# مفاهیم شی‌گرایی و پیاده سازی آن‌ها با زبان‌های C++، C#، جاوا و پایتون


**تألیف:**

**دکتر جواد وحیدی**
**دکتر همایون موتمنی**
**دکتر رمضان عباس نژادورزی**



فن‌آوری نوین

| | |
|---|---|
| سرشناسه | وحیدی، جواد، ۱۳۴۸- |
| عنوان و نام پدیدآور | مفاهیم شی‌گرایی و پیاده سازی آها با زبان‌های C++، C#، جاوا و پایتون/ تالیف جواد وحیدی،رمضان عباس‌نژادورزی، همایون موتمنی |
| مشخصات نشر | بابل :فناوری نوین ، ۱۳۹۷. |
| مشخصات ظاهری | ۲۹۶ ص. :مصور |
| شابک | ۳۴۰۰۰۰ ریال978-600-7272-23-7: |
| وضعیت فهرست نویسی | فیپا |
| یاداشت | کتاب نامه |
| موضوع | برنامه‌نویسی شی گرا |
| موضوع | Object-oriented programming (Computer science) |
| موضوع | شی گرایی (کامپیوتر) |
| موضوع | Object-oriented methods (Computer science) |
| شناسه افزوده | موتمنی، همایون، ۱۳۵۰ - |
| شناسه افزوده | عباس‌نژاد ورزی، رمضان، ۱۳۴۸ - |
| رده بندی کنگره | QA۷۶/۱۳۹۷۹ ۳و۹ش/ |
| رده بندی دیویی | 005/117 |
| شماره کتابشناسی ملی | ۵۴۰۴۸۴۰ |

**فن‌آوری نوین** **بابل، کدپستی ۴۷۱۶۷-۷۳۴۴۸** www.fanavarienovin.net **تلفن: ۰۱۱-۳۲۲۵۶۶۸۷**

**مفاهیم شی‌گرایی و پیاده سازی آها با زبان‌های C++، C#، جاوا و پایتون**

تألیف: جواد وحیدی / رمضان عباس نژادورزی / همایون موتمنی

نوبت چاپ: چاپ اول

سال چاپ: پاییز ۹۷

شمارگان: ۱۰۰۰

**قیمت**: ۳۴۰۰۰ تومان

نام چاپخانه و صحافی:

شابک: ۷-۲۳-۷۲۷۲-۶۰۰-۹۷۸

نشانی ناشر: بابل، چهارراه نواب، کاظم‌بیگی، جنب مسجد منصور کاظم‌بیگی، طبقه اول

طراح جلد: کانون آگهی و تبلیغات آبان (احمد فرجی)

تهران، خ اردیبهشت، نبش وحید نظری، پلاک ۱۴۲ تلفکس: ۶۶۴۰۰۲۲۰-۶۶۴۰۰۱۴۴

# فهرست مطالب

## مقدمه

دیدگاه شی‌گرایی از اواسط دهه ۱۹۷۰ تا اواخر ۱۹۸۰ مطرح گردید. در این دوران تلاش‌های زیادی برای ایجاد روش‌های تحلیل و طراحی شی‌گرا صورت پذیرفت. در نتیجه این تلاش‌ها بود که در طول ۵ سال (یعنی ۱۹۸۹تا ۱۹۹۴)، تعداد متدلوژی‌های شی‌گرا از کم‌تر از ۱۰ متدلوژی به بیش از ۵۰ متدلوژی رسید. تکثیر متدلوژی‌ها و زبان‌های شی گرایی و رقابت بین این‌ها به حدی بود که این دوران به عنوان **"جنگ متدلوژی‌ها "** لقب گرفت.

از یک طرف کاربران از متدلوژی‌های موجود خسته شده بودند، زیرا مجبور بودند از میان روش‌های مختلف شبیه به هم که تفاوت کمی در قدرت و قابلیت داشتند یکی را انتخاب کنند. بسیاری از این روش‌ها، مفاهیم مشترک شی‌گرایی را در قالب‌های مختلف بیان می‌کردند که این واگرایی و نبودن توافق میان این زبان-ها، کاربران تازه کار را از دنیای شی‌گرایی زده می‌کرد و آن‌ها را از این حیطه دور می‌ساخت. عدم وجود یک زبان استاندارد، برای فروشندگان محصولات نرم افزاری نیز مشکلات زیادی ایجاد کرده بود.

برنامه نویسی شی گرا در اوایل دهه ۱۹۷۰ توسط آلن کی Alan Kay طراحی شد. یعنی اولین قدم‌های این سبک برنامه نویسی توسط آلن کی برداشته شده است. اولین زبان شی‌گرا (Small Talk) توسط آلن کی طراحی شد. آلن کی گفته بود که : آن چیزی که باعث شد این فکر به ذهنم برسد نحوه عملکرد سلول‌های زیست محیطی بود. یعنی، این سبک برنامه‌نویسی از روی سلول‌های جانداران الگو برداری شده است. آن چیزی که باعث شد که آلن کی از روی سلول‌های جانداران الگو برداری کند نحوه زندگی سلول‌ها بود:

- هر سلول نمونه‌ای از اصل خودش است و هر خصوصیاتی که دارد از اصل خود (ژنتیک سلول) به ارث برده است. همچنین هر سلول رفتارهایی دارد که از اصل خود به ارث برده است.
- سلول‌ها همگی مستقل از هم زندگی می‌کنند و براساس ارسال پیام‌های شیمیایی با یکدیگر ارتباط برقرار می‌کنند. ارسال پیام به این صورت است که پیام از پوسته یکی خارج و به پوسته دیگری وارد می شود.
- سلول‌ها می‌توانند از یکدیگر متمایز شوند.

با توجه به این گفته‌ها، می‌توان متوجه شد که همان مشخصه کلاس‌ها را بیان می‌کند. یعنی، هر شی از یک کلاس تشکیل شده که ویژگی‌های آن کلاس را با خودش به ارث برده است. همان‌طور که می‌دانیم اشیا با یکدیگر ارتباط برقرار می‌کنند. نحوه ارتباط با فرستادن پیام در اشیا هنگام فراخوانی رفتارها در یک رویداد است. هر شی خودش یک شناسنامه دارد که ویژگی‌های آن شی را بیان می‌کند.

برنامه‌نویسی شی‌گرا شیوه نوینی است که در آن می‌توان قطعاتی را ایجاد کرد و در برنامه‌های مختلف مورد استفاده قرار داد. قابلیت خوانایی برنامه‌هایی که در این روش نوشته می‌شوند بالا بوده، تست، عیب‌یابی و اصلاح آن‌ها آسان است. شی‌گرایی، بر اشیا تاکید دارد. در برنامه‌نویسی شی‌گرا اشیا به صورت انتزاع مطرح می‌شوند. انتزاع: به آن چیزی می‌گویند که در مورد آن فکر می‌کنید و در یک دید کلی مطرح می‌کنید. مثلاً وقتی به یک دانه شن فکر می‌کنید ناخوداگاه فکرتان به سمت ساحل می‌رود یا وقتی به یک درخت فکر می-کنید ذهنتان به سمت جنگل متمرکز می‌شود.

این کتاب شامل ۸ فصل است که فصل‌های اول تا چهارم، مفاهیم از قبیل پیچیدگی، شی‌گرایی، کلاس‌ها، انتزاع و وراثت را مورد بررسی قرار می‌دهد. فصل‌های چهارم تا هشتم به ترتیب مفاهیم بیان شده را با زبان‌های C++، C#، جاوا و پایتون پیاده‌سازی می‌کند.

# فصل ۱
# پیچیدگی

یک پزشک، یک مهندس عمران و یک دانشمند کامپیوتر در مورد این‌که قدیمی‌ترین شغل در دنیا کدام است، بحث می‌کردند. پزشک گفت: خوب، در کتاب مقدس گفته‌شده که خداوند حوا را از گوشت دنده‌ی آدم خلق کرده است. این کار واضح است که به جراحی نیاز داشت و بنابراین، می‌توانم با قاطعیت ادعا کنم که شغل من قدیمی‌ترین شغل در جهان است.

مهندس عمران حرفش را قطع کرد و گفت، اما پیش‌تر از آن، در کتاب تورات گفته‌شده است که خداوند نظم آسمان‌ها و زمین را از بی‌نظمی خلق کرده است. این کار اولین و مطمئناً تماشایی‌ترین کار مهندس عمران است.

سپس پزشک برآشفت و گفت، شما در اشتباه هستید. شغل من قدیمی‌ترین شغل در جهان است. دانشمند کامپیوتر که پشت صندلی‌اش لم‌داده بود لبخندی زد و با اعتمادبه‌نفس گفت : **"اوه، اما شما فکر می‌کنید این بی‌نظمی را چه کسی خلق کرده است؟"**

هر چه قدر سیستم پیچیده‌تر باشد، اجزای بیش‌تری باید تفکیک گردد.

به‌ندرت پیش می‌آید یک معمار به فکر اضافه کردن یک طبقه زیر زمین در یک ساختمان صد طبقه بی‌افتد. انجام این کار خیلی هزینه‌بر خواهد بود و بدون شک منجر به شکست خواهد شد.

ضعف ما در مهار پیچیدگی نرم‌افزار سبب تأخیر در پروژه، بودجه بیش‌ازحد، ناکارایی در نیازمندی‌های مقرر پروژه می‌شود. اغلب به این شرایط ایجادشده **بحران نرم‌افزاری** می‌گوییم، اما صراحتاً باید گفت که بیماری و اختلالی که مدت زمان زیادی از وجودش سپری‌شده باشد را باید طبیعی دانست و دیگر بیماری نیست.

متأسفانه این بحران منجر به اتلاف با ارزش‌ترین شکل یعنی نیروی انسانی و همچنین از دست رفتن فرصت‌ها می‌شود. به اندازه کافی توسعه‌دهنده‌ی نرم‌افزارهای جدیدی که کاربران نیاز دارند، در همه‌جا وجود ندارد. از این گذشته، تعداد قابل‌ملاحظه‌ای از کارکنان تیم توسعه در هر سازمانی باید برای تعمیر یا نگه‌داری نرم‌افزار فرسوده و قدیمی تخصیص داده شوند. با در نظر گرفتن نقش مستقیم و نیز غیرمستقیم نرم‌افزار در جایگاه اقتصادی کشورهای صنعتی و با توجه به راه‌هایی که در آن نرم‌افزار می‌تواند مهارت فردی را تقویت کند غیرقابل‌قبول است که اجازه دهیم شرایط بحران نرم‌افزاری ادامه یابد.

## ۱–۱. ساختار سیستم‌های پیچیده

چگونه می‌توان این وضعیت بسیار بد را تغییر داد؟ ازآنجایی‌که مشکل واقعی از پیچیدگی ذاتی نرم‌افزار ناشی می‌شود، بهتر است در مورد این‌که چطور سیستم‌های پیچیده سازمان‌یافته شده‌اند، مطالعه کنیم. درواقع اگر چشمانمان را به دنیای اطرافمان باز کنیم، سیستم‌های پیچیده خوبی را خواهیم یافت. بعضی از این سیستم‌ها

مثل شاتل فضایی و تونل فرانسه – انگلستان و سازمان‌های تجاری بزرگ دست‌ساخت بشر است. البته بیش‌تر این سیستم‌های پیچیده در طبیعت یافت می‌شوند مانند سیستم گردش خون انسان و ساختار گیاه فلفل کوبایی.

**ساختار کامپیوتر شخصی**

کامپیوتر دستگاهی با پیچیدگی متوسط است. غالب آن‌ها از تعدادی واحد مشترک تشکیل‌شده‌اند: واحد پردازش مرکزی(CPU)، صفحه‌نمایش، صفحه‌کلید و تعدادی از انواع دستگاه‌های ذخیره‌سازی معمول مانند CD یا DVD و یا هارددیسک. می‌توانیم هر یک از این واحدها را تجزیه کنیم. به‌طور مثال، یک CPU شامل حافظه اصلی، واحد محاسبه و منطق (ALU) و یک گذرگاه برای دستگاه‌های جانبی متصل شده است. هرکدام از این بخش‌ها می‌توانند به بخش‌های جزء تجزیه شوند: مثلاً واحد ALU می‌تواند به بخش‌های ثبات (رجیستر) و منطق کنترل تصادفی (Random Control Logic) تقسیم شود، به‌طوری‌که هرکدام به‌تنهایی از ابتدایی‌ترین واحدها مثل گیت‌های NAND و معکوس کننده (inverter) و غیره ساخته‌شده‌اند.

اینجا است که به ماهیت سلسله‌مراتبی سیستم‌های پیچیده پی می‌بریم. یک کامپیوتر تنها به خاطر فعالیت مشترک و یکپارچه این واحدها است که به‌درستی کار می‌کند. این بخش‌های مجزا، باهم یک واحد کل را می‌سازند. با تجزیه‌ی واحدهای کامپیوتر و مطالعه‌ی عملکرد هر یک از آن‌ها می‌توانیم به چگونگی طرز کار کامپیوتر پی ببریم. بنابراین، می‌توانیم عملکرد صفحه‌کلید و هارددیسک را به‌طور جداگانه و همچنین طرز کار واحد ALU را بدون توجه به حافظه اصلی موردبررسی قرار دهیم.

نه‌تنها سیستم‌های پیچیده، سلسله‌مراتبی هستند، بلکه هرکدام از سطوح این سلسله‌مراتب نمایانگر سطح تجرید متفاوتی است، هرکدام بر پایه دیگری ساخته‌شده و هرکدام به‌تنهایی قابل‌فهم هستند. در هر سطح تجرید مجموعه‌ای از دستگاه‌ها وجود دارند که با یک‌دیگر جهت ارائه سرویس به لایه‌های بالایی همکاری می‌کنند. برای یک نیاز مشخص باید سطح تجریدی را انتخاب کرد که با آن متناسب باشد. برای مثال، اگر بخواهیم مشکل زمان‌بندی حافظه اصلی را برطرف کنیم باید سطح گیت‌بندی معماری کامپیوتر بررسی گردد. درحالی‌که این سطح تجرید برای برطرف کردن مشکلی که یک برنامه صفحه گسترده می‌تواند پیدا کند نامناسب و بی‌مورد است.

**ساختار گیاهان و حیوانات**

در علم گیاه‌شناسی، دانشمندان سعی دارند به شباهت‌ها و تفاوت‌های میان گیاهان از طریق مطالعه‌ی ریخت‌شناسی آن‌ها – شکل و ساختار آن‌ها – پی ببرند. گیاهان از ارگانیسم‌های چند سلولی پیچیده هستند و از فعالیت مشترک مجموعه‌ی اندام‌های گیاهان مختلف رفتارهای پیچیده‌ای هم چون فتوسنتز و تعرق پدید می‌آید. گیاهان شامل سه ساختار اصلی (ریشه‌ها، ساقه‌ها، برگ‌ها) هرکدام از این‌ها ساختار متفاوت و خاص دارند. برای مثال، ریشه‌ها شامل ریشه‌های فرعی، موهای ریشه، نوک ریشه، کلاهک انتهای ریشه هستند. به‌طور مشابه، یک برش عرضی از برگ شامل روپوست، بافت درونی برگ، بافت آوندی برگ است. هر یک از این ساختارها شامل مجموعه‌ای از سلول‌ها است و درون هر سلول سطح دیگری از پیچیدگی شامل واحدهایی مثل کلروپلاست، هسته اصلی و... وجود دارد. همانند ساختار کامپیوتر، اجزای گیاهان نیز از سلسله‌مراتب برخوردارند و هر سطح از این سلسله‌مراتب پیچیدگی خودش را دارد.

همه‌ی اجزای یک سطح تجرید، از راه‌های مشخصی باهم در ارتباط هستند. به‌طور مثال، در بالاترین سطح تجرید، ریشه‌ها مسئول جذب آب و املاح معدنی از خاک هستند. ریشه‌ها با ساقه‌هایی که املاح را به برگ می‌رسانند در ارتباط‌اند و برگ‌ها از آب و مواد معدنی که ساقه فراهم کرده است برای تولید غذا ( از طریق فتوسنتز) استفاده می‌کنند.

همواره حدومرز واضحی بین داخل و خارج یک سطح وجود دارد. برای مثال، می‌توان گفت که اجزای برگ در کنار هم کار می‌کنند تا کارکرد برگ به‌عنوان یک واحد کل حفظ شود و بااین‌وجود ارتباط کم و غیرمستقیم با اجزای ریشه دارد. به عبارت ساده‌تر، ارتباط تنگاتنگی بین اجزای سطوح تجرید مختلف وجود دارد.

در طراحی واحد پردازش مرکزی(CPU) کامپیوتر و همین‌طور هارددیسک از گیت‌های NAND استفاده‌شده است. در سلسله‌مراتب یک گیاه هم اشتراکات قابل‌توجهی است. برای مثال، سلول‌ها به‌عنوان یک واحد ساختمانی پایه در همه‌ی ساختارهای گیاه وجود دارد؛ درنهایت ریشه‌ها، ساقه‌ها و برگ‌های یک گیاه از سلول‌ها ساخته‌شده‌اند. اگرچه هر یک از این واحدهای پایه یک سلول است اما انواع مختلفی از سلول وجود دارد. برای مثال، سلول‌هایی با کلروپلاست یا فاقد کلروپلاست، سلول‌هایی با دیواره‌هایی که غیرقابل نفوذ آب در آن هستند و سلول‌هایی با دیواره‌های قابل‌نفوذ آب آن و حتی سلول‌های زنده و سلول‌های مرده.

در مطالعه‌ی ریخت‌شناسی گیاهان، نمی‌توان بخش‌های خاصی از گیاه را پیدا کرد که مسئولیت کل فرآیند(مثل فتوسنتز) را برعهده‌گرفته باشد. در حقیقت، واحد مرکزی که مستقیماً فعالیت‌های سطوح پایینی را هماهنگ کند؛ وجود ندارد. در عوض، بخش‌های مجزایی وجود دارند که مستقل عمل می‌کنند و هرکدام رفتار نسبتاً پیچیده‌ای دارند و در بسیاری از عملیات سطوح بالایی همکاری می‌کنند. تنها از طریق همکاری متقابلی که این مجموعه هدف‌مند دارد می‌توان عملکرد سطوح بالایی را فهمید. علم پیچیدگی این رفتار را **معلول** می‌نامد. **رفتار کل مجموعه از مجموع رفتار اجزایش بزرگ‌تر است.**

با نگاه مختصر درزمینه جانورشناسی، متوجه خواهیم شد که حیوانات چند سلولی نیز همانند گیاهان ساختار سلسله‌مراتبی دارند. مجموعه سلول‌ها تشکیل بافت را می‌دهند، بافت‌ها در کنار هم اندام‌ها را به وجود می‌آورند و مجموعه‌ی از اندام‌ها سیستم را ایجاد می‌کنند(سیستم گوارش). چاره‌ای نداریم جز این‌که کار مقرون‌به‌صرفه خداوند را متذکر باشیم. عنصر اولیه بدن حیوانات همانند گیاهان از سلول است. البته سلول‌های این دو با هم متفاوت است. برای مثال، سلول‌های گیاهان به‌وسیله‌ی دیواره‌های سلولزی سختی محصورشده‌اند، درحالی‌که سلول‌های حیوانات این‌طور نیست. باوجوداین تفاوت‌ها هر دوی این تفاوت‌ها سلولی‌اند. درواقع، این مثالی از اشتراکات است که حوزه‌های مختلفی را به هم پیوند می‌دهد. ماکانیسم‌های زیادی فراتر از سطح سلولی که بین گیاهان و حیوانات مشترک است، هر دو از نوعی سیستم عروقی برای انتقال مواد مغذی درون اندام‌ها استفاده می‌کنند و یا این‌که هر دو تمایزشان را از طریق جنسیت یکی از اندام‌های همان‌گونه نشان می‌دهند.

### ساختار ماده

مطالعه‌ی زمینه‌های متفاوتی مانند ستاره‌شناسی و فیزیک هسته‌ای نمونه‌های فوق‌العاده‌ای از سیستم‌های پیچیده هستند. با کندوکاو این دو هم ساختار سلسله‌مراتبی مشاهده می‌شود. ستاره شناسان به مطالعه‌ی

کهکشان‌هایی که به‌صورت خوشه مرتب‌شده‌اند، می‌پردازند. ستاره‌ها، سیاره‌ها، خرده‌سنگ‌ها اجزاء سازنده کهکشان راه شیری هستند. به‌طور مشابه یک فیزیکدان هسته‌ای هم با یک ساختار سلسله‌مراتبی در مقیاسی متفاوت روبه‌رو است. اتم‌ها از الکترون‌ها، پروتون‌ها و نوترون‌ها ساخته‌شده‌اند.

به نظر می‌رسد الکترون‌ها ذره‌ی بنیادی باشد اما پروتن‌ها و نوترون‌ها و ذره‌های دیگر از واحدهای بنیادی به نام **کوارک** ساخته‌شده‌اند. وجه اشتراک بزرگی که در اجزای سهیم وجود دارد باعث وحدت سلسله‌مراتب اصلی خواهد شد. به بیان دقیق‌تر، جهان ما تنها توسط چهار نیروی گرانش زمین، نیروی برهم‌کنش الکترومغناطیسی، نیروی هسته‌ای قوی و نیروی هسته‌ای ضعیف کار می‌کند. قوانین فیزیکی زیادی هم چون قانون پایستگی انرژی و تکانه که این چهار نیرو در آن درگیرند، در کهکشان‌ها و همین‌طور کوارک‌ها به کار بسته‌شده است.

**ساختار نهادهای اجتماعی**

به‌عنوان آخرین نمونه از سیستم‌های پیچیده، به ساختار نهادهای اجتماعی روی آوردیم. گروهی از انسان‌ها برای به انجام رساندن کارهایی که تنهایی قادر به انجام آن نیستند، به هم می‌پیوندند. برخی از سازمان‌ها موقتی هستند و برخی دیگر برای همیشه پایدار می‌مانند. هر چه قدر سازمان‌ها رشد بیش‌تری می‌یابند، سلسله‌مراتب مجزایی شکل می‌گیرد. مؤسسات چندملیتی از شرکت‌هایی که از چندین بخش و بخش‌هایی که از چندین شعبه و شعباتی که شامل دفاتر محلی است، تشکیل می‌شوند. اگر سازمان دوام داشته باشد، حدومرز این بخش-ها تغییر خواهد کرد و در طول زمان یک سلسله‌مراتب بسیار با ثبات حاصل می‌شود. روابط میان بخش‌های مختلف یک سازمان بزرگ درست شبیه به همان ارتباطی است که بین قطعات کامپیوتر یا گیاهان و یا حتی کهکشان یافتیم. میزان تعامل میان کارمندان یک دفتر کار از تعاملشان با دفترهای دیگر بیش‌تر است. معمولاً یک کارمند پست تعاملی با مدیر اجرایی یک شرکت ندارد، اما تعامل مداومی با کارکنان دفتر پست خواهد داشت. البته از طرفی این دو سطح متفاوت از طریق مکانیسم‌های مشترکی متحد می‌شوند. کارمند و مجری هر دو از طریق یک تشکیلات مالی یکسان حقوق دریافت می‌کنند و یا هر دو از امکانات مشترکی مانند سیستم تلفن شرکت جهت به انجام رساندن کارهایشان استفاده می‌کنند.

## ۲-۱. پیچیدگی ذاتی نرم‌افزار

ستاره‌ی رو به افول و نزدیک فروپاشی، کودک در حال یادگیری که چطور بخواند، حمله‌ی گلبول‌های سفید خون به ویروس‌ها، این‌ها مواردی از دنیای واقعی هستند که به‌راستی درگیر حیرت‌آورترین نوع پیچیدگی هستند. نرم‌افزار هم ویژگی‌های سبک سیستم پیچیده را دارا است گرچه پیچیدگی آن نوع متفاوتی دارد. همان-طور که کتاب‌ها اشاره کردند : انیشتین اظهار می‌کرد که می‌بایست تبین ساده‌ای از طبیعت وجود داشته باشد. چراکه خداوند غیرقابل‌اعتماد و تصادفی کار نمی‌کند. به‌هیچ‌وجه چنین اعتماد و ثباتی در کار نرم‌افزار نیست که کار مهندسین نرم‌افزار را راحت کند. بسیاری از پیچیدگی‌هایی که آن‌ها باید کنترل کنند، تصادفی هستند.

### تعریف پیچیدگی نرم‌افزار[1]

دریافتیم که بعضی از سیستم‌های نرم‌افزاری پیچیده نیستند. این‌ها عمدتاً نرم‌افزارهای معمولی هستند که توسط یک شخص تعیین‌شده، ساخته‌شده و نگه‌داری شده و استفاده می‌گردد که این فرد معمولاً یک برنامه-نویس آماتور و یا توسعه‌دهنده حرفه‌ای است که به‌تنهایی کار می‌کند. منظورمان این نیست که همه‌ی سیستم‌ها، معمولی و فاقد ظرافت هستند. قصد کم‌ارزش کردن کار این افراد را نداریم. بعضی از سیستم‌ها اهداف محدودی با طول عمر بسیار کوتاه دارند. می‌توانیم به‌جای تلاش برای استفاده مجدد، اصلاح و گسترش قابلیت-های آن‌ها، از آن‌ها چشم‌پوشی کرده و دور بریزیم. چنین برنامه‌هایی بیش‌تر از این‌که توسعه‌شان سخت باشد، خسته‌کننده هستند؛ درنتیجه یادگیری این‌که چطور آن‌ها را طراحی کنیم برای‌مان جذاب نخواهد بود.

در عوض، به چالش‌های مربوط به توسعه‌ی نرم‌افزارهایی که آن را نرم‌افزارهای industrial-strength می-خوانیم علاقه‌مندیم. برنامه‌هایی که مجموعه‌ی بسیار زیادی از رفتارها را ارائه می‌دهند. برای مثال، سیستم‌های واکنشی که یا خودشان را از طریق رویدادهای دنیای اطراف هدایت می‌کنند و یا هدایت می‌شوند و در هر زمان و مکانی جزء منابع کمیاب هستند؛ نرم‌افزارهایی که یکپارچگی صدها هزار رکورد اطلاعات را حفظ می-کنند. درحالی‌که هم‌زمان اجازه بروز رسانی و پرس‌وجو هم وجود ندارد؛ سیستم‌هایی که موجودیت‌های دنیای واقعی را کنترل می‌کنند و فرمان می‌دهند. مانند: مسیر جریان هوا، ترافیک بزرگراه. سیستم‌های نرم‌افزاری هم تمایل به طول عمر بالا دارند و بسیاری از کاربران به‌جایی می‌رسند که در کارهای اصلی‌شان به آن وابسته می-شوند. در دنیای نرم‌افزارهای صنعتی گرا با چارچوب‌های ( framworkهایی) روبه رو خواهیم بود که ایجاد برنامه‌های کاربردی خاص محور و برنامه‌هایی که جنبه‌های هوش بشری را تقلید می‌کنند را آسان‌تر می‌کنند. گرچه چنین برنامه‌های کاربردی معمولاً محصول تحقیق و توسعه هستند، ولی ازلحاظ پیچیدگی کم‌تر از آن‌ها نیست چراکه آن‌ها روش‌ها و مصنوعات توسعه‌ی اکتشافی و افزایشی هستند.

ویژگی متمایز نرم‌افزار صنعتی گرا این است که گرچه غیرممکن نیست اما به‌شدت برای یک نفر توسعه‌دهنده سخت و دشوار است که تمام ظرافت‌های طراحی آن را درک کند. بی‌پرده باید گفت که پیچیدگی برخی از سیستم‌ها از ظرفیت فکری و ذهنی انسان‌ها فراتر است. افسوس که به نظر می‌رسد این پیچیدگی، ویژگی ذاتی نرم‌افزارهای بزرگ‌شده باشد. منظورمان از ذاتی این است که ممکن است پیچیدگی را مهار کنیم، اما هرگز نمی‌توانیم کاری کنیم از بین برود.

### چرا نرم‌افزار ذاتاً پیچیده است

همان‌طوری که بروکس(Broks) گفته است : پیچیدگی نرم‌افزار یک ویژگی ذاتی است نه یک امر تصادفی. می‌بینیم که این پیچیدگی ذاتی از چهار عنصر مشتق شده است : **پیچیدگی محدوده‌ی مسئله**[2]، **مشکل مدیریت فرآیند توسعه، تغییرپذیری احتمالی نرم‌افزار، مشکلات توصیف رفتار سیستم‌های مجزا**.

---

[1]. Complexity of software [2]. Problem domain

### پیچیدگی محدوده‌ی مسئله

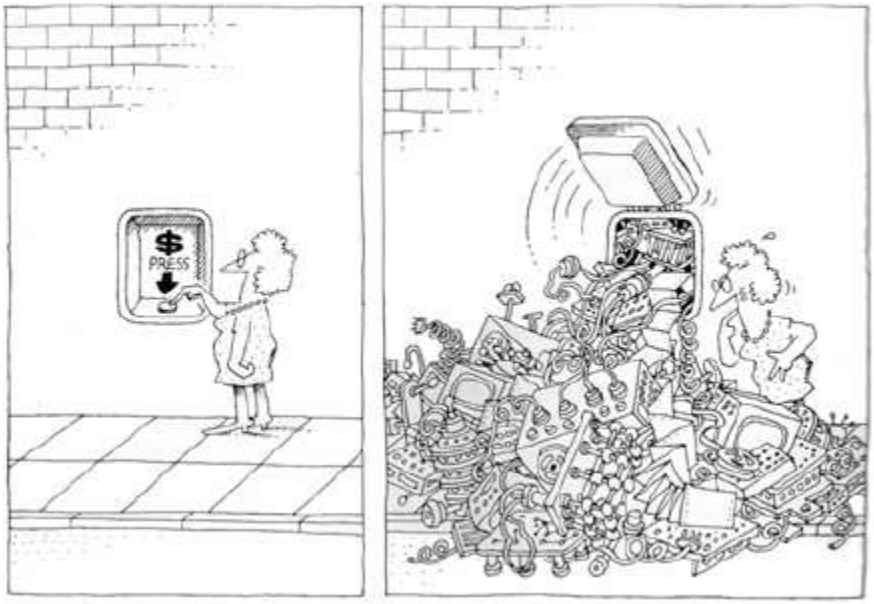
مسائلی که قرار است از طریق نرم‌افزار آن‌ها را حل کنیم، اغلب درگیر پیچیدگی‌های اجتناب‌ناپذیری هستند، به‌طوری‌که هزاران تناقض و یا حتی نیازهای متضاد در آن می‌یابیم.

در مورد نیازهای سیستم الکترونیکی هواپیمای چند موتوره، سیستم سوئیچینگ تلفن سلولی و یا یک ربات خودکار فکر کنید.

اما حال باید همه‌ی نیازمندی‌های غیر کارکردی(عملکردی) مانند قابلیت استفاده، کارایی، هزینه، قابلیت بقا، قابلیت اطمینان را هم به آن اضافه کرد. این پیچیدگی خارجی همان چیزی است که سبب پیچیدگی تصادفی می‌شود که بروکس در موردش نوشته بود.

این پیچیدگی خارجی معمولاً از خلأ ارتباطی میان کاربران سیستم و توسعه‌دهندگان آن ناشی می‌شود. برای کاربران دشوار است که توصیف درستی از نیازهایشان را ارائه دهند، به شکلی که توسعه‌دهنده بتواند آن را درک کند. در برخی موارد ممکن است کاربران نظرات مبهمی ازآنچه که در یک سیستم نرم‌افزاری نیاز دارند، نداشته باشند. این موضوع خیلی تقصیر کاربران یا توسعه‌دهندگان نیست، بلکه به خاطر این است که هرکدام از این دو طرف فاقد مهارت حوزه‌ی کاری طرف مقابلشان هستند. کاربران و توسعه‌دهندگان دید متفاوتی از طبیعت مسئله دارند و این امر باعث می‌شود که فرض‌های متفاوتی از راه حل داشته باشند. درحالی‌که حاضر ابزارهای کمی برای استخراج این نیازمندی‌ها در دست داریم. شیوه‌ی معمول بیان نیازمندی-ها، استفاده از حجم وسیعی از متن و گاهی اوقات همراه با ترسیم شکل است.

اسنادی که فهم آن‌ها دشوار است معمولاً دستخوش برداشت‌های متفاوتی می‌شوند و گاهی اوقات به‌جای نیازمندی‌های اصلی، اجزاء دیگری لحاظ می‌شود.




الف: وظیفه تیم توسعه‌ی نرم‌افزار، برنامه‌ریزی در جهت ایجاد توهم سادگی است.

مسئله‌ی دیگر این است که اغلب اوقات نیازمندی‌های یک سیستم نرم‌افزاری در طول دوره توسعه‌اش تغییر می‌کنند، عمدتاً به خاطر این‌که دوره حیات پروژه توسعه نرم‌افزار زیاد می‌شود، قاعده و اصل مسئله تغییر می‌کند. مشاهده‌ی تولیدات اولیه، مانند طرح اسناد و نمونه‌ها[1] و استفاده از سیستم‌های که یک بار نصب می‌شود و ازلحاظ عملیاتی شامل توابع اصلی هستند کاربران را به سمت درک و بیان بهتر نیازهای واقعی‌شان هدایت می‌کنند. هم‌زمان، این فرآیند به توسعه‌دهندگان نیز کمک می‌کند که بر محدوده و حوزه‌ی مسئله تسلط پیدا کنند و به آن‌ها این امکان را می‌دهد که سؤالات بهتری بپرسند تا نقاط تاریک رفتار مطلوب سیستم روشن گردد. چون سیستم نرم‌افزاری بزرگ یک سرمایه‌گذاری ثابت محسوب می‌شود، توانایی آن را نداریم که به خاطر تغییر کردن همیشگی نیازمندی‌ها، یک سیستم نرم‌افزاری را دور بی‌اندازیم( پیش‌بینی‌شده یا نشده)، نرم‌افزارها همواره درحال تکامل‌اند وضعیتی که به اشتباه نگه‌داری نرم‌افزار تلقی می‌شود. اگر بخواهیم بیش‌تر دقیق باشیم، زمانی که خطاها را تصحیح می‌کنیم نگه‌داری صورت می‌گیرد. زمانی که نیازمندی‌ها تغییر می‌کند و به تغییرات پاسخ می‌دهیم تکامل صورت می‌گیرد و هرگاه از شیوه‌های خاص استفاده کنیم تا نرم‌افزارهای روبه‌زوال و قدیمی به کار خود ادامه دهند، محافظت(presentation)گویند. متأسفانه، واقعیت از این خبر می‌دهد که درصد زیادی از منابع توسعه نرم‌افزار در محافظت نرم‌افزار صرف می‌شود.

**مشکل مدیریت فرآیند توسعه**

وظیفه‌ی اولیه تیم توسعه نرم‌افزار، مهندسی توهم(خطای حسی) سادگی است- محافظت کاربران از حجم وسیع پیچیدگی – یقیناً اندازه ویژگی مهمی در سیستم نرم‌افزاری نیست. می‌کوشیم با ابداع مکانیزم‌های هوشمندانه و قدرت‌مند که حس سادگی را به ما می‌دهد و همچنین از طریق استفاده مجدد از کدها و framworkهای طرح‌های موجود، کد نویسی را به حداقل برسانیم. اگرچه گاهی اوقات حجم زیاد نیازمندی‌های نرم‌افزار اجتناب‌ناپذیر است و چه بخواهیم یک نرم‌افزار بزرگ بنویسیم و چه از نرم‌افزارهای موجود مجدداً استفاده کنیم به ما فشار وارد می‌کند تا چند دهه‌ی قبل، برنامه‌های زبان اسمبلی با چند هزار خط کد محدوده‌ی توانایی مهندسین نرم‌افزار را تحت‌فشار قرار می‌دهد. اما امروزه سیستم‌های ارائه‌شده با اندازه صدها هزار و یا حتی میلیون‌ها خط کد (که همه آن‌ها با زبان‌های سطح بالا نوشته می‌شود) غیرعادی نیست. حتی یک فرد به‌تنهایی نمی‌تواند یک سیستم را بفهمد. حتی یک سیستم پیاده‌سازی شده را با روش‌های دقیق تجزیه کنیم به صدها و گاهی اوقات هزاران ماژول مستقل خواهیم رسید. این حجم کار نیاز به تیمی از توسعه‌دهنده دارد و به‌طور ایدئال ما کوچک‌ترین تیم ممکن را استفاده می‌کنیم. بدون توجه به‌اندازه‌ی تیم، همیشه چالش‌های عمده‌ای در ارتباط با تیم توسعه وجود دارد. توسعه‌دهنده‌ی بیش‌تر به معنی ارتباطات پیچیده‌ی بیش‌تر و ازاین‌رو سختی هماهنگ‌سازی بیش‌تر است به‌ویژه اگر تیم ازلحاظ مکانی از هم دور باشند که اغلب هم همین‌طور است. با تیمی از توسعه کنندگان، حفظ هماهنگی و یکپارچگی طرح همواره چالش کلیدی مدیریت است.

---

[1]. Prototype

معمولاً یک شرکت خانه‌سازی برای خودش مزرعه‌ی درختی برای تولید الوار موردنیازش ندارد. این بسیار غیرمعمول است که یک شرکت ساخت‌وساز، برای ساخت تیر حمال ساختمان‌هایش یک کارخانه ذوب‌آهن بسازد. در صنعت نرم‌افزار هم‌چنین کاری معمول است. نرم‌افزار از نهایت انعطاف‌پذیری برخوردار است، پس این احتمال برای توسعه‌دهندگان وجود دارد که هرگونه انتزاعی را بیان کنند.

همان‌طور که صنعت ساخت‌وساز برای کیفیت مواد خام، اصول و استانداردهای ساختمانی یکسانی دارند. در صنعت نرم‌افزار هم، چنین استانداردهایی هست. به‌عنوان نتیجه باید گفت توسعه نرم‌افزار تجارت پرکاری است.

**مشکلات شناخت رفتار سیستم‌های مجزا**

اگر یک توپ را به هوا پرتاب کنیم به‌طورقطع می‌توانیم مسیر برگشت آن را پیش‌بینی کنیم چراکه می‌دانیم در شرایط عادی، قوانین خاص فیزیک اعمال می‌شوند. اگر فقط به خاطر این‌که توپ را کمی تندتر به هوا پرتاب کردیم، در وسط مسیر پرواز توپ به‌طور اتفاقی بایستد و دوباره مستقیم رو به بالا پرتاب شود قطعاً خیلی تعجب خواهیم کرد. اما در شبیه‌سازی حرکت این توپ در نرم‌افزاری که به‌طور کامل رفع اشکال نشده به‌سادگی این نوع رفتارها رخ می‌دهد.

درون یک برنامه کاربردی بزرگ، ممکن است صدها و یا هزاران متغیر و هم‌چنین بیش از یک نخ کنترلی وجود داشته باشد. مجموعه‌ی کامل این متغیرها، آدرس و مقدار فعلی‌شان و پشته‌ی فراخوانی‌های هر فرآیند درون سیستم، حالت فعلی برنامه کاربردی را تشکیل می‌دهد. چون نرم‌افزار را در کامپیوترهای دیجیتالی اجرا می‌کنیم سیستمی باحالت‌های گسسته داریم. بالعکس سیستم‌های آنالوگ همانند حرکت توپ پرتاب‌شده سیستم‌های پیوسته هستند. پارنس(parnes) می‌گوید: زمانی که می‌گوییم یک سیستم به‌وسیله‌ی یک تابع پیوسته تعریف‌شده است، درواقع می‌گوییم که آن سیستم شامل رویدادهای ناگهانی نیست. با تغییرات کوچک در ورودی‌ها به همان نسبت هم در خروجی‌ها تغییرات کوچکی خواهیم داشت. از آن‌طرف سیستم‌های گسسته با ویژگی زیادشان دارای تعداد محدودی از حالت‌های ممکن هستند؛ در سیستم‌های بزرگ انفجار ترکیبی باعث می‌شود که این تعداد خیلی زیاد شود. باید سیستم‌مان را همواره با تفکیک وظایف طراحی کنیم به‌طوری‌که رفتار یک بخش از سیستم کم‌ترین تأثیر را در بخش‌های دیگر داشته باشد. هر رویداد بیگانه با سیستم نرم‌افزاری پتانسیل این را دارد که سیستم را در حالت جدیدی قرار دهد، از این گذشته نگاشت از یک حالت به حالت دیگر همیشه قطعی نیست. در بدترین شرایط، یک رویداد خارجی ممکن است حالت یک سیستم را خراب کند، به خاطر این‌که طراحانش حساب تأثیر متقابل میان رویدادها را نکرده بودند. زمانی که سیستم پیش رانش کشتی به علت سرریز ریاضی از کار می‌افتد که آن نیز براثر این‌که شخصی داده‌ی نادرستی را در سیستم حفاظت وارد کرده است(یک رویداد واقعی). حال اهمیت این موضوع را درک می‌کنیم. ترقی شگرفی در عیوب سیستم وابسته به نرم‌افزار، مثلاً در سیستم‌های مترو، اتومبیل‌ها، ماهواره‌ها، سیستم‌های کنترل ترافیک هوایی، سیستم‌های انبارداری و... به وجود آمده است. در سیستم‌های پیوسته این نوع رفتارها بعید خواهد بود، اما در سیستم‌های گسسته هر رویداد خارجی می‌تواند هر بخش از حالت درونی سیستم را تحت تأثیر قرار دهند. بدون تردید این موضوع انگیزه اصلی خواهد بود تا نرم‌افزارهای‌مان را جدی‌تر آزمایش کنیم، اما برای همه به جز بیش‌تر سیستم‌های معمولی، تست کامل غیرممکن است. برای مدل‌سازی رفتار کامل

سیستم‌های گسسته، هیچ‌گونه ابزار ریاضی یا قدرت فکری در اختیار نداریم. به همین خاطر باید سطح قابل قبولی از اطمینان در خصوص صحت و درستی آن‌ها راضی باشیم.

## ۳-۱. پنج ویژگی سیستم پیچیده

با توجه به طبیعت این پیچیدگی، به این نتیجه می‌رسیم که پنج ویژگی معمول در همه‌ی سیستم‌های پیچیده وجود دارد.

### ساختار سلسله‌مراتبی

بیشتر اوقات، پیچیدگی به شکل سلسله‌مراتبی در خواهد آمد، به استناد آن، سیستم پیچیده از زیرسیستم‌های مرتبط به هم تشکیل‌شده که این زیرسیستم‌ها هم زیرسیستم‌های خود را دارند و الی آخر... تا جایی که به پایین‌ترین سطح مؤلفه‌های پایه برسیم.

سیمون اشاره کرد که : " **حقیقت این است که سیستم‌های پیچیده زیادی قابلیت تجزیه شدن دارند، ساختار سلسله‌مراتبی عامل اصلی آسان‌سازی است که این امکان را می‌دهد بفهمیم، توصیف کنیم و حتی سیستم‌ها و بخش‌های آن‌ها را ببینیم. مسلماً تنها آن سیستم‌هایی را می‌توانیم بفهمیم که ساختار سلسله‌مراتبی دارند.**

**مهم است درک کنیم که معماری یک سیستم پیچیده تابع مؤلفه‌هایش و هم‌چنین روابط سلسله‌مراتبی میان این مؤلفه‌ها است.**" همه سیستم‌ها، زیرسیستم دارند و همه‌ی سیستم‌ها بخش‌هایی از سیستم‌های بزرگ‌تر هستند.

ب: معماری یک سیستم پیچیده، تابعی از کامپونت‌ها و هم‌چنین روابط سلسله‌مراتبی بین آن‌ها است.

### مؤلفه‌های پایه‌ی نسبی

با توجه به ماهیت مؤلفه‌های پایه سیستم پیچیده، تجربه‌ی ما می‌گوید که : انتخاب این‌که کدام مؤلفه‌ی سیستم، پایه‌ی هستند نسبتاً اختیاری است و به‌دقت مشاهده ما از سیستم بستگی دارد. چیزی که برای یک مشاهده‌گر، مؤلفه‌ی اولیه هست، ممکن است برای دیگری، در سطح بالاتری از تجرید باشد.

### تفکیک وظایف

سیمون(simon) سیستم‌های سلسله‌مراتبی را تجزیه‌پذیر می‌نامد چراکه آن‌ها می‌توانند به بخش‌های مشخصی تقسیم شوند. او آن‌ها را تقریباً تجزیه‌پذیر می‌نامد چون‌که بخش‌های آن‌ها کاملاً مستقل نیستند. این مطالب ما را به سمت ویژگی مشترک دیگری در سیستم‌های پیچیده هدایت می‌کند.

معمولاً ارتباطات درون مولفه‌ای از ارتباطات بین مولفه‌ای سخت‌تر است. این واقعیت نتیجه جداسازی پویایی با فراوانی بالا این تفاوت بین ارتباطات درون مولفه‌ای و بین مولفه‌ای یک تفکیک وظایفی میان بخش-های مختلف سیستم فراهم می‌کند، به‌طوری‌که امکان بررسی هر بخش به‌تنهایی وجود دارد.

### الگوهای مشخص

همان‌طور که بحث کردیم، بسیاری از سیستم‌های پیچیده به‌اصطلاح مقرون‌به‌صرفه ( economy of exoression) انجام شده‌اند. سایمون می‌گوید:

" معمولاً سیستم‌های سلسله‌مراتبی تنها از تعدادی از انواع مختلف زیرسیستم با ترکیب و چیدمان متنوع ساخته‌شده‌اند." به‌عبارت‌دیگر، سیستم‌های پیچیده الگوهای رایجی دارند. ممکن است این الگوها شامل

استفاده‌ی مجدد از مؤلفه‌های کوچک مثل سلول که هم در حیوانات و هم در گیاهان یافت شده‌اند و یا ساختارهای بزرگ‌تر مثل سیستم‌های عروقی که باز هم در هر دو وجود دارند، باشند.

## ۴-۱. شکل رایج سیستم پیچیده

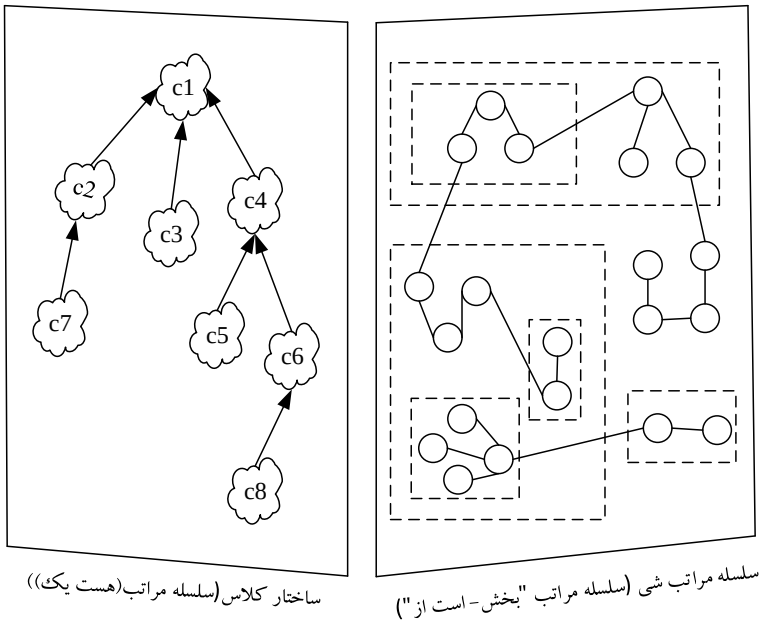
واژه‌ی سلسله‌مراتب را به طرز غیرحقیقی استفاده می‌کردیم. بسیاری از سیستم‌ها این‌طور نیست که فقط یک سلسله‌مراتب داشته باشند. در عوض دریافتیم که معمولاً سلسله‌مراتب‌های مختلف زیادی درون همان سیستم وجود دارند. برای مثال، ممکن است برای بررسی یک هواپیما آن را به سیستم پیش رانش آن و سیستم کنترل پرداز و... تجزیه کنیم. این تجزیه حاکی از یک سلسله‌مراتب ساختاری یا **" part of "** (**بخشی-از**) دارد.

متناوباً، می‌توانیم سیستم را به روش کاملاً عمودی تقسیم کنیم. برای مثال، موتور توربوفن نوع خاصی از موتور جت است و prat whitney TF30 یک نوع خاصی از موتور توربوفن هست. به‌بیان‌دیگر، موتور جت نماینده‌ی کلیت ویژگی‌های مشترک در همه‌ی انواع جت‌ها است؛ موتور توربوفن فقط یک نوع تخصصی است از موتورهای جت با ویژگی‌هایی که آن را مثلاً از موتورهای رم جت متمایز می‌کند.

این نوع دوم به سلسله‌مراتب **"is-a"** (هست-یک) اشاره دارد. در تجاربمان دریافتیم که لازم است یک سیستم را از هر دو جنبه بررسی کنیم، به سلسله‌مراتب "is-a" به‌اندازه‌ی سلسله‌ی مراتب "part of" توجه شود. به دلایلی که در ادامه خواهید دید سلسله‌مراتب‌ها را به ترتیب ساختار کلاس و ساختار شیء می‌نامیم.

اجازه دهید برای کسانی که با تکنولوژی شیء آشنا هستند، روشن کنیم. در این مورد، هرجایی که در مورد ساختار کلاس و ساختار شیء صحبت می‌کنیم منظورمان کلاس‌ها و اشیاء که به هنگام کد نویسی نرم‌افزارمان ایجاد می‌کنیم نیست. منظورمان کلاس‌ها و اشیاء در یک سطح انتزاع بالاتر است. مثلاً یک موتور جت، بدنه‌ی هواپیما، انواع مختلف صندلی، زیرسیستم خلبان خودکار و الی آخر... از بحث‌های اخیری که در مورد ویژگی‌های سیستم‌های پیچیده داشتیم به یاد دارید که مؤلفه‌های پایه به مشاهده‌گر بستگی دارد.

در شکل ۱-۱ دو سلسله‌مراتب عمومی از سیستم را می‌بینیم. هرکدام از این سلسله‌مراتب‌ها با انتزاعی‌ترین کلاس‌ها و اشیای که بر روی مؤلفه‌های اولیه‌ی شان ساخته‌شده بودند، لایه‌بندی شده است. این‌که کدام کلاس و شیء به‌عنوان اولیه شناخته شود، نسبی است. با بررسی سطح مشخص، سطح دیگری از پیچیدگی آشکار می‌شود. مخصوصاً بین بخش‌های ساختار شیء، همکاری نزدیکی بین اشیاء همان سطح وجود دارد.



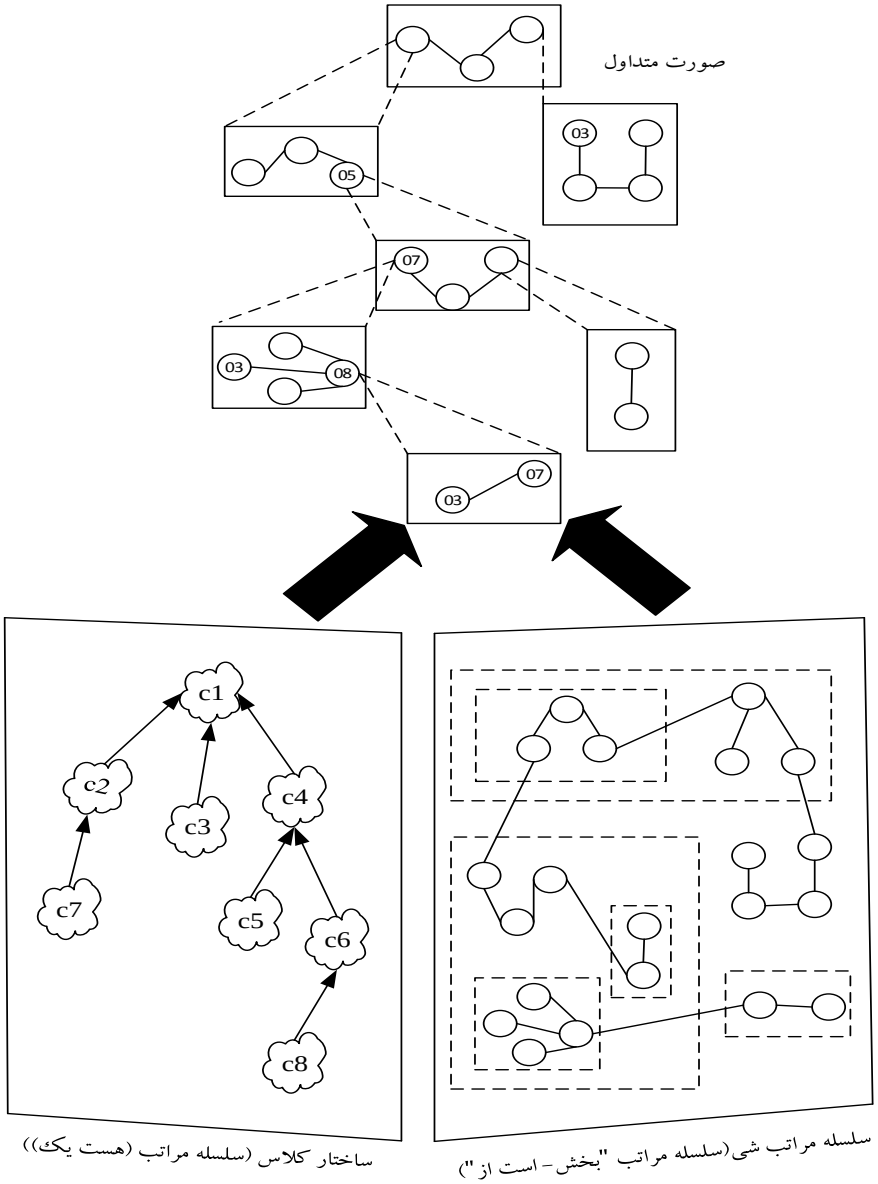

شکل ۱-۱ سلسله‌مراتب‌های کلیدی سیستم‌های پیچیده.

با ترکیب مفاهیم کلاس و شیء یا پنج ویژگی یک سیستم پیچیده دریافتیم که تقریباً همه‌ی سیستم‌های پیچیده یک قالب رایجی را می‌پذیرند، همان‌طور که در شکل ۲-۱ نشان داده‌شده است. مجموعاً ساختار کلاس و شیء در یک سیستم به‌عنوان معماری آن محسوب می‌شود.

همچنین توجه کنید که ساختار کلاس و شیء به‌طور کامل مستقل نیستند. به بیان دقیق‌تر، هر شیء در ساختار شیء، به معنی یک نمونه خاص از تعدادی کلاس است (در شکل۲-۱، به کلاس‌های $c_3$، $c_5$، $c_7$، $c_8$ و تعدادی نمونه o3، o5، o7، o8 توجه کنید) . همان‌طور که در شکل می‌بینید درون یک سیستم پیچیده، معمولاً اشیاء بسیار زیادی از انواع اشیاء وجود دارد. با نمایش سلسله‌مراتب‌های "is-a" و "part-of"، افزونگی سیستم را با دقت نشان دادیم. اگر ساختار کلاس سیستم را نداشته باشیم مجبوریم اطلاعات مربوط به هر کلاس مشخص را تکرار کنیم. باوجود ساختار کلاس، این صفات مشترک در یکجا ثبت می‌شود.

از همان ساختار کلاس، روش‌های مختلفی برای مرتب کردن اشیاء وجود دارد. درواقع هیچ معماری خاصی نمی‌تواند "درست" تلقی شود. این همان چیزی است که معماری سیستم را دشوار می‌کند – پیدا کردن تعادل بین روش‌های مختلفی که مؤلفه‌های سیستم می‌توانند سازمان پیدا کند. پنج ویژگی سیستم‌های پیچیده و نیازهای کاربران سیستم، تجربه به ما نشان داده است که بیش‌تر سیستم‌های نرم‌افزاری موفق آن‌هایی هستند که طراحی‌شان شامل ساختارهای کلاس و شیء خوب مهندسی‌شده باشند و متضمن پنج ویژگی سیستم‌های پیچیده که در بخش‌های قبلی عنوان شد، باشند. مبادا اهمیت این نظر نادیده گرفته شود، اجازه دهید که صریح باشیم : خیلی به‌ندرت با سیستم‌های نرم‌افزاری که سر وقت تحویل داده می‌شوند یا این‌که در محدوده بودجه

هزینه صرف می‌کنند و یا این‌که نیازمندی‌های‌شان برآورده شوند، مواجه می‌شویم مگر این‌که با نکاتی که گفتیم طراحی‌شده باشند.




شکل ۲-۱. صورت متداول یک سیستم پیچیده.

### محدودیت‌های توانایی انسان در تعامل با پیچیدگی

اگر می‌دانیم که، طراحی سیستم‌های نرم‌افزاری پیچیده باید شبیه چه چیزی باشد پس چرا هنوز مسائل جدی در توسعه‌ی موفق آن‌ها داریم؟ این مفهوم پیچیدگی سازمان‌یافته‌ی نرم‌افزار نسبتاً جدید است. بااین‌وجود

هنوز عامل دیگری وجود دارد که سایه افکنده است: محدودیت‌های اساسی توانایی انسان برای ارتباط با پیچیدگی.

زمانی که شروع به تحلیل سیستم‌های نرم‌افزاری می‌کنیم، بخش‌های زیادی را می‌یابیم که باید به روش‌های پیچیده زیادی و با اشتراکات قابل‌ملاحظه‌ای که چه در بخش‌ها و چه در ارتباطات وجود دارند باهم ارتباط برقرار کنند. زمانی که کاری می‌کنیم تا از طریق فرآیند طراحی موجب سازمان‌دهی این پیچیدگی شویم باید هم‌زمان در مورد همه‌چیز فکر کنیم. برای مثال، در سیستم کنترل ترافیک هوایی، باید هم‌زمان حالت هواپیماهای مختلف را اداره کنیم، شامل ویژگی‌های چون مکان، سرعت، مسیر. مخصوصاً در مورد سیستم‌های گسسته باید با فضای حالت نسبتاً بزرگ، پیچیده و غیرقطعی کار کنیم. متأسفانه کاملاً غیرممکن است کسی بتواند این همه جزییات را هم‌زمان اداره کند. تجربه روانشناسان، مانند مایلر می‌گوید: بیش‌ترین تعداد قطعه اطلاعاتی که یک شخص می‌تواند هم‌زمان بفهمد $7 \pm 2$ قطعه است. این ناشی از ظرفیت حافظه‌ی کوتاه‌مدت انسان است. سایمون نشان داد که سرعت پردازش یک عامل بازدارنده است: ذهن در حدود پنج ثانیه زمان می‌برد تا یک قطعه اطلاعات جدید را بپذیرد.

بنابراین، با یک وضعیت دشوار مواجه‌ایم. یک طرف پیچیدگی سیستم‌های نرم‌افزاری که قصد توسعه‌ی آن را داریم سیر افزایشی دارند. ولی هنوز محدودیت‌های بنیادی در توانایی ما با کنار آمدن با این پیچیدگی وجود دارد. چطور این مشکل را باید حل کرد؟

### نقش تجزیه[1]

تکنیک مهار کردن پیچیدگی از زمان‌های قدیم شناخته‌شده است : از تقسیم امپراطوری (رده‌بندی کردن سیستم حکم‌رانی). زمان طراحی یک سیستم نرم‌افزاری پیچیده، این ضروری است که سیستم را به بخش‌های کوچک‌تر و کوچک‌تر تجزیه کنیم، به‌طوری‌که بتوانیم هرکدامشان را مستقلاً اصلاح کنیم. با این روش، خیلی از محدودیت‌هایی که در قدرت ادراک انسان وجود دارد را رفع می‌کنیم: با فهم هر سطح از سیستم فقط باید تعداد کمی از بخش‌ها (نه کل بخش‌های سیستم) را هم‌زمان بفهمیم. تجزیه هوشمندانه با تقسیم‌بندی فضای حالت سیستم، به پیچیدگی ذاتی نرم‌افزار می‌پردازد.

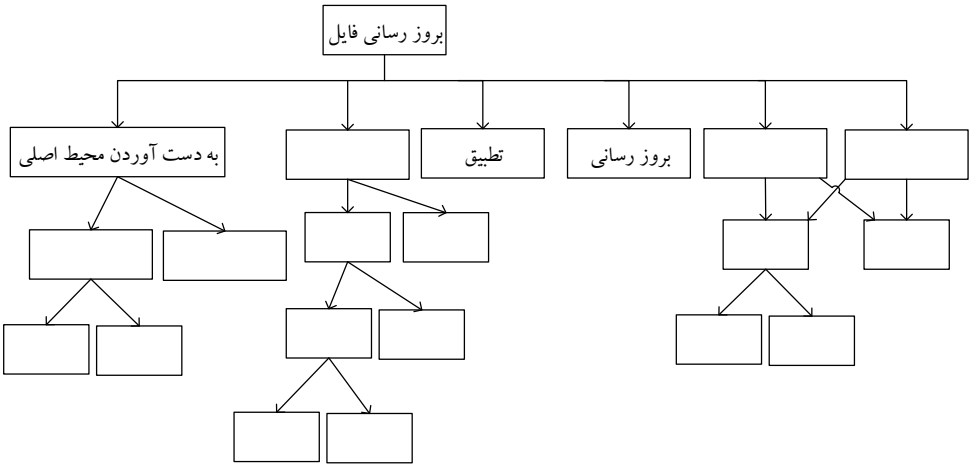
### تجزیه الگوریتمی[2]

به بیش‌تر ما، طراحی ساخت‌یافته بالا به پایین آموزش داده‌شده است و تجزیه ساده‌شده را تجزیه الگوریتمی می‌دانیم، که در آن هرکدام از ماژول‌های سیستم به معنی یک عمل اصلی در نرم‌افزار کلی است.

شکل ۳-۱ یک نمونه از تولیدات طراحی ساخت‌یافته را نشان می‌دهد. یک نمودار ساختار، که روابط بین واحدهای عملیاتی را نشان می‌دهد. این نمودار ساختار خاص، بخش‌های طراحی یک برنامه‌ای که محتویات یک فایل اصلی را بروز می‌کند را شرح می‌دهد. نمودار به‌طور اتوماتیک از دیاگرام جریان داده که به‌وسیله‌ی ابزار سیستم خبره که قواعد طراحی ساخت‌یافته در آن گنجانده تولید می‌شود.

---

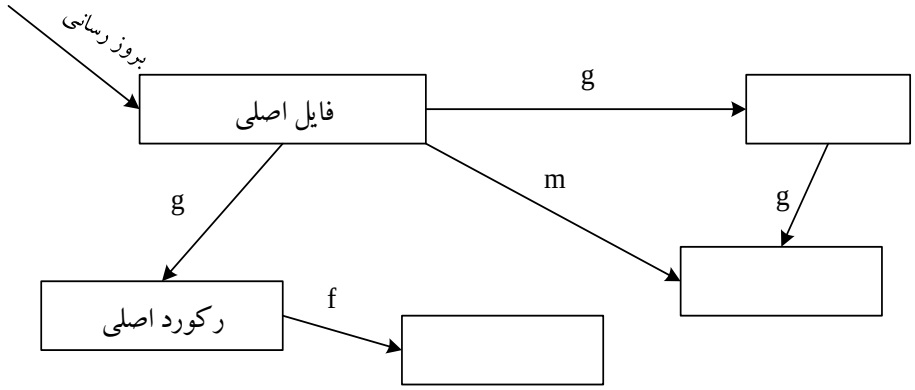
[1]. Decomposition [2]. Algorithmic Decomposition




**شکل ۳-۱. تجزیه‌ی الگوریتمی**

### تجزیه شیءگرا[1]:

می‌گوییم که یک تجزیه دیگری برای همان مسئله وجود دارد. در شکل ۴-۱ سیستم را بر طبق انتزاع‌های کلیدی محدوده مسئله تجزیه کرده‌ایم. به‌جای این‌که مسئله را به گام‌هایی مثل Get fomated update و Add checksum تجزیه کنیم، اشیایی مثل master file و checksum را شناسایی کرده‌ایم که مستقیماً از واژه‌های محدوده‌ی مسئله به دست آمدند.




**۴-۱. تجزیه‌ی شیءگرا**

اگرچه هر دو طراحی، مسئله را حل می‌کنند آن‌ها این کار را با روش‌های کاملاً متفاوت انجام می‌دهند. در این دو تجزیه جهان را به‌عنوان یک مجموعه عامل‌های مستقل می‌بینیم که باهم برای اجرای یک رفتار سطح بالاتر همکاری می‌کنند. Getformatedupdate به‌عنوان یک الگوریتم مستقل نیست بلکه عملیاتی است که به شیء fileofupdates پیوند می‌خورد. فراخوانی این عملیات شیء دیگری را به نام updatetocard را به وجود

[1]. Object-oriented Decomposition

می‌آورد. از این طریق، هر شیء راه حل رفتار منحصربه‌فرد خودش را بیان می‌کند و هرکدام تعدادی از اشیاء دنیای واقعی را مدل می‌کنند. از این دید، اشیاء کاری را انجام می‌دهند، و با ارسال پیام از آن‌ها می‌خواهیم که آن کار را برای‌مان انجام دهند. چون تجزیه‌مان بر اساس اشیاء صورت گرفته است و نه الگوریتم. آن را یک تجزیه شیءگرا می‌نامیم.

**تجزیه الگوریتمی در مقابل تجزیه شیءگرا:**

کدام روش تجزیه کردن درست است. الگوریتمی یا شیءگرا؟ درواقع این سؤال شوخی است چراکه جواب درست این است که هر دو دیدگاه مهم هستند: دیدگاه الگوریتمی، ترتیب رویدادها را برجسته می‌کند و دیدگاه تجزیه شیءگرا، بر عامل‌ها، چه آن‌هایی که موجب عملی می‌شوند و چه آن‌هایی که تابعی از یک عمل انجام‌شده هستند، تکیه دارد.

**طبقه‌بندی متدهای تحلیل و طراحی**

تمایز بین واژه‌های متد[1] و متدولوژی[2] می‌تواند مفید باشد. متد یک روال منظمی است برای خلق مجموعه‌ای از مدل‌هایی که جنبه‌های مختلف سیستم‌های نرم‌افزاری تحت توسعه را توصیف می‌کند و با استفاده از تعدادی علائم خوش- تعریف نشان داده می‌شوند. یک متدولوژی مجموعه‌ای از متدهای به کار گرفته‌شده در چرخه‌ی توسعه نرم‌افزار است که به‌وسیله فرآیندها، روال‌ها و روش‌های فیلسوفانه و کلی یکی شده‌اند. متدها به سه دلیل اهمیت دارند. مهم‌ترین آن‌ها این است که متدها به توسعه‌ی سیستم‌های نرم‌افزاری پیچیده نظم می‌بخشند. آن‌ها محصولاتی را تعریف می‌کنند که به‌عنوان یک ابزار مشترک برای ارتباط میان اعضای تیم توسعه‌دهنده به کار می‌رود.

متدها در پاسخ به رشد پیچیدگی سیستم‌های نرم‌افزاری به وجود آمده‌اند. در روزهای نخستین حرفه‌ی کامپیوتر، کسی برنامه‌های بزرگ نمی‌نوشت چراکه توانایی ماشین‌های‌مان خیلی محدود بودند. مهم‌ترین محدودیت‌ها در ساخت سیستم‌ها، عمدتاً ناشی از سخت‌افزار بودند : ماشین‌ها حافظه اصلی بسیار کمی داشتند، برنامه‌ها می‌بایست با تأخیر بسیار زیاد دستگاه‌های حافظه ثانویه مثل درام‌های مغناطیسی کنار بی‌آیند، پردازشگرها زمان سیکل به اندازه صدها میکرو ثانیه داشتند. در سال‌های ۱۹۶۰ تا ۱۹۷۰، اقتصاد حرفه کامپیوتر شروع به تغییرات چشم‌گیری کرد، چون قیمت سخت‌افزار تنزل یافت و قابلیت‌های کامپیوتر رشد کرد. به‌عنوان نتیجه، این بهتر بود و اکنون عاقبت اقتصادی شدن این شد که برنامه‌هایی با پیچیدگی بیش‌تری ماشینی شده‌اند. زبان‌های برنامه‌نویسی سطح بالاتری به‌عنوان ابزارهای مهم وارد صحنه شدند. زبان‌هایی که بهره‌وری فردی توسعه‌دهنده و کل تیم را بهبود بخشیدند شگفت‌آورتر آن‌که ما را تحت‌فشار گذاشت تا سیستم‌های نرم‌افزاری با پیچیدگی‌های بیش‌تری بسازیم. متدهای طراحی زیادی در طول سال‌های ۱۹۶۰ تا ۱۹۷۰ پیشنهاد شد که به این پیچیدگی درحال رشد پرداختند. مهم‌ترین آن‌ها طراحی ساخت‌یافته‌ی بالا به پایین بود که به طراحی

---

[1]. Method [2]. methodology

مرکب شناخته می‌شد. این متد بر روی توپولوژی زبان‌های سطح بالای قدیمی مثل فورترن و کوبول اثر گذاشته بود. در این زبان‌ها، واحدهای اولیه تجزیه، زیر برنامه است و برنامه نهایی شکل درخت به خود می‌گیرد به‌طوری‌که زیر برنامه‌ها با فراخوانی زیر برنامه‌های دیگر، کارشان را انجام می‌دهند. این دقیقاً همان نگاه طراحی ساخت‌یافته بالا به پایین است. با اعمال تجزیه الگوریتمی یک مسئله‌ی بزرگ به گام‌های کوچک می‌شکند.

از ۱۹۶۰ تا ۱۹۷۰ تابه‌حال، کامپیوترهایی با قابلیت‌های بسیار زیاد به وجود آمدند. ارزش طراحی ساخت‌یافته تغییری نکرد اما همان‌طور که stein گفت : برنامه‌نویسی ساخت‌یافته برای زمانی که برنامه‌های کاربردی بیش از ۱۰۰/۰۰۰ خط کد داشته باشند به کار می‌رود تا آن را خرد کند. چندین و چند متد طراحی پیشنهاد شد، بیش‌تر آن‌ها برای اداره کردن نقاط ضعف طراحی ساخت‌یافته‌ی بالا به پایین ابداع شدند. sommerrille می‌گوید، بیش‌تر متدها می‌توانند با یکی از سه نوع زیر طبقه‌بندی شوند:

- طراحی ساخت‌یافته بالا به پایین[1]
- طراحی مبتنی بر داده[2]
- طراحی شیءگرا[3]

طراحی ساخت‌یافته به موضوع تجرید داده و پنهان‌سازی اطلاعات نمی‌پردازد. طراحی ساخت‌یافته، برای سیستم‌های بی‌اندازه پیچیده اثر خوبی ندارد و این متد عمدتاً برای استفاده در زبان‌های برنامه‌نویسی شیءگرا و شیء محور مناسب نیست.

طراحی مبتنی بر داده، با نگاشت ورودی‌های سیستم به خروجی‌ها، ساختار یک سیستم نرم‌افزاری را می‌سازد. مانند طراحی ساخت‌یافته، طراحی مبتنی بر داده به‌طور موفقیت‌آمیزی در تعدادی از زمینه‌های پیچیده به کار گرفته‌شده است مخصوصاً سیستم‌های مدیریت اطلاعات که این شامل رابطه مستقیم بین ورودی‌ها و خروجی‌های سیستم است، اما مستلزم ارتباط ناچیز با رویدادهای بحران زمانی است.

مفهوم زیر بنایی تحلیل شیءگرا این است که یک شخص باید سیستم‌های نرم‌افزاری را به‌عنوان مجموعه‌ای از اشیاء هماهنگ مدل‌سازی کند، و اشیاء خاص را به‌عنوان نمونه‌ای از یک کلاس فرض کند. بازتاب تحلیل و طراحی شیءگرا را می‌توان مستقیماً در توپولوژی زبان‌های برنامه‌نویسی سطح بالا مثل small talk، object pascal، c++، clos، Ada، Eiffcl، python، visual c# و java دید.

هرچند، واقعیت می‌گوید که نمی‌توانیم یک سیستم پیچیده را هم‌زمان با هر دو این روش‌ها بسازیم. برای این‌که هر دو دیدگاه عمودی دارند. باید تجزیه یک سیستم را با یک روش چه الگوریتمی و چه اشیاء شروع کنیم و آن‌وقت از ساختار حاصل‌شده به‌عنوان framework برای نشان دادن دیدگاه دیگر استفاده کنیم. تجربه نشان می‌دهد که اول باید روش شیءگرا را به کار بگیریم چراکه این دیدگاه در سازمان دادن به پیچیدگی ذاتی سیستم‌های نرم‌افزاری کمک بهتری می‌کنند. همان‌طور که این دیدگاه به ما در توصیف پیچیدگی سازمان‌یافته سیستم‌های پیچیده‌ی گوناگون مثل کامپیوترها، گیاهان، کهکشان‌ها و جوامع اجتماعی بزرگ

---

[1]. Top-Down structured design [2].Data-driven design [3]. Object-oriented design

کمک کرد. هرچند در ادامه بیش‌تر بحث خواهیم کرد، تجزیه شیء‌گرا مزیت‌های بسیار چشم‌گیری نسبت به تجزیه الگوریتمی دارد. تجزیه شیء‌گرا، سیستم‌های کوچک‌تر را از طریق استفاده مجدد مکانیسم‌های مشترک حاصل می‌سازد که مقرون‌به‌صرفه است. سیستم‌های شیء‌گرا انعطاف‌پذیری بیش‌تری برای تغییرات دارند و بنابراین بهتر قادرند در طول زمان رشد و تکامل یابند چون‌که طراحی‌شان بر فرم‌های میانی پایدار( stable intermediate forms ) استوار شده است. از طرفی تجزیه شیء‌گرا ریسک ساخت سیستم‌های پیچیده را به‌شدت کاهش می‌دهد چون‌که آن‌ها طوری طراحی‌شده‌اند که از سیستم‌های کوچک‌تری که تاکنون به آن اطمینان داشتیم به‌صورت افزایشی تکامل یابند. از این گذشته، تجزیه شیء‌گرا، با کمک کردن به ما در تصمیم‌گیری راجع به جداسازی ارتباطات در یک فضای حالت بزرگ، به موضوع پیچیدگی سیستم می‌پردازد.

**نقش انتزاع یا تجرید[1] (The Role of abstraction)**

پیش‌تر از تجربه‌ی مایلر گفتیم، از این‌که او نتیجه گرفته بود که یک فرد در یک‌لحظه فقط در حدود $7 \pm 2$ قطعه اطلاعات را می‌تواند درک کند. این عدد مستقل از محتوا و مضمون اطلاعات است. همان‌طور که مایلر گفت: گستره‌ی قضاوت مطلق و ظرفیت حافظه بلافصل محدودیت‌های سختی بر مقدار اطلاعاتی که قادریم دریافت کنیم، پردازش کنیم و به خاطر بسپاریم، تحمیل می‌کند. با سازمان‌دهی ورودی محرک به‌طور هم‌زمان و با ابعاد مختلف و به‌طور پی‌درپی و با زنجیره‌ای از قطعات، موفق به شکستن مسئله می‌شویم... این تنگنا اطلاعاتی... به‌اصطلاح جدید، به این فرآیند **تقطیع[2] یا تجرید** می‌گوییم.

همان‌طور که wulf *توصیف کرد:* " ما (انسان‌ها) تکنیک فوق‌العاده قدرتمند برای رفتار با پیچیدگی ایجاد کردیم. آن را خلاصه می‌کنیم. نمی‌توانیم کل یک شیء پیچیده را خلاصه کنیم، ترجیح می‌دهیم از جزییات غیرضروری آن چشم‌پوشی کنیم و در عوض به کلیات و مدل آرمانی آن می‌پردازیم.

برای مثال، زمانی که به بررسی فتوسنتز در گیاه می‌پردازیم، باید بر روی واکنش‌های شیمیایی در سلول‌های خاصی در برگ تمرکز کنیم و بخش‌های دیگر گیاه مثل ریشه‌ها و ساقه‌ها را نادیده بگیریم. بااین‌حال هنوز هم توسط تعداد چیزهایی که هم‌زمان می‌توانیم درک کنیم تحت‌فشاریم، اما از طریق تجرید، از قطعه‌های اطلاعات با محتوای معنایی بزرگ استفاده می‌کنیم. درست است اگر دیدی شیء‌گرا داشته باشیم چراکه اشیاء به‌عنوان تجریدی از موجودیت‌های دنیای واقع، نمایانگر گروهی به‌هم‌پیوسته و منسجم از اطلاعات است.

**نقش سلسله‌مراتب**

روش دیگر برای زیاد کردن محتوای معنایی قطعات جداگانه اطلاعات این است که سلسله‌مراتب‌های شیء و کلاس موجود در سیستم نرم‌افزاری پیچیده را بشناسیم. ساختار شیء اهمیت دارد چراکه توضیح می‌دهد چگونه اشیاء مختلف از طریق الگوهای ارتباطی که به آن مکانیزم می‌گوییم باهم همکاری می‌کنند. ساختار کلاس هم به همان اندازه مهم است چراکه رفتار و ساختار مشترک درون سیستم را برجسته می‌کند. بنابراین به‌جای بررسی فتوسنتز تک‌تک سلول‌های برگ یک گیاه، کافی است یک سلول را بررسی کنیم چون انتظار

---

[1]. Abstraction [2]. Chunking

داریم همه سلول‌های دیگر رفتار مشابهی را بروز دهند. اگرچه فرض می‌کنیم که هر نمونه از یک نوع مشخص شیء، متمایز است، اما می‌توان فرض کرد که رفتار یکسانی را با همه‌ی نمونه‌های دیگر آن نوع شیء سهیم باشد. با دسته‌بندی اشیاء به گروه‌هایی از تجریدهای مرتبط به هم (مثل انواع سلول گیاهان در برابر سلول حیوانات) ویژگی‌های مشترک و متمایز اشیاء مختلف را از هم تشخیص می‌دهیم، و کمک می‌کند که پیچیدگی آن را کنترل کنیم.

شناسایی سلسله‌مراتب‌های درون یک سیستم نرم‌افزاری پیچیده کار آسانی نیست چون نیاز دارد که الگوهای بین اشیاء زیادی کشف شود که هرکدام ممکن است تعدادی رفتار بی‌اندازه پیچیده را شامل شود. به‌محض این‌که این سلسله‌مراتب‌ها نشان داده می‌شود، باوجود ساختار پیچیده آن و فهم ما از آن، همه‌چیز ساده و آسان می‌شود.

## ۵– ۱. طراحی سیستم‌های پیچیده

شیوه‌ی کار همه‌ی رشته‌های مهندسی- چه مهندسی عمران، مکانیک، شیمی، برق و یا مهندسی نرم‌افزار باشد- شامل هر دو عنصر هنر و علم است. همان‌طور که petroski بیان کرد، "مفهوم یک نقشه و طرح برای سازه‌ی جدید می‌تواند شامل تا اندازه‌ی تخیل و تااندازه‌ای تلفیقی از علم و تجربه و به‌مانند هر هنرمندی برای شروع به کاغذ یا بوم نقاشی نیاز دارد و وقتی‌که طرح توسط مهندس (در حکم هنرمند) بیان می‌شود، باید توسط مهندس(در حکم دانشمند) با یک برنامه بسیار دقیق از متدهای علمی تحلیل شود به‌مانند کاری که همه-ی دانشمندان می‌کنند. سختی برنامه‌نویسی استفاده گسترده از تجرید است و بنابراین به ترکیبی از توانایی‌های ریاضیدان و نگرش مهندس کاردان نیاز دارد.

### مهندسی به‌عنوان یک علم و یک هنر

نقش مهندس طراح، زمانی که بخواهد یک سیستم کاملاً جدید را طراحی کند، سخت است. مخصوصاً در مورد سیستم‌های واکنشی و سیستم‌هایی برای زمان و کنترل، اغلب می‌خواهیم نرم‌افزار را برای یک سری از نیازمندی‌های کاملاً منحصربه‌فرد بنویسیم و بتواند بر روی پردازنده‌هایی که مخصوصاً برای این سیستم ساخته‌شده است، اجرا شود. در موارد دیگر، مثل ایجاد چارچوب (framework)، ابزارهایی برای تحقیقات در زمینه هوش مصنوعی، سیستم‌های مدیریت اطلاعات، ممکن است که یک محیط هدف پایدار و هوش تعریفی داشته باشیم اما ممکن است نیازمندی‌های‌مان، تکنولوژی نرم‌افزار را از یک بعد و یا بیش‌تر تحت‌فشار قرار دهد. برای مثال، ممکن است بخواهیم که سیستم هواپیماهایی که سریع‌تر هستند، ظرفیت بیش‌تری داشته باشد و یا از ریشه کارکرد آن را بهبود ببخشیم. در همه این وضعیت‌ها، باید از مکانیسم‌ها و تجریدهای آزمایش‌شده و ثابت‌شده (به گفته سایمون فرم‌های میانی پایدار (stable intermediate forms)) به‌عنوان اساس ساخت سیستم‌های جدید استفاده کنیم. باوجود یک کتابخانه‌ی بزرگ از قطعات نرم‌افزاری قابل‌استفاده‌ی مجدد، مهندسین نرم‌افزار باید این بخش‌ها را با روش‌هایی مبتکرانه روی هم سوار کنند تا نیازمندی‌های ضمنی و معین برآورده شود، درست مثل نقاش و یا موسیقیدان باید محدودیت‌های وسیله‌اش را نادیده بگیرد.

# فصل ۲ مدل شیء

تکنولوژی شیءگرایی بر پایه‌ی مهندسی دقیقی ساخته شده است، اجزایش را مجموعاً **مدل شیء توسعه** یا ساده‌تر بگوئیم **مدل شیء**[1] می‌نامیم. مدل شیء شامل اصول انتزاع، بسته‌بندی (مخفی‌سازی)، پیمانه‌بندی، سلسله‌مراتبی، نوع‌گذاری[2]، هم‌روندی، ماندگاری است. هیچ کدام از این اصول، به تنهایی جدید نیستند.

شکی نیست که تحلیل و طراحی شیءگرا، اساساً با رویکردهای طراحی ساخت‌یافته‌ی قدیمی متفاوت است:

تحلیل و طراحی شیءگرا، به طرز فکر متفاوتی درزمینه‌ی تجزیه نیاز دارد و نوعی از معماری نرم‌افزار را ارائه می‌کند که عمدتاً خارج از حوزه‌ی فرهنگ طراحی **ساخت‌یافته**[3] است.

## ۱-۲. برنامه‌نویسی شیءگرا

برنامه‌نویسی شیءگرا یک متد پیاده‌سازی است که در آن برنامه‌ها به‌صورت مجموعه‌هایی از اشیاء هستند که با هم همکاری دارند، هر کدام از آن‌ها ثمره‌ی یک نمونه از تعدادی کلاس است و کلاس‌هایشان اعضایی از سلسله‌مراتب کلاس‌ها هستند که از طریق روابط وراثت متحد شدند.

سه بخش مهم در این تعریف وجود دارد که عبارت‌اند از:

۱) برنامه‌نویسی شیءگرا از اشیاء به‌عنوان اجزای منطقی پایه استفاده می‌کنند نه الگوریتم (سلسله‌مراتب "Part - of").

۲) هر شیء یک نمونه از تعدادی کلاس است.

۳) کلاس‌ها ممکن است از طریق روابط وراثت با هم‌دیگر ارتباط برقرار کنند (سلسله‌مراتب "is-a").

یک برنامه ممکن است شیءگرا به نظر رسد، اما اگر هر کدام از سه مورد بالا را رعایت نکرده باشد، دیگر برنامه‌ی شیءگرا نخواهد بود. به‌ویژه، برنامه‌نویسی بدون وراثت مشخصاً شیءگرا نیست و صرفاً برنامه‌نویسی، با انواع داده انتزاعی است. با این تعریف، بعضی از زبان‌ها شیءگرا هستند و برخی دیگر نیستند. Stroustrup می‌گوید که "زبان شیءگرا" هر معنی داشته باشد، باید به معنای زبانی باشد که مکانیزم‌هایی دارد که سبک شیءگرایی برنامه‌نویسی را به خوبی پشتیبانی نمی‌کند. یک زبان، هر سبک برنامه‌نویسی را به خوبی پشتیبانی می‌کند اگر آن زبان امکاناتی را فراهم کند که آن را برای استفاده از آن سبک مناسب سازد. یک زبان تکنیکی را پشتیبانی نمی‌کند که برای نوشتن برنامه‌ها تلاش و مهارت استثنایی را بطلبد. در این صورت، زبان صرفاً برنامه‌ساز را قادر به استفاده از تکنیک‌ها می‌سازد. هر کسی می‌تواند برنامه‌نویسی شیءگرا را در زبان‌های غیر شیءگرایی مثل پاسکال یا حتی کوبول یا اسمبلی سر هم کند، اما انجام چنین کاری بی‌فایده است.

---

[1]. Object model [2]. Typing [3]. Structured design

Cardelli و Wegner می‌گویند: یک زبان تنها در صورتی شیءگرا است که نیازمندی‌های زیر را برآورده سازد:

- آن زبان اشیایی را پشتیبانی کند که انتزاع‌های داده همراه با ربطی به نام رابط عملیات و یک حالت فعلی پنهان هستند.
- اشیاء یک نوع (کلاس) تابعه دارند.
- انواع (کلاس‌ها) ممکن است صفاتی را از نوع بالاتر (کلاس بالاتر) به ارث ببرند.

در زبانی که از وراثت پشتیبانی می‌کند می‌توان رابطه‌های " is-a " میان انواع را نشان داد. برای مثال، رز قرمز نوعی گل است و یک گل یک نوع گیاه است. اگر یک زبان پشتیبانی مستقیمی از وراثت فراهم نکند، پس شیءگرا نیست. Cardelli و Wegner چنین زبان‌هایی را تمیز می‌دهند و به جای شیءگرا به آن‌ها **مبتنی بر شیء** می‌گویند. طبق این تعریف، Java, C#, CLOS, Eiffel, C++, Object pascal, small talk همگی شیءگرا هستند و Ada 83 مبتنی بر شیء است (پشتیبانی شیءگرایی در Ada 95 اضافه شد).

## ۲-۲. طراحی شیءگرا

تأکید بر متدهای برنامه‌نویسی، در درجه‌ی اول بر روی استفاده‌ی درست و مؤثر از مکانیسم‌های خاص زبان است. در مقابل متدهای طراحی بر ساخت درست و مؤثر سیستم پیچیده تأکید دارد. طراحی شیءگرا (OOD) چیست؟ می‌گوییم:

**طراحی شیءگرا یک متد طراحی است که دربرگیرنده‌ی فرآیند تجزیه‌ی شیءگرا و نمادگذاری برای تشریح مدل‌های فیزیکی و منطقی و هم‌چنین مدل‌های ایستا و پویایی از سیستم تحت طراحی می‌باشد.** دو بخش مهم در این تعریف وجود دارد که عبارت‌اند از:

۱) طراحی شیءگرا منجر به تجزیه‌ی شیءگرا می‌شود.

۲) طراحی شیءگرا از نماد گذاری‌های مختلفی برای بیان مدل‌های طراحی منطقی (ساختار کلاس و شیء) و فیزیکی (ساختار فرآیند و ماژول) سیستم، و هم‌چنین جنبه‌های ایستا و پویایی یک سیستم استفاده می‌کند.

پشتیبانی از تجزیه شیءگرا چیزی است که باعث طراحی شیءگرا متفاوت از طراحی ساخت‌یافته می‌شود:

اولی از انتزاع‌های کلاس و شیء برای ساختار منطقی سیستم‌ها استفاده می‌کند، دومی از انتزاع‌های الگوریتمی استفاده می‌کند. برای رجوع به هرگونه متدی که منجر به تجزیه‌ی شیءگرا می‌شود از واژه‌ی طراحی شیءگرا استفاده خواهیم کرد.

## ۳-۲. تحلیل شیءگرا

تحلیل شیءگرا[1] (OOA) بر ساخت مدل‌های دنیای واقعی با استفاده از دید شیءگرایی از جهان تأکید دارد:

---

1. Object- Oriented Analysis (OOA)

تحلیل شیءگرا یک متد تحلیل است که نیازمندی‌ها را از منظر کلاس‌ها و اشیای موجود در واژگان دامنه مسئله موردبررسی قرار می‌دهد.

چگونه OOA، OOD[1]، OOP[2] به هم مرتبط می‌شوند؟ درواقع، تحلیل شیءگرا مدل‌هایی را تأمین می‌کند که از آن برای شروع طراحی شیءگرا استفاده می‌کنیم؛ سپس محصولات طراحی شیءگرا می‌تواند به‌عنوان نقشه‌ای برای پیاده‌سازی کامل یک سیستم با استفاده از متدهای برنامه‌نویسی شیءگرا به کار گرفته شود.

## ۴-۲. اجزای مدل شیء

Glasgow, Jenkins مشاهده کردند که «بیش‌تر برنامه‌نویسان با یک زبان و تنها یک سبک برنامه‌نویسی کار می‌کنند» آن‌ها با الگویی که آن زبان تحمیل می‌کرد برنامه می‌نوشتند. غالباً آن‌ها در معرض راه‌های جایگزینی برای فکر کردن در مورد مسئله نیستند و ازاین‌رو، در درک مزیت‌های انتخاب یک سبک مناسب‌تر برای مسئله در دست بررسی مشکل دارند. Bobrow، Stefik یک سبک برنامه‌نویسی تعریف کردند به‌عنوان «یک روش سازمان‌دهی برنامه‌ها بر اساس برخی از مدل‌های ذهنی مفهومی برنامه‌نویسی و یک زبان مناسب برای امکان این‌که برنامه‌ها در آن سبک کامل نوشته شوند. از این گذشته آن‌ها خاطرنشان کردند که پنج نوع اصلی سبک برنامه‌نویسی وجود دارد، که در زیر همراه با انواع انتزاع‌هایی که آن‌ها به کار می‌برند، آمده‌اند:

۱) رویه گرا (Proce dure- oriented) الگوریتم‌ها
۲) شیءگرا (Object- oriented) کلاس‌ها و اشیاء
۳) منطق‌گرا (Logic- oriented) اهداف، معمولاً به‌صورت حساب گزاره‌ای بیان می‌شوند.
۴) قانون‌گرا (قاعده گرا) (Rule-oriented) قواعد if-Then
۵) محدودیت گرا (Constraint-oriented) روابط ثابت

یک سبک برنامه‌نویسی که برای همه نوع برنامه‌ی کاربردی بهترین باشد، وجود ندارد. برای مثال، برنامه‌نویسی قانون‌گرا برای طراحی پایگاه دانش بهترین گزینه است و برنامه‌نویسی رویه گرا برای طراحی عملیات با محاسبات فشرده و سخت مناسب خواهد بود. از دید ما، سبک شیءگرا برای سطح گسترده‌ای از برنامه‌های کاربردی مناسب است. درواقع، این مدل برنامه‌نویسی به‌عنوان چارچوب معماری انجام وظیفه می‌کند که در آن مدل‌های دیگر را به کار می‌بریم.

هر کدام از این سبک‌های برنامه‌نویسی بر روی چارچوب‌های مفهومی خودشان پایه‌ریزی شده‌اند. هر کدام به طرز فکر متفاوتی نیاز دارند، روش متفاوتی برای فکر کردن در مورد مسئله، چارچوب مفهومی همه‌ی چیزهای شیءگرا، مدل شیء است. چهار عنصر اصلی در این مدل وجود دارد که عبارت‌اند از:

۱) انتزاع[3] ۲) بسته‌بندی[4] ۳) پیمانه‌بندی (ماژولاریتی)[5] ۴) سلسله‌مراتب[6]

منظورمان از اصلی این است که مدلی که هر یک از این عناصر را نداشته باشد، شیءگرا نیست. سه عنصر فرعی در مدل شیء نیز وجود دارد که عبارت‌اند از:

۱) نوع‌گذاری ۲) هم‌زمانی[7] ۳) ماندگاری[8]

---

[1]. Object- oriented Design [2]. Object- Oriented Programming [3]. Abstraction
[4]. Encapsulation [5]. Modularity [6]. Hierarchy [7]. Concurrency [8].Persistence

منظورمان از فرعی این است که هر کدام از این عناصر یک واحد سودمند، اما نه ضروری، در مدل شیء است. بدون این چارچوب مفهومی، احتمالاً با یک زبانی مثل Eiffel, C++, Object Pascal, Small talk یا Ada برنامه‌نویسی می‌کنید اما طرح‌تان در حدّ برنامه‌های کاربردی به زبان FORTRAN، Pascal یا C خواهد بود. قدرت گویایی زبان شیءگرایی که برای پیاده‌سازی استفاده می‌کنید را از دست می‌دهید و یا به عبارتی ضایع می‌کنید. از همه مهم‌تر، به احتمالی زیاد بر پیچیدگی مسئله‌ی موردبررسی کنترلی نداریم.

## ۱-۴-۲. مفهوم انتزاع

انتزاع یکی از روش‌های بنیادین است که برای فائق آمدن بر پیچیدگی به کار می‌رود. Dahl، Di jkstra، Hoare بیان می‌کنند که "انتزاع از شناسایی شباهت‌های بین اشیاء خاص، شرایط یا فرآیندها در دنیای واقعی پدید می‌آید، و بر این شباهت‌ها توجه کنیم و فعلاً تفاوت‌ها را نادیده بگیریم. Shaw انتزاع را به‌عنوان "**یک توصیف ساده‌شده، یا مشخصات، از یک سیستم که بر روی برخی از جزئیات یا ویژگی‌های سیستم تأکید دارد درحالی‌که بقیه را می‌پوشاند. یک انتزاع خوب آن است که بر روی جزئیاتی که برای خواننده یا کاربر مهم هستند تأکید داشته باشد و جزئیاتی که حداقل در آن لحظه، بی‌اهمیت و انحرافی هستند را پنهان کند**" تعریف می‌کند. Navmann, Gray, Berzins پیشنهاد می‌کنند که "**یک مفهوم، واجد شرایط عنوان انتزاع است تنها اگر آن مفهوم بتواند مستقل از مکانیسمی که بالاخره برای تحقق آن استفاده خواهد شد، توصیف بشود، درک شود و تحلیل شود**". با ترکیب این دیدگاه‌های مختلف، انتزاع را به‌صورت زیر تعریف می‌کنیم:

یک انتزاع بر خصوصیات اساسی یک شیء که آن را از انواع دیگر اشیاء متمایز می‌کند، دلالت دارد و بدین ترتیب محدوده‌ی مفهومی مشخصی نسبت به دید بیننده ارائه می‌دهد.

یک انتزاع به بُعد بیرونی[1] یک شیء می‌پردازد و رفتار ذاتی و اساسی یک شیء را از پیاده‌سازی و اجرایش جدا می‌کند.


**انتزاع بر مشخصه‌های اصلی و عمده‌ی شیء نسبت به منظر (دید) بیننده تمرکز دارد.**

[1] .outside view

تعیین مجموعه‌ی درست انتزاع برای یک دامنه‌ی داده‌شده، اصلی‌ترین مسئله در طراحی شیءگرا است. به خاطر اهمیت این موضوع، کل فصل چهار در مورد این موضوع صحبت کردیم.

یک طیفی از انتزاع وجود دارد. از اشیایی که به‌دقت موجودیت‌های دامنه‌ی مسئله را مدل‌سازی می‌کنند تا اشیای که در حقیقت هیچ دلیلی برای وجودشان وجود ندارد، از ضروری‌ترین آن‌ها تا بی‌اهمیت‌ترین آن‌ها، انواع انتزاع شامل موارد زیر هستند:

- **انتزاع موجودیت**[1]: یک شیء که یک مدل مفیدی از موجودیت دامنه‌ی مسئله یا دامنه‌ی راه حل ارائه می‌دهد.
- **انتزاع عمل**[2]: یک شیء که مجموعه‌ی کلی از عملکرد را فراهم می‌کند که هر کدامشان همان نوع تابع را اجرا می‌کنند.
- **انتزاع ماشین مجازی**[3]: عبارت است از مجموعه‌ای از اعمال متکی بر یک سطح پایین‌تر از خود و یا این‌که خود به‌عنوان یک سطح پایین‌تر برای استفاده یک سطح بالاتر مورداستفاده قرار می‌گیرند.
- **انتزاع همسان**[4]: یک شیء که مجموعه‌ای از عملکردهایی که هیچ ارتباطی با هم دیگر ندارند را بسته‌بندی می‌کند.

در تلاش هستیم که انتزاع‌های موجودیت ایجاد کنیم چون‌که آن‌ها مستقیماً با اصطلاحات یک دامنه‌ی مسئله‌ی مشخص برابری می‌کنند.

یک کلاینت شیءای است که از منابع یک شیء دیگر (به نام سرور) استفاده می‌کند، می‌توانیم رفتار یک شیء را با بررسی سرویس‌هایی که برای اشیاء دیگر فراهم می‌کند و حتی عملیاتی که آن شیء بر روی اشیاء دیگر ممکن است اجرا کند، توصیف کنیم. این دید ما را بر تمرکز روی جنبه‌ی بیرونی یک شی وادار می‌سازد و به سمت چیزی که Meyer "**مدل قراردادی برنامه‌نویسی**[5]" می‌خواند، سوق می‌دهد. مدل قراردادی برنامه‌نویسی: دید خارجی هر شیء یک قراردادی را مشخص می‌کند که ممکن است اشیاء دیگر بر آن متکی باشند و به نوبه‌ی خود باید توسط دید داخلی خود شیء هم به آن عمل شود (اغلب در همکاری با اشیاء دیگر). این قرارداد همه‌ی فرضیاتی را که یک شیء کلاینت ممکن است در رفتار با یک شیء سرور تصور کند را وضع می‌کند. به‌عبارت‌دیگر، این قرارداد مسئولیت‌های یک شیء را در بر می‌گیرد. یعنی، رفتاری که برای آن، شیء مسئول است. به‌طور جداگانه، هر عملیاتی که به این قرارداد عرضه می‌شود یک امضای منحصربه‌فرد شامل آرگومان‌های ظاهری و نوع برگشتی دارد. کل مجموعه عملیاتی که یک کلاینت ممکن است بر روی یک شیء اجرا کند، به همراه ترتیب مجازی که در آن، عملیات ممکن است فراخوانی شوند را **پروتکل**[6] شیء می‌نامیم. یک پروتکل بر شیوه‌هایی که یک شیء ممکن است کنش و واکنش داشته باشد دلالت دارد و بدین ترتیب همه‌ی دید خارجی ایستا و پویایی یک انتزاع را تشکیل می‌دهد.

---

[1]. Entity abstraction
[2]. Action abstraction
[3]. Virtual Machine abstraction
[4]. Coicidental abstraction
[5].Contract model
[6].Portocol

اساس فکری هر انتزاع، مفهوم ثبات[۱] است. یک ثابت[۲] تعدادی شرط بولی (درست و غلط) (False, True) است که درستی‌اش باید حفظ شوند. برای هر عملیات مرتبط با شیء، ممکن است پیش‌شرط‌ها[۳] (ثابت‌های در نظر گرفته‌شده توسط عملیات) و پس شرط‌هایی[۴] (ثابت‌هایی که توسط عملیات برآورده می‌شود) تعریف کنیم. نقض هر ثابت، قرارداد مرتبط با آن انتزاع را زیر پا می‌گذارد. اگر یک پیش‌شرط نقض شود، بدین معنی است که یک کلاینت بخشی از قرارداد که مربوط به خودش بود را اجرا نکرده، بنابراین، سرور نمی‌تواند با اطمینان اقدام کند. به‌طور مشابه، اگر یک پس شرط نقض شود، این یعنی این‌که یک سرور بخشی از قرارداد که مربوط به خودش است را اجرا نکرده است و به همین خاطر کلاینت‌هایش نمی‌توانند اعتمادی به رفتار سرور داشته باشند. یک استثنا[۵] اشاره‌ای است به این‌که تعدادی ثابت برآورده نشده است یا نمی‌توان برآورده کرد. زبان‌های خاصی این امکان را به اشیا می‌دهند که استثناها را رد کنند به‌طوری‌که پردازش را متوقف می‌کنند و تعدادی از اشیاء دیگر را از وجود مشکلی هشدار می‌دهند که به نوبه خود ممکن است متوجه استثنا شوند و مشکل را مدیریت کنند. از سوی دیگر واژه‌های عملیات[۶]، متد[۷] و تابع عضو[۸] از سه فرهنگ مختلف برنامه‌نویسی شکل گرفتند (C++, Small talk, Ada به ترتیب). آن‌ها عملاً هم‌معنی هستند و آن‌ها را به جای همدیگر استفاده خواهیم کرد. همه‌ی انتزاع‌ها صفات ایستا[۹] و هم‌چنین پویا[۱۰] دارند. برای مثال، یک شیء فایل، یک مقدار مشخصی از فضای حافظه را اشغال می‌کند؛ هر فایلی یک اسم و یک محتوا دارد. این‌ها صفات ایستا هستند. مقدار هر یک از این صفت‌ها نسبت به طول عمر شی، پویا است. هر شیء فایل ممکن است ازلحاظ اندازه رشد کند و یا کوچک شود، نام فایل ممکن است تغییر کند، محتوایش ممکن است تغییر کند. در برنامه‌نویسی به سبک رویه گرا، عملی که مقادیر پویای شیء را تغییر می‌دهد، بخش اصلی همه‌ی برنامه‌ها است. اشیا زمانی تغییر می‌کنند که زیر برنامه‌ها فراخوانی می‌شوند و دستورات اجرا می‌شوند. در برنامه‌نویسی به روش قاعده گرا، اشیاء زمانی تغییر می‌کنند که رویدادهای جدید باعث می‌شوند که قواعد کار کنند که ممکن است به نوبه‌ی خود باعث اجرای قواعد دیگر شود و الی‌آخر. در برنامه‌نویسی به سبک شیء‌گرا، اشیا هر وقت که یک شیء را به کار می‌اندازیم تغییر می‌کنند (مثلاً زمانی که یک پیام را به یک شیء می‌فرستیم). بنابراین، فراخوانی یک عمل بر روی یک شیء باعث تعدادی واکنش از سوی شیء می‌شود. چه عملیاتی را می‌توانیم بر روی یک شیء اجرا کنیم و چطور واکنش‌های آن شی، رفتار شیء را تشکیل می‌دهد.

## مثال‌هایی از انتزاع

اجازه دهید این مفاهیم را با چند مثال شرح دهیم. نحوه‌ی پیدا کردن انتزاع‌های درست در یک مسئله‌ی معین را به فصل ۴ موکول می‌کنیم.

در یک مزرعه‌ی هیدروپونیک، گیاهان در یک محلول مغذی، بدون شن، ماسه یا خاک‌های دیگر رشد می‌کنند. نگهداری مناسب فضای گلخانه کار ظریفی است و به نوع گیاهی که در آن رشد می‌کند و سن آن بستگی دارد. یکی باید فاکتورهای مختلفی مثل دما، رطوبت، نور، PH، غلظت مواد مغذی را کنترل کند. در یک مزرعه بزرگ، استفاده از سیستم خودکاری که به‌طور مداوم این عوامل را نظارت و تنظیم می‌کند، رایج

---

[1].invariance [2]. invariant [3]. precondition [4]. Post condition [5]. Exception [6]. Operation
[7]. Method [8]. member function [9].static [10].Dynamic