پیوست ۲ مفاهیم شیءگرایی

دیدگاه **شیءگرا** چیست؟ چرا یک روش، شیءگرا در نظر گرفته می‌شود؟ شیء چیست؟ با فراگیر شدن مفاهیم شیءگرایی طی دو دهه‌ی ۱۹۸۰ و ۱۹۹۰، آرا و عقاید متفاوت و متعددی درباره‌ی پاسخ‌های درست به این پرسش‌ها مطرح شد، ولی امروزه، دیدگاهی یکپارچه در خصوص مفاهیم شیءگرایی وجود دارد. این پیوست به منظور ارائه‌ی دیدگاهی مختصر درباره‌ی این مبحث و معرفی مفاهیم و اصطلاح‌های پایه طراحی شده است.

برای درک دیدگاه شیءگرا، مثالی از یک شیء واقعی- چیزی که هم اکنون روی آن نشسته‌اید- یعنی صندلی را در نظر بگیرید. **Chair** زیرکلاسی از یک کلاس بسیار بزرگ‌تر است که آن را **PieceOfFurniture** (اثاثیه) می‌نامیم. تک تک صندلی‌ها اعضای کلاس Chair هستند (که معمولاً نمونه‌های کلاس خوانده می‌شوند). مجموعه‌ای از خصیصه‌های کلی را می‌توان با هر شیء از کلاس **PieceOfFurniture** مرتبط ساخت. برای مثال، همه‌ی اثاثیه‌ها دارای قیمت، ابعاد، وزن، مکان، رنگ و بسیاری خصیصه‌های ممکن هستند. هرگاه درباره‌ی یک میز، صندلی، کاناپه یا کمد سخن به میان می‌آید، این خصیصه‌ها کاربرد دارند. از آن‌جا که **Chair** عضوی از **PieceOfFurniture** است، **Chair** همه خصیصه‌های تعریف شده برای کلاس را به ارث می‌برد.

ما تلاش کردیم که تعریفی روایی از کلاس را با توصیف خصیصه‌های آن ارائه دهیم، ولی چیزی کم است. هر شیء در کلاس PieceOfFurniture را می‌توان به شیوه‌های گوناگون دستکاری کرد. می‌توان آن را خرید و فروخت، اصلاح فیزیکی کرد (مثلاً پایه‌ها را اره کرد تا کوتاه‌تر شوند یا به رنگ ارغوانی درآورد)، یا از مکانی به مکان دیگر انتقال داد. هر کدام از این اَعمال (سرویس‌ها یا متدها) یک یا چند خصیصه از شیء را اصلاح می‌کنند. برای مثال، اگر خصیصه‌ی مکان، یک قلم‌داده‌ی مرکب باشد که به صورت زیر تعریف می‌شود:

```
Location=building+floor+room
```

در آن صورت عملی به نام ()move یک یا چند قلم‌داده‌ای (building، floor، یا room) را که خصیصه‌ی location را تشکیل می‌دهند، اصلاح خواهد کرد. برای این منظور، ()move باید درباره‌ی این اقلام داده‌ای "آگاهی" داشته باشد. عمل ()move برای یک صندلی یا میز قابل استفاده است مادامی که هر دو

آن‌ها نمونه‌هایی از کلاس **PieceOfFurniture** باشند– اَعمال معتبر برای ***buy()***، ***sell()***، ***weigh()***، به عنوان بخشی از تعریف کلاس مشخص شده‌اند و همه‌ی نمونه‌های کلاس آن‌ها را به ارث می‌برند.

کلاس **Chair** (و همه‌ی اشیا به طور کلی) داده‌ها (مقادیر خصیصه‌هایی که صندلی را تعریف می‌کنند)، اَعمال (کنش‌هایی که برای تغییر دادن خصیصه‌های صندلی به کار می‌روند)، اشیای دیگر، ثابت‌ها (مقادیر تعیین شده) و سایر اطلاعات مرتبط را محصور می‌کنند. محصورسازی[1] به این معناست که کلیه‌ی این اطلاعات تحت یک نام واحد بسته‌بندی می‌شوند و می‌توان آن‌ها را به عنوان یک مولفه از برنامه دوباره استفاده کرد.

اکنون که چند مفهوم پایه را معرفی کردیم، تعریفی رسمی‌تر برای شیءگرایی بهتر معنا پیدا خواهد کرد. کود و یوردون [Coa91] این اصطلاح را چنین تعریف می‌کنند:

ارتباطات + وراثت + طبقه‌بندی + اشیا = شیءگرایی

سه تا از این مفاهیم را قبلاً معرفی کردیم. درباره‌ی ارتباطات نیز بعداً در همین پیوست بحث خواهیم کرد.

کلاس‌ها و اشیا

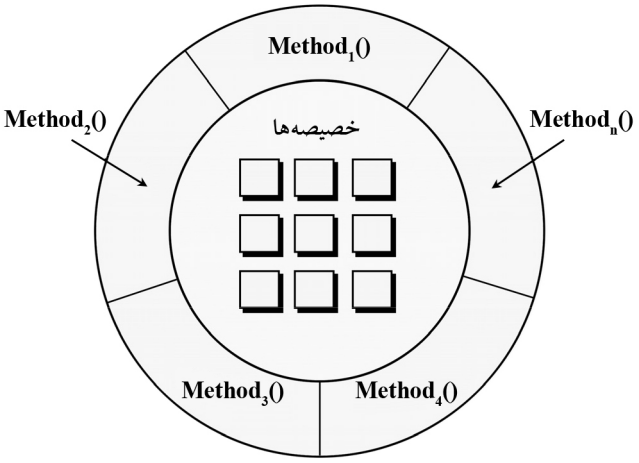
کلاس، مفهومی شیءگراست که داده‌ها و انتزاع‌های رویه‌ای لازم برای توصیف محتوا و رفتار یک نهاد حقیقی را محصورسازی می‌کند. انتزاع‌های داده‌ای که کلاس را توصیف می‌کنند توسط محتوا "دیواری" از انتزاع‌های رویه‌ای محصور می‌شوند [Tay90] (شکل پ ۲–۱) که به نحوی قادر به دستکاری داده‌ها هستند. در یک کلاس که خوب طراحی شده باشد، تنها راه برای رسیدن به خصیصه‌ها (و عمل کردن روی آن‌ها) عبور از یکی از متدهایی است که "دیوار" نشان داده شده در شکل را تشکیل می‌دهند. بنابراین، کلاس، داده‌ها (درون دیوار) و پردازشی که داده‌ها را دستکاری می‌کند (یعنی متدهای تشکیل‌دهنده‌ی دیوار) را بسته‌بندی می‌کند. بدین ترتیب داده‌ها از دید خارجی پنهان می‌مانند (فصل ۱۲) و تأثیر اثرات جانبی ناشی از تغییرات کاهش می‌یابد. از آن‌جا که متدها تمایل به دستکاری تعداد محدودی از خصیصه‌ها دارند، انسجام آن‌ها بهبود می‌یابد و چون ارتباطات تنها از طریق متدهایی رخ می‌دهد که "دیوار" را تشکیل می‌دهند، کلاس با قدرت کمتری با عناصر دیگر سیستم متصل می‌شود[2].

به بیان دیگر، کلاس، توصیفی تعمیم‌یافته (مثلاً یک قالب یا نقشه‌ی پلان) است که مجموعه‌ای از اشیای مشابه را توصیف می‌کند. طبق تعریف، اشیا، نمونه‌هایی از یک کلاس مشخص هستند و خصیصه‌ها و

---

1. Encapsulation

۲. به هر حال، لازم به ذکر است که اتصال می‌تواند در سیستم‌های شیءگرا به مشکلی جدی تبدیل شود. این مشکل هنگامی پیش می‌آید که کلاس‌هایی از بخش‌های گوناگون سیستم به عنوان انواع داده برای خصیصه‌ها و شناسه‌هایی برای متدها به‌کار روند. گرچه دستیابی به اشیا ممکن است از طریق فراخوانی رویه‌ها نباشد، این بدان معنا نخواهد بود که اتصال الزاماً پایین است بلکه فقط پایین‌تر از دستیابی مستقیم به درون اشیا خواهد بود.



شکل پ ۲-۱ طرحی از یک کلاس

اَعمال در دسترس برای دستکاری این خصیصه‌ها را به ارث می‌برند. اَبَرکلاس (که غالباً *کلاس پایه* نامیده می‌شود) تعمیمی است از یک مجموعه کلاس که با آن در ارتباط هستند. زیرکلاس، نمونه‌ی تخصصی از یک اَبَرکلاس است. برای مثال، اَبَرکلاس **MotorVehicle** تعمیمی از کلاس‌های **Truck**، **SUV**، **Automobile** و **Van** است. زیرکلاس **Automobile** همه‌ی خصیصه‌های **MotorVehicle** را به ارث می‌برد، ولی علاوه بر آن، شامل خصیصه‌های اضافی است که تنها مختص خودروهاست.

از این تعاریف چنین برمی‌آید که سلسله‌مراتبی از کلاس‌ها وجود دارد که در آن، خصیصه‌ها و اَعمال اَبَرکلاس برای زیرکلاس‌هایی به ارث گذاشته می‌شود که هر کدام ممکن است خصیصه‌ها و متدهای "خصوصی" اضافی به آن بیفزایید. برای مثال، اَعمال ()sitOn و ()turn ممکن است خاص زیرکلاس **Chair** باشد.

خصیصه‌ها

دانستید که خصیصه‌ها به کلاس‌ها متصل هستند و به نحوی کلاس را توصیف می‌کنند. یک خصیصه ممکن است مقدار تعریف شده توسط دامنه‌ای معین را به خود بگیرد. در اکثر موارد، دامنه تنها مجموعه‌ای از مقادیر مشخص است. برای مثال، فرض کنید که کلاس **Automobile** دارای خصیصه‌ی color است. دامنه‌ی مقادیر مربوط به خصیصه‌ی color عبارت است از {white, black, silver, gray, blue, red, yellow, green}. این دامنه در وضعیت‌های پیچیده‌تر می‌تواند خود یک کلاس باشد. با ادامه‌ی این مثال، کلاس **Automobile** دارای خصیصه‌ی powerTrain است که خودش یک کلاس است. کلاس **PowerTrain** حاوی خصیصه‌هایی است که موتور و سیستم انتقال نیروی خودرو را توصیف می‌کنند.

ویژگی‌ها[1] (مقادیر دامنه) را می‌توان با نسبت دادن یک مقدار پیش‌فرض (ویژگی) به یک خصیصه تکمیل کرد. برای مثال، خصیصه‌ی color دارای پیش‌فرض **white** است. همچنین ممکن است در ربط دادن

1. Feature

احتمالی به یک ویژگی خاص مفید واقع شود؛ برای این منظور، یک جفت {احتمال، مقدار} نسبت داده می‌شود. خصیصه‌ی color را برای خودرو در نظر بگیرید. در برخی کاربردها (مثل برنامه‌ریزی برای تولید) ممکن است نسبت دادن احتمالی به هر کدام از رنگ‌ها ضرورت یابد (مثلاً احتمال سفید یا سیاه بودن رنگ خودرو بالاتر باشد).

اَعمال، متدها و سرویس‌ها

یک شیء، داده (که به صورت مجموعه‌ای از خصیصه‌ها نمایش داده می‌شود) و الگوریتم‌هایی را که داده‌ها را پردازش می‌کنند، محصورسازی می‌کند. این الگوریتم‌ها را عمل، متد یا سرویس می‌نامند[1] و می‌توان آن‌ها را به عنوان مولفه‌های پردازشی در نظر گرفت.

هر کدام از اَعمال محصورسازی‌شده توسط یک شیء یکی از رفتارهای شیء را به نمایش می‌گذارد. برای مثال، عمل GetColor() برای شیء **Automobile** رنگ ذخیره شده در خصیصه‌ی color را استخراج می‌کند. وجود این عمل بدان معناست که کلاس **Automobile** طوری طراحی شده است که یک محرک (ما این محرک را پیام می‌نامیم) دریافت کند که رنگ نمونه‌ی خاصی از یک کلاس را دریافت کند. هر گاه که شیءای یک محرک دریافت کند، رفتاری را آغاز می‌کند. این کار می‌تواند به سادگی بازیابی رنگ خودرو باشد یا به پیچیدگی شروع زنجیره‌ای از محرک‌ها که در میان انواع اشیای گوناگون تبادل می‌شوند. در مورد دوم، مثالی را در نظر بگیرید که در آن محرک اولیه‌ای که توسط **Object1** دریافت می‌شود، به تولید دو محرک دیگر می‌انجامد که **Object2** و **Object3** ارسال می‌شوند. عمل محصورسازی شده در اشیای دوم و سوم روی این محرک‌ها عمل کرده اطلاعات مورد نیاز شیء اول را فراهم می‌سازند. **Object1** سپس از اطلاعات بازگردانده شده استفاده می‌کند تا رفتار درخواست شده توسط محرک نخست را از خود به نمایش بگذارد.

تحلیل شیءگرا و مفاهیم طراحی

در مدل‌سازی نیازمندی‌ها (که مدل‌سازی تحلیل نیز خوانده می‌شود) آن‌چه که در وهله‌ی نخست، کانون توجه قرار می‌گیرد، کلاس‌هایی است که مستقیماً از صورت مسأله استخراج می‌شود. این کلاس‌های نهادی معمولاً نشانگر چیزهایی هستند که قرار است در یک پایگاه داده نگهداری شوند و در سراسر طول عمر اپلیکیشن ماندگار خواهند ماند (مگر این‌که به طور مشخص حذف شوند).

طراحی، مجموعه کلاس‌های موجودیت را پالایش می‌کند و توسعه می‌دهد. کلاس‌های مرزی و کنترلگر طی طراحی، توسعه داده و/یا پالایش می‌شوند. کلاس‌های مرزی، واسطی (مثلاً صفحه‌ی تعامل یا گزارش‌های چاپی) را ایجاد می‌کنند که کاربر می‌بیند و به هنگام استفاده از نرم‌افزار با آن در تعامل است. کلاس‌های مرزی طوری طراحی می‌شوند که مسئولیت شیوه‌ی مدیریت ارائه‌ی اشیای موجودیت به کاربران برعهده‌ی آن‌ها باشد.

---

۱. در حیطه‌ی این بحث، از اصطلاح عمل استفاده خواهیم کرد ولی واژه‌های متد و سرویس نیز به همین اندازه رایج هستند.

کلاس‌های کنترلگر طوری طراحی می‌شوند که (۱) ایجاد یا به‌هنگام‌سازی اشیای موجودیت، (۲) نمونه‌سازی از اشیای مرزی با کسب اطلاعات از اشیا، (۳) برقراری ارتباطات پیچیده میان مجموعه‌های اشیا و (۴) معتبرسازی داده‌های تبادل شده میان اشیا و میان کاربر و برنامه مدیریت کنند.

مفاهیم بحث شده در پاراگراف‌های زیر می‌توانند در کار تحلیل و طراحی مفید واقع شوند:

وراثت. وراثت یکی از وجوه تمایز کلیدی میان سیستم‌های سنتی و شیءگراست. زیرکلاس **Y** همه‌ی خصیصه‌ها و اَعمال اَبَرکلاس **X** خودش را به ارث می‌برد. این بدان معناست که همه‌ی ساختارهای داده‌ای و الگوریتم‌هایی که در آغاز برای **X** طراحی و پیاده‌سازی شده بوده‌اند، بلافاصله برای **Y** نیز در دسترس قرار دارند- نیاز به هیچ کار بیشتری نیست و استفاده دوباره به طور مستقیم امکان‌پذیر است. هرگونه تغییر در خصیصه‌ها یا اَعمال موجود در یک اَبَرکلاس بلافاصله برای همه‌ی زیرکلاس‌های آن به ارث گذاشته می‌شود. بنابراین، سلسله‌مراتب کلاس‌ها به راهکاری تبدیل می‌شود که از طریق آن می‌توان تغییرات را (در سطوح عالی) بلافاصله در سراسر سیستم پراکنده ساخت.

شایان ذکر است که در هر سطح از سلسله‌مراتب کلاس‌ها، خصیصه‌ها و اَعمال جدیدی را می‌توان به خصیصه‌ها و اَعمال ارث برده شده از سطوح بالاتر اضافه کرد. در واقع، کلاس جدیدی که قرار است ایجاد شود، هر چه که باشد، چند گزینه فرا روی شماست:

- کلاس را می‌توان از ابتدا طراحی کرد و ساخت. یعنی، از وراثت استفاده نکرد.
- سلسله‌مراتب کلاس‌ها را جستجو کرد تا معلوم شود که آیا کلاسی در سطوح بالاتر حاوی اکثر خصیصه‌ها و اَعمال موردنیاز هست. کلاس جدید از کلاس بالاتر ارث می‌برد و سپس برحسب نیاز، چیزهایی می‌توان به آن افزود.
- به سلسله‌مراتب کلاس‌ها می‌توان سازمانی دوباره داد به طوری که خصیصه‌ها و اَعمال موردنیاز از کلاس جدید قابل ارث بردن باشند.
- خصوصیات یک کلاس موجود را می‌توان تعریف مجدد[1] کرد و نسخه‌های متفاوت دیگری از خصیصه‌ها و اَعمال را برای کلاس جدید پیاده‌سازی کرد.

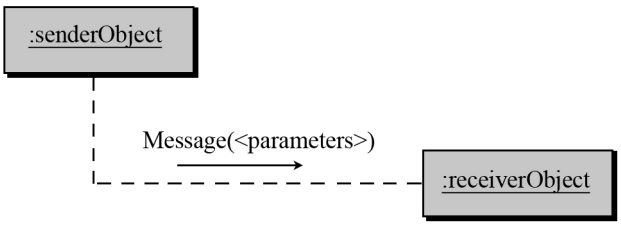
وراثت، همانند کلیه مفاهیم بنیادی طراحی می‌تواند مزایای چشمگیری برای طراحی به همراه داشته باشد، ولی در صورت استفاده‌ی نامناسب[2] می‌تواند بیهوده باعث پیچیده شدن طراحی شود و به ایجاد نرم‌افزاری منجر شود که مستعد خطا بوده نگهداری از آن دشوار است.

پیام‌ها. کلاس‌ها باید با یکدیگر تعامل کنند تا اهداف طراحی حاصل شود. یک پیام باعث می‌شود رفتار خاصی در شیء دریافت‌کننده رخ دهد. این رفتار هنگامی رخ می‌دهد که یک عمل اجرا شود.

---

1. Override

۲. برای مثال، اکثراً طراحان به طراحی زیرکلاسی که وارث خصیصه‌ها و اَعمال بیش از دو زیرکلاس باشد (که گاهی «وراثت چندگانه» خوانده می شود) علاقه ندارند.



شکل پ ۲-۲ طرحی از یک کلاس

نموداری از تعامل میان اشیا در شکل پ ۲-۲ نشان داده شده است. یک عمل در **SenderObject** تولید پیامی به شکل messege(<parameters>) تولید می‌کند که در آن، پارامترها **RecieverObject** را به عنوان شیءای که توسط پیام تحریک می‌شود، عملی در داخل **RecieverObject** که قرار است پیام را دریافت کند و اقلام داده‌ای که اطلاعات لازم برای موفق شدن عمل لازم را فراهم می‌سازند، تعیین می‌کند. همکاری تعریف شده میان کلاس‌ها به عنوان بخشی از مدل نیازمندی‌ها، راهنمایی مفیدی در طراحی پیام‌ها فراهم می‌سازد.

کاکس [Cox86] تعامل میان کلاس‌ها را به شیوه‌ی زیر توصیف می‌کند:

> از یک شیء [کلاس] درخواست می‌شود که یکی از اَعمال خودش را با ارسال پیامی به آن و ذکر وظیفه در این پیام، اجرا کند. [شیء] دریافت‌کننده ابتدا با انتخاب عملی که نام پیام را پیاده‌سازی می‌کند، اجرای این عمل و سپس بازگرداندن کنترل به فراخواننده، به این پیام پاسخ می‌دهد. پیام‌رسانی شیرازه‌ی سیستم شیءگراست و آن را به هم متصل می‌کند. پیام‌ها دیدی از رفتار تک‌تک اشیا و کل سیستم شیءگرا به دست می‌دهد.

چندریختی. خصوصیتی است که تا حد زیادی تلاش لازم برای بسط طراحی یک سیستم شیءگرای موجود را کاهش می‌دهد. برای درک چندریختی، یک برنامه‌ی سنتی را در نظر بگیرید که باید چهار نوع نمودار متفاوت: خطی، دایره‌ای، هیستوگرام و کیویات (Kiviat) را رسم کند. به طور ایده‌آل، هنگامی که داده‌ها برای نوع خاصی از نمودار جمع‌آوری شد، هیستوگرام خودش باید رسم شود. برای نیل به این مقصود در یک برنامه‌ی سنتی (و حفظ انسجام پیمانه‌ها)، لازم خواهد بود که برای هر نوع نمودار یک پیمانه‌ی ترسیمی جداگانه توسعه داده شود. پس در مرحله طراحی، منطق کنترلی مشابه با رویه‌ی زیر باید در نظر گرفته شود:

```
case of graphtype:
   if graphtype = linegraph then DrawLineGraph (data)
   if graphtype = piechart then DrawPieChart (data)
   if graphtype = histogram then DrawHisto (data)
   if graphtype = kiviat then DrawKiviat (data)
endcase
```

گرچه این طراحی به طور قابل قبول صریح است، ممکن است افزودن انواع جدید نمودار نیاز به ترفند داشته باشد. به این معنی که برای هر نوع نمودار باید یک پیمانه‌ی ترسیمی جدید ایجاد شود و سپس منطق کنترلی باید به‌هنگام شود تا نوع نمودار جدید منعکس شود.

برای حل این مشکل در یک سیستم شیءگرا، کلیه نمودارها زیرکلاسی از کلاس عمومی **Graph** می‌شوند. با به‌کارگیری مفهومی به نام تعریف مجدد (overloading) [Tay90] هر زیرکلاس، عملی

تحت نام draw تعریف می‌کند. یک شیء می‌تواند به هر کدام از اشیای نمونه‌برداری شده از هر کدام از زیرکلاس‌ها یک پیام draw ارسال کند. شیءای که پیام را دریافت می‌کند عمل draw خودش را فراخوانی می‌کند تا نمودار مناسب را ایجاد کند. بنابراین، طراحی به صورت زیر کاهش می‌یابد:

```
draw < graphtype>
```

هنگامی که قرار باشد نوع جدیدی از نمودار به سیستم اضافه شود، یک زیرکلاس با عمل draw خودش ایجاد می‌شود. ولی در هر شیءای که می‌خواهد نموداری رسم شود، هیچ تغییراتی لازم نیست چون پیام draw<graph type> بدون تغییر باقی می‌ماند. به طور خلاصه، با چندریختی می‌تواند کاری کرد که چند عمل مختلف دارای نام یکسان باشند. این به نوبه‌ی خود باعث می‌شود که اشیا از هم مستقل شوند و میزان اتصال کاهش یابد.

کلاس‌های طراحی. مدل نیازمندی‌ها، مجموعه‌ی کاملی از کلاس‌های طراحی را تعریف می‌کند. هر کدام از این کلاس‌ها عنصری از دامنه‌ی مسأله را تعریف می‌کند که بر جنبه‌هایی از مسأله تأکید دارند که برای کاربر یا مشتری قابل مشاهده‌اند. سطح انتزاع یک کلاس تحلیل، نسبتاً بالاست. به موازاتی که مدل طراحی تکامل می‌یابد، تیم نرم‌افزاری باید مجموعه‌ای از کلاس‌های طراحی را تعریف کند که (۱) با فراهم ساختن جزییات طراحی مربوط به پیاده‌سازی کلاس‌ها، کلاس‌های تحلیل را پالایش کنند و (۲) مجموعه‌ی جدیدی از کلاس‌های طراحی را ایجاد کنند که یک زیرساخت نرم‌افزاری برای پشتیبانی از راهکار تجاری پیاده‌سازی نماید. پنج نوع متفاوت از کلاس‌های طراحی وجود دارد که هر کدام لایه‌ی متفاوتی از معماری طراحی را نشان می‌دهند [Amb01]:

- کلاس‌های واسط کاربر همه‌ی انتزاع‌هایی را تعریف می‌کنند که برای تعامل انسان با کامپیوتر ضروری‌اند.
- کلاس‌های دامنه‌ی تجاری غالباً شکل پالایش یافته‌ای از کلاس‌های تحلیل هستند که قبلاً تعریف شدند. این کلاس‌ها خصیصه‌ها و اَعمال (متدهایی) را تعریف می‌کنند که برای پیاده‌سازی عنصری از دامنه‌ی تجاری مورد نیازند.
- **کلاس‌های پردازشی**، انتزاع‌های تجاری سطح پایینی را پیاده‌سازی می‌کنند که برای مدیریت کامل کلاس‌های دامنه‌ی تجاری موردنیازند.
- **کلاس‌های ماندگار**، انبارهای داده‌ای (مثلاً پایگاه داده) را نشان می‌دهند که ورای اجرای نرم‌افزار باقی می‌مانند.
- **کلاس‌های سیستمی**، وظایف کنترلی و مدیریت نرم‌افزار را پیاده‌سازی می‌کنند که سیستم را قادر به کار و برقراری ارتباط در داخل محیط کامپیوتری خودش و نیز با جهان خارج می‌سازند.

تیم نرم‌افزاری به موازات تکامل یافتن طراحی معماری، باید مجموعه‌ی کاملی از خصیصه‌ها و اَعمال را برای هر کلاس طراحی توسعه دهد. سطح انتزاع با تبدیل هر کلاس تحلیل به یک نمایش طراحی،

کاهش می‌یابد. یعنی، کلاس‌های تحلیل اشیا (و متدهای مرتبط با آن‌ها را) با به‌کارگیری زبان خاص دامنه‌ی تجاری به نمایش می‌گذارند. در کلاس‌های طراحی، جزییات فنی به مراتب بیشتری به عنوان راهنمایی برای پیاده‌سازی ارائه می‌شود.

آرلو و نوی‌اشتات [Arl02] پیشنهاد می‌کنند که هر کلاس طراحی باید مرور شود تا از "خوش‌فرم" بودن آن اطمینان حاصل شود. آن‌ها چهار خصوصیت برای کلاس طراحی "خوش‌فرم" تعریف می‌کنند. **کامل و کافی.** کلاس طراحی باید محصورسازی کاملی از همه‌ی خصیصه‌ها و متدهایی باشد که به لحاظ منطقی انتظار می‌رود (بر اساس تفسیر آگاهانه‌ای از نام کلاس) برای آن کلاس وجود داشته باشد. برای مثال، کلاس **Scene** که برای یک نرم‌افزار مونتاژ ویدیویی تعریف می‌شود، تنها در صورتی کامل است که حاوی همه خصیصه‌ها و متدهایی باشد که منطقاً انتظار می‌رود برای ایجاد یک صحنه‌ی ویدیویی وجود داشته باشند. کفایت کلاس ایجاب می‌کند که کلاس طراحی تنها حاوی آن دسته از متدهایی باشد که برای دستیابی به اهداف کلاس کافی‌اند، نه بیشتر و نه کمتر.

یگانگی[1]. متدهای مرتبط با یک کلاس طراحی باید تنها یک وظیفه‌ی خاص را برای کلاس مدنظر قرار دهند. هنگامی که وظیفه‌ای با یک متد پیاده‌سازی می‌شود، کلاس نباید راه دیگری برای دستیابی به همان وظیفه داشته باشد. برای مثال، کلاس **VideoClip** در نرم‌افزار مونتاژ ویدیویی ممکن است دارای خصیصه‌های start-point و end-point باشد که نقاط آغازی و پایانی کلیپ را نشان می‌دهند (توجه دارید که تصویر ویدیویی خام بار شده در سیستم ممکن است از کلیپی که استفاده می‌شود، بلندتر باشد). متدهای setStartPoint() و setEndPoint() تنها روش‌های ممکن برای تعیین نقاط شروع و پایان کلیپ هستند.

انسجام بالا[2]. کلاس طراحی منسجم کلاسی است که بر آن فکری واحد حاکم است. یعنی دارای مجموعه‌ای متمرکز از مسئولیت‌ها بوده با فکری واحد، خصیصه‌ها و متدها را برای پیاده‌سازی آن مسئولیت‌ها به کار می‌گیرد. برای مثال، کلاس **VideoClip** در نرم‌افزار مونتاژ ویدیویی حاوی مجموعه‌ای از متدها برای مونتاژ کلیپ ویدیویی است. مادامی که هر متد صرفاً خصیصه‌های مرتبط با کلیپ ویدیویی را کانون توجه قرار می‌دهد، انسجام حفظ می‌شود.

اتصال پایین[3]. در داخل مدل طراحی، لازم است که کلاس‌های طراحی با یکدیگر همکاری داشته باشند. به هر حال، همکاری باید در کمترین سطح قابل قبول حفظ شود. اگر یک مدل طراحی دارای اتصال بالا باشد (همه کلاس‌های طراحی با سایر کلاس‌های طراحی همکاری داشته باشند)، پیاده‌سازی آزمایش و نگهداری سیستم دشوار می‌شود. به طور کلی، کلاس‌های طراحی در یک زیرسیستم تنها باید به آگاهی از کلاس‌های دیگر محدود شوند. این محدودیت، که قانون دِمِتِر [Lie03] نامیده می‌شود[4]، می‌گوید که یک متد تنها باید به متدهای موجود در کلاس‌های مجاور پیام ارسال کند.

---

1. Primitiveness 2. High cohession 3. Low coupling

۴. یک راه کمتر رسمی برای بیان قانون دمتر چنین است: «هر واحد باید فقط با دوستانش حرف بزند و با غریبه‌ها هم‌کلام نشود.»