوغ مهندسی نرم‌افزار منجر به روش‌های تحلیل، طراحی و برنامه‌سازی شیءگرا شده است که همگی برنامه‌سازی در مقیاس بزرگ را نشانه گرفته‌اند؛
- طُرق مختلف برنامه‌سازی وجود دارد: رویه‌گرا (procedure–oriented)، شیءگرا (object–oriented)، منطق‌گرا (logic–oriented) و محدودیت‌گرا (constraint–oriented)؛
- مدل شیء چهارچوب مفهومی برای روش‌های شیءگرا را فراهم می‌کند؛ مدل شیء دربرگیرنده‌ی اصول زیر است: تجرید (abstraction)، دربرگیری (encapsulation)، پیمانه‌ای بودن (modularity)، سلسله‌مرتب (hierarchy)، نوع بندی (typing)، توازی / هم‌وجودی (concurrency)، و ماندگاری (persist tance)؛
- تجرید، مبین خصوصیات اصلی یک شیء است که آن‌را از تمام انواع شیءهای دیگر متمایز می‌کند و در نتیجه محدوده‌های مفهومیِ کاملاً تعریف شده‌ای، را با توجه به دید ناظر، فراهم می‌کند؛
- دربرگیری، فرایند پنهان کردن تمام جزئیات شیء است که در خصوصیات اصلی شیء نقشی ندارند؛
- پیمانه‌ای بودن، خصوصیتی از یک سیستم است که منجر به تفکیک سیستم به مجموعه‌ای از مؤلفه‌های منسجم و با وابستگی کم شده است؛
- سلسله مراتب، یک رتبه‌بندی یا مرتب کردن تجریدهاست؛
- نوع‌بندی، تحمیل کلاس شیءهاست، به‌ترتیبی که شیءهای نوع‌های متفاوت با یکدیگر نتوانند عوض شوند، یا حداکثر این‌که تنها از طُرق بسیار محدود عوض شوند؛
- توازی (هم‌وجودی)، خصوصیتی است که یک شیء فعال را از شیء غیر فعال متمایز می‌کند؛
- ماندگاری خصوصیتی از یک شیء است که از طریق آن، وجود شیء از زمان و / یا مکان فراتر می‌رود؛
- کاربرد مدل شیء منجر به سیستم‌هایی می‌شود که پنج ویژگی سیستم‌های پیچیده‌ی کاملاً ساخت‌یافته را دارد.

فصل ۳

کلاس‌ها و شیء‌ها (Classes & Objects)

۳- ۱ ماهیت شیئی *(The Nature of an Object)*

شیء دارای حالت (state)، رفتار (behavior) و هویت (identity) است؛ ساختار (structure) و رفتار شیء‌های مثل هم در کلاس (class) مشترک آنها تعریف می‌شود؛ واژه‌های رویداد (instance) و شیء، قابل تعویض با یکدیگرند.



شکل ۳-۱: شیء دارای حالت است، رفتارهای کاملاً تعریف شده‌ای را از خود نشان می‌دهد و دارای هویت (شناسه) منحصربه‌فردی است

حالت شیء *(State)*

حالت یک شیء دربرگیرنده‌ی تمام خاصیت‌های (معمولاً ایستا) آن شیء و همین‌طور مقادیر (معمولاً پویا) جاری هر یک از این خاصیت‌ها است.

رفتار شیء *(Behavior)*

هیچ شیء به‌صورت منفرد و منزوی وجود ندارد. شیءها مورد تأثیر شیءهای دیگر قرار می‌گیرند و یا شیءهای دیگر را مورد تأثیر قرار می‌دهند. لذا می‌توان گفت:

> رفتار یک شیء عبارت از اینست که چگونه عمل می‌کند و یا عکس‌العمل نشان می‌دهد، برحسب تغییرات حالتش و ارسال پیام (message passing). به‌عبارت دیگر، رفتار شیء کاملاً به‌وسیله‌ی عمل‌هایش (its action) تعریف می‌شود [7].

عمل ≈ پیام.

در زبان‌هایی مثل Smalltalk، صحبت از این می‌کنیم که یک شیء به شیء دیگر پیام می‌فرستد. عموماً یک پیام، عملی است که یک شیء روی شیء دیگر انجام می‌دهد، گرچه مکانیزم به‌کار رفته متفاوت است. برای منظور ما، واژه‌ی عمل (operation) و پیام (message) می‌توانند به‌جای یکدیگر بکار روند.

روش‌ها (methods).
روش ≈ تابع عضویت

در زبان‌های برنامه‌سازی شیءگرا (object–oriented) و مبتنی بر شیء (object–based) عملیاتی که مشتری‌ها (clients) روی شیءها ممکن است انجام دهند، نوعاً به‌صورت روش‌ها (methods) که قسمتی از کلاس آن شیء است تعریف می‌شوند. C++ از واژه‌ی تابع عضویت (member function) برای بیان همین مفهوم استفاده می‌کند، لذا ما از واژه‌های روش (method) و تابع عضویت (member function) به‌صورت قابل تعویض با یکدیگر استفاده می‌کنیم.

اعمال متداول روی شیءها (معنی عملیات)

در عمل نوعاً یک مشتری (client) پنج عمل زیر را روی یک شیء انجام می‌دهد:

- تغییر دهنده (modifier): عملی که حالت (state) یک شیء را تغییر می‌دهد. (عمل نویسنده)؛
- انتخاب کننده (selector): عملی که به حالتی از شیء دسترسی پیدا می‌کند ولی آن حالت را تغییر نمی‌دهد. (عمل خواننده)؛
- تکرارگر (integrator): عملی که اجازه می‌دهد تمام قسمت‌های شیء با یک ترکیب کاملاً مشخصی، مورد دسترسی قرار گیرند.

در بعضی زبان‌های برنامه‌سازی مثل CLOS, C++, Smalltalk معرفی دو نوع دیگر از عملیات ممکن است:

- سازنده (constructor): عملی که یک شیء را به‌وجود می‌آورد / یا به حالتش مقدار اولیه می‌دهد؛
- تخریب کننده (destructor): عملی که حالت یک شیء را رها (free) می‌کند یا خود شیء را نابود می‌کند.

شیءها به‌عنوان ماشین

وجود حالت در شیء این معنی را می‌دهد که ترتیب عملیات مهم است. این نکته، این مطلب را به ذهن می‌آورد که هر شیء مثل یک ماشین مستقل کوچک است. در واقع برای بعضی شیءها، این ترتیب زمانی عملیات آنقدر فراگیر و نافذ است که می‌توانیم رفتار شیء را به‌وسیله‌ی ماشین با حالت‌های محدود (finite state machine) مشخص کنیم.

شیءهای فعال و منفعل *(Active and passive object)*

شیءها ممکن است فعال (active) و یا منفعل (passive) باشند. شیء فعال آن است که ریسمان کنترلی (thread of control) مالِ خود را داشته باشد، در حالی‌که شیء منفعل این‌طور نیست. شیءهای فعال عموماً خودمختار (autonomous) هستند. بدین معنی که بدون این‌که شیء دیگری روی آنها عملی انجام دهد، خودش می‌تواند رفتاری را نشان دهد. از سوی دیگر شیءهای منفعل (passive) تنها وقتی که صراحتاً شیء دیگری روی آنها عمل می‌کند، دست‌خوش یک تغییر حالت شوند. بدین ترتیب شیءهای فعال به‌عنوان بنیان کنترل در سیستم‌ها عمل می‌کنند. اگر سیستم ما شامل چند ریسمان کنترل باشد، در این‌صورت ما معمولاً چند شیء فعال داریم. سیستم‌های ردیفی دقیقاً تنها یک شیء فعال در هر لحظه دارند (یعنی دقیقاً یک ریسمان کنترل).

هویت / شناسه *(Identity)*

هویت، خاصیتی از شیء است که آن شیء را از تمام شیءهای دیگر متمایز می‌کند [10].

عامل بسیاری از خطاها.

عدم تمایز اسم یک شیء و خود آن شیء، عامل بسیاری از خطاها در زبان‌های برنامه‌سازی شیءگرا است.

۳- ۲ انواع روابط شیءها با یکدیگر *(Kinds of Relationships)*

رابطه‌ی بین دو شیء دربرگیرنده‌ی تمام فرضیاتی است که یک شیء از شیء دیگر دارد، ازجمله این‌که، چه عمل‌هایی می‌توان انجام داد و این‌که چه رفتارهایی در مقابل انجام خواهد شد. دو نوع سلسله مراتب شیءها در طراحی شیءگرا مورد توجه است:

رابطه‌ی بکارگیری.

- رابطه‌ی به‌کارگیری (using relationship) یا ارشدیت (seniority) [14]، که یک شیء، شیء دیگر را به‌کار می‌برد؛

رابطه‌ی دربرداشتن.

- رابطه‌ی دربرداشتن (containing relationship) یا رابطه‌ی پدر، فرزندی (parent / child) که یک شیء در داخل شیء دیگر قرار می‌گیرد.

نقش شیء *(The Role of object)*

در هر رابطه‌ی استفاده در بین شیءها، هر شیء ممکن است یکی از سه نقش زیر را بازی کند:

- فعال یا خواهان (actor): شیئی که می‌تواند روی شیءهای دیگر عمل کند، ولی هیچ‌گاه به‌وسیله‌ی شیءهای دیگر مورد عمل قرار نگیرد؛ واژه‌های actor, active, object معادل یکدیگر گرفته می‌شوند؛
- خدمتگذار (server): شیئی که هرگز روی شیءهای دیگر عمل نمی‌کند؛ تنها به‌وسیله‌ی شیءهای دیگر مورد عمل قرار می‌گیرد؛
- واسطه (agent): شیئی که هم می‌تواند روی شیءهای دیگر عمل کند و هم توسط شیءهای دیگر مورد عمل قرار گیرد؛ معمولاً یک شیء واسطه برای انجام کاری از طرف یک شیء فعال (actor) یا شیء واسطه‌ی (agent) دیگر به‌وجود می‌آید.

معنی همگام‌سازی شیءها *(The Meaning of Synchronization)*

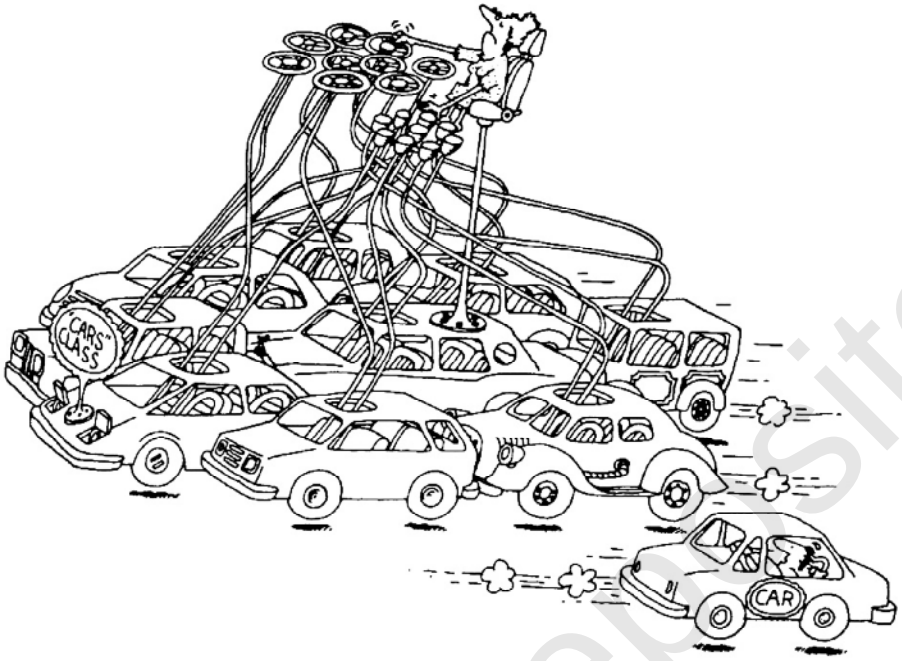
هرگاه یک شیء به شیء دیگری که با آن رابطه‌ی استفاده (using relationship) دارد پیامی می‌فرستد، این دو شیء باید به‌گونه‌ای همگام شوند. برای شیءهایی که در یک کاربرد کاملاً ردیفی هستند، این همگام‌سازی معمولاً به‌وسیله‌ی فراخوانی زیربرنامه‌ها صورت می‌گیرد. با این‌حال برای شیءهایی که در چند ریسمان کنترل (multiple thread of control) درگیر هستند، همگام‌سازی پیچیده‌تری برای برخورد با مسأله ممانعت دوجانبه (mutual exclusion) که در سیستم‌های هم‌وجود (concurrent) پیش می‌آید، لازم است. این مطلب ما را به نوع دیگری از طبقه‌بندی شیءها هدایت می‌کند:

- شیء ردیفی (sequential object): یک شیء منفعل (passive) که معانی آن تنها با وجود یک ریسمان کنترل ضمانت می‌شود؛
- شیء مسدود (Blocking object): یک شیء منفعل که معانی آن با وجود چند ریسمان کنترل ضمانت می‌شود؛
- شیء هم‌وجود (concurrent object): یک شیء فعال (active) که معانی آن با وجود چند ریسمان کنترل ضمانت می‌شود.

۳- ۳ ماهیتِ کلاسی *(The Nature of a Class)*

در حالی‌که شیء یک موجدیت مشخص و موجود در زمان و مکان است، کلاس تنها نمایشگر یک تجرید است.

کلاس مجموعه‌ای از شیءهاست، شیءهایی که داری ساختار و رفتار مشترک هستند.



شکل ۳-۲: کلاس نمایشگر مجموعه‌ای از شیء‌ها با ساختار و رفتار مشترک است

دیدگاه بیرونی و دیدگاه داخلی یک کلاس *(The Outside and Inside Views of a Class)*

- واسطِ (interface) یک کلاس بیانگر دید خارجی است. بنابراین تأکید بر تجرید (abstraction) دارد. این در حالی است که ساختار و اسرار رفتارش را پنهان می‌کند. این واسط به‌طور عمده شامل تعاریفِ (declaration) تمام اعمالی است که روی تمام رویدادهای (شیء‌های) این کلاس قابل اعمال باشد. در عین حال این واسط ممکن است حاوی تعریف کلاس‌ها، ثابت‌ها، متغیرها و موارد استثنای دیگر برای تکمیل تجرید باشد؛
- پیاده سازیِ (implementation) یک کلاس بر خلاف واسط (interface)، دیدگاه درونی یک کلاس است. دیدگاه درونی در برگیرنده‌ی اسرار و رفتار شیء می‌باشد. پیاده‌سازی کلاس به‌طور عمده مشتمل بر پیاده‌سازی تمام عمل‌های (operation) تعریف شده در واسطِ آن کلاس است.

تقسیم بندی واسط کلاس

می‌توانیم دیدگاه بیرونی یا واسط (interface) از کلاس را به سه قسمت زیر تقسیم کنیم:

- همگانی (public): تعریفی که بخش قابل مشاهده از واسط کلاس را برای تمام مشتری‌های (clients) این کلاس معرفی می‌کند. البته مشتری‌هایی که برایش قابل رؤیت هستند؛
- حفاظت شده (protected): تعریفی که بخشی از واسط که تنها برای زیرکلاس‌ها (sub classes) قابل مشاهده هستند را معرفی می‌کند؛
- خصوصی (private): تعریفی که بخشی از واسط که برای هیچ کلاس دیگر قابل مشاهده نیست را معرفی می‌کند.

از زبان‌هایی که استفاده می‌کنیم، C++ بهترین کار را می‌کند که به تولیدکننده‌ی نرم‌افزار اجازه می‌دهد، صراحتاً این قسمت‌های متفاوت از واسطِ کلاس را متمایز سازد.

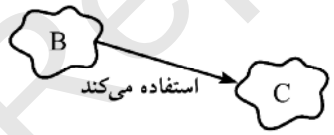
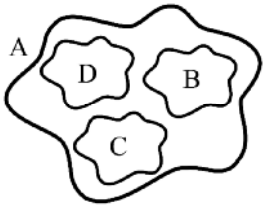
۳- ۴ انواع روابط کلاس‌ها با یکدیگر *(Kinds of relationship)*

اکثر زبان‌های برنامه‌سازی شیءگرا یا مبتنی بر شیء، ترکیبی از روابط زیر بین کلاس‌ها را حمایت می‌کنند:

- رابطه‌ی وراثت (inheritance relationships):
  یگانه؛
  چندگانه.
- رابطه‌ی استفاده (using relationships):
  واسط یک کلاس از کلاس دیگر استفاده می‌کند؛
  پیاده‌سازی یک کلاس، از کلاس دیگر استفاده می‌کند. کلاس استفاده شده در کلاس استفاده‌کننده پنهان خواهد بود.
- رابطه‌ی رویداد (instantiation relationship)؛
- رابطه‌ی کلاس برتر / اَبَرکلاس (meta class relationship).

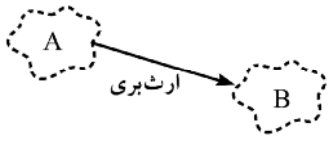
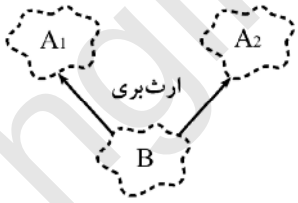
خلاصه در مورد روابط شیءها

جدول ۳-۱: روابط شیءها

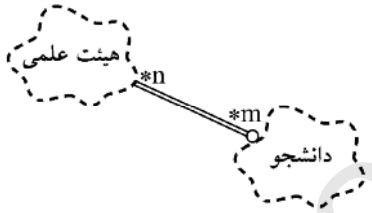
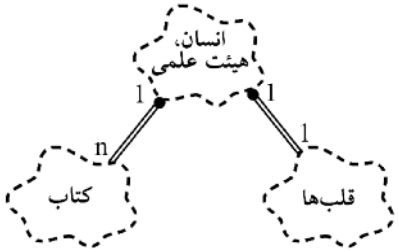
استفاده:	دربرداشتن:
	
شیء A از شیء B استفاده می‌کند.	شیء A شیءهای C، B و D را دربردارد.

خلاصه در مورد روابط کلاس‌ها

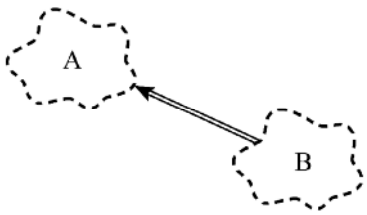
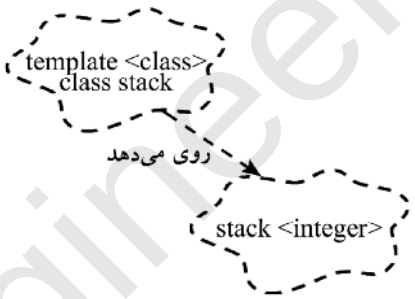
جدول ۳-۲: روابط کلاس‌ها

ارث بری یگانه:	ارث بری چندگانه:
	
کلاس B از کلاس A ارث می‌برد (ارث‌بری یگانه).	کلاس B از کلاس‌های A_1 و A_2 ارث می‌برد (ارث‌بری چندگانه)

جدول ۳-۳: روابط کلاس‌ها

استفاده (در واسط):	استفاده (در پیاده سازی):
	
کلاس مورد استفاده، برای مشتری‌های کلاس استفاده‌کننده نیز باید قابل رویت باشند. کلاس دانشجو از کلاس هیئت علمی (در بخش واسط) استفاده می‌کند (m دانشجو از n هیات علمی).	کلاس مورد استفاده برای مشتری‌های کلاس استفاده‌کننده نباید قابل رویت باشد. کلاس مورد استفاده بخشی از اسرار کلاس استفاده‌کننده است. کلاس انسان، هیئت علمی (در پیاده سازی عملکرد خود) از کلاس قلب‌ها استفاده می‌کند (یک عضو هیئت علمی از یک کتاب). کلاس عضو هیئت علمی (در پیاده‌سازی عملکرد خود) از کلاس کتاب استفاده می‌کند (یک عضو هیئت علمی از $n \geq 0$ کتاب).

جدول ۳-۴: روابط کلاس‌ها

آبَر کلاس (Meta class):	رویداد (Instantiation):
	
B با A رابطه‌ی آبَر کلاسی دارد، به این معنی که کلاس B، به‌عنوان یک شیء عضو کلاس A است.	کلاس انباره‌های مربوط به template رویدادی از کلاس عمومی stack است.

تذکر: گرچه C++ صراحتاً از آبَرکلاس حمایت نمی‌کند ولی برای متغیرها و روش‌های کلاس امکاناتی را فراهم کرده‌است. شخص می‌تواند یک شیء عضو (member object) یا تابع عضو (member function) را به‌صوت static معرفی کند، به این معنی که چنین عضوهایی بین تمام رویدادهای آن کلاس مشترک است (مثل این‌که بخواهیم تعدادی ساعت مختلف داشته باشیم ولی همگی آنها دقیقه و ثانیه‌ی یکسانی را نمایش دهند).

در Java از رابطه‌ی آبَر کلاس عملاً زیاد استفاده شده است.

چند شکلی *(Polymorphism)*

اصولاً چند شکلی مفهومی از تئوری نوع (type theory) می‌باشد. در مفهوم چند شکلی، یک اسم ممکن است به معنی شیءهایی از کلاس‌های مختلف باشد؛ کلاس‌هایی که به‌وسیله‌ی یک سوپر کلاس (super class) به یکدیگر مرتبط هستند. بنابراین هر شیء که با این اسم مشخص شده است قادر به پاسخ‌گویی مجموعه‌ای از عمل‌ها به‌صورت‌های مختلف باشد.

زبان‌هایی مثل Pascal، بر اساس این ایده هستند که، توابع، رویه‌ها و بنابراین عملگرها دارای نوع منحصربه‌فرد هستند. چنین زبان‌هایی یک شکلی (mono morphic) نامیده می‌شوند.

بارگذاری اضافی *(Over loading)*

در زبان‌های برنامه‌نویسی جدید، این‌که بتوان برای علامتی مثل "+" بیش از یک معنی تعریف کرد را بارگذاری اضافی (over loading) می‌گویند.

۳- ۵ نقش متقابل کلاس‌ها و شیءها *(The Interplay of Classes and Objects)*

ویژگی‌های کلاس‌ها

کلاس‌ها و شیءها مفاهیمی مجزا و در عین حال مرتبط به هم هستند. به‌خصوص هر شیء رویدادی از یک یا چند کلاس است، و هر کلاس دارای صفر یا بیشتر از صفر رویداد (شیء) می‌باشد. عملاً در کاربردها، کلاس‌ها ایستا هستند؛ بنابراین، وجود، معنا و روابط آنها قبل از اجرای برنامه تثبیت شده است. همین‌طور، کلاس‌های غالب شیءها ایستا هستند و در نتیجه

ویژگی‌های شیءها.

وقتی که شیئی ایجاد شد، کلاسش ثابت است. در مقابل، شیءها نوعاً با یک نرخ شدید در طول زندگیِ یک کاربرد، ایجاد شده و از بین می‌روند.

دو وظیفه‌ی اصلی تولیدکننده.

در طی تحلیل و مراحل اولیه‌ی طراحی، تولیدکننده (developer) دو وظیفه‌ی اصلی دارد:

- شناسایی و تعیین کلاس‌ها و شیءها که لغت‌نامه‌ی (vocabulary) زمینه‌ی مسأله را تشکیل می‌دهند؛
- ابداع ساختارهایی که به‌وسیله‌ی آنها مجموعه‌ای از شیءها با یکدیگر کار کرده و رفتارهایی را ایجاد کنند که جواب‌گوی خواسته‌های مسأله باشند.

تجریدهای کلیدی. مکانیزم‌ها.

در مجموع چنین کلاس‌ها و شیءهایی را تجریدهای کلیدی (key abstractions) مسأله می‌نامیم، و این ساختارهای همکاری را مکانیزم‌های (mechanisms) پیاده‌سازی می‌نامیم.

دیدگاه خارجی.

در طی این مرحله (در طی مرحله‌ی تحلیل و مراحل اولیه‌ی طراحی) از ایجاد و توسعه‌ی نرم‌افزار، تمرکز تولیدکننده باید روی دیدگاه خارجی (outside view)، تجریدهای کلیدی و مکانیزم‌ها باشد. این دیدگاه (دیدگاه خارجی) نمایشگر چهارچوب منطقی سیستم است، و بنابراین دربرگیرنده‌ی ساختار کلاس (class structure) و ساختار شیء

دیدگاه داخلی.

(object structure) از سیستم است. در مراحل بعدی طراحی و گذر به مرحله‌ی پیاده‌سازی وظیفه‌ی تولیدکننده تغییر می‌کند:

> تمرکز در آن موقع روی دیدگاه داخلی (inside view) از تجریدهای کلیدی و مکانیزم‌ها، و روی نمایش فیزیکی آنها خواهد بود. این تصمیم‌های طراحی (در مورد تجریدهای کلیدی و مکانیزم‌ها) را ممکن است به‌عنوان قسمتی از معماریِ مؤلفه‌های سیستم (systems modal architecture) و معماری فرایند سیستم (process architecture) بیان کنیم.

معماری مؤلفه‌ها و فرایندها.

۳- ۶ ساخت کلاس‌ها و شیءهای باکیفیت *(On Bridling Ovality Classes and Objects)*

فرایند افزایشی و تکراری.

بر اساس تجربه‌ی ما، طراحی کلاس‌ها و شیءها، یک فرایند افزایشی (incremental) و تکرار شونده (iterative) است. صادقانه می‌گوییم، که به‌جز تجریدهای بسیار ساده، هیچ‌گاه قادر نبوده‌ایم که یک کلاس را در ابتدا به‌طور دقیق تعریف کنیم. البته پالایش تجریدهای اولیه با هزینه است و لذا می‌خواهیم هرچه سریعتر به شیءها و کلاس‌های باکیفیت برسیم.

معیار ارزیابی.

پنج معیار زیر را برای ارزیابی کیفیت کلاس‌ها و شیءها پیشنهاد می‌کنیم:

- عدم وابستگی (coupling)، قبلاً درکلاس بحث شده؛
- انسجام (cohesion)، قبلاً در کلاس بحث شده؛
- کافی بودن (sufficiency)، موردی اضافه نباشد؛
- کامل بودن (completeness)، موردی جا نمانده باشد؛
- سادگی (primitiveness).

منظور ما از کافی بودن اینست که کلاس یا مؤلفه، خصوصیات کافی از تجرید را دربر بگیرد تا تراکنشی بامعنی و کارآمد را میسّر سازد.

کافی بودن در مقابل کامل بودن.

منظور ما از کامل بودن اینست که واسط کلاس یا مؤلفه، دربردارنده‌ی تمام خصوصیات بامعنی تجرید باشد.

درحالی‌که کافی بودن مبیّن کمترین واسط (minimal interface) است. کامل بودن موردی است که تمام جنبه‌های تجرید را دربر می‌گیرد. بنابراین واسط یک کلاس یا مؤلفه کامل آنقدر عمومی است که برای هر مشتری (client) قابل استفاده است.

اعمال ساده (primitive operations) آنهایی هستند که بتوانند تنها با دسترسی به نمایش تجرید، به‌طور کارآمد پیاده‌سازی شوند.

رهیافت‌هایی برای انتخاب عمل‌ها یا متدها *(Heuristics for Choosing Operation)*

۱. از جنبه‌ی عملکرد (trade offos of functional semantics)؛

سادگی عمل‌ها. جدایی عمل‌های بی‌ارتباط.

در داخل یک کلاس شیوه‌ی ما اینست که تمام عملیات را ساده نگه داریم، به‌طوریکه هر نمایشگر یک رفتار کوچک (small) و کاملاً تعریف شده (well defined) باشد. این روش‌ها را (fine–grained) می‌نامیم. مایل هستم عمل‌هایی که با یکدیگر ارتباط ندارند را نیز جدا از هم ترتیب دهیم. در عین حال تفکیک عمل‌ها از یکدیگر و یا ادغام آنها مسلماً در سهولت استفاده از آنها و یا پیاده‌سازی و موارد این چنین تأثیر دارد.

معیارهایی برای تفکیک متدها.

Herbert و O'Brine معیارهای زیر را برای این تفکیک (method) مطرح می‌کنند:

- قابلیت استفاده‌ی مجدد (reusability): آیا این رفتار (behavior) می‌تواند در موارد بیشتری مورد استفاده قرار گیرد؟
- پیچیدگی (complexity): پیاده‌سازی این رفتار چقد مشکل است؟
- به‌کارگیری (applicability): چقدر این رفتار، مرتبط با آن نوع (type) است؟
- آگاهی پیاده‌سازی (implementation knowledge): آیا پیاده‌سازی این رفتار به اطلاع از جزئیات یک type بستگی دارد؟

عمل‌های بامعنی.

در هر حال ما تعریف عمل‌های بامعنی را انتخاب می‌کنیم.

زیربرنامه‌های آزاد.

در عین حال در زبان‌هایی مثل Ada, CLOS, Object Pascal, C++ تعریف زیربرنامه‌ای آزاد (free subprograms) امکان پذیر است. این زیربرنامه را ممکن است در کلاس همه‌بهرها (class utilities) قرار دهیم. در C++ یک زیربرنامه‌ی آزاد، یک تابع غیر عضو (non-member function) می‌باشد.

۲. از جنبه‌ی زمان و حافظه (trade-offs of time and soace semantics)؛

ما دریافتیم که بیان معانی هم‌وجودی (concurrency) برای هر یک از عمل‌ها (operation) مفید است. همچنان‌که بیان هم‌وجودی برای کلیت شیء مفید است. به‌دلیل این‌که ممکن است عملیات مختلف نیاز به انواع مختلف همگام‌سازی (synchronization) داشته باشند. بنابراین انتقال پیام ممکن است به یکی از صورت‌های زیر باشد:

- همگام (synchronous): یک عمل (operator) تنها موقعی شروع می‌شود که فرستنده، عمل (action) را شروع کرده است و دریافت‌کننده، آماده‌ی دریافت پیام است؛ فرستنده و دریافت‌کننده هر دو به‌طور نامحدود صبر می‌کنند تا هر دو طرف برای ادامه آماده باشند؛
- ترک‌کننده (balking): مثل همگام، با این تفاوت که در صورتی‌که دریافت‌کننده بلافاصله آماده نباشد، فرستنده عمل را ترک می‌کند؛
- انقضاء (timeout): مثل همگام، با این تفاوت که فرستنده تنها برای مدت مشخصی برای آماده شدن دریافت‌کننده، صبر می‌کند؛
- ناهمگام (asynchronous): فرستنده بدون توجه به این‌که دریافت‌کننده انتظار پیامی را دارد یا نه، ممکن است یک action را آغاز نماید.

برای هر یک از عمل‌ها چگونگی همگام‌سازی می‌تواند انتخاب گردد، اما تنها بعد از این‌که در مورد معانی عملکردی (functional semantics) تصمیم‌گیری شد.

رهیافت‌هایی برای انتخاب رابطه‌ها

قانون Demeter

انتخاب رابطه بین کلاس‌ها و رابطه‌ی بین شیءها به انتخاب عمل‌ها بستگی دارد. یک رهنمود مفید در انتخاب رابطه‌ی شیءها، قانون Demeter است. این قانون می‌گوید:

یک سطحی. محدود.

> "متدها یا عمل‌های یک کلاس به هر حال نباید به ساختار هیچ کلاسی به‌جز ساختار کلاسِ بلافاصله سطح بالاترش، بستگی داشته باشد. به‌علاوه هر متدی باید تنها به شیءهای مجموعه‌ی محدودی از کلاس‌ها پیام بفرستد.

Meyer می‌گوید:

وراثت رویداد

> "بین دو کلاس A و B رابطه‌ی وراثت موقعی مناسب است که هر رویداد از B را بتوان رویدادی از A دانست. رابطه‌ی استفاده موقعی مناسب است که هر رویداد از B نمایشگر یک یا چند ویژگی (attribute) از A باشد. "

نقش مکانیزم‌ها و قابلیت رؤیت *(Role of Mechanisms & Visibility)* برای انتخاب رابطه‌ها قابل توجه است.

در طول فرایند طراحی، گاهی بیان صریح چگونگی قابلیت رؤیت یک شیء برای شیء دیگر مفید است، اساساً سه راه برای این‌که شیء x را برای شیء y قابل رؤیت کند وجود دارد:

- قلمرو لُغَوی یکسان (same lexical scope): y در قلمرو x است؛ بنابراین x صراحتاً می‌تواند y را نام ببرد؛
- پارامتر (parameter): y به‌عنوان پارامتر برای بعضی عملیات که قابل به‌کارگیری روی x است ارسال (pass) می‌شود؛
- فیلد (field): y عنصری از x است.

یک اختلاف (variation) روی هر یک از این‌ها، ایده‌ی قابلیت رؤیت مشترک (shared visibility) می‌باشد. به‌عنوان مثال، ممکن است y فیلدی از x باشد. اما y ممکن است به طُرق مختلف برای شیءهای دیگر قابل رؤیت باشد. در Smalltalk این نوع قابلیت رؤیت معمولاً نشان دهنده‌ی وابستگی بین دو شیء است. قابلیت رؤیت مشترک مستلزم اشتراک ساختاری (structural sharing) است. به این معنی که یک شیء دسترسی انحصاری به دیگری را ندارد. یعنی حالت شیء ممکن است به بیش از یک طریق تغییر داده شود. چنین رابطه‌های قابلیت رؤیت غالباً غیر قابل اجتناب هستند، و لذا بیان صریح این حقیقت در طی فرایند طراحی مفید است.

ره‌یافت‌هایی برای پیاده‌سازی

انتخاب چگونگی نمایش کلاس یا شیء. از این جهت که کپسوله (encapsulated) باشد.

انتخاب محل قرار دادن یک کلاس یا شیء در یک مؤلفه. برای زبان‌هایی مثل CLOS ,C++ که از ساختار مؤلفه حمایت می‌کنند مهم است. قابلیت رؤیت و پنهان سازی اطلاعات رهنمود ما برای انتخاب است.عوامل متعددی در این تصمیم دخالت دارند. حتی موارد غیر تکنیکی مثل مستند سازی.

۳- ۷ خلاصه‌ی فصل سوم کتاب طراحی شیءگرا / کلاس‌ها و شیءها*(Classes & Objects)*

- شیء دارای حالت، رفتار و هویت است؛
- ساختار و رفتار شیءهای مشابه در کلاس مشترک آنها تعریف می‌شود؛
- حالت شیء دربرگیرنده‌ی تمام خواص (معمولاً ایستا) از شیء و همچنین مقادیرِ (معمولاً پویا) جاری از هر یک از این خواص است؛

- رفتار عبارت از این است که چگونه یک شیء عمل و عکس‌العمل می‌کند، بر حسب تغییرات در حالتش و بر حسب ارسال پیام؛
- هویت خصوصیتی از شیء است که آن را از تمام شیءهای دیگر متمایز می‌کند؛
- دو نوع سلسله مراتب شیءها عبارتند از روابط "استفاده" و "شمول" (دربرداشتن)؛
- کلاس مجموعه‌ای از شیءهاست که دارای ساختار و رفتار مشترک هستند؛
- چهار نوع سلسله مراتب کلاس‌ها عبارتند از: وراثت، استفاده، رویداد (instantiation) و اَبَرکلاس (meta class)؛
- تجریدهای کلیدی، کلاس‌ها و شیءهایی هستند که لغت‌نامه‌ی (vocabulary) زمینه‌ی مسأله را تشکیل می‌دهند؛
- مکانیزم (mechanism) یک ساختار است که به‌وسیله‌ی آن مجموعه‌ای از شیءها با هم همکاری می‌کنند تا رفتاری را به‌وجود آورند که جواب‌گوی (بعضی از) خواسته‌های مسأله باشد؛
- کیفیت یک تجرید ممکن است به‌وسیله‌ی وابستگی کم (coupling)، انسجام (cohesion)، کفایت (sufficiency)، کامل بودن (completeness) و سادگی (primitiveness) اندازه‌گیری شود.

فصل ۴

دسته‌بندی (Classification)

دسته‌بندی ابزاری است که به‌وسیله‌ی آن دانش را مرتب می‌کنیم. در طراحی شیءگرا درک شباهت بین چیزها به ما اجازه می‌دهد که مشترکاتِ بین تجریدهای کلیدی (key abstractions) و مکانیزم‌ها (mechanisms) را معلوم کرده، و در نهایت ما را به طرح‌های ساده‌تر هدایت می‌کند.

۴- ۱ دسته‌بندی و طراحی شیءگرا *(Classification and Object Oriented Design)*

مشکل‌ترین قسمت طراحی شیءگرا.

شناسایی و تعیین کلاس‌ها و شیءها مشکل‌ترین قسمت طراحی شیءگراست. تجربه‌ی ما نشان می‌دهد که شناسایی و تعیین کلاس‌ها و شیءها متضمّنِ هم کشف، (discovery) و هم ابداع (invention) است. در ضمنِ کشف، ما به درک "تجریدهای کلیدی" و "مکانیزم"های زمینه‌ی مسأله‌ی خود می‌رسیم. و در ضمنِ ابداع، ما تجربه‌های تعمیم داده شده و همین‌طور مکانیزم‌های جدید برای تنظیم نحوه‌ی همکاری شیءها را ابداع می‌کنیم. در نهایت، کشف و ابداع هر دو، مسائل دسته‌بندی هستند. دسته‌بندی مسأله‌ی پیدا کردن شباهت‌هاست. ما وقتی دسته‌بندی می‌کنیم، دنبال گروه‌بندی چیزها هستیم که دارای ساختار مشترک و یا رفتار مشترک هستند.

کشف و ابداع.

دسته‌بندی و تمام جوانب طراحی شیءگرا در تجرید کلیدی.

دسته‌بندی به تمام جوانب طراحی شیءگرا مربوط است. دسته‌بندی به ما کمک می‌کند تا سلسله مراتب‌های تعمیم (generalization)، خصوصی شدن (specialization)، و اجتماع (aggregation) در بین کلاس‌ها را شناسایی و تعیین کنیم.

در ابداع مکانیزم.

با درک الگوهای مشترک تراکنش شیءها، ما به ابداع "مکانیزم‌ها" که روح پیاده‌سازی، هستند می‌رسیم.

در تصمیم‌گیری فرایندها.

دسته‌بندی همچنین ما را به تصمیم در مورد مؤلفه‌ها (modularization) هدایت می‌کند. ممکن است ما در مورد گذاشتن بعضی کلاس‌ها و شیءها در مؤلفه‌ی مشترک یا مؤلفه‌های متفاوت، بسته به شباهت‌ها تصمیم بگیریم. انسجام (cohesion) و عدم وابستگی (coupling) معیارهای این شباهت‌ها هستند.

در تصمیم‌گیری مؤلفه‌ها

دسته‌بندی در تخصیص فرایندها به پردازنده‌ها نیز ربط دارد. بسته به این‌که فرایندها از نظر عملکردی چقدر به یکدیگر مرتبط هستند، ممکن است آنها را به یک پردازنده و یا پردازنده‌های متفاوت نسبت دهیم.

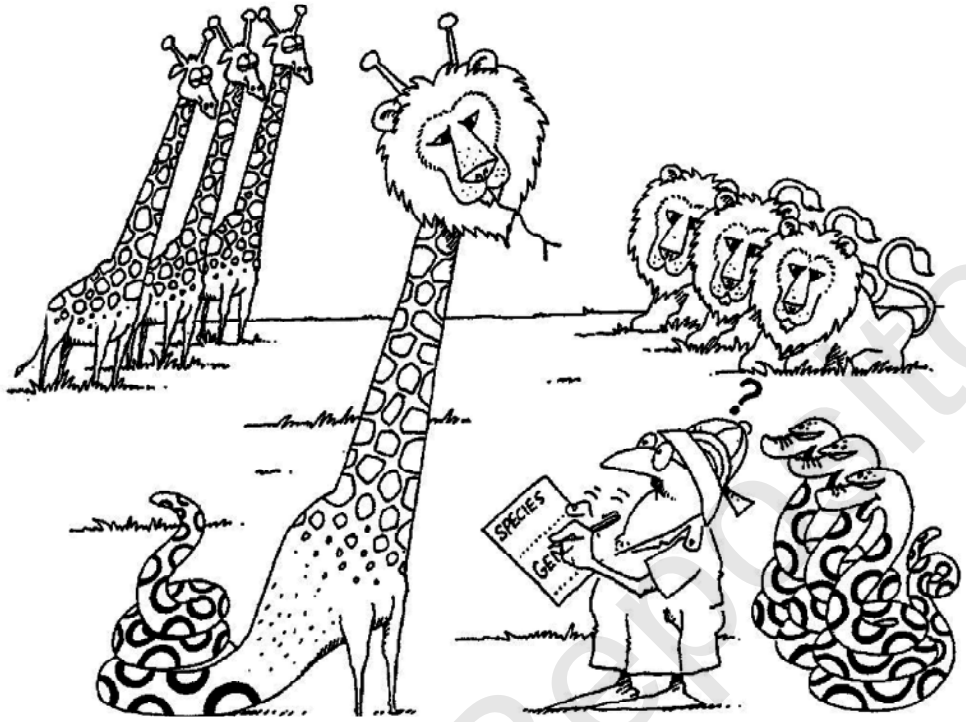
دشواری دسته‌بندی *(The Difficulty of Classification)*

حتی در نظام‌های کاملاً مشخص نیز، دسته‌بندی به میزان زیادی به دلیلِ دسته‌بندی بستگی دارد. ...

این حقیقت که "دسته‌بندی هوشمندانه مشکل است" یک اطلاع جدید است! ناظران مختلف، یک چیز واحد را به طُرق مختلفی دسته‌بندی می‌کنند.

ماهیت افزایشی و تکراری دسته‌بندی.

دسته‌بندی ماهیت افزایش یابنده (فزاینده) (incremental) و تکرار شونده (تکراری) (iterative) دارد.



شکل ۴-۱: دسته‌بندی، ابزاری است که به‌وسیله‌ی آن دانش را مرتب می‌کنیم (دسته‌بندی هوشمندانه مشکل است)

روند تولید تجرید.

تولید هر تجرید واحد معمولاً روند مشترکی را دنبال می‌کند. ابتدا، مسائل به‌طور پراکنده حل می‌شوند. با افزایش تجربه، بعضی راه‌حل‌ها بهتر از راه‌حل‌های دیگر عمل می‌کنند، و نوعی فرهنگ عامه (folklore) به‌طور غیر رسمی از فردی به فرد دیگر منتقل می‌شود. بالاخره، راه‌حل‌های مفید به‌طور سیستماتیک‌تری درک شده و مورد تحلیل قرار می‌گیرند (تبلور شاخه‌ای مختلف در نرم‌افزار یک مثال است).

تأثیر ماهیت افزایشی و تکراری دسته‌بندی

ماهیت فزاینده و تکراری دسته‌بندی، مستقیماً در ساخت سلسله مراتب شیءها و کلاس‌ها در طراحی سیستم‌های نرم‌افزاری پیچیده، تأثیر می‌گذارد. در عمل، در نظر گرفتن یک "ساختار کلاس" در بدو طراحی و تجدید نظر در مورد آن در طول زمان یک امر متداول است. تنها در مراحل ابتدایی، وقتی که مشتریان (clients) ساخته شده‌اند که از این ساختار استفاده می‌کنند، در آن موقع در مورد کیفیت دسته‌بندی می‌توانیم ارزیابی کنیم. بر اساس تجربه، ممکن است از کلاس‌های موجود بخواهیم زیرکلاس‌هایی را به‌وجود آوریم (اشتقاق یا انشعاب derivation). ممکن است بخواهیم یک کلاس بزرگ را به چند کلاس کوچک تقسیم کنیم (تفکیک factorization)، یا با اتحاد چند کلاس، یک کلاس بزرگتر به‌وجود آوریم (ترکیب composition). گاهی حتی ممکن است مشرکاتی را درک کنیم که قبلاً متوجه نشده بودیم و به کلاس جدیدی برسیم (تجرید abstraction) [10].

اشتقاق.
تفکیک.
ترکیب.
تجرید.

ممکن است از کلاس‌های موجود به جمع‌بندی‌های جدیدی برسیم:

- اشتقاق یا انشعاب (derivation): به‌وجود آوردن زیرکلاس‌هایی از کلاس‌های موجود؛
  مثلاً: به‌وجود آوردن زیرکلاس شیرهای آفریقایی یا زیرکلاس شیرهای باغ وحش از کلاس شیرها؛
  مثلاً: به‌وجود آوردن زیرکلاس چند وجهی‌ها از کلاس شکل‌های هندسی؛
  مثلاً: به‌وجود آوردن یک زیرعنوان جدید در نگارش.
- تفکیک (decomposition factorization): تقسیم یک کلاس بزرگ به چندکلاس کوچکتر؛
  مثلاً: به‌وجود آوردن زیرکلاس‌های شیرهای آفریقایی و شیرهای غیر آفریقایی از کلاس شیرها؛
  مثلاً: تقسیم کل شکل‌ها به شکل‌های هندسی و غیرهندسی؛
  تفاوت با مورد قبل در اینست که مجموع زیرکلاس‌های اخیر کل کلاس اولیه را به‌دست می‌دهد؛

مثلاً: تفکیک یک عنوان به چند عنوان.

- ترکیب (composition): اتحاد چند کلاس برای به‌وجود آوردن کلاس بزرگتر.

مثلاً: به‌وجود آوردن کلاس برتر راه‌روندگان از کلاس‌های شیرها؛ زرافه‌ها؛

مثلاً: ترکیب کلاس‌های مربع‌ها، مثلث‌ها و پنج ضلعی‌ها و به‌وجود آوردن کلاس چند وجهی‌ها؛

مثلاً: ترکیب دو زیرعنوان در یک عنوان بزرگتر.

ممکن است با بررسی‌های مجدد به کلاس‌های جدیدی برسیم که قبلاً در نظر نداشته‌ایم (abstraction).

مثلاً:

قبلاً به کلاس شیرها و زرافه‌ها رسیده بودیم و حالا متوجه شدیم که کلاس جدید مورچه‌ها را بهتر است در نظر بگیریم؛

قبلاً به کلاس اجسام توجه کرده بودیم. و حالا به کلاس "سایه‌ها" رسیدیم؛

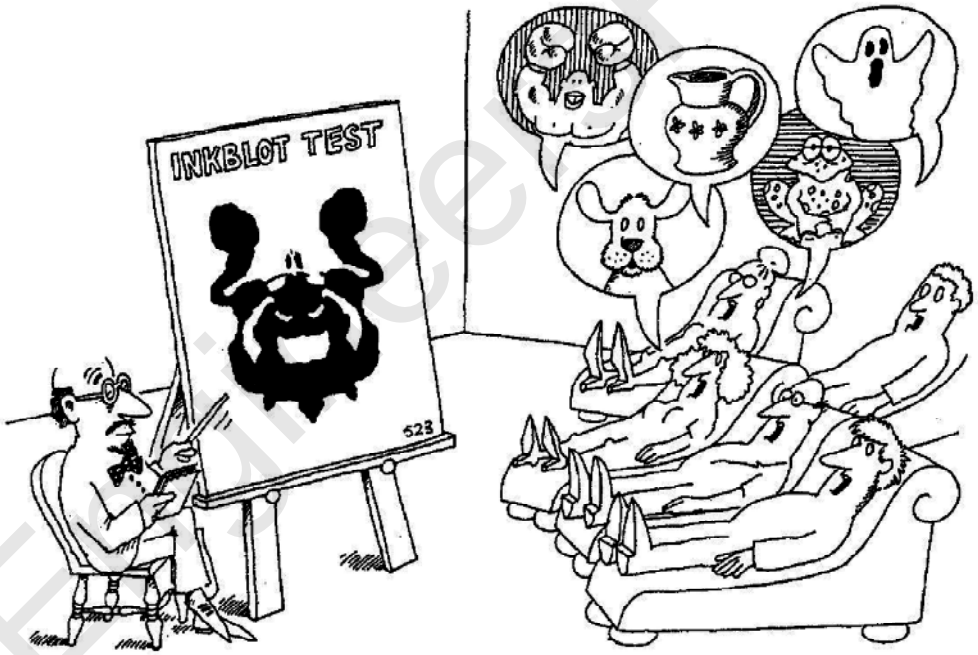
مثلاً نوشتن یک فصل کاملاً جدید.

دو دلیل مهم برای دشواری دسته‌بندی:

دلیل اول:

وابستگی به ناظر.

چیزی به‌عنوان دسته‌بندی کامل (perfect) وجود ندارد، گرچه مطمئناً بعضی از دسته‌بندی‌ها از دسته‌بندی‌های دیگر بهتر هستند. حداقل به اندازه‌ی تعداد کسانی که متصدی امری می‌شوند، می‌توان دنیا را به سیستم‌هایی از شیء‌ها تقسیم کرد. هر دسته‌بندی به دیدگاه ناظری که دسته‌بندی می‌کند، بستگی دارد.



شکل ۴-۲: ناظران متفاوت، یک شیء خاص را به طُرق مختلف دسته‌بندی می‌کنند

دوم اینکه:

نیاز به بینش خلاق.

دسته‌بندی هوشمندانه به میزان قابل توجهی نیاز به بینش خلاّق (creative insight) دارد. گاهی پاسخ روشن است، گاهی سلیقه است، و گاهی انتخاب عناصر مناسب، یک نکته‌ی کلیدی و حساس از تحلیل است [13].

۴- ۲ شناسایی و تعیین کلاس‌ها و شیءها *(Identifying Classes & Objects)*

سابقه‌ی طولانی مشکل دسته‌بندی.

مسأله‌ی دسته‌بندی مورد توجه تعداد بیشماری از فلاسفه، زبان شناسان، دانشمندان علوم اداری، راضی‌دانان، و حتی قبل از افلاطون بوده است. مطالعه‌ی تجارب آنها و به‌کارگیری آنها در طراحی شیءگرا مغتنم است.

سه دیدگاه کلی به دسته‌بندی.

از نظر تاریخی سه دیدگاه کلی به دسته‌بندی بوده است:

- دسته‌بندی کلاسیک (classical categorization)؛
- خوشه‌بندی مفهومی (conceptual clustering)؛
- تئوری خواص (property theory) [15].

تجربه‌ای برای شناسایی شیءها و کلاس‌ها.

بر اساس تجربه، ما کلاس‌ها و شیءها را ابتدا بر اساس خواص زمینه‌ی مورد نظر شناسایی و تعیین می‌کنیم.

معمولاً این تجریدها (برای انتخاب) وجود دارد، به‌دلیل این‌که مستقیماً بخشی از لغت‌نامه‌ی فضای مسأله را تشکیل می‌دهد [27]. اگر این دیدگاه به ساختار کلاس (class structure) رضایت‌بخشی منجر نشود سپس به خوشه‌بندی (clustering) شیء، به‌وسیله‌ی مفاهیم می‌پردازیم. اگر این هم شکست بخورد، بالاخره دسته بندی به‌وسیله‌ی association را مورد توجه قرار می‌دهیم، که از طریق آن خوشه‌های (clusters) شیءها بر اساس این‌که چقدر هر یک از آنها با بعضی شیءهای نمونه، شباهت دارند تعریف می‌شوند.

به‌علاوه سه دیدگاه فوق (در انتهای پاراگراف قبل) اساس تئوریک تحلیل شیءگرا ، تحلیل زمینه، و متدهای دیگر (که ما از آنها برای شناسایی و تعیین کلاس‌ها و شیءها در طراحی سیستم‌های بزرگ نرم‌افزاری بزرگ استفاده می‌کنیم) را فراهم می‌کند.

شناسایی شیءها و کلاس‌ها با تحلیل شیءگرا.

شناسایی و تعیین کلاس‌ها و شیءها به‌وسیله‌ی تحلیل شیءگرا

حدس تحلیل و طراحی شیءگرا روش نیست.
تحلیل شیءگرا.
طراحی شیءگرا.

گرچه تمرکز روی تحلیل یا روی طراحی کاملاً از هم جدا نیستند، ولی حد بین تحلیل و طراحی تا حدودی مشکّک (fuzzy) است. در تحلیل شیءگرا ما دنبال مدل کردن جهان (مسأله) با شناسایی و تعیین شیءها و کلاس‌هایی هستیم که فرهنگ لغت‌نامه‌ی زمینه‌ی مسأله را تشکیل می‌دهد؛ و در طراحی شیءگرا ما تجریدها و مکانیزم‌ها را ابداع می‌کنیم به گونه‌ای که رفتار مورد نیاز این مدل را فراهم آورد. لذا تحلیل شیءگرا جبهه‌ی ایده‌آل برای طراحی شیءگرا ست.

نامزدهایی برای شیءها و کلاس‌ها.

Shalor و Mellor موارد زیر را به‌عنوان نامزدهایی برای استخراج کلاس‌ها و شیءها پیشنهاد می‌کنند:

- چیزهای ملموس (tangible things): ماشین‌ها، داده‌های سنجش از راه دور (Telemetry data)، حس کننده‌های فشار؛
- نقش‌ها (roles): مادر، معلم، سیاست‌مدار؛
- حوادث (events): فرود، وقفه، درخواست؛
- تراکنش‌ها (interactions): قرض، ملاقات، تقاطع.

از دیدگاه مدل کردن داده‌ها (data modeling)، Ross لیست مشابهی ارائه می‌کند [29]:

- مردم (people)؛
- محل‌ها (places)؛
- چیزها (things)؛
- سازمان‌ها (organization)؛
- مفاهیم (concopts)؛
- حوادث (events).

Coad و Yourdon مجموعه‌ی دیگری از شیءهای بالقوه را پیشنهاد می‌کنند [30]:

- ساختار (structure): روابط "نوعی از" (kind of) و "جزئی از" (part of)
- سیستم‌های دیگر (other systems): سیستم‌های خارجی که با کاربرد مورد نظر ما در تراکنش هستند؛
- دستگاه‌ها (devices): دستگاه‌هایی که با کاربرد مورد نظر ما در تراکنش هستند؛

- حوادث؛
- نقش‌ها؛
- محل‌ها؛
- واحدهای سازمانی.

شناسایی و تحلیل شیء‌ها و کلاس‌ها با تحلیل زمینه.

شناسایی و تحلیل شیء‌ها به‌وسیله‌ی تحلیل زمینه *(Domain Analysis)*

ما تحلیل زمینه را به‌صورت زیر تعریف می‌کنیم:

تعریف تحلیل زمینه

"تلاشی برای شناسایی و تعیین شیء‌ها، عمل‌ها (operations)، و روابط که خبرگان زمینه، اهمیت آنها را دریافته‌اند [31]."

Moore و Bailain گام‌های زیر را برای تحلیل زمینه پیشنهاد کرده‌اند [31]:

- ساخت یک مدل عمومی (خبریک) از زمینه با مشورت خبرگان زمینه؛
- بررسی سیستم‌های موجود در زمینه و نمایش این فهم، با یک قالب عمومی؛
- شاسایی و تعیین شباهت‌ها و تفاوت‌های بین سیستم‌ها با مشورت خبرگان زمینه؛
- پالایش مدل خبریک برای جواب‌گویی به سیستم‌های موجود.

شناسایی شیء‌ها و کلاس‌ها از شرح غیر رسمی مسأله.

دیدگاه‌های دیگر برای شناسایی و تعیین شیء‌ها و کلاس‌ها *(Alternate Approaches)*

تحلیل شیء‌گرا و تحلیل زمینه دو روشی هستند که ما برای طراحی شیء‌گرا ترجیح می‌دهیم. با این‌حال دو دیدگاه دیگر نیز مطرح شده است:

- شرح غیر رسمی (informal description): این روش ابتدا به‌وسیله‌ی Abbott [33] پیشنهاد شد، پیشنهاد این است که شرح غیر رسمی در مورد مسأله (یا قسمتی از مسأله) نوشته شود. و سپس اسم‌ها و فعل‌ها مشخص شوند. اسم‌ها نامزدهایی برای شیء‌ها هستند و فعل‌ها نامزدهایی برای عملیات روی آنها هستند. و این تکنیک خود منجر به روش‌های خودکار شده است [34]. این روش از این جهت که ساده است و طراح را موظف به کار با فرهنگ زمینه می‌کند مفید است. ولی مسئله پیچیده را جواب‌گو نیست. پیچیدگی‌های زبان طبیعی و وابسته شدن شیء‌ها به شیوه‌ی نگارش از مشکلات دیگر است؛

اسم ≈ شیء
فعل ≈ عمل

شناسایی شیء‌ها و کلاس‌ها از تحلیل ساخت‌یافته.

- تحلیل ساخت‌یافته (structured analysis): این روش استفاده از محصول تحلیل ساخت‌یافته را به‌عنوان جبهه‌ی (front end) طراحی شیء‌گرا معرفی می‌کند. این روش قابل توجه است، از این جهت که تعدادی از تحلیل‌گران، با تحلیل ساخت‌یافته آشنا هستند و ابزار (CASE tools) متعددی برای مکانیزه کردن آن وجود دارد.

دو خطر.

تجربه‌ی ما می‌گوید که تحلیل ساخت‌یافته می‌تواند به‌عنوان یک جبهه‌ی مناسب برای طراحی شیء‌گرا باشد، اما به شرطی که طراح از افتادن در دامن طراحی ساخت‌یافته مقاومت کند. خطر جدی دیگر، این حقیقت است که بسیاری از تحلیل‌گران گرایش به این دارند که نمودار جریان داده‌ها را طوری بنویسند که منعکس کننده‌ی یک طرح باشد، تا این‌که مسأله مورد نظر را مدل کند.

۴-۳ شناسایی و تعیین تجریدهای کلیدی *(Identifying key Abstraction)*

یک تجرید کلیدی، یک کلاس یا شیء است که قسمتی از لغت‌نامه‌ی (فرهنگ) زمینه‌ی مسأله را تشکیل می‌دهد. ارزش اصلی شناسایی چنین تجریدهایی، اینست که حدود مسأله‌ی ما را مشخص می‌کند، آنها چیزهایی که در سیستم ما هستند و در نتیجه در طراحی ما تأثیر می‌گذارند را جلوه می‌دهند. و چیزهایی که خارج سیستم هستند را کنار می‌گذارند. تعیین تجریدهای کلیدی بسیار به زمینه بستگی دارد.

کشف و ابداع.

شناسایی و تعیین تجریدهای کلیدی متضمّن دو فرایند است: کشف (discovery) و ابداع (intention)، در طی کشف، ما تجریدهایی که به‌وسیله‌ی خبرگان زمینه بکار رفته است را درک می‌کنیم، وقتی خبرگان زمینه در مورد چیزی صحبت

یک مثال.

می‌کنند، در این‌صورت آن تجرید معمولاً مهم است [40]. در طی ابداع، ما کلاس‌ها و شیءهای جدیدی را که الزاماً جزئی از زمینه‌ی مسأله نیستند، ولی برای طراحی یا پیاده‌سازی مفید هستند را ایجاد می‌کنیم. به‌عنوان مثال، مشتری عابر بانک از واژه‌های شماره حساب، دریافت و پرداخت صحبت می‌کند؛ این لغات بخشی از لغت‌نامه‌ی زمینه هستند. تولیدکننده‌ی چنین سیستمی، از همین تجریدها استفاده می‌کند، در عین حال باید تجریدهای جدیدی مثل بانک‌های اطلاعاتی، مدیر صفحه‌ی نمایش، را معرفی کند. این تجریدها عواملی از آن طراحی خاص هستند و نه از زمینه‌ی مسأله.

مهمترین راهِ شناسایی تجریدهای کلیدی.

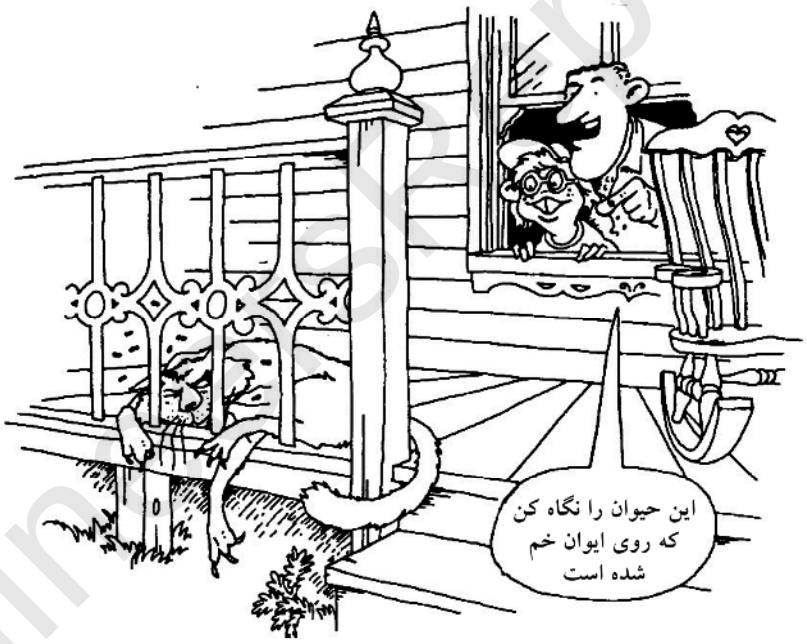
شاید مهمترین راهِ شناسایی تجریدهای کلیدی، نگاه کردن به مسأله یا طرح و دیدن این‌که آیا تجریدهایی که شبیه کلاس‌ها و شیءهای موجود باشد وجود دارند یا نه. چون این یک مسأله دسته‌بندی است، ما می‌توانیم از هر یک از روش‌های کلاسیک یا مدرن دسته‌بندی استفاده کنیم (روش‌هایی که در این فصل اشاره شد). این دیدگاه تأکید بر استفاده‌ی مجدد از این تجریدها (که ذاتی طراحی شیءگراست) دارد.

پالایش تجریدهای کلیدی

پالایش تجریدهای کلیدی.

سطح تجرید.

هنگامی‌که یک تجرید کلیدی را به‌عنوان کاندید در نظر گرفتیم، باید آن را بر اساس معیارهایی که در فصل قبل گفته شد ارزیابی کنیم و در صورت لزوم تجدید نظر کنیم. قرار دادن کلاس‌ها و شیءها در سطح درستی از تجرید مشکل است.



شکل ۴-۳: کلاس‌ها و شیءها باید در سطح مناسبی از تجرید باشند: نه خیلی بالا و نه خیلی پایین

نام‌گذاری مناسب.

چند معیار نام‌گذاری

نام‌گذاری مناسب چیزها، به ترتیبی که منعکس کننده‌ی معانی آنها باشد و گویای تجرید مورد نظر ما باشد، مهم است. ما موارد زیر را پیشنهاد می‌کنیم:

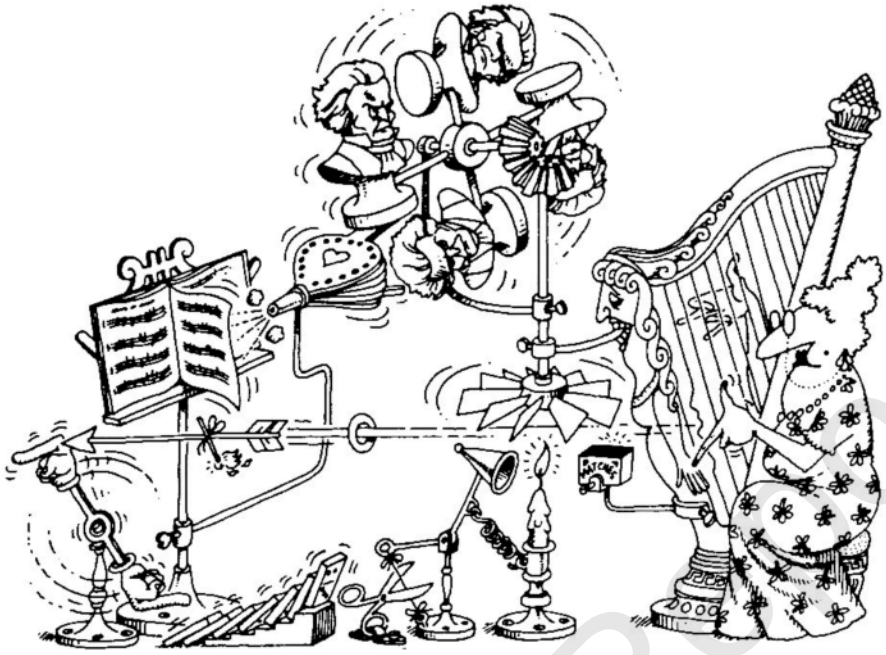
- شیءها باید با اسامی خاص نام‌گذاری شوند. (مثل پودر لباسشویی الف)؛
- کلاس‌ها باید با اسامی عام نام‌گذاری شوند. (مثل مواد پاک‌کننده)؛
- عمل‌های تغییر (modifier operations) با فعل‌های معلوم نام‌گذاری شوند. (مثل شستن)؛
- عمل‌های انتخابی (selector operations) باید مبیّن یک سؤال باشند یا باید با فعل‌های بودن (tobe) نام‌گذاری شوند. (مثل تمیز است).

۴- ۴ شناسایی و تعیین مکانیزم‌ها *(Identifying Mechanisms)*

در طراحی شیءگرا ما ابتدا با تعیین تجریدهای کلیدی شروع می‌کنیم تا مدلی از واقعیت را شکل دهیم؛ تنها بعد از آنست که ما به این تجریدها رفتار (behavior) اضافه می‌کنیم تا به رفتار ملموس سیستم برسیم[46] .

تعریف مکانیزم

از واژه‌ی مکانیزم (mechanisms) برای بیان هر ساختاری که به‌وسیله‌ی آن شیء‌ها با یکدیگر کار می‌کنند تا رفتار لازم برای جواب‌گویی خواسته‌ها را فراهم آورند، استفاده می‌کنیم.



شکل ۴-۴: به‌وسیله‌ی مکانیزم‌ها شیء‌ها با یکدیگر همکاری می‌کنند تا رفتارهای سطح بالاتری به‌وجود آورند

مکانیزم‌ها، روح طراحی.

در حالی‌که تجریدهای کلیدی منعکس‌کننده‌ی لغت‌نامه‌ی زمینه‌ی مسأله هستند، مکانیزم‌ها روح طراحی هستند. در طی فرایند طراحی تولیدکننده‌ی نرم‌افزار، نه تنها طراحی هر یک از کلاس‌ها را باید مورد توجه قرار دهد، بلکه باید متوجه چگونگی عملکرد متقابل رویدادهای کلاس‌ها (شیء‌ها) نیز باشد. وقتی‌که یک تولیدکننده‌ی نرم‌افزار در مورد یک مکانیزمِ بخصوص تصمیم بگیرد، کار در بین شیء‌ها به‌وسیله‌ی تعریف متدهای مناسب در کلاس‌های مناسب تقسیم شده است.

... مکانیزم‌ها نمایشگر سطح دیگری از استفاده‌ی مجددِ طراحی (Design reuse) می‌باشند، سطحی بالاتر از استفاده‌ی مجدد از هر یک از کلاس‌ها.

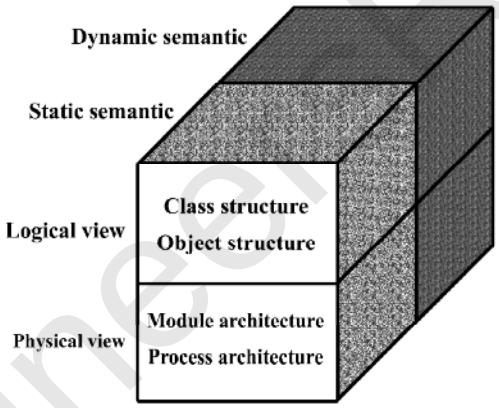
۴- ۵ خلاصه‌ی فصل چهارم کتاب طراحی شیء‌گرا / طبقه بندی *(Classification)*

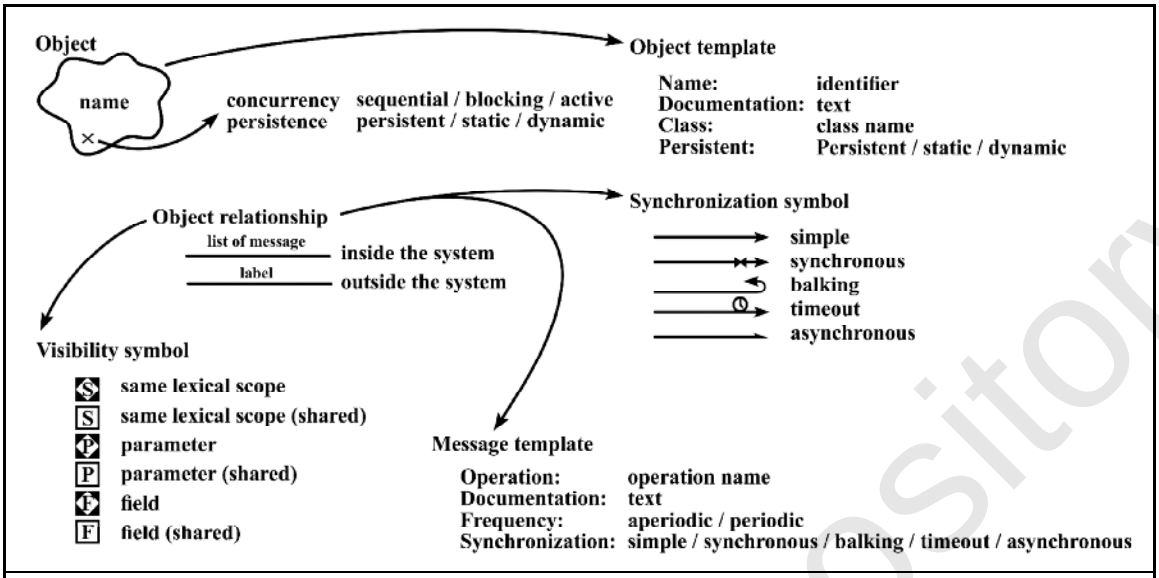
- شناسایی و تعیین کلاس‌ها نکته‌ی اصلی در طراحی شیء‌گراست؛ شناسایی و تعیین، هم اکتشاف و هم ابداع را در بر دارد؛
- کلاس‌بندی (classification) اصولاً یک مسأله‌ی خوشه‌بندی (clustering) است؛
- طبقه‌بندی یک فرایند اضافه شونده (incremental) و تکرار شونده (iterative) می‌باشد، و از این جهت مشکل اینست که مجموعه‌ی خاصی از شیء‌ها را ممکن است به طُرق مختلف طبقه‌بندی کرد؛
- سه دیدگاه به طبقه‌بندی عبارتند از: طبقه‌بندی شیئی (طبقه‌بندی به‌وسیله‌ی خصوصیات)، خوشه‌بندی مفهومی (conceptual clustering) یا طبقه‌بندی به‌وسیله‌ی مفاهیم، و تئوری خواص (property theory یا طبقه‌بندی بوسیله‌ی انتساب به یک خصوصیت)؛
- طراحی شیء‌گرا پیشنهاد می‌کند که چیزهای ملموس، مثل نقش‌ها (roles)، حوادث (events)، و تراکنش‌ها (interactions) کاندیدهایی برای کلاس‌ها و شیء‌ها می‌باشند؛
- تحلیل زمینه دنبال شناسایی و تعیین کلاس‌ها و شیء‌هایی است که در کاربردهای مشابه در یک زمینه‌ی مشترک هستند؛
- تجریدهای کلیدی منعکس کننده‌ی لغت‌نامه‌ی زمینه است و ممکن است از قلمرو مسأله کشف شود، یا به‌عنوان قسمتی از طرح، ابداع گردد؛
- مکانیزم‌ها روح طرح‌ها و نمایشگر تصمیم‌های استراتژیک طراحی در مورد همکاری انواع مختلف و متعدد شیء‌ها هستند.

فصل ۵

علامت‌گذاری (The Notation)

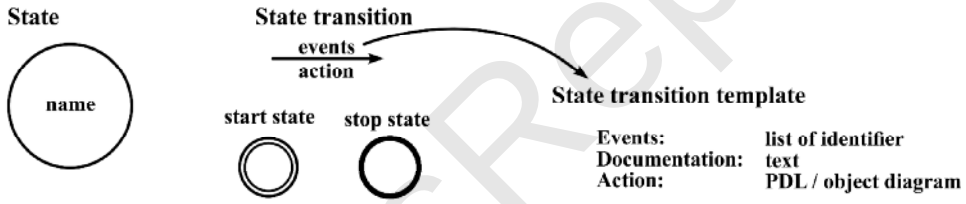
جدول ۵-۱: مدلی برای طراحی شیء‌گرا

The Models of Object-Oriented Design Supporting multiple, interrelated views 
The Process of Object-Oriented Design Supporting the incremental and iterative process of round-trip gest off design. Identify the classes and object at a given level of abstraction. Identify the semantics of these classes and objects; Identify the relationship among these classes and objects; Implement these classes and objects.
Object Design Illustrates the object structure, including the specification of indidual object and their relationship.



State Transition Diagram

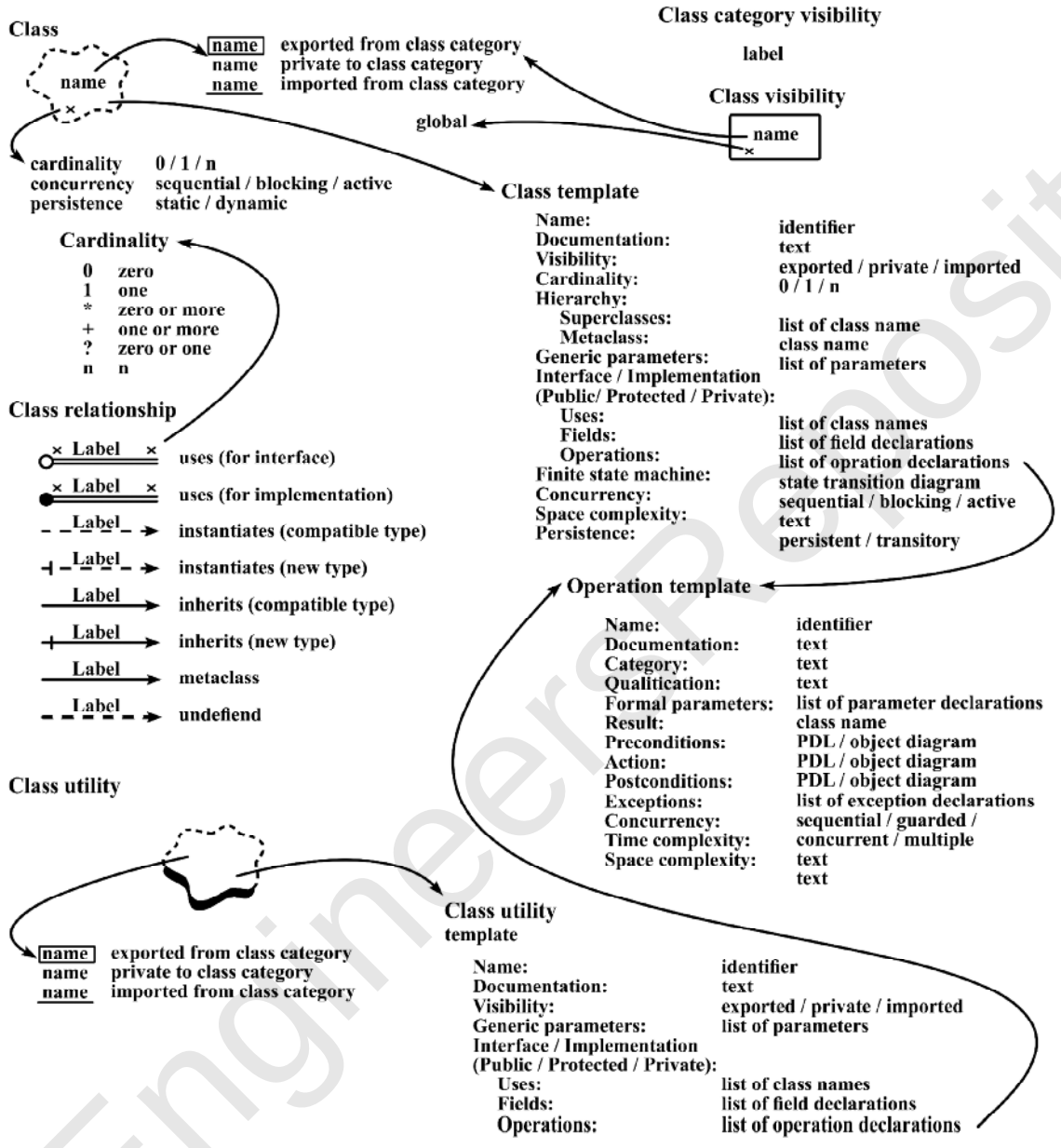
Part of an object diagram, illustrates the order of events among a set of objects.



جدول ۵-۲: نمودار کلاس

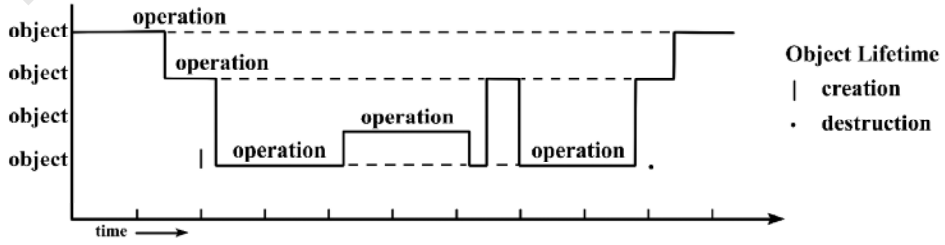
Class Diagram

Illustrates the class structure, including the specification of individual classes and their relationships.



Timing Diagram

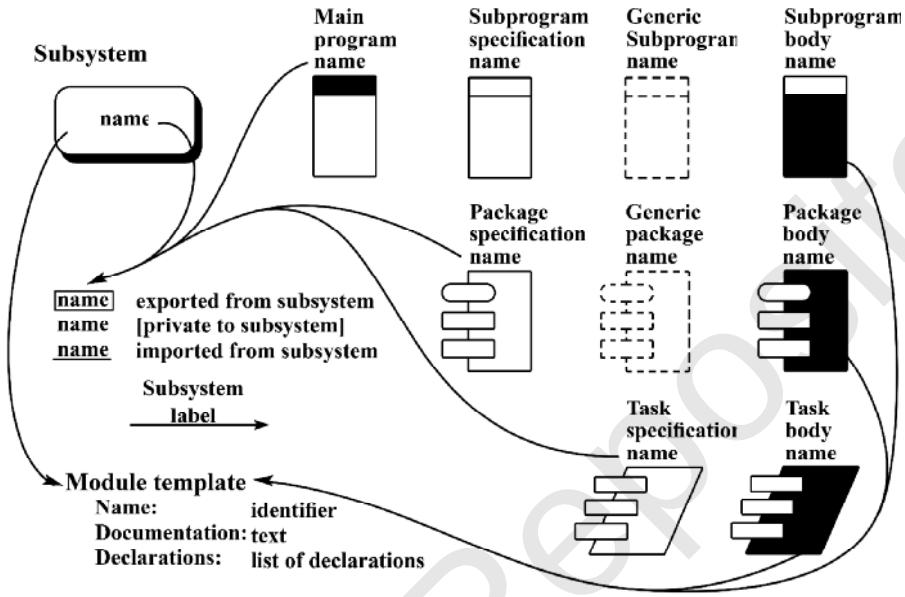
Part of object diagram, illustrates the order of events among a set of objects.



جدول ۵- ۳: نمودار مؤلفه

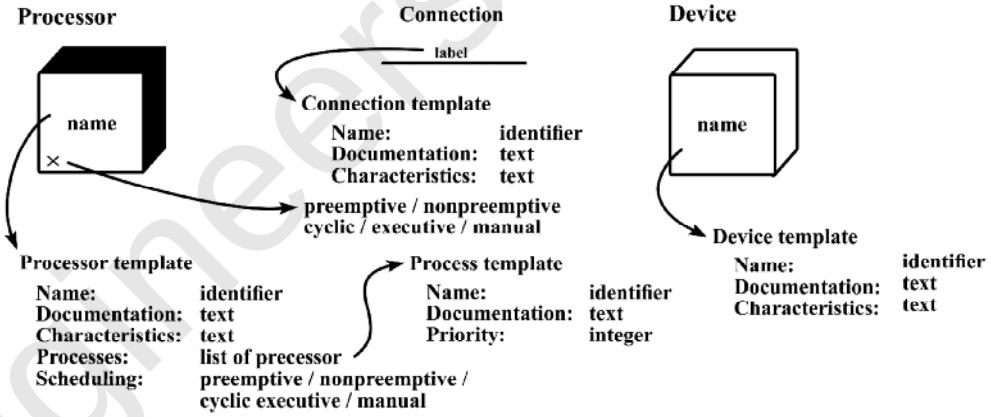
Module Diagram

Illustrates the physical packaging of classes and objects into modules.



Process Diagram

Illustrates the allocation of processes to processors.



دلایل علامت‌گذاری

داشتن علامت‌گذاریِ گویا و کاملاً تعریف شده مهم است چرا که:

۱. علامت‌گذاریِ استاندارد، فرموله کردن یک طرح و انتقال آن به دیگران را ممکن می‌سازد؛

۲. یک علامت‌گذاری خوب، تمرکز روی مسایل پیشرفته را میسّر می‌سازد؛

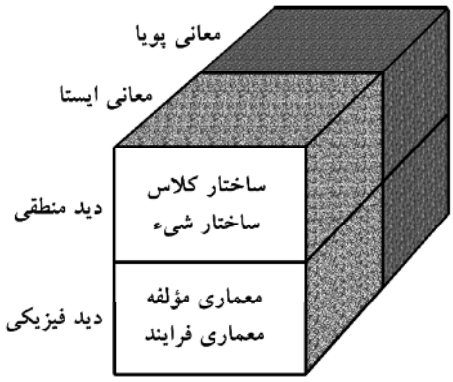
۳. استفاده از علامت‌گذاری گویا (expressive)، این امکان را به‌وجود می‌آورد که اکثر کار یکنواخت و چک‌های سازگاری (consistency) و صحتِ (correctness) طرح با استفاده از ابزار خودکار حذف گردد.

۵- ۱ عناصر علامت‌گذاری طراحی شیءگرا *(Elements of the Notation)*

به‌کارگیری این علامت‌گذاری در سیستم‌های کوچک و بزرگ.

یک نوع نمودار برای دریافت مناسب تمام جزئیات یک سیستم نرم‌افزاری پیچیده کافی نیست، دیدگاه‌های متعدد لازم است. شکل زیر (که قبلاً هم آمده است) مدل‌های مختلفی را، که برای طراحی شیءگرا مهم می‌دانیم، نمایش می‌دهد. ما دریافته‌ایم که این علامت‌گذاری قابل به‌کارگیری در سیستم‌های کوچک (چند صد خط دستوری) و همین‌طور در

سیستم‌های بزرگ (چندین میلیون خط دستوری) می‌باشد. البته ضرورتی ندارد که همه‌ی جوانب آن همیشه مورد استفاده قرار گیرد.



شکل ۵-۱: مدل‌های مختلف، برای طراحی شیءگرا

این فصل برای بیانِ نحوه‌ی نگارش (syntax) و معانی (semantics) علامت‌گذاری، برای طراحی شیءگراست.

مدل‌های منطقی و فیزیکی *(Logical Versve Physical Models)*

موارد بنیانی در طراحی شیءگرا.

در عمل، ما دریافته‌ایم که جدا کردن انواع مختلف تصمیم‌گیری‌های طراحی، یک نکته‌ی اساسی است. از جمله، یک تولیدکننده‌ی نرم‌افزار، باید موارد بنیانی زیر را در طراحی شیءگرا مد نظر قرار دهد:

- چه کلاس‌هایی وجود دارد و آنها چگونه با یکدیگر مرتبط هستند؟
- چه مکانیزم‌هایی برای تنظیم چگونگی همکاری شیءها به‌کار رفته است؟
- هر یک از کلاس‌ها وشیءها باید در کجا معرفی شوند؟
- یک فرایند (process) باید به کدام پردازنده نسبت داده شود، و برای یک پردازنده‌ی بخصوص، فرایندهای متعددش چگونه زمان‌بندی می‌شوند؟

چهار نمودار برای پاسخ‌گویی به موارد بنیانی.

پاسخ چهار سؤال فوق را به ترتیب در نمودارهای زیر می‌توان تبیین کرد:

- نمودارهای کلاسی (class diagrams)؛
- نمودارهای شیئی (object diagrams)؛
- نمودارهای مؤلفه (module diagrams)؛
- نمودارهای فرایند (process diagrams).

دید منطقی با نمودار کلاس شیء

نمودارهای چهارگانه‌ی فوق علامت‌گذاری اصلی طراحی شیءگرا را تشکیل می‌دهند.

نمودارهای کلاس و شیء قسمتی از دید منطقی یک سیستم هستند، چرا که در خدمت تبیین وجود و معنی تجریدهای کلیدیِ تشکیل دهنده‌ی طرح می‌باشند.

دید فیزیکی با نمودارهای مؤلفه و فرایند.

نمودارهای مؤلفه و فرایند قسمتی از ساختار فیزیکی سیستم هستند، به دلیل این‌که برای تبیین دقیق، عناصرِ سخت‌افزاری و نرم‌افزاریِ یک پیاده‌سازی هستند.

معانی ایستا و پویا *(Static Versve Dynamic Semantics)*

نمودارهای چهارگانه‌ی فوق عموماً ایستا هستند. با این‌حال در تمام سیستم‌های شدیداً نرم‌افزاری، وقایع به‌طور پویا اتفاق می‌افتد:

> شیءها به‌وجود می‌آیند و از بین می‌روند، شیءها با ترتیب‌های گوناگون به یکدیگر پیام (Message) می‌فرستند، و در بعضی سیستم‌ها، پیام‌ها به‌طور همزمان ارسال می‌شوند.

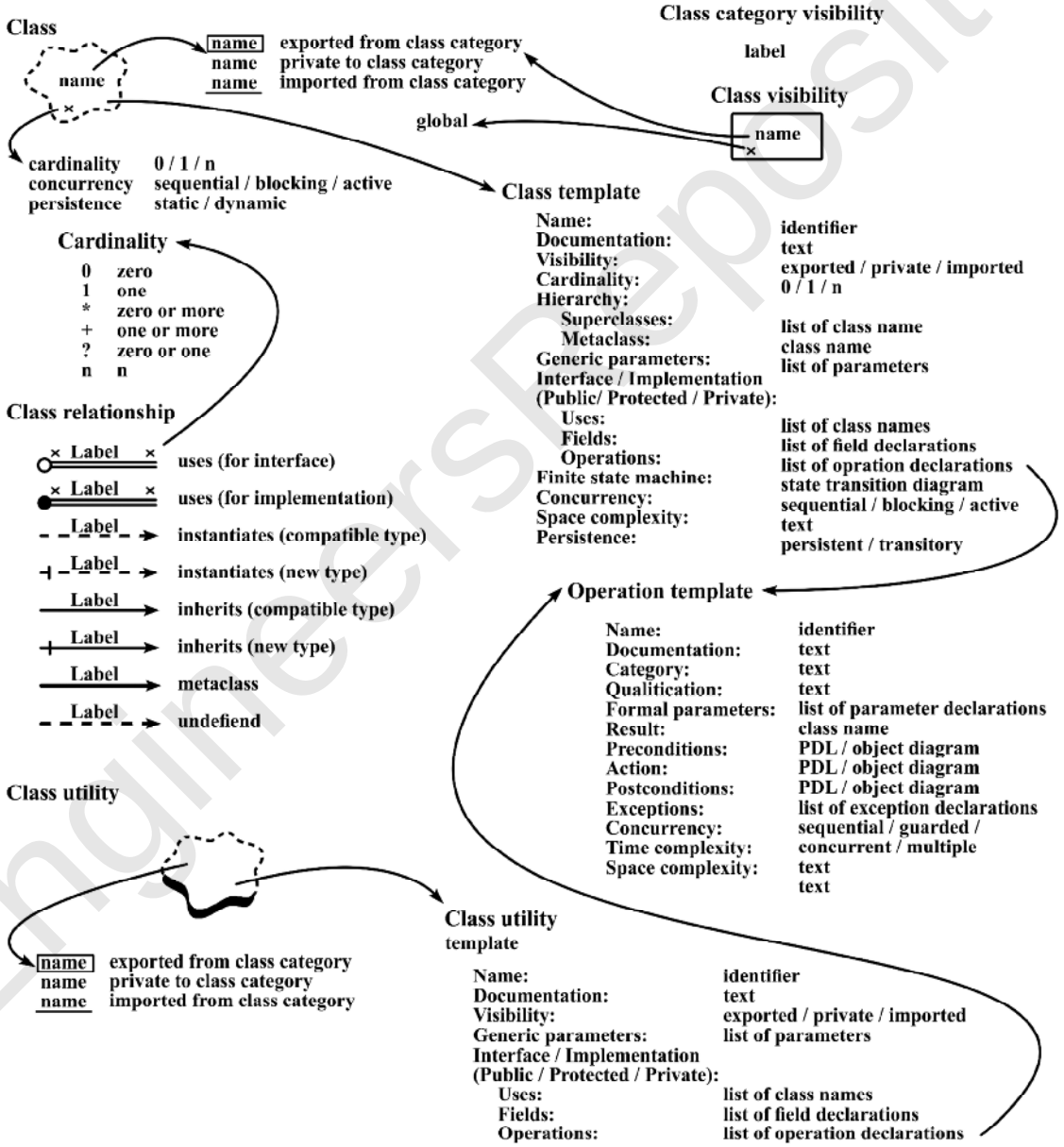
در طراحی شیءگرا، معانی پویای طرح را توسط دو نمودار دیگر بیان می‌کنیم:

- نمودارهای تغییر حالت (state transition diagrams)؛
- نمودارهای زمانی (timing diagrams).

هر کلاس ممکن است یک نمودارِ تغییر حالت داشته باشد، که به‌وسیله‌ی آن ترتیب زمانی حوادثی که می‌تواند هر یک از رویدادهای این کلاس را تحت تأثیر قرار دهد، بیان می‌شود. یک نمودارِ شیءِ واحد، نمایشگر یک لحظه از یک شیء است (مگر این‌که آن شیء گذرا transitory باشد)؛ بنابراین، ممکن است از یک نمودار زمانی همراه هر نمودار شیء استفاده کنیم تا ترتیب زمانی ارسال و ارزیابی پیام‌ها را نشان دهیم. در ادامه مجموعه علامت‌گذاری طراحی شیءگرا به‌طور فشرده ارائه می‌گردد:

۵-۲ نمودار کلاس *(Class Diagram)*

بیانگر ساختار کلاس، شامل مشخصات هر کلاس و روابط کلاس‌هاست.



شکل ۵-۲: نمودار کلاس

- کاردنیالیتی (cardinality): تعداد رویدادهایی که یک کلاس ممکن است داشته باشد؛ تعداد رویدادهایی که در یک "رابطه‌ی استفاده" (using class relationship) شرکت می‌کند؛
- همه‌بهرِ کلاس (class utility): مجموعه‌ای از زیربرنامه‌های آزاد؛

- قابلیت رؤیت (visibility): این توان که یک تجرید، تجرید دیگری را (و در نتیجه منابع مورد مراجعه در دید خارجی را) ببنید. تجریدها تنها در جایی برای دیگری قابل رؤیت هستند که در قلمروشان هم‌پوشانی (overlap) داشته باشند.

در ادامه هر یک از اجزاء مطرح شده در شکل فوق تشریح می‌شوند.

نمودارهای کلاس *(Class diagram)*

از نمودار کلاس برای نمایش وجود کلاس‌ها و رابطه‌ی بین آنها در طرح منطقیِ (logical design) سیستم استفاده می‌شود. برای یک سیستم کوچک، معمولاً یک نمودار کلاس کافیست، اما طراحی اکثر سیستم‌ها نیاز به مجموعه‌ای از این نمودارهای کلاسی برای نمایش ساختار کلاس (class structure) دارد.

عناصر ساختار کلاس.

سه جزء اصلی یک ساختار کلاس عبارتند از:

- کلاس‌ها (classes)؛
- رابطه‌ی کلاس‌ها (class relationships)؛
- همه‌بهرهای کلاس (class utilities) (برای زبان‌هایی که تعریف زیربرنامه‌های آزاد free subprograms را حمایت می‌کنند).

کلاس‌ها *(Classes)*

از شکلک (Icon) زیر برای بیان کلاس استفاده می‌شود.

شکل ۵-۳: شکلک کلاس

بعضی این شکل را ابر (cloud) می‌نامند. مشتریان کلاس تنها بر رویدادهای کلاس می‌توانند عمل کنند، نه خود کلاس. نام کلاس لازم است و در داخل ابر قید می‌شود. اسم کلاس باید در گروه کلاس‌ها (class category) منحصربه‌فرد باشد.

رابطه‌ی کلاس‌ها

رابطه‌ی کلاس‌ها

قبلاً انواع رابطه بین کلاس‌ها را (در فصل سوم) دیدیم. شامل وراثت، استفاده، رویداد و کلاس برتر. شکلک‌های زیر بیانگر این روابط هستند.

جدول ۵-۴: رابطه‌ی کلاس‌ها

استفاده می‌کند (برای واسط)	برچسب
استفاده می‌کند (برای پیاده‌سازی)	برچسب
روی می‌دهد (instantiates) (نوع قابل تطبیق compatible)	برچسب
روی می‌دهد (نوع جدید) / کلاس باید مجازی باشد.	برچسب
ارث می‌برد (نوع قابل تطبیق)	برچسب
ارث می‌برد (نوع جدید) / پدر مجازی بوده است.	برچسب
کلاس برتر (meta class)	برچسب
تعریف نشده	برچسب

هر رابطه ممکن است دارای یک برچسب برای مستندسازیِ اسم یا نقش آن رابطه، باشد.

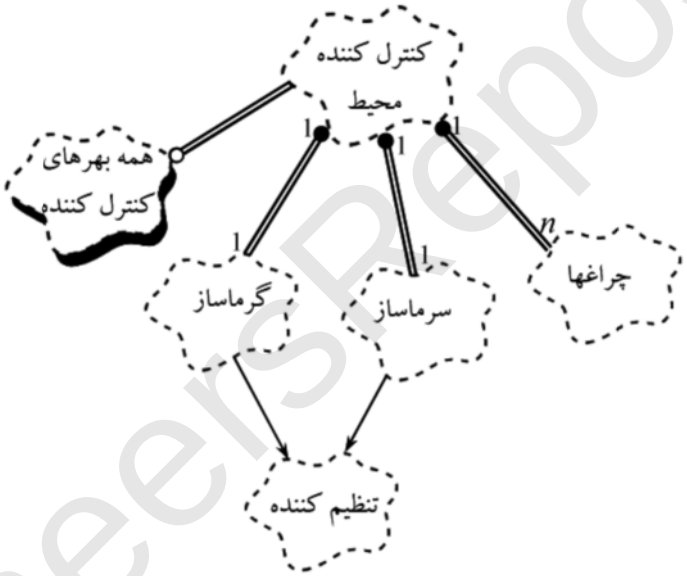
کاردینالیتی.

در بعضی موارد، به‌خصوص در مدل کردن کلاس‌هایی که جزئی از یک بانک اطلاعاتی را شکل می‌دهند، نمایش کاردینالیتی (cardinality) در بین کلاس‌هایی که از یکدیگر استفاده می‌کنند بسیار ارزشمند است.

در شکل قبل × می‌تواند یکی از موارد زیر باشد:

جدول ۵-:۵کاردینالیتی

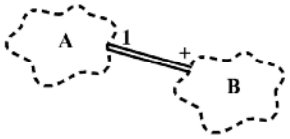
0	صفر
1	یک
*	صفر یا بیشتر
+	یک یا بیشتر
?	صفر یا یک
n	n



شکل ۵-۴: یک نمودار کلاس

مثال از کاردینالیتی.

به‌عنوان مثال، اگر در بین کلاس‌های A و B از علامت "1" نزدیک A و از علامت "+" نزدیک B استفاده کنیم، گویای این است که برای هر رویداد از A ممکن است یک، یا بیشتر از یک رویداد از B داشته باشیم و برای هر رویداد از B ، دقیقاً یک رویداد از A وجود دارد.



شکل ۵- : ۵مثالی از کاردینالیتی

تذکر: رابطه‌ی رویداد تنها برای زبانهایی که به نوعی از parameterized class یا generic mechanism حمایت می‌کنند، معنی دارد.

رابطه‌ی تعریف نشده که با خط‌چین و بدون پیکان نمایش داده شده است، گویای این نکته است که طراح رابطه‌ای را بین دو کلاس بیان می‌کند ولی نوع دقیق این رابطه بعداً مشخص خواهد شد.

همه‌بهرهای کلاس *(Class Utilities)*

برای نمایش همه‌بهرهای کلاس از شکلک زیر استفاده می‌شود.

شکل ۵- ۶: شکلک همه‌بهرهای کلاس

(مثل شکلک کلاس با این تفاوت که سایه‌دار شده است). این شکلک بیانگر یک زیربرنامه‌ی آزاد (free subprogram) یا مجموعه‌ای از زیربرنامه‌های آزاد است. اگر زبان مورد استفاده از این ساختار حمایت نکند، می‌تواند حذف شود.

گروه‌های کلاس‌ها و قابلیت رؤیت آنها *(Class Categories and Class Category Visibility)*

نوعاً نمودار کلاس می‌تواند شامل یک یا چند ده کلاس باشد. با این‌حال برای یک سیستم بزرگ ممکن است ساختار کلاس شامل چند هزار کلاس باشد. در چنین مواردی قطعاً نمایش تمام کلاس‌ها در یک نمودار کلاس میسّر نیست. لذا سازمان دادن کلاس‌ها در "گروه کلاس" (class category) مطرح می‌شود.

در تعداد معدودی از زبان‌های برنامه‌سازی شیءگرا، می‌توان کلاس‌های تودرتو (کلاس در داخل کلاس) داشت. در چنین مواردی، تصمیم‌گیری در مورد کلاس‌ها، تصمیم مهمی است. نکته‌ی مهمتر در طراحی، تعیین گروه کلاس‌ها، نقش زمینه (که به‌عنوان subject areas نیز نامیده می‌شود) است. این نکته برای برنامه‌سازان بسیار اساسی است ولی اکثر زبان‌های برنامه‌سازی شیءگرا از ساختار گروه کلاس‌ها حمایت نمی‌کنند. در عمل هر سیستم بزرگ دارای یک نمودار کلاس سطح بالا (top level class diagram)، شامل گروه‌های کلاس‌ها در بالاترین سطح تجرید است. چنین نموداری فهم عمومی از معماری سیستم را برای تولیدکننده میسّر می‌سازد.

نمودار کلاسِ سطح بالا.

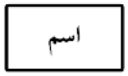
از شکلک جعبه برای نمایش گروه کلاس‌ها استفاده می‌شود. هر گروه کلاس به یک نمودار کلاس گسترش می‌یابد. هر نمودار کلاس ممکن است شامل هم گروه‌های کلاس‌ها، و هم کلاس‌های منفرد باشد.

شکلک برای گروه کلاس‌ها.

قابلیت رؤیت گروه کلاس *(Class Category Visibility)*

نظر به این‌که هر گروه کلاس‌ها بیانگر یک دربرگیری (encapsulation) است. ممکن است بعضی از عناصر آن برای خارج از آن گروه کلاس‌ها قابل رؤیت (visible) باشد. بعضی ممکن است خصوصیِ خود گروه کلاس‌ها و بعضی از گروه‌های دیگر وارد (import) شده باشند. از شکلک زیر برای بیان این مفاهیم استفاده شده است.

قابلیت رؤیت گروه کلاس‌ها.



شکل ۵-۷: گروه کلاس‌ها

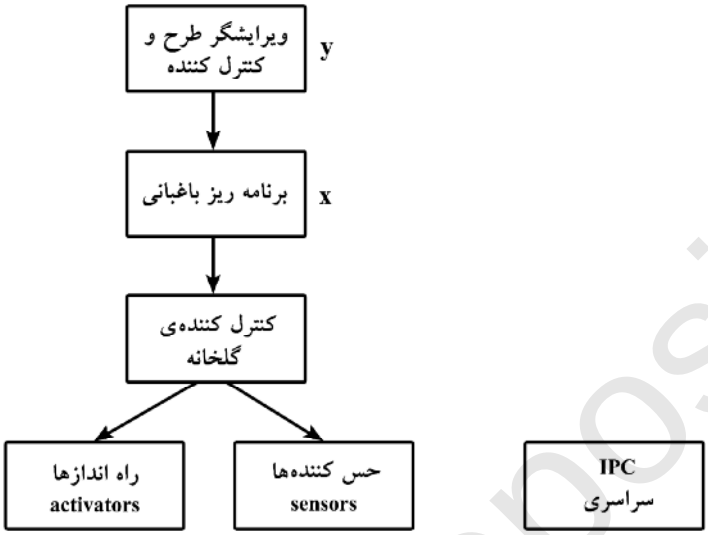
جدول ۵-۶: رابطه‌ی بین گروه‌های کلاس‌ها

اسم	قابل رویت برای دیگران
اسم	خصوصی برای همین گروه کلاس
اسم	وارد شده از گروه کلاس دیگر

رابطه‌ی بین گروه‌های کلاس‌ها به‌وسیله‌ی پیکان نمایش داده می‌شود. برای نمایش این‌که گروه کلاس‌های y، کلاس‌هایی را که در گروه x تعریف شده است را وارد می‌کند (یعنی x برای y قابل رؤیت است)، می‌توانیم یک کمان

جهت‌دار از y به x رسم کنیم. به این کمان که بیانگر رؤیت است می‌توان یک برچسب نسبت داد تا این رابطه یا نقش این رابطه را مستند سازی نماید.

گروه کلاس‌های سراسری.



شکل ۵- ۸: یک نمودار کلاسی بالا، نموداری از گروه‌های کلاس‌ها

گروه کلاس‌های y، کلاس‌هایی را که در گروه x تعریف شده‌اند، را وارد می‌کند. یعنی x برای y قابل رویت است.

meta class کلاسی که رویدادهایش کلاس هستند. کلاس کلاس‌ها super class کلاسی که از آن کلاس دیگر (که زیرکلاس بلافاصل نامیده می‌شود) ارث می‌برد.

برای نمایش گروه کلاس‌هایی که قبلاً تعریف شده‌اند و می‌خواهیم در تمام گروه‌های کلاس‌ها قابل رؤیت باشد، از درج واژه‌ی "سراسری" (global) در داخل جعبه‌ی مربوط به چنین گروه‌هایی استفاده می‌کنیم. در مثال بالا گروه IPC برای تمام گروه‌های کلاس‌های شکل فوق قابل رؤیت است. این بدین معنی است که تمام مواردی که توسط این گروه کلاس‌های سراسری صادر می‌شود، به‌وسیله‌ی هر یک از گروه‌های کلاس‌های دیگر از همان نمودار وارد (import) شده است.

الگوهای نمودار کلاس *(Class Diagram Templates)*

گرچه نمودارهای فوق بسیار کمک کننده است و مرور ساختار را تسهیل می‌نماید ولی کافی نیست. بخصوص باید به‌گونه‌ای مفهوم هر کلاس (اَبَر کلاسش، فیدهایش و اعمالش) را بیان کنیم.

جدول ۵- ۷: الگوی نمودار کلاس

: متغیر	اسم :	Name :	: identifier	
: متن	مستندات :	Documentation :	: text	
: وارد شده، خصوصی، صادر شده	قابل رویت نسبت به گروه کلاس‌های دربرگیرنده :	Visibility :	: exported / private / imported	در مورد کلیت یک کلاس
: ۰ / ۱ / n	کاردینالیتی :	Cardinality :	: 0 / 1 / n	
سلسله مراتب		Hierarchy		
: لیست اسامی کلاس‌هایی که این مستقیماً از آنها ارث می‌برد	سوپر کلاس‌ها	Super classes	: list of class names	قابل رؤیت درمورد اجزاء این کلاس

	: class name	Meta classes	متا کلاس‌ها	: اسم کلاس (کلاسی که این کلاس یک رویداد از آن است)
	: list of parameters	Generic parameters	پارامترهای جنریک	: جنریک پارامترها
	Interface / Implementations		واسط یا پیاده‌سازی	
	(Public / Protected / Private) :		(عمومی / حفاظت شده / خصوصی) :	
	: list of class names	Users	استفاده می‌کند	: لیست اسامی کلاس‌ها
	: list of field declarations	Fields	فیلدها	: فهرست تعریف فیلدها
	: list of operation declarations	Operations	عمل‌ها	: لیست تعریف عمل‌ها
	: state transition diagram	Finite state machine	ماشین با حالت‌های محدود	: نمودار تغییر حالت
	: sequential / blocking / active	Concurrency	هم‌وجودی	: ردیفی / مسدودی / فعال
	: text	Space complexity	پیچیدگی فضا	: متن
	: static / dynamic	Persistence	ماندگاری	: ایستا / پویا

ما برای هر یک از کلاس‌های نمودار کلاسی، یک الگوی کلاسی (class template) نیز داریم.

تذکر: این الگو نباید با الگوی مطرح شده در زبان C++ برای کلاس‌های پارامتری (parameterized class) اشتباه گرفته شود.

الگوی کلاس تمام جنبه‌های مهم هر کلاس را که در فصل سوم مطرح شد، دربر می‌گیرد. این الگو تقریباً وارد جزئیات شده است. اما در عمل انتظار نمی‌رود که تولیدکننده‌ی نرم‌افزار تمام این اجزاء را پر کند، مگر این‌که نمایش چنین جزئیاتی لازم باشد. در حقیقت، در مراحل اولیه طراحی این چنین الگوهایی به‌طور پراکنده پر می‌شوند؛ با پیشرفت طراحی، اجزاء بیشتری مشخص شده و اضافه می‌گردند. در صورتی‌که زبان پیاده‌سازی به اندازه‌ی کافی گویا باشد، رها کردن این الگوها و بیان مستقیم طرح به‌وسیله‌ی زبان مربوطه معقول است. در ادامه به شرح اجزاء این الگو (در صورت لزوم) می‌پردازیم.

جدول ۵-۸: شرح اجزاء الگوی نمودار کلاس

اسم (name):	بدون شرح
مستند (document):	شرحی در مورد این کلاس
قابلیت رویت:	بیانگر این نکته است که نسبت به گروه کلاس‌های دربرگیرنده، این کلاس صادر شده (قابل رؤیت توسط دیگران)، وارد شده (import) و یا خصوصی همین گروه کلاس‌هاست.
کاردینالیتی:	بیانگر این است که چه تعداد رویداد (instance) را برای این کلاس اجازه می‌دهیم. نوعاً این مقدار، 0 ، 1 و n است.
سلسله مراتب	این فیلد بیانگر نقش این کلاس در سلسله مراتب کلاس‌هاست. بسته به زمان پیاده‌سازی مورد استفاده‌ی، این کلاس ممکن است، صفر، یک یا بیشتر، super class و نیز class mete داشته باشد. در هر حال کلاس‌هایی که در اینجا ذکر شده است، از روابط بین کلاس‌ها، که در نمودار کلاس نمایش داده شد، استخراج شده است؛ دلیل مطرح شدن آنها در الگو، تنها به‌خاطر کامل و قابل فهم بودن الگو است (البته

	در صورت اجازه دادن زبان).
generic parameters:	در صورتی‌که زبان مورد استفاده اجازه دهد، این جزء از الگوی کلاس، فراهم کننده‌ی پارامترهایی به این کلاس است (به‌عنوان مثال، پارامترهای جنریک برای Ada ، یا ماکروها و کلاس‌های پارامتریک (parameterized classes) برای C++).
public / protected / private uses :fields operations	سه فیلد بعدی تا چهار بار تکرار شده است: سه بار برای واسط (interface) و یک بار برای پیاده‌سازیِ (implementation) این کلاس. در صورتی‌که زبان پیاده‌سازی اجازه دهد، واسط کلاس را می‌توان به سه قسمت عمومی (public)، حفاظت شده (protected) و خصوصی (private) تقسیم کرد. بنابراین C++ ممکن است از تمام این سه قسمت استفاده کند. Ada تنها از قسمت‌های عمومی و خصوصی استفاده می‌کند و Object Pascal تنها از قسمت‌های public استفاده می‌کند. در هر صورت این بخش از الگوی کلاسی مهمترین قسمت آن است؛ چرا که در این قسمت است که دیدگاه خارجی کلاس بیان می‌شود. این دیدگاه خارجی دربردارنده‌ی متغیرهای رویداد (instance variables) و متغیرهای کلاس (class variables) –که در طی field ها مستند شده است– و همچنین تمام عمل‌ها (operations) می‌باشد. برای هر فیلد ممکن است نام آن را مستند کنیم، این‌که ثابت یا متغیر است؛ کلاسش، هرگونه محدودیت آن (مثلاً: فیلد ما عدد صحیح است و مقدارش در محدوده‌ی ۱ تا ۱۰۰ است) و این‌که این فیلد چگونه مقدار اولیه می‌گیرد. باز هم مجبور نیستیم که تمام این چیزها را مستند کنیم، تنها مواردی که برای ما مهم است را مستند می‌کنیم. با این‌حال الگو به اندازه‌ی کافی کامل هست تا تمام جنبه‌های مهم فیلد را بیان کند. در ادامه، جزء عمل‌ها (operations) مبیّن لیستی از عمل‌ها (با الگوی خاص خود) است که به اختصار شرح می‌دهیم. جزء uses شبیه meta class و super class است که در اینجا به‌خاطر کامل و قابل فهم بودن قرار داده‌ایم، در واقع تمام روابط استفاده (using relationship) در نمودار شیء به‌صورت گرافیکی برقرار شده است. چهار عنصر بعدی از الگوی کلاس، مستندکننده‌ی معانی پویا و رفتار زمان (time) و حافظه‌ی (space) رویدادهای این کلاس است.
ماشین با حالت‌های محدود:	این جزء از الگوی کلاس نمایش دهنده‌ی یک نمودار تغییر حالت (state transition) است که به‌زودی به تفصیل مطرح خواهیم کرد.
هم‌وجودی :(concurrency)	این جزء بیانگر این است که آیا رویدادهای این کلاس ردیفی (sequential)، مسدودی (blocking)، یا فعال (active) است. طبیعتاً این معانی با معانی هم‌وجودی مربوط به عمل‌ها باید تطابق داشته باشد. به عنوان مثال اگر کلاسی به‌صورت ردیفی مشخص شده است، در این‌صورت هیچ یک از عمل‌هایش نمی‌تواند معانی guarded، concurrent یا چندگانه (multiple) داشته باشد.
پیچیدگی (space & time complexity)	این جزء از الگوی کلاس مستند کننده‌ی زمان و فضای مورد مصرف رویدادهای این کلاس است. این اندازه می‌تواند بر حسب واحدهای واقعی حافظه، یا بر حسب مفاهیم نسبی (نوعاً بر حسب علامت‌گذاری big O) بیان گردد.
ماندگاری :(persistence)	آخرین جزء الگوی کلاسی مستند کننده‌ی ماندگاری شیءهای این کلاس است. یک شیء ماندگار (persistence object) موردی است که حالت و کلاسش بعد از طول زندگی برنامه‌ای که آن را به‌وجود آورده است، باقی بماند. یک شیء انتقالی (transitory object) موردی است که بعد از مرگ برنامه‌ی به‌وجود آورنده‌اش، باقی نماند.

زبان‌های مختلف.

در مورد هر فیلد.

اجزاء دیگر الگوی نمودار کلاس.

باز هم تأکید می‌کنیم که ذکر تمام موارد لازم نیست. در هر صورت موارد مهم و ضروری را طراح می‌تواند در این قالب برای هر کلاسی مستند نماید. ضمن این‌که بعضی مفاهیم در عین حال در نموداهای مختلف به‌صورت گرافیکی بیان می‌شوند.

الگوی همه‌بهر کلاس *(Class Utility Templates)*

در ادامه الگوی همه‌بهر کلاس آمده است. نظر به این‌که همه‌بهر کلاس بیانگر مجموعه‌ای از زیر برنامه‌های آزاد (free sub programs) (و گاهی بیانگر ثابت‌ها و متغیرهای سراسری است) می‌باشد، الگوی آن زیرمجموعه‌ای از الگوی کلاس است.

جدول ۵-۹: الگوی همه‌بهر کلاس

: معرف	اسم	Name	: identifier
: متن	مستندات	Documentation	: text
: صادر شده، خصوصی، وارد شده	قابلیت رویت	Visibility	: exported/private/imported
	پارامترهای جنریک	Generic parameters	: list of parameters
واسط / پیاده‌سازی :		Interface / Implementation :	
: لیست اسامی کلاس‌ها	استفاده می‌کند	Uses	: list of class name
: لیست تعریف فیلدها	فیلدها	Field	: list of field declaration
: لیست تعریف عمل‌ها	عمل‌ها	Operations	:list of operation declaration

الگوی عمل‌ها *(Operation Templates)*

هم الگوهای کلاس‌ها و هم الگوی همه‌بهری کلاس ممکن است شامل لیستی از عمل‌ها باشند. اغلب بیان اسم هر عمل، پارامترها و معانی آنها (با یک متن آزاد) کافیست. اگر نیاز به مستندسازی دقیقتر باشد از الگوی زیر می‌توان استفاده کرد.

جدول ۵-۱۰: الگوی عمل‌ها

Name	: Identifier
Documentation	: text
Category	: text
Qualification	: text
Formal parameters	: list of parameter declarations
Result	: class name
Preconditions	: PDL / object diagram
Action	: PDL / object diagram
Post conditions	: PDL / object diagram
Exceptions	: list of exception declaration
Conevrrency	: sequential / guarded / concurrent / multiple
Time Complexity	: text
Space	: text

جدول ۵-۱۱:شرح اجزاء الگوی عمل‌ها

name:	نام
document:	مستندات
گروه (category):	ممکن است عمل‌ها را با معیارهایی گروه‌بندی کنیم، مثلاً گروه modifier ها که حالت‌ها را تغییر می‌دهند، یا selector ها که موجب تغییر حالت نمی‌شوند. این جزء برای مواقعی که کلاس دارای عملیات زیادی است، مفید می‌باشد.
توصیف (qualification)	در زبان CLOS از این فیلد بیشتر استفاده شده است. این‌که این عمل (operation) نمایشگر یک متد اصلی است یا نه، این‌که این عمل یک متدِ before، after یا around است یا نه (و یا تلفیقی از تعریف نوع متدهای کلاس) هست یا نه، می‌تواند در اینجا مستند گردد. به مطالب صفحه‌ی ۱۰۴، پاراگراف آخر مراجعه شود. (۱۰۴ از کتاب Booch). در مورد Invoke شدن این متد است.
پارامترهای رسمی :	
نتیجه (result):	تنها در صورتی‌که متد یک تابع باشد، نوع مقدار برگشتی از تابع، در اینجا قید می‌گردد. فیلدهای قبلی از عملیات، جنبه‌های ایستای عمل (operation) را مشخص می‌کرد. در چهار فیلد بعدی معانی پویای عمل مورد توجه قرار می‌گیرد.
عملکرد (action):	بیان معنی operation به‌وسیله‌ی یک متن آزاد و یا به‌صورت رسمی با بیان سه فیلد زیر: پیش شرط (pre condition): شرط لازم برای انجام عمل؛ پس شرط (post condition): شرایط بعد از انجام عمل؛ استثنا (exceptions): شرایط استثنایی. معنای یک operation ممکن است گاهی به نمودار شیء (که نمایشگر رابطه‌ی شیءهای شرکت‌کننده در عمل است) و معنای پویای آن عمل (که به‌وسیله‌ی نمودار زمانی و یا Program Description Language ≡ PDL بیان شده است) اشاره داشته باشد.
هم‌وجودی (concurrency):	یک operation ممکن است ردیفی باشد. به این معنی که تنها با وجود یک ریسمان کنترل معنای آن عمل تضمین می‌شود. ممکن است چند ریسمان کنترلی، در معنا پیدا کردن یک عمل مطرح باشد. اما با نگرش‌های مختلف به همگام سازی: Guarded Concurrent Multiple
پیچیدگی زمانی (time complexity):	متن در مورد زمان مورد مصرف.
حافظه (space complexity):	متن در مورد حافظه‌ی مورد مصرف.

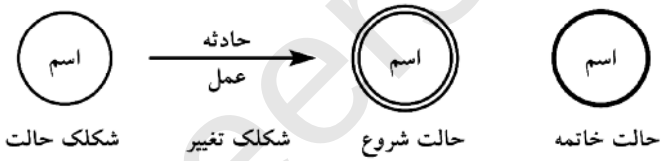
۵- ۳ نمودارهای تغییر حالت *(State Transition Diagrams)*

تنها با نگاه کردن به نمودارِ کلاس، رفتار پویای رویدادهای هر یک از کلاس‌ها مشهود نیست. رفتار پویای کلاس‌ها را توسط نمودارهای تغییر حالت (state transition diagrams) بهتر می‌توان مستند کرد. این نمودارها بیانگر فضای حالت‌های (state space) یک کلاس، حوادثی (events) که موجب انتقال از یک حالت به حالت دیگر می‌شود، و همچنین بیانگر عملِ (action) حاصل از تغییر حالت است. بنابراین نمودارهای تغییر حالت با بخش‌های دیگر علامت‌گذاری مرتبط است: الگوی کلاس ممکن است شامل یک نمودار تغییر حالت باشد و عمل‌هایی که در یک نمودار تغییر حالتِ بخصوص مطرح شده است ممکن است به نمودارهایِ شیء دیگر اشاره کند.

حالت‌ها توسط دایره و نام آن حالت نمایش داده می‌شود. نامِ حالت لازم است و باید در آن نمودار تغییر حالت منحصربه‌فرد باشد.

نوعاً نمودارهای تغییر حالت مربوط به یک کلاس، دارای حالت شروع و یا حالت خاتمه نیستند: هنگامی‌که یک شیء ایجاد می‌شود، بسته به شرایط محیطش وارد یک حالت می‌شود، و وقتی آن شیء از بین می‌رود، تمام حالت‌های مربوط به آن شیء ازبین می‌رود (moot). در مواردی که نمایش صریح حالت‌های شروع و خاتمه لازم باشد به ترتیب از دایره‌ی دوخطی و دوخطی پر شده (دایره‌ای با محیط ضخیم) استفاده می‌شود.

تغییر حالت: تنها رابطه‌ای که در بین حالت معنی دارد، تغییر حالت است، تغییر از یک حالت به حالت دیگر و همچنین تغییر یا انتقال از یک حالت به همان حالت. تغییر حالت توسط کمان جهت‌دار نمایش داده می‌شود. این کمان باید دارای برچسب (label) باشد. این برچسب گویای نام حداقل یک حادثه‌ای است که موجب این انتقال شده است، و یا مربوط به عمل حاصل است.



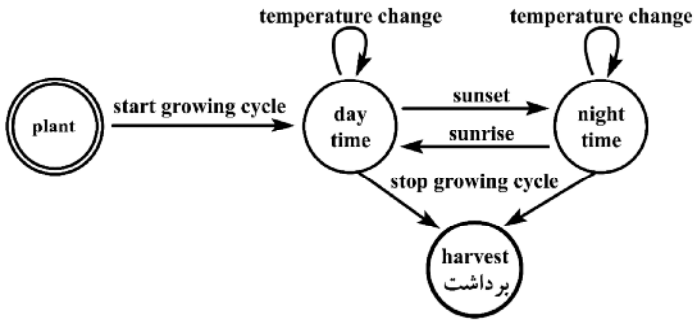
شکل ۵-۹: نمودار تغییر حالت

الگوی تغییر حالت *(State Transition Template)*

مثل کلاس‌ها، همه‌بهرهای کلاس و عملیات، برای تغییر حالت هم الگو وجود دارد. نظر به این‌که هر تغییر حالت را با یک عمل (action) مربوط و جمع می‌کنیم، این نوع ماشین با حالات محدود (finite state machine) را مدل "روی لبه‌ها" (mealy model) می‌نامند بر خلاف مدل "روی گره‌ها" (more model) که عمل‌ها به حالت‌ها نسبت داده می‌شوند.

جدول ۵-۱۲:الگوی تغییر حالت

: سیستمی از معرف‌ها	حادثه‌ها	Event	: list of identifier
: متن	مستند	Documentation	: text
: PDL / نمودار شیء	عمل	Action	: PDL / object diagram



شکل ۵-۱۰: مثالی از نمودار تغییر حالت

۵- ۴ نمودارهای شیء *(Object Diagrams)*

وظیفه‌ی نمودار

از نمودار شیء برای نمایش وجود شیءها و رابطه‌ی بین آنها در طرح منطقی سیستم استفاده می‌شود. یک نمودارِ شیءِ واحد، نمایشگر تمام یا قسمتی از شیء (object structure) از سیستم است. نوعاً برای طرح یک سیستم، مجموعه‌ای از نمودارهای شیء لازم است.

هدف از نمودار

هدف از هر نمودار شیء، بیان معانی مکانیزم‌های کلیدی (key mechanisms) در طرح منطقی است. در طرح یک سیستم، کلاس‌ها عمدتاً ایستا هستند، در حالی که شیءها بیشتر انتقالی (transitory) هستند. به‌طوری‌که بسیاری از آنها در حین یک اجرا ممکن است ایجاد و نابود شوند، بنابراین از نمودار شیء برای گرفتن معانی پویای (dynamic semantics) عملیات و ماشین‌های با حالت محدود (state machines finite) استفاده می‌کنیم. بنابراین یک نمودار شیء، خاصِ یک لحظه از یک حادثه‌ی گذرا را نشان می‌دهد. از این جهت نمودارهای شیء، نمونه‌ای (protypical) هستند: هر یک نمایشگر تراکنش‌هایی است که می‌تواند در بین مجموعه‌ای از شیءها بروز کند.

رابطه‌ی نمودار کلاس و نمودار شیء.

روابط مهمی بین نمودارهای کلاس و نمودارهای شیء برای به موازات هم کردن نقش متقابل کلاس‌ها و شیءها وجود دارد. به‌ویژه، هر شیء از نمودار شیء بیانگر یک رویداد (رویداد مشخص یا دلخواه) از یک کلاس است. به‌علاوه عملیاتی که در یک نمودار شیء به‌کار رفته است باید با عملیاتی که در کلاس‌های مربوطه تعریف شده است، سازگار (consistent) باشد. به‌عنوان مثال، اگر شیء S پیام یا عمل M را برای شیء R می‌فرستد، در این‌صورت عمل (operation) M باید برای کلاس مربوطه به S تعریف شده باشد. البته این سازگاری دوطرفه است. اگر در نمودارهای کلاس تغییراتی اعمال شد، تغییراتِ نظیر در نمودارهای شیء نیز باید داده شود.

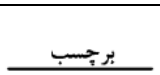
طرح منطقی سیستم. چرا هم نمودار کلاس و هم نمودار شیء؟

برای مستند کردن طرح منطقی سیستم، هم به نمودارهای کلاس و هم به نمودارهای شیء نیاز داریم. چراکه این نمودارها نمایشگر تصمیم‌های طراحی کاملاً متفاوت هستند. نمودارهای کلاس تجریدهای کلیدی از سیستم را مستند می‌کنند و نمودارهای شیء نمایشگر مکانیزم‌های مهم پردازش کننده‌ی این تجریدها هستند. دو عنصر مهم نمودار شیء، شیءها و رابطه‌ی شیءها می‌باشند.

نام شیء

شیءها. از شکلکِ مشابه شکلک مربوط به کلاس‌ها ولی به‌صورت خط پر استفاده می‌شود. احتیاجی نیست که اسم شیء منحصربه‌فرد باشد. در حقیقت بسیاری از شیءهای یک برنامه هیچ اسمی ندارند که صراحتاً در سطح کد مبدأ (source) شناخته شده باشند. به این دلیل ضرورتی ندارد که اسامی شیء دقیق باشد ولی می‌تواند گویای رویدادی نامعین (indefinite) که نماینده‌ی تجرید است، باشد. مثلاً "یک سردکننده"، "یک کتاب". در صورت لزوم می‌توان رویدادهای خاصی را نام‌گذاری کرد. مثلاً گل‌خانه‌ی شماره‌ی ۷.

روابط شیءها. رابطه‌ی بین دو شیء بیانگر این است که این شیءها می‌توانند به یکدیگر پیام بفرستند. نظر به این‌که پیام‌ها نوعاً دوطرفه (bidirectional) هستند، از خطوط بدون پیکان برای نمایش رابطه‌ها استفاده می‌کنیم: خط پر (solid) برای رابطه‌هایی از شیءها که در نرم‌افزار داخل سیستم بتوانند مدل شوند. خط نازک (gray) برای مواردی که خارج از سیستم هستند (مثلاً برای مستند کردن حلقه‌ی بازخورد، در سیستم‌های گرمایی و سرمایی).

	شکلک برای نمایش شیء
	داخل سیستم
	خارج سیستم

شکل ۵-۱۱: کلمات پیام، عمل و متد تقریباً معادل هم به‌کار می‌روند و معادل تابع عضویت

برای سیستم‌هایی که صرفاً ردیفی هستند، رسم یک خط ساده جهت‌دار برای بیان تراکنش بین دو شیء کافی است. اما در مواردی که چند ریسمان کنترل (multiple threads of control) وجود داشته باشد کمی مشکل‌تر است. به‌عنوان مثال، اگر دو شیء فعال و خواهان (active) هستند، در این‌صورت پیامی از S به R ممکن است تأخیر انداخته شود (چراکه S نمی‌تواند منتظر پاسخ بماند). ممکن است پیام‌هایی داشته باشیم که به یک شیء مشغول، وقفه (interrupt) بدهد. در مجموع پنج نوع همگام‌سازی پیام‌ها وجود دارند، که در فصل ۳ شرح داده شده‌اند: ساده (simple)، همگام (synchronous)، (balking)، (timeout) و ناهمگام (asynchronous). شکل زیر این شکلک‌ها را نمایش می‌دهد، هر کدام از شکلک‌ها ممکن است برچسبی (label) از فهرست اسامی پیام‌ها داشته باشد.

	simple
	synchronous
	balking
	timeout
	asynchronous

شکل ۵-۱۲: شکلک‌های همگام‌سازی پیام‌ها

قابلیت رؤیت شیء‌ها و هماهنگی *(Object Visibility and Synchronization)*

می‌توانیم به دو طریق جزئیات بیشتری از نمودار شیء را مشخص کنیم.

قابلیت رؤیت شیء، **اولاً:** می‌توانیم این نکته که چگونه دو شیء یکدیگر را می‌بینند، را مستند کنیم (این مطلب را تنها در مواردی که لازم هست، اضافه می‌کنیم). به شش صورت یک شیء می‌تواند برای شیء دیگر قابل رؤیت باشد. در شکل زیر این شش راه به‌وسیله‌ی شکلک‌ها نشان داده شده است.

	Same lexical scope
	Same lexical scope (shared)
	parameter
	Parameter (shared)

F	field
F	Field (shared)

شکل ۵-۱۳: قدبلیت رؤیت یک شیء برای شیء دیگر

به‌عنوان مثال، اگر شیء R یک فیلد مشترک (shared field) از شیء S است، در این صورت این قابلیت رؤیت را با قرار دادن شکلک "F" همراه رابطه‌ی R و S ، و نزدیک شیء R قرار می‌دهیم.

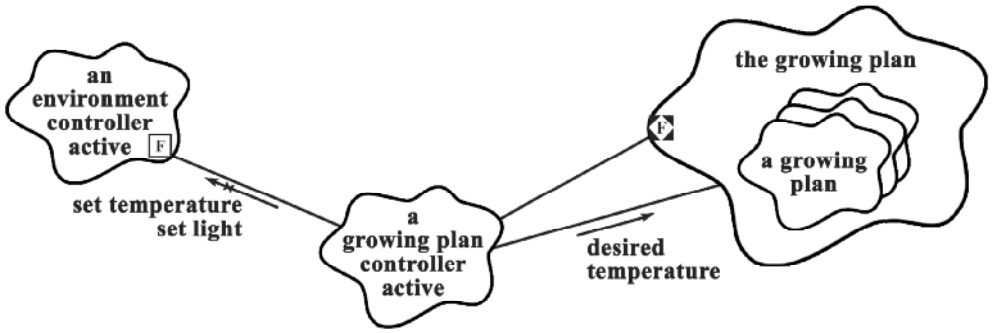
شکل ۵-۱۴: مثالی برای قابلیت رؤیت یک شیء برای شیء دیگر

با قرار دادن مناسب شیءها می‌توانیم به درک خود از قابلیت رؤیت در بین شیءها کمک کنیم. مثلاً، نوعاً شیءهای خواهان و فعال (actor) را در بالای این نمودارها و شیءهای خدمتگذار (server) را در پایین قرار می‌دهیم. همین‌طور برای نمایش تجمع و انبوهی (aggregation)، ممکن است شکلک شیء را درون دیگری جای دهیم.

هماهنگی پیام‌ها *(Message Synchronization)*

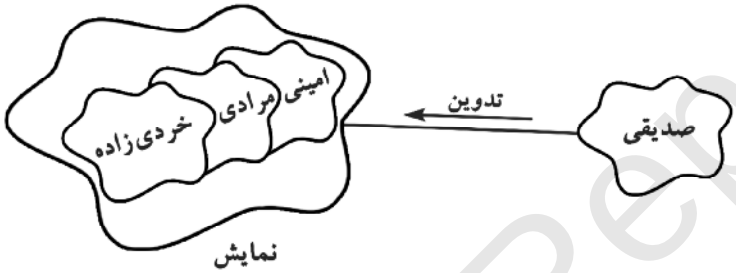
ثانیاً: نمایش چگونگی تراکنش شیءها نیز مهم است. علامت‌گذاری ما اجازه می‌دهد تا پیام‌های ارسالی از شیء S به شیء R را (با قرار دادن جهت روی رابطه و با برچسب‌گذاری اسم پیام) نمایش دهیم. جهت این خط، شیئی را نشان می‌دهد که روی آن عمل شده است. گرچه داده‌ها ممکن است در همان جهت یا جهت مخالف، جریان یابند (هم جهت در صورتی‌که یک پارامتر به عمل باشد، و در خلاف جهت اگر این داده حاصل فراخوان تابع باشد).

مثالی از یک نمودار شیء. شکل زیر مثالی از نمودار شیء است. شیئی که به نام a growing plan controller می‌باشد دو پیامِ set temperature و set lights را برای شیئی به نام an environment controller می‌فرستد. ارسال پیام بین این شیء فعال به‌صورت همگام (synchronous) می‌باشد. به‌علاوه نشان داده شده است که an environment controller فیلدی از a growing plan، می‌باشد، اما مشترک است، به این معنی که برای شیئی که توسط این فیلد مشخص شده است (aliases) وجود دارد. مجموعه‌ی the growing plan هم فیلدی از a growing plan controller می‌باشد، ولی فیلدی غیرمشترک است، به این معنی که شناسه‌ی (identity) این شیء هیچ جای دیگر شناخته شده نیست. شیئی که دارای نام the growing plan است، نمایشگر یک مجموعه، شامل شیءهای متمایز growing–plan می‌باشد، و برای این‌که نمایش داده شود که آنها nested هستند، آنها در شکلکِ the growing plan گذاشته شده‌اند. می‌توان از همین روش برای نمایش تودرتویی کلاس‌ها، همه‌بهرهای کلاس‌ها و مؤلفه‌ها استفاده کرد. در این طرح شیء a growing plan controller ممکن است پیامِ (انتخاب a selector) desired temperature را به هر یک از growing plan ها بفرستد. در اینجا معانی انتقال پیام‌ها ساده است (simple)، به‌دلیل این‌که شیءهای growing–plan ردیفی هستند.



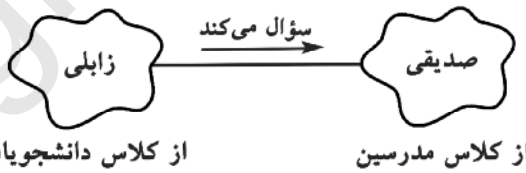
شکل ۵-۱۵: یک نمودار شیء

مفهوم ضرورت همگام‌سازی. در مواردی که بیش از یک خط کنترلی وجود دارد.



شکل ۵-۱۶: مثالی برای نمودار شیء

شکل ۵-۱۷: مثالی برای نمودار شیء



شکل ۵-۱۸: مثالی برای نمودار شیء

الگوی نمودار شیء *(Object Diagram Templates)*

شیءها و پیام‌های نمودار شیء هر یک الگویی برای ارائه‌ی اطلاعات بیشتر دارند.

الگوی شیء (object template). جدول زیر الگو برای شیء است. این الگوی کلاس شیء، که گویای عملیاتی است که مشتری‌ها (client) ممکن است روی آن شیء، در نمودار شیء انجام دهند را مستند می‌کند. این الگو در عین حال بیان ماندگاری (persistence) شیء را میسّر می‌سازد. خصوصیتی از شیء که باید با ویژگی ماندگاری که در کلاس مربوطه تعریف شده است، مطابقت داشته باشد. به‌ویژه اگر در الگوی کلاس مربوطه ذکر شده است که، تمام رویدادها گذرا (transitory) هستند، در این‌صورت ماندگاریِ این شیء فقط می‌تواند ایستا (static)، (یعنی شیء در تمام طول اجرای

برنامه وجود دارد)، یا پویا، (یعنی شیء در حین اجرای برنامه به‌صورت پویا ایجاد و نابود می‌گردد) باشد. اگر الگوی کلاس مبیّن این باشد که رویدادها ممکن است ماندگار باشند، در این‌صورت شیء می‌تواند ایستا، پویا، و یا ماندگار باشد (یعنی بعد از اتمام برنامه‌ای که طی آن، ایجاد شده بود، وجود داشته باشد).

جدول ۵-۱۳: الگوی نمودار شیء

: معرف	اسم	Name	: identifier
: متن	مستند	Documentation	: text
: نام کلاس	کلاس	Class	: class - name
: ماندگار / ایستا / پویا	ماندگاری	Persistence	: persistence / static / dynamic

الگوی پیام (message template). این الگو اطلاعات بیشتری از عمل تعریف شده برای کلاس یک شیء که به نوبه‌ی خود جزئیات بیشتری از معنای عمل است، را ارائه می‌نماید. می‌توان اطلاعات زمانی را نیز بیان کرد. مثل این‌که آیا این پیام به‌صورت دوره‌ای (periodically) ارسال می‌شود و یا نه، و اگر می‌شود با چه نرخی (how often). این جزئیات بخصوص در طراحی کاربردهایی که زمان در آنها حیاتی است، بسیار مفید است.

جدول ۵-۱۴: الگوی پیام

: نام عمل	عمل	Operation :	operation name
: متن	مستند	Document :	text
: ادواری / غیر ادواری	تکرار	Frequency :	periodic / periodic
: ساده/ همگام/ ترک کننده/ انقضا/ غیرهمگام	همگام سازی	Synchronization :	simple / synchronous / balking / timeout / asynchronous

اضافه کردن اطلاعاتِ زمانی در اینجا، ساخت ابزارهای تحلیلی برای تعیین زمان لازم برای تکمیلِ عملیات و برای ارزیابی مکانیزم‌ها را میسّر می‌سازد.

۵-۵ نمودارهای زمانی *(Timing Diagrams)*

نمودارهای شیء به‌خودی‌خود ایستا هستند: مجموعه‌ای از شیءها را که با یکدیگر همکاری می‌کنند و به یکدیگر پیام می‌فرستند، را نشان می‌دهند. اما نمودارهای شیء، جریان کنترل (flow of control) و یا ترتیب حوادث (events) را نشان نمی‌دهند. نمودارهای تغییر حالت کلاس‌ها هم کمکی نمی‌کنند؛ چرا که تنها چگونگی انجام تغییرات در یک شیء واحد را نشان می‌دهد و نه تغییرات مجموعه‌ای از شیءهایی که با یکدیگر همکاری می‌کنند. برای مستندسازی غالب مکانیزم‌ها، اطلاعات دیگری لازم است، لذا در علامت‌گذاری خود سه راه ممکن برای مستند سازی پویایی انتقال پیام‌ها در نمودار شیء را منظور می‌کنیم:

دیدگاه اول: شماره‌گذاری پیام‌ها

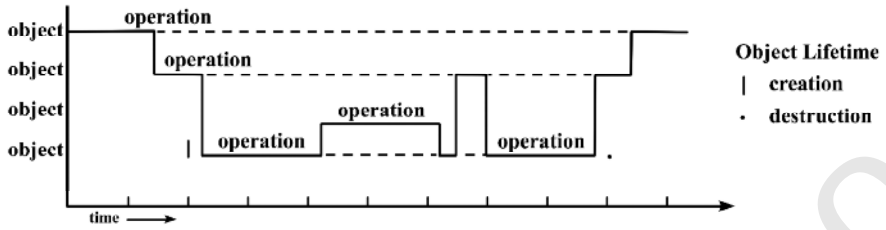
۱. دیدگاه اول خیلی ساده است. هر یک از پیام‌ها در نمودار شیء را شماره‌گذاری می‌کنیم به‌گونه‌ای که مبیّن ترتیب نسبی آنها برای فعال شدن (invoke) باشد. ابتدا پیام ۱ ارسال می‌شود؛ سپس پیام ۲ و الی آخر. این روش برای یک گردشِ کنترل کاملاً ردیفی کار می‌کند، اما اگر بخواهیم گردش کنترل شرطی را بیان کنیم کافی نیست (مثلاً اگر شرط C برقرار بود پیام M ارسال شود، در غیر این‌صورت پیام N ارسال شود)؛

دیدگاه دوم: PDL

۲. در دیدگاه دوم، باهر نمودار شیء Program Description Language ≡ PDL را هم برای بیان ترتیب حادثه‌ها در نظر می‌گیریم. برای اغلب طرح‌ها، PDL انتخاب می‌شود؛ چرا که گویا (expressive)، قابل فهم و مساعد برای خودکارسازی است.

دیدگاه سوم:

۳. دیدگاه سومی را که مفید یافته‌ایم، ریشه در نمودارهای زمانی (که توسط مهندسین سخت‌افزار به‌کار می‌رود) دارد. نمودار زمانی (timing diagram) گرافی است که زمان (واقعی یا نسبی) را روی محور افقی و شیءها را روی محور عمودی در نظر می‌گیرد. مثل شیءها، تنها مواردی را در نظر می‌گیریم که تراکنش متقابل آنها مستندسازی یک مکانیزم باشد. با حرکت از چپ به راست ممکن است عملی فعال invoke شود. به‌عنوان مثال ممکن است با عمل ۱ برای شیء R شروع کنیم، با گذشت زمان، این عمل، عمل ۲ را برای شیء S فعال می‌کند؛ که به نوبه‌ی خود پیام ۳ را برای شیء T می‌فرستد. خط‌چین‌ها مبین تودرتویی (nesting) پیام‌هاست. به‌عنوان مثال، با کامل شدن عمل ۳، کنترل به عمل ۲ بازمی‌گردد، که در این‌صورت آزاد است که چیزهای دیگر را انجام دهد.

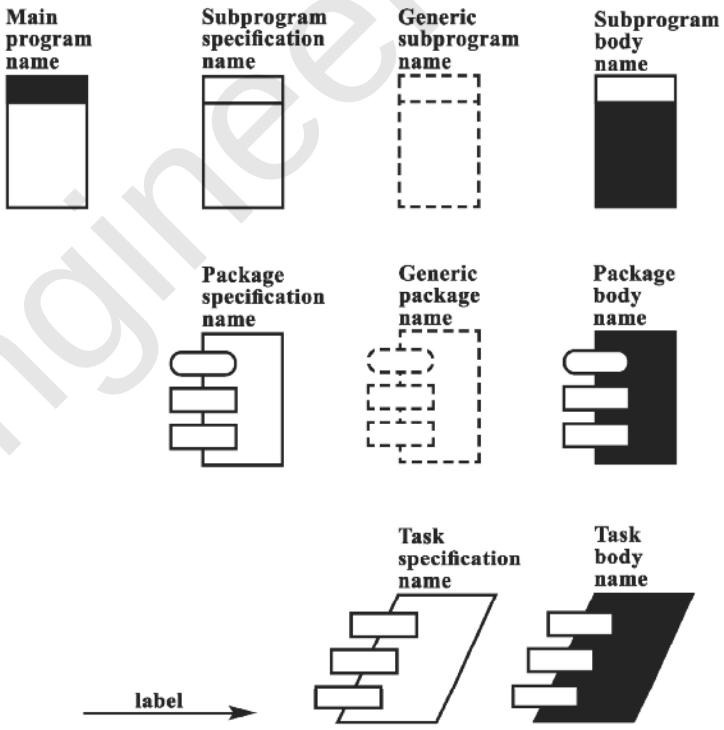


شکل ۵-۱۹: نمودار زمانی نمودار زمانی قسمتی از نمودار شیء و بیانگر ترتیب بروز حوادث در بین مجموعه‌ای از شیءهاست

۵- ۶ نمودارهای مؤلفه *(Module diagrams)*

از نمودارهای مؤلفه برای نمایش انتساب کلاس‌ها و شیءها به مؤلفه‌ها، در طرح فیزیکی سیستم، استفاده می‌شود. در بعضی زبان‌ها مثل C++ از ساختار مؤلفه حمایت می‌شود.

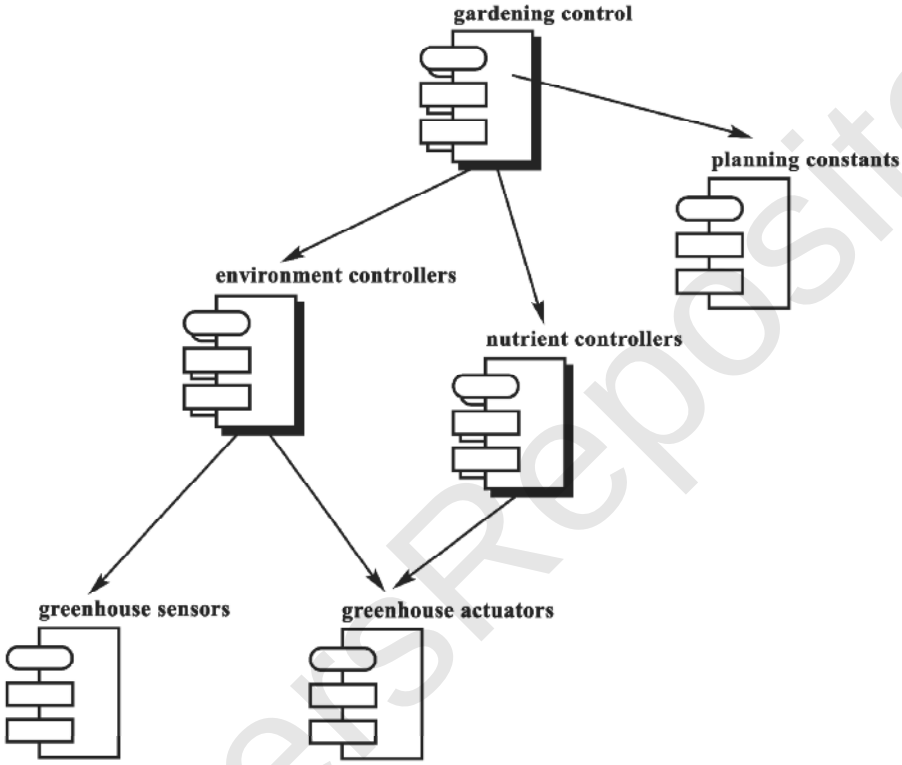
دو عنصر مهم معماری مؤلفه‌ها (module architecture)، مؤلفه‌ها و قابلیت رؤیت مؤلفه (module visibility) هستند. مؤلفه‌ها. شکل زیر بیانگر شکلک‌های انواع مؤلفه‌هایی است که در نمودار مؤلفه از آنها استفاده می‌کنیم.



شکل ۵-۲۰: نمودارهای مؤلفه

قابلیت رؤیت مؤلفه‌ها *(Module visibility)*

تنها رابطه‌ی ممکن بین دو مؤلفه وابستگی کامپایل (compilation dependency) است که توسط کمانی نمایش داده می‌شود. ممکن است با اضافه کردن یک برچسب (label) معنای این رابطه را مستند کنیم. به‌عنوان مثال برای نمایش این‌که مؤلفه‌ی G به مؤلفه‌ی H وابسته است، یک کمان از G به H رسم می‌کنیم، به این معنی که H برای G قابل رؤیت است.



شکل ۵- ۲۱: یک نمودار مؤلفه[1]

زیرسیستم‌ها *(Subsystems)*

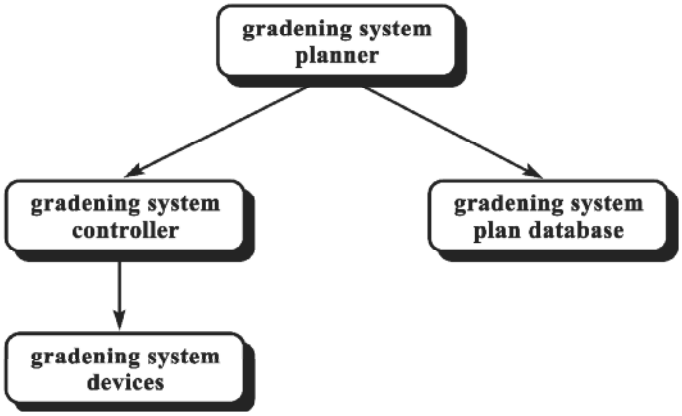
یک سیستم بزرگ ممکن است صدها مؤلفه داشته باشد. در چنین مواردی برای درک بهتر سیستم، از زیرسیستم‌ها استفاده می‌شود. زیرسیستم‌ها نمایشگر خوشه‌هایی (clusters) از مؤلفه‌های مرتبط به یکدیگرند و برای برنامه‌سازی در سطح کلان (programming-in-the-large) بسیار کمک‌کننده هستند. متأسفانه زبان‌های برنامه‌سازی موجود از این ساختار حمایت نمی‌کنند. گرچه محیط‌های تولید نرم‌افزار (software development environment) از این ساختار حمایت می‌کنند.

در عمل، یک سیستم بزرگ در بالاترین سطح تجرید دارای (مؤلفه‌هایی از) زیرسیستم است. معماری فیزیکی بزرگتر برای کل سیستم است. در اینجا هر زیرسیستم نمایشگر یک نمودار مؤلفه است (مشابه گروه کلاس‌ها که قبلاً در همین فصل مطرح شد).

شکل ۵-۲۲: شکلک برای زیرسیستم

---

[1] *module diagram*



شکل ۵-۲۳: یک نمودار مؤلفه‌ی سطح بالا (نمودار زیرسیستم‌ها)[1]

الگو برای نمودار مؤلفه *(Module Diagram Templates)*

شامل اجزاء زیر:

جدول ۵-۱۵: الگو برای نمودار مؤلفه

اسم	Name :	identifier
مستندات	Documentation :	text
تعاریف : فهرست تعاریف	Declarations :	list of declaration

معانی پویای الگوی مؤلفه *(Module Template Dynamic Semantics)*

تمام نمودارهایی که تابه‌حال بررسی کرده‌ایم، هم دارای معانی ایستا و هم دارای معانی پویا هستند؛ گرچه نمودارهای مؤلفه نوعاً ایستا هستند. با این‌حال در بعضی کاربردها، ممکن است مؤلفه‌ها دارای معانی پویا هم باشند. به‌عنوان مثال برای کاربردی که حافظه‌ی مورد نیاز را در اختیار ندارد، ممکن است برای اجرا از swap کردن کد مؤلفه‌های مختلف به حافظه و overlay استفاده کنیم. این چنین معانی پویا را می‌توان توسط نمودارهای زمانی (timing diagram) و یا Program Description Language ≡ PDL تشریح کنیم.

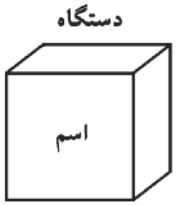
۵- ۷ نمودار فرایند *(Process Diagram)*

معماری فرایند (process architecture). در سیستم‌های بزرگ ممکن است از بیش از یک کامپیوتر استفاده کنیم. از نمودار فرایند برای نمایش فرایندها (processes) به پردازنده‌ها استفاده می‌شود. و این نمودار هم بیانگر طرح فیزیکی (physical design) سیستم است. حتی برای سیستم‌هایی که روی تنها یک پردازنده اجرا می‌شوند نیز نمودارهای فرایند مفید هستند، اگر پیاده‌سازی ما متضمّن دستگاه یا شیءهای فعال (active objects) باشد. به این معنی که چند پردازش (یا چند خط کنترل) در یک زمان وجود دارد.

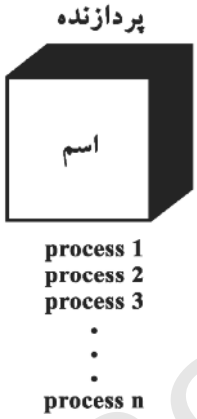
شکل زیر شکلک نمایشگر پردازنده‌ها و دستگاه‌ها و مثالی از نمودار فرایند را نشان می‌دهد.

---

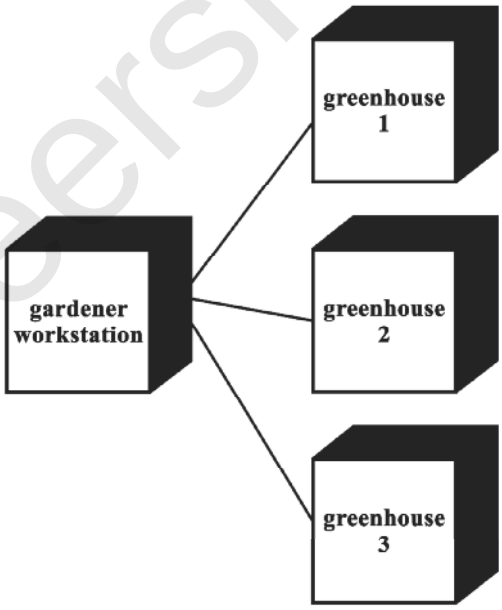
[1] *top-level module diagram*



شکل ۵-۲۴: شکلک دستگاه



شکل ۵-۲۵: شکلک پردازنده



شکل ۵-۲۶: یک نمودار فرایند

الگو برای پردازنده

جدول ۵-۱۶: الگو برای پردازنده

Processor		
Name :	identifier	اسم :
Documentation :	text	مستندات :

Characteristics :	text	مشخصات پردازنده :
Processor :	list of processes	لیست فرایندهای منتسب به این پردازنده :
Scheduling :	preemptive / non preemptive / cycle[1] / executive[2] / manual[3]	چگونگی زمان بندی :

الگوی پراسس

جدول ۵-۱۷: الگوی پراسس

Processor		
Name :	identifier	اسم :
Documentation :	text	مستندات :
Priority :	integer	اولویت این پراسس :

الگوی دستگاه

جدول ۵-۱۸: الگوی دستگاه

Device		
Name :	identifier	اسم :
Documentation :	text	مستندات :
Characteristics :	text	مشخصات این دستگاه :

۵- ۸ محصول طراحی شیءگرا

محصول اساسی طراحی شیءگرا، نوعاً مجموعه‌ای از نمودارهای کلاسی، نمودارهای شی، نمودارهای مؤلفه و نمودارهای فرایند است. در رأس طراحی ما، بالاترین نمودار کلاس (بیانگر تجریدهای اصلی در طرح منطقی سیستم)، بالاترین نمودار مؤلفه (بیانگر اجزاء کلیدی از ایجاد فیزیکی سیستم) و نمودار اصلی فرایند (نمایشگر اجزاء کلیدی طرح فیزیکی سخت‌افزار) می‌باشد. مکانیزم‌های به‌کار رفته در پیاده‌سازی ما، در نمودارهای شیء مطرح شده‌اند. مسلماً استفاده از ابزار برای تهیه‌ی این نمودار کمک‌کننده است.

تغییر مقیاس *(Scaling Up and Sealing Down)*

ما دریافته‌ایم که این علامت‌گذاری هم برای سیستم‌های کوچک (چند صد خطی) و هم سیستم‌های بسیار بزرگ (چند میلیون خطی) قابل به‌کارگیری است. نکته‌ی دیگر این‌که این علامت‌گذاری مستقل از زبان برنامه‌سازی مورد استفاده در پیاده‌سازی بوده و برای زبان‌های مختلف شیءگرا قابل به‌کارگیری است.

۵- ۹ خلاصه‌ی فصل پنجم کتاب طراحی شیءگرا / علامتگذاری *(Notation)*

- طراحی عبارت از رسم نمودار نیست؛ نمودار یک طرح را دربر دارد؛

---

[1] هر پراسس محدوده‌ی مشخصی از زمان دارد
[2] زمان‌بندی با الگریتمی خاص
[3] زمان‌بندی توسط کاربر از خارج سیستم

- در طراحی یک سیستم پیچیده، این مهم است که از چند نقطه نظر، مورد توجه قرار گیرد: از نظر ساختار منطقی، از نظر ساختار فیزیکی، از نظر معانی ایستا (static semantics) و از نظر معانی پویا (dynamic semantic)؛
- علامت‌گذاری برای طراحی شیءگرا شامل چهار نمودار اصلی (نمودارهای کلاس، نمودارهای شیء، نمودارهای مؤلفه و نمودارهای فرایند) و همین‌طور دو نمودار کمکی (نمودار تغییر حالت و نمودار زمان‌بندی) می‌باشد؛
- نمودار کلاس (class diagram) برای نشان دادن وجود کلاس‌ها و روابط بین آنها در طرح منطقی سیستم به‌کار می‌رود؛ یک نمودار کلاس، نمایشگر تمام یا بخشی از ساختار کلاس (class structure) از یک سیستم است؛
- نمودار شیء (object diagram) برای نشان دادن وجود شیءها و روابط بین آنها در طرح منطقی سیستم به‌کار می‌رود؛ یک نمودار شیء، نمایشگر تمام یا بخشی از ساختار شیء (object structure) از یک سیستم بوده و بیانگر معانی تجریدهای کلیدی در طرح منطقی است. یک نمودار شیء واحد، نمایشگر یک لحظه‌ی (snapshot) زمانی از شیء و در غیراین‌صورت نمایشگر یک حادثه‌ی گذرا (transitory event) یا پیکربندی (configuration) از شیء است؛
- نمودار مؤلفه (module diagram) برای نشان دادن تخصیص کلاس‌ها و شیءها به مؤلفه‌ها در طرح فیزیکی (Physical design) یک سیستم است؛ یک نمودار مؤلفه نمایشگر تمام یا بخشی از معماری مؤلفه‌ای (architecture of a system) یک سیستم است؛
- نمودار فرایند (process diagram) برای نشان دادن تخصیص فرایندها به پردازنده‌ها در طرح فیزیکی یک سیستم به‌کار می‌رود؛ نمودار فرایند نمایشگر تمام یا بخشی از معماری فرایند (process architecture) از یک سیستم است؛
- نمودار تغییر حالت (state transition diagram) برای نشان دادن فضای حالت از یک رویداد، از یک کلاس، داده شده، و برای نمایش حادثه‌ای که موجب انتقال از یک حالت به حالت دیگر شده، و عمل‌هایی که از این تغییر حالت نتیجه می‌شوند، به‌کار می‌رود؛
- نمودار زمانی (timing diagram) برای نشان دادن تراکنش‌های پویا (dynamic transaction) بین شیءهای مختلف در نمودار شیء به‌کار می‌رود؛
- **تذکر مهم:** یک دلیل پرداختن به این همه جزئیات در این متن فشرده، توجه دادن به مجموعه مواردی است که در طراحی شیءگرا می‌تواند (و بهتر است) مورد توجه قرار گیرد.

فصل ۶

فرایند طراحی شیءگرا (The Process)

طراحی شیءگرا فرایندی نیست که با یک مشخصات خواسته‌ها شروع، و با گزارشی برای پیاده‌سازی تمام شود. با تجربه‌ی ما فرایند طراحی شیءگرا عموماً ترتیب زیر از وقایع را دنبال می‌کند:

- شناسایی و تعیین (identify) کلاس‌ها و شیءها در یک سطح مشخصی از تجرید؛
- شناسایی و تعیین معانی این کلاس‌ها و شیءها؛
- شناسایی و تعیین روابط بین این کلاس‌ها و شیءها؛
- پیاده‌سازی این کلاس‌ها و شیءها.

طراحی شیءگرا، یک فرایند افزایش یابنده است: شناسایی و تعیین کلاس‌ها و شیءهای جدید معمولاً موجب می‌شود که ما معانی و روابط کلاس‌ها و شیءهای موجود را پالایش کرده و ارتقاء دهیم.

طراحی شیءگرا در عین حال تکرار شونده (iterative) است: پیاده‌سازی کلاس‌ها و شیءها غالباً ما را به کشف یا ابداع کلاس‌ها و شیءهای جدید هدایت می‌کند، کلاس‌ها و شیءهایی که حضورشان موجب ساده شدن و عمومی‌تر شدن طرح ما می‌شود. در ادامه هر یک از گام‌های فوق مورد بررسی قرار می‌گیرد.

۶- ۱ شناسایی و تعیین کلاس‌ها و شیءها *(Identifying Classes and Objects)*

گام اول.

اولین گام شامل دو فعالیت است:

- کشف تجریدهای کلیدی در فضای مسأله (کلاس‌ها و شیءهای مهم)؛
- ابداع مکانیزم‌های مهم (ساختارهایی که به‌وسیله‌ی آنها مجموعه‌ای از شیءها با یکدیگر همکاری کرده و رفتارهایی را ایجاد کنند که جواب‌گوی (بخشی) از خواسته‌های مسأله باشند).

این کار با استفاده از تکنیک‌هایی که در فصل ۴ مطرح شده انجام می‌شود. اصولاً تولیدکننده باید به‌عنوان یک تجریدگر (abstractionist) عمل کند. باید لغت‌نامه‌ی زمینه‌ی مسأله را (با مطالعه‌ی خواسته‌های مسأله و با ارتباط با خبرگان محیط مسأله) یاد بگیرد.

محصولات اوّلین گام

محصولات این گام می‌تواند خیلی رسمی تا خیلی غیر رسمی باشد. به‌صورت غیر رسمی می‌تواند تنها فهرستی از اسامیِ با معنی شیءها و کلاس‌های مسأله باشد.از سوی دیگر، می‌تواند به‌طور رسمی مبیّن معانی این تجریدها و مکانیزم‌ها با پر کردن الگوهای مربوطه باشد (الگوهای علامت‌گذاری در فصل پنجم آمده است).

محصولات دیگر.

در اکثر موارد این گام وقت چندانی نمی‌گیرد. غالباً یک طراح ارشد، فهرستی از کلاس‌ها و شیءهای کاندید را پیشنهاد می‌کند. اگر تفکیک فیزیکی (physical decomposition) سیستم مهم باشد (همچنان‌که در موارد موجود در زیرقراردادها (sub contracts) مهم است)، در این‌صورت، پیش‌نویس‌های طرح‌های مؤلفه‌ها (module design) نیز در این زمان تهیه می‌شود. و بالاخره اگر عوامل سخت‌افزاری برای سیستم وجود داشته باشد، در این‌صورت طرح اولیه‌ی نمودارهای فرایند (process diagrams) نیز مفید است.

پیش‌نویس طرح مؤلفه.

نمودار فرایند.

۶- ۲ شناسایی و تعیین معانی کلاس‌ها و شیءها

(Identifying the Semantics of Classes and Objects)

گام دوم.

گام دوم شامل یک فعالیت اصلی است:

تعیین معنی‌های کلاس‌ها و شیءهای مشخص شده در گام اول.

از دید خارجی.

در اینجا تولیدکننده‌ی نرم‌افزار به‌صورت یک ناظر خارجی و مستقل عمل می‌کند و سعی می‌کند به هر کلاس از نقطه‌نظر واسط آن (interface) نگاه کند؛ تا چیزهایی که ما می‌توانیم با هر رویداد (instance) از هر کلاس انجام دهیم و همچنین چیزهایی را که هر شیء می‌تواند با شیء دیگر انجام دهد را شناسایی و تعیین نماییم.

مشکل‌تر و وقت‌گیرتر.

این گام از گام قبلی بسیار مشکل‌تر و وقت‌گیرتر است. پیدا کردن کلاس‌ها و شیءها ساده است؛ تصمیم‌گیری در مورد پروتکل هر شیء مشکل است. به همین دلیل فرایند طراحی شیءگرا در این نقطه تکرار شونده می‌گردد. تصمیم‌گیری در مورد پروتکل یک شیء، ممکن است به تغییر شیء دیگر منجر گردد. عملاً وجود تجریدهای کلیدی ما تغییر نمی‌کند، بلکه ما تنها حدود آنها را تغییر می‌دهیم.

متنی در مورد هر شیء.

یک روش خوب برای هدایت این فعالیت‌ها، نوشتن متنی در مورد هر شیء است. متنی که چرخه‌ی زندگی هر شیء از ایجاد تا مرگ و رفتار آن تعریف کند.

محصولات این مرحله

محصولات دیگر گام دوم.

محصولات این مرحله ماهیت افزایش یابنده‌ی طراحی شیءگرا را منعکس می‌کنند. برای مستندسازی تصمیم‌ها در مورد معنیِ هر شیء و کلاس، ما عموماً الگوهایی که در مرحله‌ی قبل (مطابق قواعد علامت‌گذاری) رسم کرده‌ایم را پالایش (refine) می‌کنیم. ما باید تمام معانی ایستا و پویای هر تجرید کلیدی و مکانیزم را به بهترین وجهی که می‌توانیم به موقع مستند کنیم. ممکن است نمودارهای شیء (object diagrams) جدید را هم برای مستند کردن مکانیزم‌هایی که احتمالاً در این مرحله ابداع کرده‌ایم، رسم کنیم. در اینجا ممکن است در مورد نمونه‌سازی بعضی از قسمت‌های طرح تصمیم بگیریم.

گام سوم.

۶- ۳ شناسایی و تعیین روابط بین کلاس‌ها و شیءها

(Identifying the Relationship Among Classes and Objects)

اظهار نظر قطعی در مورد انواع روابط بین کلاس‌ها.

فعالیت‌ها: گام سوم عمدتاً بسطِ فعالیتهای گام قبلی است. در اینجا بطور دقیق تعیین می‌کنم که چگونه چیزها در داخل سیستم با یکدیگر در تراکنش هستند. در رابطه با تجریدهای کلیدی، این بدین معنی است که ما باید در مورد روابط استفاده (using)، وراثت (inheritance) و انواع دیگر رابطه بین کلاس‌ها اظهار نظر قطعی کنیم. در رابطه با شیءها در پیاده‌سازی ما، این بدین معنی است که ما باید معانی ایستا و پویای هر مکانیزم را مشخص کنیم.

دو فعالیت مرتبط.

در اینجا دو فعالیت مربوط وجود دارد که ممکن است موجب تجدید نظر در محصولات قبلی ما گردد:

۱. کشف الگوها.

۱. اول این‌که ما باید الگوها (patterns) را کشف کنیم: الگوها در میان کلاس‌ها، که موجب می‌شود ما ساختار کلاسِ سیستم (systems class structure) را درک و ساده کنیم؛ و الگوها در بینِ مجموعه‌های شیءهای همکاری‌کننده، که ما را به تعمیم مکانیزم‌هایی که هم‌اکنون در طرح ما وجود دارند هدایت می‌کند. این قسمت از طراحی تمام مهارت‌های خلاّق طراح را می‌طلبد. توفیق در این قسمت است که یک طرح خوب را از یک طرح عالی متمایز می‌کند؛

۲. تصمیم در مورد قابلیت رؤیت.

۲. دوم این‌که ما باید در مورد قابلیت رؤیت (visibility) تصمیم بگیریم: این‌که کلاس‌ها چگونه یکدیگر را می‌بینند، چگونه شیءها یکدیگر را می‌بینند، و مطلب دیگری که به همان اندازه مهم است این‌که چه کلاس‌هایی و چه شیءهایی یکدیگر را نباید ببینند. این تصمیم‌ها به ما کمک می‌کند تا در مورد دسته‌بندی (packaging) در طراحی معماری مؤلفه‌های (modal architecture) سیستم تصمیم بگیریم.

کشف الگوها و تصمیم‌ها در مورد قابلیت رؤیت ممکن است موجب تجدید نظر در مورد پروتکل‌های کلاس‌های مراحل مختلف قبل گردد.

کارت‌های "کلاس- مسئولیت- همکاری"

استفاده از کارت‌های (کلاس- مسئولیت- همکاری ≡ CRC ≡ Class, Responsibility, Collaboration) برای هدایت این فعالیت‌ها مفید است. هر کارتِ "کلاس- مسئولیت- همکاری"، یک کارت ۵×۳ اینچ فیش برداری است که برای هر کلاس از شیءها تهیه می‌شود. در این کارت نام کلاس، مسئولیت‌های آن کلاس (رفتار آن کلاس)، و کلاس‌های دیگری که این کلاس با آنها همکاری می‌کند قید می‌گردد. طراح می‌تواند این کارت‌ها را به‌صورت فضایی سازمان داده و محتوای آنها را بر اساس یادداشت‌های چرخه‌ی عمرش (که در مرحله‌ی قبل ذکر شد) تغییر دهد.

محتوای کارت‌های کلاس-مسئولیت- همکاری.

بازهم دید خارجی.

تأکید می‌کنیم که در این مرحله از طراحی نیز هنوز تمرکز ما روی دید خارجی از تجریدهای کلیدی و مکانیزم‌های موجود می‌باشد.

محصولات مرحله‌ی سوم

تکمیل مدل‌های منطقی.

شامل تکمیل مدل‌های منطقی طراحی است. در اینجا نمودارهای کلاس از مراحل قبل را با توجه به الگوی کشف شده و تصمیم‌های قابلیت رؤیت، پالایش می‌کنیم. همچنین جزئیات نمودارهای شیء (که مکانیزم‌های اصلی را مستند می‌کنند) را تکمیل می‌کنیم.برای ایجاد این نمودارها از علامت‌گذاری فصل پنجم استفاده می‌کنیم. ابتدا شیءهایی را که می‌دانیم با هم همکاری می‌کنند، را رسم می‌کنیم. سپس هر زوج از شیءها را در نظر گرفته و در موردِ رابطه‌ی بین آن‌دو سؤال می‌کنیم. اگر جواب مثبت باشد دو سؤال بعدی را مطرح می‌کنیم که: این دو شیء چگونه به هم مربوط می‌شوند و هر یک چه پیامی برای دیگری می‌فرستد؟ این کار معمولاً به تجدید نظر در مورد پروتکل کلاس‌های مربوطه منجر می‌شود. واین تجدید نظر، به نوبه‌ی خود موجب کشفِ الگوی دیگر می‌گردد.

محصولات دیگر.

در اینجا، ممکن است ایجاد نمودارهای اصلی مؤلفه را شروع کنیم. در این گام ممکن است به ایجاد و توسعه‌ی نمونه‌ها (prototypes) نیز ادامه دهیم. ممکن است نمونه‌های جدیدی را ایجاد کنیم و یا نمونه‌های قبل را پالایش کنیم.

گام چهارم.

۶- ۴ پیاده‌سازی کلاس‌ها و شیءها *(Implementing Classes and Objects)*

فعالیت‌ها: چهارمین گام الزاماً آخرین گام نیست. این گام شامل دو فعالیت است:

- تصمیم‌گیری‌های طراحی با توجه به نمایش کلاس‌ها و شیءهایی که ابداع کرده‌ایم؛
- تخصیص کلاس‌ها و شیءها به مؤلفه‌ها، و تخصیص برنامه‌ها به پردازنده‌ها.

دید داخلی.

اینجا اولین جا در طراحی است که به دید داخلی هر کلاس و مؤلفه می‌پردازیم، تا در مورد چگونگی پیاده‌سازی رفتارِ آن، تصمیم بگیریم.

به‌جز در مورد تجریدها و مکانیزم‌هایِ بسیار ابتدایی، معمولاً در این مرحله ناچار به بازگشت به مرحله‌ی اول می‌شویم، و این فرایند را دوباره در دید داخلیِ کلاس‌ها و مؤلفه‌های موجود اعمال می‌کنیم.

محصولات: در این گام از طراحی، ما باید واسطِ دقیقِ هر کلاس یا شیءِ مهم را (در سطحی از تجرید) مشخص کرده باشیم. لذا محصولات ما شامل پالایش ساختارکلاس، و بخصوص تکمیل قسمت پیاده‌سازی از الگوی کلاس می‌باشد. تکمیل قسمت‌های مشابه از نمودار مؤلفه، و در صورت لزوم، در نمودار فرایندهای سیستم.

۶- ۵ خلاصه‌ی فصل ششم از کتاب طراحی شیءگرا / فرایند طراحی شیءگرا

(The Process)

- فرایند طراحی شیءگرا نه بالا به پایین و نه پایین به بالا است؛ بلکه بهترین بیان برای طراحی شیءگرا، طراحی مرکّب دوره‌ای (round–trip gestalt design) می‌باشد، که ایجاد و توسعه‌ی سیستم به‌صورت اضافه شونده/ فزاینده (incremented) و تکرار شونده (iterative)، از طریق پالایش دیدگاه‌های متفاوت و در عین حال سازگار و هماهنگ (consistent) به تمامیّت سیستم، صورت می‌گیرد؛
- فرایند طراحی شیءگرا با کشف کلاس‌ها و شیءهای زمینه‌ی مسأله شروع می‌شود؛ و هنگامی پایان می‌یابد که ما دریابیم هیچ تجرید ابتدایی (primitive) و مکانیزم جدیدی وجود ندارد، یا هنگامی‌که کلاس‌ها و شیءهایی که تا حالا کشف کرده‌ایم بتوانند به‌وسیله‌ی ترکیب مؤلفه‌های نرم‌افزاری موجود پیاده‌سازی شوند؛
- اولین گام در فرایند طراحی شیءگرا شامل شناسایی و تعیین کلاس‌ها و شیءها در یک سطح داده شده از تجرید است؛ در اینجا مهمترین فعالیت‌ها عبارتند از کشف تجریدهای کلیدی و ابداع مکانیزم‌های مهم؛
- دومین گام شامل شناسایی و تعیین معانی (semantics) این کلاس‌ها و شیءهاست؛ کار مهم تولیدکننده در اینجا اینست که به‌عنوان یک بیگانه‌ی مستقل و مجزا، به هر یک از کلاس‌ها از نقطه نظر واسط (interface) نگاه کند؛
- سومین گام شامل شناسایی و تعیین رابطه‌ی بین این کلاس‌ها و شیءهاست؛ در اینجا با توجه به معانی ایستا و پویای تجریدهای کلیدی و مکانیزم‌های مهم، چگونگی تعامل چیزها با سیستم را برقرار و مشخص می‌کنیم؛
- چهارمین گام شامل پیاده‌سازی این کلاس‌ها و شیءهاست؛ کار مهم در اینجا شامل انتخاب یک نمایش (representation) برای هر یک از کلاس‌ها و شیءها، و تخصیص کلاس‌ها و شیءها به مؤلفه‌ها، و تخصیص برنامه‌ها به فرایندهاست؛ این گام الزاماً گام آخر نیست، برای تکمیل کردن، معمولاً لازم است که، تمام فرایند را، برای سطح پایین‌تری از تجرید، تکرار کنیم.

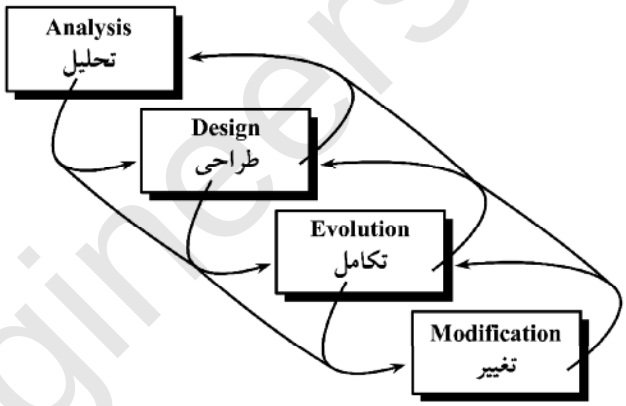
فصل ۷

نکات علمی (Pragmatics)

در این فصل نکات علمی در طراحی شیء‌گرا، نقش این روش در چرخه‌ی عمر نرم‌افزار و تأثیر آن در مدیریت، مورد بررسی قرار می‌گیرد.

۷- ۱ طراحی شیء‌گرا در چرخه‌ی عمر نرم‌افزار

(Object-Oriented Design in the Life Cycle)



شکل ۷- ۱: طراحی شیء‌گرا در چرخه‌ی تولید نرم‌افزار

تحلیل *(Analysis)*

فعالیت‌ها (the activates of analysis): حَدِّ بین تحلیل و طراحی می‌تواند گاهی مشکک (Fuzzy) باشد. به‌عنوان مثال تجریدهای کلیدی از زمینه‌ی مسأله را می‌توان هم به‌عنوان قسمتی از تحلیل و یا قسمتی از طراحی دانست. ولی هدف‌های تحلیل و طراحی کاملاً متفاوت هستند. در تحلیل، ما (با شناسایی کلاس‌ها و شیء‌هایی که فرهنگ زمینه‌ی مسأله را تشکیل می‌دهند) دنبال مدلی از دنیای مسأله هستیم؛ در طراحی شیء‌گرا، ما تجریدها و مکانیزم‌هایی را ابداع می‌کنیم که رفتار مورد نیاز این مدل را فراهم کنند. به عبارت دیگر تحلیل، رفتارِ مطلوبِ سیستمی که باید بسازیم را به ما می‌گوید، در حالی‌که طراحی، منشور (blueprints) پیاده‌سازی آن سیستم را به‌وجود می‌آورد (تعریف تحلیل در فصل دوم آمده است).

در تحلیل.
در طراحی.
به عبارت دیگر...

روش‌های تحلیل (analysis methods): انواع مختلفی از روش‌های تحلیل وجود دارد، که بعضی از آنها برای این‌که مقدمِ (front end) طراحی شیء‌گرا قرار گیرند، مناسب هستند. تحلیل ساخت‌یافته مقدم جاذبی برای طراحی شیء‌گراست؛

جاذبه‌ی تحلیل ساخت‌یافته

عمدتاً به‌خاطر این‌که متداول‌ترین روش، طراحیِ جاری است (1991) و افراد زیادی در این رابطه آموزش دیده‌اند، با این‌حال تحلیل ساخت‌یافته، مقدم بهینه برای طراحی شیءگرا نیست. دانستن این‌که چه موقع یک نمودار جریان داده‌ها نمایش مدل مهمی از مسأله را متوقف و در عوض یک تصمیم طراحی را مطرح می‌کند، مشکل است. به‌علاوه گاهی تحلیل ساخت‌یافته می‌تواند دید الگوریتم‌گرا (algorithm oriented) به مسأله را برای همیشه باقی گذارد، که این طراحی شیءگرا را بسیار مشکل‌تر خواهد کرد.

تحلیل ساخت‌یافته بهینه نیست

فرایند تحلیل شیءگرا، به طراحی شیءگرا کاملاً نزدیک است. به این معنی که محصول تحلیل شیءگرا تقریباً مستقیماً در ابتدای فرایند طراحی شیءگرا می‌تواند مورد استفاده قرار گیرد.

تناسب تحلیل شیءگرا

روش‌های تحلیل دیگری مثل تکنیک‌های تحلیل و طراحی ساخت‌یافته (Structured Analysis and Design Techniques ≡ SADT) [16]، نیز به‌عنوان مقدم طراحی شیءگرا مورد استفاده قرار گرفته‌اند.

طراحی *(Design)*

شروع فرایند طراحی: هرگاه یک مدل رسمی یا غیر رسمی (و احتمالاً ناقص) از مسأله وجود داشته باشد، طراحی شیءگرا می‌تواند شروع شود. ایده‌ی مقداری تحلیل، مقداری طراحی (analysis a Little design a little) می‌تواند مورد توجه باشد (فرایند طراحی در فصل ششم مفصل‌تر بحث شده است).

خاتمه‌ی فرایند طراحی: تعیین این نکته هم مشکل است. قاعده‌ی ره‌یافتی که ما استفاده می‌کنیم این است که موقعی فرد می‌تواند طراحی را متوقف کند که تجریدهای کلیدی آنقدر ساده هستند که به تفکیک بیشتر نیاز ندارند، ولی می‌توانند از مؤلفه‌هایِ نرم‌افزار قابلِ استفاده‌ی مجدد ترکیب گردند.

تکامل *(Evolution)*

فعالیت‌های تکامل (the activities of evolution): تکامل یک سیستم در چرخه‌ی ایجاد توسعه‌ی نرم‌افزار به‌صورت شیءگرا، ترکیبی از جنبه‌های سنتیِ برنامه‌نویسی (coding)، آزمایش (testing) و مجتمع‌سازی (integration) می‌باشد. لذا مجتمع‌سازی یک‌باره (big-bang) در رابطه با طراحی شیءگرا وجود ندارد. در عوض فرایند ایجاد و توسعه‌ی سیستم، نتیجه‌ی تولید اضافه شونده‌ی دنباله‌ای از نمونه‌ها است، که در نهایت به‌صورت پیاده‌سازی نهایی ظاهر می‌شود. این روند افزایش یابنده‌ی ایجاد و توسعه امتیازاتی دربر دارد:

برنامه‌سازی، آزمایش ومجتمع‌سازی سنتی.

مزایای روند افزایش یابنده و تکرارشونده

مزایای روند افزایش یابنده و تکرار ونده.

- بازخورد مهم برای کاربران موقعی فراهم می‌شود که بیشترین نیاز به آن هست، بیشترین استفاده را دارد، و بیشترین معنی را دارد؛
- کاربران می‌توانند گونه‌های مختلفی از اسکلت سیستم را ببینند، و لذا به آرامی از سیستم قدیمی خود به سیستم جدیدشان انتقال یابند؛

انتقال آرام به سیستم جدید.

- تأخیر در زمان‌بندی پروژه، بعید است موجب قطع پروژه گردد؛

به موقع بودن.

- واسطه‌های اصلی سیستم زودتر و بیشتر آزمایش می‌شوند؛

روند مناسب آزمایش.

- آزمایش منابع، عادلانه‌تر توزیع می‌گردد؛
- پیاده‌کنندگان می‌توانند نتایج زودرس از سیستم در حال کار را ببینند، که موجب بالا رفتن روحیه‌ی آنها می‌شود؛

روحیه‌ی کاری.

- در صورت کوتاهی زمان، برنامه‌نویسی و آزمایش می‌توانند قبل از اتمام طراحی، شروع شوند [19]؛

کوتاه کردن زمان.

- طرح‌های متفاوت و رسیدن به تصمیم‌های بهتر می‌تواند مطرح باشد.

طرح‌های متفاوت.

ایجاد و توسعه‌ی سیستم عمدتاً تلاشی برای جواب‌گویی به تعدادی از محدودیت‌های رقیب (از جمله عملکرد (functionality)، زمان و فضا) است. شخص همیشه محدود به بزرگترین تنگناهاست. با پیاده‌سازی تکاملی، ما می‌توانیم میزان اهمیت هر یک از محدودیت‌ها را درک کرده و لذا بهتر می‌توانیم سیستم را در طول زمان تنظیم کنیم.

انواع تغییرات تکاملی *(Kinds of Evolutionary Charles)* :

در عمل انواع تغییرات زیر در طرحِ کلّیِ تکامل تدریجی سیستم مطرح می‌شود:

- اضافه کردن یک کلاس جدید؛
- تغییر پیاده‌سازی یک کلاس؛
- تغییر نمایش یک کلاس؛
- سازماندهی مجدد ساختار کلاس؛
- تغییر واسط یک کلاس.

هر یک از انواع تغییرات فوق دارای دلایل و همچنین هزینه‌های خاص خود است.

تغییر *(Modification)*

بعد از پیاده‌سازی.

با بلوغ یک سیستم نرم‌افزاری پیاده شده، نکاتی مشاهده می‌شود:

قانون تداوم تغییر.

- برنامه‌ای که در محیط واقعی به‌کار رفته است الزاماً یا باید تغییر کند و یا این‌که از مفید بودن آن در محیطش کاسته شود قانون تداوم تغییر (the law of continuing elang)؛

قانون ازدیاد پیچیدگی.

- وقتی یک برنامه‌ی متعامل تغییر می‌کند، ساختارش پیچیده می‌شود، مگر اینکه تلاش‌های خاصی برای جلوگیری از این پدیده صورت گیرد قانون ازدیاد پیچیدگی (the law of increasing complexity) [22].

تفاوت اندک مراحل تغییر و تکامل.

ما دریافته‌ایم که فعالیت‌هایِ مرحله‌ی تغییر سیستم با فعالیت‌های زمان تکامل سیستم تفاوت‌های اندکی دارد. به‌ویژه اگر در طراحی شیءگرا، خوب عمل کرده باشیم، اضافه کردن قابلیت‌های جدید (new functionality) و یا تغییر بعضی از رفتارهای موجود بطور عادی انجام می‌شود. و بخصوص این راحتی را در مورد سیستم‌های بزرگ نیز دریافته‌ام.

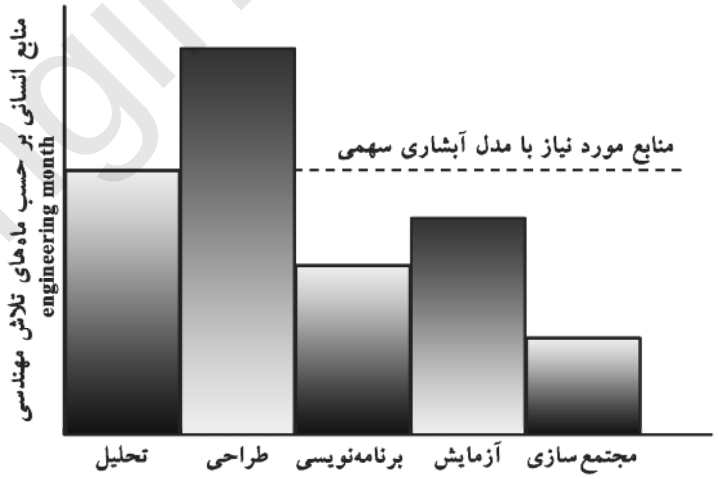
۷- ۲ مدیریت پروژه *(Managing a Project)*

زمان کمتر.

تخصیص منابع (resource allocation): ما دریافته‌ایم که به‌کارگیری طراحی شیءگرا موجب کوتاه شدن زمان ایجاد و توسعه‌ی development سیستم، موجب ارتقاء کیفی نرم‌افزار، یعنی نزدیکی بیشترِ آن با نیازهای واقعی می‌گردد. یکی از

منابع کمتر.

جنبه‌های اعجاب‌آور مدیریت پروژه‌ها با استفاده از طراحی شیءگرا اینست که کاهش عمومی منابع مورد نیاز و تغییر در زمانِ کنار رفتن نیروها محسوس است. شکل۷-۲ بیانگر این پدیده است.



شکل ۷-۲: منابع انسانی برحسب ماه‌های تلاش مهندسی

با طراحی شیءگرا، منابع باید به‌گونه‌ی دیگری تخصیص یابد. منابع در بخش تحلیل تفاوت چندانی در مقایسه با مدل سنتی آبشاری ندارند. اما برای طراحی، منابع بیشتری لازم است چراکه کار بیشتری انجام می‌گیرد. این ازدیاد به معنی بیشتر

کردن افراد نیست؛ معمولاً به معنی کار بیشتر چند نیروی خوب برای مدت طولانی‌تر است. برای برنامه‌نویسی نیروی کمتری لازم است به‌خاطر این‌که کار باقیمانده محدود است. آزمایش هم منابع کمتری احتیاج دارد. بیشتر به‌دلیل این‌که اضافه کردن توانایی جدید یک کلاس، عمدتاً با تغییر کلاسی که رفتار صمیمی دارد انجام می‌شود. بلاخره منابع مورد نیاز در قسمت مجتمع‌سازی خیلی کمتر است چراکه مجتمع‌سازی به تدریج و به‌صورت افزایش یابنده در طول چرخه‌ی عمرِ ایجاد و توسعه انجام می‌شود. در مجموع منابع مورد نیاز برای روش شیءگرا معمولاً کمتر یا مساوی منابع مورد نیاز با دیدگاه‌های سنتی، و کیفیت محصول خیلی بهتر است.

۷- ۳ مهارت‌های تصمیم‌های ایجاد و توسعه *(Development Team Skills)*

با دیدگاه شیءگرا در نقش تصمیم‌های تولیدکننده‌ی نرم‌افزار نیز تغییراتی می‌بینیم. چهار نوع تولیدکننده مورد نیاز است، به ترتیب ارشدیت:

- معماران سیستم (system architects)؛
- طراحان کلاس (class designer)؛
- پیاده‌کنندگان کلاس (class Implementers)؛
- برنامه‌نویسان کاربردی (application programmers).

تیم‌های کوچک.

دید شیءگرا امکان استفاده از تیم‌های کوچکتر را میسّر می‌سازد. ایجاد بیش از یک میلیون خط برنامه‌ی کیفی توسط ۳۰ تا ۴۰ نفر در طول یک‌سال غیر عادی نیست. با توجه به استفاده از ابزار در دیدگاه شیءگرا؛ وجود ابزارمند tool smith و کتابدار (librarian) نیز در تیم‌ها کمک‌کننده / ضروری است.

حمایت ابزار *(Tool Support)*

انواع ابزار مفید.

ما حداقل شش نوع ابزار مختلف که در طراحی شیءگرا مفید هستند را شناسایی و تعیین کرده‌ایم:

- سیستم متکی بر گرافیک برای نشان دادن علامت‌گذاری‌های تعریف شده (فصل۵) در طراحی شیءگرا. این ابزار در تمام چرخه‌ی عمر روند ایجاد و توسعه‌ی نرم افزار با دید شیءگرا می‌تواند کمک‌کننده باشد؛
- مرورکننده (browser) که ساختار کلاس و معماری سیستم را می‌داند و در مواردی مثل سلسله‌مراتب و تعریف کلاس‌ها، پیدا کردن یک کلاس می‌تواند کمک کند. مرورکننده‌ها ابزار مهمی در طراحی شیءگرا هستند؛
- کامپایلر افزایش یابنده (incremented compiler)؛
- اشکال‌زُدا (debuggers)؛
- ابزار مدیریت پیکربندی و کنترل گونه‌ها (configuration management & version control tools) بخصوص برای پروژه‌های بزرگ؛
- کتاب‌دار کلاس‌ها (class librarian).

با توجه به ابزار فوق‌الذکر، یک ابزارمند (tool smith) و کتاب‌دار (librarian) در تیم‌های تولید و ایجاد نرم‌افزار با دید شیءگرا مفید / ضروری خواهد بود.

۷- ۴ مزایا و خطرات طراحی شیءگرا

(The Benefits & Risks of Object-Oriented Design)

مزایا.

مزایا:

- برخورد با مشکل ایجاد و توسعه‌ی سیستم‌های بزرگ؛
- ساخت سیستم‌های پیچیده خوش‌ساخت (well structure) (اصولاً مزایای مدل شیءگرا)؛
- استفاده از امکانات زبان‌های برنامه‌سازی شیءگرا یا مبتنی بر شیء؛
- تشویق استفاده‌ی مجدد از نرم‌افزار (reuse)؛

- محصول منعطف‌تر در مقابل تغییرات؛
- کاهش خطرات تولید؛
- استفاده از توان ادراکی انسان؛
- امکان تقلیل زمان تولید و اندازه‌ی کد (بر اساس تجربه‌ها).

خطرات

دو زمینه‌ی خطر باید مورد توجه قرار گیرد:

۱. کارایی (performance)؛

۲. هزینه‌های شروع (start-up cost).

خطرات کارایی:

- ارسال پیام در مقابل فراخوانی عادی کندتر است؛
- به‌خاطر تجریدها؛
- به‌خاطر بارِ کلاس‌ها (یک کلاس ممکن است زیرکلاس‌های بسیار زیادی داشته باشد)؛
- به‌خاطر رفتار صفحه‌بندی (paging behavior) کاربردهای در حال اجرا؛
- به‌خاطر تخصیص پویا و تخریب شیء‌ها.

بعضی موارد مورد تردید است. مثل این‌که نشان داده شده برنامه‌های C++ از برنامه معادل C سریع‌تر است[31].

خطرات هزینه‌های شروع:

- نیاز به ابزار؛
- عدم وجود نرم‌افزارهای قبلی که بتوانند مورد استفاده‌ی مجدد قرار گیرند (در شروع)؛
- نیاز به آموزش نیروها (با یادگیری زبان شیء‌گرا مسأله حل نمی‌شود. بلکه تغییر در تفکر لازم است).

۷- ۵ انتقال به طراحی شیء‌گرا *(The Transition to Object-Oriented Design)*

تغییر تفکر لازم است. اقدامات زیر را برای رسیدن به این تفکر پیشنهاد می‌کنیم:

- برگزاری آموزش‌های رسمی هم برای تولیدکنندگان و هم برای مدیران روی عناصر مدل شیء؛
- استفاده از طراحی شیء‌گرا ابتدا در پروژه‌هایی که ریسک خطرات کمی دارند؛ اجازه دادن، به موارد خطای اعضای تیم؛ استفاده از این اعضاء تیم‌ها در پروژه‌های دیگر برای هدایت پروژه با دید شیء‌گرا؛
- قرار دادن تولیدکنندگان و مدیران در معرضِ سیستم‌های شیء‌گرای خوش‌ساخت (well structured).

۷- ۶ جایگاه موجودیت‌های انسانی از محیط عملیاتی در سیستم کامپیوتری مورد نظر

استفاده از مدل شیء به‌عنوان یک مدل مهندسی نرم‌افزار (برای ایجاد، توسعه و نگهداری نرم‌افزار) همواره با این حقیقت روبرو است که (بخصوص در مراحل تحلیل و طراحی) عوامل انسانی متعددی در کل سیستم نقش دارند. معمولاً نیروهای انسانی که هر یک، یک شیء از محیط عملیاتی محسوب می‌شوند را می‌توان در کلاس‌هایی طبقه‌بندی کرد.

در مثال کتابخانه. کلاس‌های مراجعین کتاب‌داران و پردازش‌کنندگان کتاب، مطرح می‌شوند.

سؤال این‌جاست که این شیء‌ها و کلاس‌ها چه جایگاهی در تحلیل، طراحی و بخصوص در پیاده‌سازی سیستم دارند؟

- در تحلیل قطعاً نقش دارند. (غیر از کمکی که به ما برای فهم زمینه‌ی عملیاتی می‌کنند) آنها مهمترین پردازشگران محیط اجرایی هستند. یک سیستم کامپیوتری قرار است زیر مجموعه‌ای از فعالیت‌های آنها را انجام دهد. در حالی‌که جایگاه تشکیلاتی آن شخص در کل محیط عملیاتی معمولاً همچنان پابرجاست، و احتمالاً با تغییرات کمی روبروست. پس در تحلیل‌ها قطعاً باید مورد توجه باشند؛

  از نظر وظایف و مسئولیت‌ها؛

از نظر روش انجام وظایف؛
از نظر واحدها یا شیءهای دیگر که با آنها در تراکنش هستند.

- در طراحی نیز انسان‌ها و کلاس‌هایی که تشکیل می‌دهند، جایگاه ویژه‌ای دارند:
از نظر منظور کردن شیءهایی که بتوانند وظایف مورد نظر از نیروی انسانی را به‌عهده بگیرند؛
از نظر ارتباط با شیءها و کلاس‌هایی از برنامه که جایگاه عملکرد نیروهای انسانی دیگر را در کل سیستم می‌گیرند؛
از نظر ایجاد ابزار (نرم‌افزار) لازم برای تراکنش با انسان.
چه انسانی که کامپیوتر زیرمجموعه‌ای از وظایف او را به‌عهده گرفته است.
چه انسان (هایی) که خود، مراجعه‌کننده‌ی انسان وظیفه‌مند اول بوده‌اند.

شکل ۷-۳: یک انسان در محیط عملیاتی به‌عنوان client

شکل ۷- ۴:یک انسان در محیط عملیاتی به‌عنوان server

شکل ۷-۵: جایگاه موجودیت‌های انسانی از محیط عملیاتی در سیستم کامپیوتری

به‌جای هر انسانی که نقش اجرایی دارد، مؤلفه‌ای از سیستم باید وظایف مربوطه را به‌عهده گیرد.
برای هر انسانی که نقش مراجعه‌کننده یا مشتری را دارد، مؤلفه‌ای (هایی) از سیستم باید واسط لازم با او را به‌وجود آورد. به‌عبارت دیگر واسط‌هایی بین کاربر و واحد(های) مسؤل انجام وظایف باشند.

۷- ۷ خلاصه‌ی فصل هفتم از کتاب طراحی شیءگرا / نکات علمی در طراحی شیءگرا *(Pragmatism)*

- چرخه‌ی زندگی ایجاد و توسعه‌ی نرم‌افزار با استفاده از طراحی شیءگرا تکیه بر ایجاد سیستم به‌صورت اضافه شونده و تکرار شونده دارد؛
- تحلیل ساخت‌یافته، آغاز جالبی برای طراحی شیءگراست؛ تحلیل شیءگرا مقدّمِ (front end) مؤثرتری است؛
- هرگاه که یک مدل رسمی یا غیر رسمی (احتمالاً ناقص) از مسأله وجود داشته باشد، طراحی می‌تواند شروع شود؛ و زمانی طراحی متوقف می‌شود که ترکیب، به‌جای تفکیک حاکم باشد؛
- در طراحی شیءگرا هرگز یک حادثه‌ی بزرگِ (big-bang) یک‌باره برای مجتمع‌سازی (integration) سیستم وجود ندارد؛

- در طی تکامل سیستمی که با استفاده از طراحی شیءگرا به‌دست آمده است، چندین نوع از تغییرات روی طرح قابل پیش‌بینی است؛ هر یک از این تغییرات متضمّن هزینه‌های متفاوتی هستند؛
- استفاده از طراحی شیءگرا روی مدیریت نیرو، معیارهای ارزیابی (milestones) و محصول، مدیریت گونه‌های مختلف محصول (release management)، اطمینان از کیفیت و ابزار حمایتی تأثیر می‌گذارد؛
- به‌کارگیری طراحی شیءگرا دارای مزایای بسیار زیاد و در عین حال با ریسک‌هایی همراه است؛ تجربه نشان داده است که این مزایا بسیار فراتر از ریسک‌ها می‌باشند؛
- در یک سازمان، تغییر وضع به استفاده از مدل شیءگرا، محتاج تغییر در اندیشه است؛ یادگیری یک زبان شیءگرا یا مبتنی بر شیء کافی نیست.