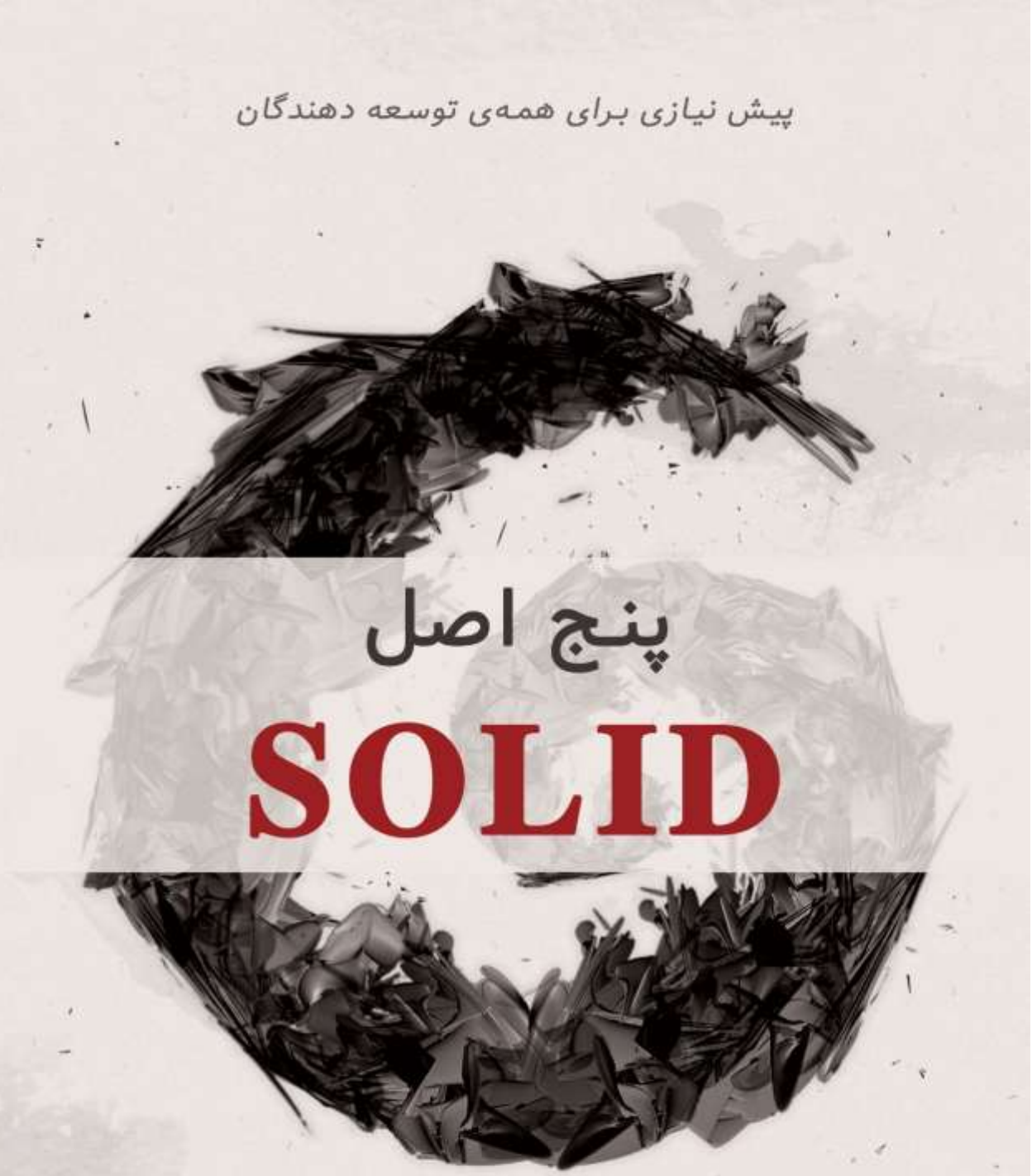
پیش نیازی برای همه‌ی توسعه دهندگان



# پنج اصل

# SOLID

نوشته‌ی
**رحمت اله (صدرا) عیسی پناه املشی**

# پنج اصل S.O.L.I.D

## پیش نیازی برای همه‌ی توسعه دهندگان

رحمت اله (صدرا) عیسی پناه املشی

**انتشارات ناقوس**

سرشناسه : عیسی پناه املشی, رحمت اله,۱۳۷۰-
Esa Panah Amlashi, Rahmatollah
عنوان و نام پدیدآور : پنج اصلS.O.L.I.D : پیش نیازی برای همه توسعه دهندگان/
رحمت اله (صدرا) عیسی پناه املشی
مشخصات نشر : تهران: ناقوس ,۱۳۹۷
مشخصات ظاهری : ۱۶۴ص
شابک : ۱-۱۶۲-۴۷۳-۶۰۰-۹۷۸
وضعیت فهرست‌نویسی : فیپا
یادداشت : کتابنامه: ص .۱۵۳.
موضوع : برنامه نویسی شی گرا
موضوع : Object-oriented Programming ( Computer science)
موضوع : نرمد افزار کاربردی — طراحی و توسعه
موضوع : Application Software -- Development
موضوع : تولید نرم افزار به روش اجایل — مدیریت
موضوع : Agile Software Development -- Management
رده بندی کنگره : ۱۳۹۷ ۹پ ۹ ع/QA۷۶/۶۴
رده بندی دیویی : ۰۰۵/۱
شماره کتابشناسی ملی : ۵۵۷۴۷۶۳




## انتشارات ناقوس

نام کتاب : پنج اصلS.O.L.I.D : پیش نیازی برای همه توسعه دهندگان
ناشر : انتشارات ناقوس
نویسنده : رحمت اله (صدرا) عیسی پناه املشی
چاپ اول : ۱۳۹۷
تیراژ : ۱۰۰ نسخه
قیمت : ۴۲۰۰۰۰ ریال
چاپ و صحافی: نارنجستان
شابک : ۱-۱۶۲-۴۷۳-۶۰۰-۹۷۸
ISBN : 978-600-473-162-1



انتشارات ناقوس
۱-بلوار کشاورز-خیابان ۱۶آذر-کوچه پارسی-پلاک ۱۰
تلفن و فاکس : ٦٦٤٧٨٩٥٤- ٦٦٤٧٨٩٥١

# قدردانی

در اینجا بر خود لازم دانسته که نهایت سپاس و قدردانی را نسبت به همکارانم در مرکز تحقیق وتوسعه شرکت دانش‌بنیان پارس فایبرنت، ابراز نموده و توفیق روز افزون دست اندرکاران آن مرکز تحقیقاتی و پژوهشی را در امر ارتقاء کیفی صنعت فناوری اطلاعات کشور از درگاه خداوندگار آرزو نمایم. این کتاب را تقدیم می کنم به همکاران عزیزم:

- فرامرز صالح پور
- مجید حشمتی
- رضا مشتاقی
- مهسا محسنیان
- مهدی مشتاقی
- شایان حسینی

و تقدیم به آنان که زندگی خود را وقف ***آگاهی*** انسان ها و مبارزه با *جهل* می‌نمایند

# فهرست

# پیشگفتار

*«Don't Listen to People Who Say You Don't Need Them!»*

## مقدمه

اگر احساس می کنید در نوشتن یک برنامه گیر کرده اید، این نکته را در نظر بگیرید که آموزش سنتی متداول، ممکن است شما را برای ورود به دنیای توسعه نرم افزار و برنامه نویسی آماده نکرده باشد. کن مازاکا[1] می گوید:

> *" از آنجا که برخلاف مراکز آکادمیک، شکست و تقلب در دنیای برنامه نویسی خوب تلقی می شود. بنابراین هرگز نمی توانید دنیای برنامه نویسی را به طور کامل درک کنید، بنابراین به جای تمرکز بر روی ریزه کاری ها، بر روی مفاهیم کلی متمرکز می شوید."*

ممکن است با سال‌ها برنامه‌نویسی و تمرین با موارد و سوژه‌های مختلف آشنا شوید، اما بدون دانستن اصول، قواعد و مفاهیم اصلی، هرگز نخواهید توانست آن بینش لازم جهت توسعه‌ی یک نرم افزار مطلوب را بدست آورید.

از آنجایی که در حال حاضر اکثر تیم‌های برنامه‌نویسی که در حال فعالیت اند با زبان‌ها و تکنولوژی‌هایی مبتنی بر شئ گرایی فعالیت می‌کنند، پس بر همه واجب است که با اصول و قواعد مربوط به آن آشنا شوند. این اصول و قواعد شامل موارد زیر است:

- *Testing*
- *Design Patterns*
- *SOLID*
- *Refactoring*

شاید گزینه‌های دیگری نیز وجود داشته باشند، اما این ها واجب‌ترین چیزهایی است که هر برنامه‌نویس باید از آن آگاه باشد. هر برنامه‌ای (حتی ساده‌ترین‌شان) احتیاج دارد مورد تست و آزمون قرار گیرد، برای اینکه برنامه شما قابل تست گرفتن باشد، ناگزیرید که با استفاده از الگوی‌های طراحی و همچنین اصول SOLID کدهای خود را بهینه و قابل تست کنید و برای اینکه بتوانید کدهای گذشته‌ی خود را برای این کار آماده کنید لازم است با نحوه بازنویسی و ریفکتور کردن آن‌ها آشنایی کامل داشته باشید. این اصول از ضروری‌ترین مفاهیمی هستند که هر برنامه‌نویس باید با آن‌ها آشنا باشد.

---

[1] Ken Mazaika

من برای یادگیری و فهم اصول SOLID به منابع مختلفی رجوع کردم؛ از ویدئوها و کتاب آنکل باب (که مطرح کننده این مفهوم است) گرفته تا سایت‌ها و مقاله‌های مختلفی که در منابع به آن‌ها اشاره کرده‌ام. کتاب پیش‌رو حاصل تلاش بنده در گردآوری و ارائه‌ی این مفاهیم به صورت ساده و در دسترس است. به قول *سایه* (هوشنگ ابتهاج) نام این کتاب «سالید به سعی صدرا» است. امید آنکه حق مطلب ادا شده باشد.

از خوانندگان محترم تقاضا دارم در صورتی که در مباحث مطرح شده نظری داشته یا احیانا در کتاب اشتباه فنی/غیرفنی وجود دارد، از طریق ایمیل یا یکی از راه‌های ارتباطی که در ادامه ذکر خواهم کرد به اطلاع بنده برسانند:

*Email*: amlashi.sadra@gmail.com | *website*: isapanah.com

## واژه‌ها

زبان انگلیسی، زبانِ کار و بیزنس است، زبان فنی است، زبان مهندسی است. اگر می‌خواهید در کار و حرفه‌ی خود موفق شوید، لازم است که بر آن مسلط شوید. از همین رو، در خط-های مختلف کتاب سعی کرده‌ام واژه‌ها و نام‌های مختلفِ یک عبارتِ خاص را بیان کنم . برای مثال Function را درجایی تابع، در جایی متد و حتی در جایی همان عبارت فینگلیشِ فانکشن نوشته‌ام. به امید اینکه با عبارت فنی و انواع مختلف آن در خلل مطالعه‌ی این کتاب آشنا شده و به همین سبب آن واژه ملکه‌ی ذهن‌تان شود.

## نحوه‌ی مطالعه‌ی این کتاب

این کتاب در ۷ فصل نوشته شده است. در فصل اول به مباحث مقدماتی و پیش از ورود به مبحث SOLID پرداخته‌ام .سپس در ۵ فصل آن اصول و قواعد را تبیین و در فصل آخر به چیزهایی برای مطالعه بیشتر اشاره کرده‌ام.

در فصول مربوط به اصل‌های SOLID در ابتدا به تشریح مفهوم آن اصل پرداخته‌ام .سپس با یک مثال مستقیما وارد کدها شده‌ام. در قسمت‌های میانی هر فصل بخشی به نام تحلیل و بررسی و بازنویسی مشاهده می‌کنید که در آن مشکلات موجود در مثال را بررسی کرده‌ام و سپس بر اساس دانسته‌های قبلی و تحلیل و بررسی‌های پیشینی که انجام داده بودیم، اقدام به

بازنویسی یا به اصطلاح Refactoring کرده‌ام. مثال‌هایی که در این کتاب زده‌ام، در زبان *JAVA* بوده است. در بخش انتهایی هر اصل، به نکات و تکنیک‌های ضروری جهت مدیریت و اعمال هرکدام از این اصول پرداخته‌ام.

از خوانندگان محترم این کتاب انتظار می‌رود با مبانی برنامه‌نویسی، Design Patternها، دیاگرام UML و همچنین تست نویسی (به ویژه unit-test) آشنایی داشته باشند.

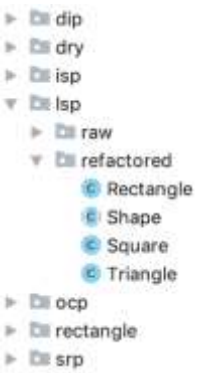
## بلاک کد

هرکدام از مثال‌هایی که زده‌ام شامل بلاک کدهایی می‌شود که کدهای آن مثال را در آن‌ها قرار داده‌ام. تمام تلاش خود را کرده‌ام تا بلاک‌کدها بین صفحه‌هات تقسیم نشده تا دچار سردرگمی شما نشود . به همین دلیل در برخی از بخش‌ها مشاهده می‌کنید که برای کامل افتادن یک بلاک کد در صفحه، بخش‌هایی از صفحه قبلی یا همان صفحه، خالی است (حقیقتاً سخت‌ترین بخش نوشتن کتاب‌هایی با موضوع برنامه‌نویسی، همین مدیریت کدها در صفحات است).

## مخزن گیتهاب

تمام کدهایی که در این کتاب نوشته شده‌اند، به صورت پروژه‌های جداگانه‌ای در گیتهاب نیز قابل دسترسی هستند. می‌توانید آن پروژه را از لینک زیر دریافت کرده و عیناً در محیط توسعه ی خود اجرا کنید:

https://github.com/sadra/SOLID




در این مخزن به ازای هر اصل پوشه‌ای وجود دارد که با نام مخفف آن اصل مشخص شده است، سپس هر پروژه یک فایل با نام raw دارد که شامل کدها بدون بازنویسی و ریفکتورینگ است، و پوشه‌هایی که با نام refactored مشخص شده‌اند، مربوط به کدهای پس از باز نویسی و اعمال اصل مربوطه می‌باشند.

از آن‌جایی که مثال‌های عینی در زبان‌های مختلف، می‌تواند به درک بهتر آن مفـاهیم کمـک کننـد، از شـــمـا خواننـده‌ی محترم

دعوت می‌کنم تا برای این پروژه در توسعه و ایجاد مثال‌هایی مشابه در زبان‌های برنامه‌نویسی دیگر، مشارکت کنید.

## منابع

برای نوشتن این کتاب، علاوه بر تجربه‌ی قبلی اینجانب از منابعی دیگری همچون ویدئوها و کتاب Clean Code از رابرت سی مارتین یا همان آنکل باب، ویدئوهای وبسایت pluralsight با موضوعSOLID و همچنین کتاب Working Effectively with Legacy Code اثر مایکل فیدرز بهره برده‌ام.

## سخن پایانی

امیدوارم این مباحث و همچنین بطور خاص این کتاب، جهت کسب آن بینش لازم در برنامه-نویسی کمک شایانی کند. در نهایت برای همه‌ی شما آرزوی به روزی و موفقیت دارم.

# مقدمات

*«Of course bad code can be cleaned up. But it’s very expensive.»*

## مقدمه

قبل از اینکه به اصول پنج‌گانه SOLID بپردازیم، لازم است تا با برخی مفاهیم آشنا شده و آن‌ها را بررسی کنیم. از همین رو، اولین فصل این کتاب، به این مقدمات و مفاهیم اختصاص خواهد داشت.

## SOLID چیست؟

اصول پنج‌گانه طراحی شئ گرا برای اولین بار توسط رابرت سی مارتین معروف به عمو باب[1] تعریف شد؛ سپس در سال ۲۰۰۰ آقای Michael Feathers با استفاده از اولین حروف هریک از نام‌های این اصول ۵ گانه، نام SOLID را برای آنها انتخاب کرد.

وقتی این اصول در کنار یکدیگر در طراحی و پیاده‌سازی یک برنامه اعمال می‌شوند، به احتمال زیاد آن سیستم قابلیت این را خواهد داشت که به آسانی قابل توسعه و نگهداری باشد.

در حقیقت اصول سالید، دستورالعمل‌هایی هستند که می‌توان هنگام کار بر روی یک نرم‌افزار، آن‌ها را برای از بین بردن، عوامل نامطلوب در کد، اعمال کرد. این کار از طریق فراهم آوردن چارچوبی انجام می‌گیرد که با استفاده از آن، برنامه‌نویس می‌تواند کدهای برنامه را اصلاح و بازسازی کند تا آن‌ها توسعه‌پذیر و خواناتر شوند.

## مدیریت وابستگی یا Dependency Management

قبل از اینکه شیرجه‌ای عمیق در SOLID بزنیم، لازم است تا با مدیریت وابستگی و مفاهیم مرتبط با آن آشنا شویم.

---

[1] Uncle Bob

## مدیریت وابستگی چیست؟

کمتر دیده می‌شود که پروژه‌های نرم افزاری را به صورت ایزوله توسعه داده باشند. معمولا، یک پروژه وابستگی زیادی به توابعی[۱] با قابلیت استفاده مجدد دارد که به صورت کتابخانه[۲] یا اجزای[۳] جدا شده از سیستم مورد استفاده قرار می‌گیرند. مدیریت وابستگی[۴]، یک تکنیک برای شفاف سازی، حلِ پیچیدگی و استفاده از وابستگی‌های مورد نیاز یک پروژه است.


در توسعه نرم افزار، بزرگترین نگرانی، پایداری و انسجام همیشگی سیستم است؛ به بیانی دیگر، با افزایش وابستگی یک سیستم ویژگی‌هایی مانند استفاده مجدد، انعطاف پذیری و قابلیت نگهداری آن کاهش می‌یابد. مدیریت وابستگی (یا DM) به ما کمک می‌کند تا این وابستگی ها را کنترل و مدیریت کنیم.

مفاهیم و اصولی مانند SOLID یا *Object-Oriented* از ابزارها و تکنیکهای پر کاربرد در مدیریت وابستگی اند.

---

[۱] Functions

[۲] Library

[۳] Components

[۴] Dependency Management

## فاجعه در مدیریت وابستگی

به قول Uncle Bob، سیستم‌هایی که مدیریت وابستگی در آن رعایت نمی‌شود این ۴ بو (Smell) را همراه خود خواهند داشت:

- سختی یا Rigidity
- شکنندگی یا Fragility
- عدم تحرک یا Immobility
- چسبناکی یا Viscosity

اما این ۴ بو چه هستند و چه تاثیری در سیستم ما دارند. در ادامه هرکدام از این ۴ مورد را به تفصیل بررسی خواهیم کرد.

## سختی یا Rigidity

به ساده‌ترین زبان ممکن، سختی یا Rigidity یعنی ناتوانی در تغییر.


اما چه می‌شود که تغییر دادن یک سیستم سخت می‌شود؟ تغییر دادن یک سیستم زمانی سخت می‌شود که هزینه تغییر بالا رود! فرض کنید برای ساختن و تست گرفتن از یک سیستم باید ۲ ساعت وقت صرف کنید. اگر بعد از یک تغییر کوچک، مجبور شوید دوباره ۲ ساعت دیگر برای ساختن و تست گرفتن از آن سیستم وقت بگذارید، *ریجیدیتی* رخ داده است.

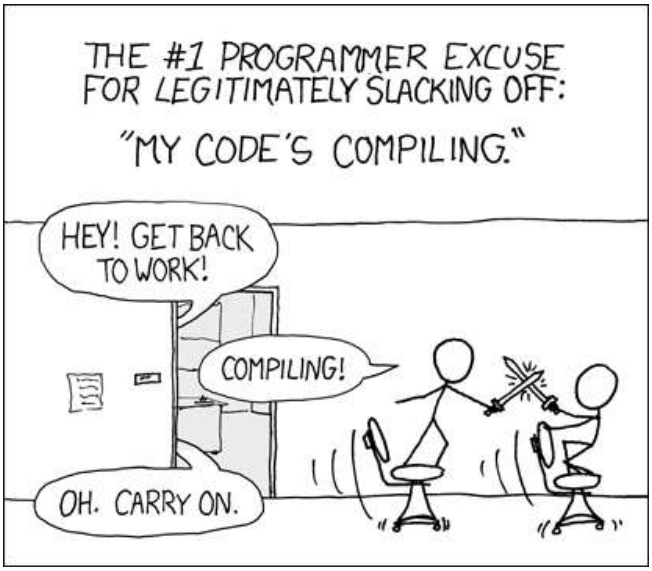
در نتیجه اگر برای تغییر دادن بخش کوچکی از کد مجبور شویم کل سیستم را مجددا *rebuild* کنیم، آنوقت آن سیستم سخت شده یا دچار Rigidty شده است.

## چطور بفهمیم دچار Rigidty شده ایم؟

- وقتی به ازای هر تغییر کوچک مجبور شویم کل سیستم را *rebuild* کنیم.
- وقتی به ازای هر تغییر، زمان زیادی برای تست و ساخت یک سیستم صرف شود.

درصورتی که بتوانیم راهی پیدا کنیم تا به ازای هر تغییر مجبور نشویم کل سیستم را دوباره بازسازی کنیم یا دست کم بتوانیم زمان تست و ساخت سیستم را به حداقل ممکن برسانیم، آنوقت از Rigidty جلوگیری کرده‌ایم.

اگر سیستم شما به ازای هر تغییر، وقت زیادی را جهت *test* و *rebuild* صرف می‌کند، نشانه این است که توسعه دهندگان کم حوصله و بی دقتی را در تیم خود دارید!




## دلایل ابتلا به Rigidity

امیدوارم هیچگاه به Rigidity مبتلا نشوید! برای اینکه بتوانید از آن پیشگیری کنید، بهتر است دلایلی که سیستمِ شما را مبتلا به Rigidity می‌کند خوب بشناسید:

- **کدها به صورت رویه‌ای یا Procedural نوشته شده‌اند:** برای مثال تمام کد در یک فایل و به صورت تو در تو با *if-else* های مختلف و زیاد و به صورت به هم پیوسته و وابسته نوشته شده است.

- **هیچ مفهوم انتزاعی یا Abstraction در کد دیده نمی‌شود:** این مورد وقتی اتفاق می‌افتد کدها در پایین‌ترین سطح ممکن نوشته شده باشد. درواقع بیش از حد به جزئیات توجه شده تا اینکه روی مفاهیم و خصوصیات شئ تمرکز شود. برای مثال به‌جای اینکه به خصوصیات یک شئ از طریق یک *method* یا *property* دسترسی داشته باشیم، با استفاده از یک پرچم ۱ بیتی در خانه nام حافظه، خصوصیت آن شئ را دریافت می‌کنیم.

- **کدها به صورت یک مفهوم کلی در سیستم تعبیه شده‌اند در حالی که با خصوصیاتی بسیار جزئی جهت استفاده در یک مورد خاص تعریف شده باشند:** برای مثال یک کد *HTML* را در نظر بگیرید که قرار است یک *table* را با استفاده از داده‌های یک ماتریس چاپ کند درحالی که در کد تعریف کرده ایم تا *Header*ها پیش زمینه‌ای مشکی با فونت *Bold* و رنگ سفید داشته باشند.

- **کامپوننت‌ها اطلاعات زیادی درباره جزئیات یکدیگر دارند، وقتی که قرار است باهم ارتباط داشته باشند:** برای مثال کلاسی داریم که یک Shape از مربع برای ما می‌سازد؛ از آن می‌خواهیم که مساحت مربع را بطور هاشور زده نمایش دهد؛ پس در آن فانکشنی می‌نویسیم که ضلع مربع را گرفته، آن را ضرب در خودش کند تا مساحت مربع بدست آمده و حدود *origin* های شکل چند ضلعی روی اسکرین بدست آید. سپس توسط فانکشن دیگری آن را به صورت هاشور زده نمایش می‌دهیم. مشکل جایی رخ می‌دهد که از همان کامپوننت - که مساحت اشکال را هاشور زده نمایش می‌دهد - بخواهیم یک Shape دیگر از دایره ساخته و مساحت آن را هاشور زده نمایش دهد!

## شکنندگی یا Fragility

مفاهیم Fragility و Rigidity بسیار بهم نزدیک اند. درواقع شکنندگی و سختی، علت و معلول یکدیگر هستند. شکنندگی یا Fragility اشاره دارد به اینکه هر موقع تغییری در سیستم ایجاد می‌کنید در بخش (یا بخش‌های) دیگری از سیستم - که حتی هیچ ربطی با آن قسمت ندارد - با خطا و مشکل مواجه می‌شوید.

فرض کنید یک ماشین داریم که رادیوی آن مشکل دارد، رادیو را درست می‌کنیم و پس از روشن کردن ماشین متوجه می‌شویم که شیشه برقی کار نمی‌کند!

همانطور که تغییرات افزایش پیدا می‌کنند، نگهداری سیستم نیز سخت تر می‌شود؛ زیرا هر بار که تغییری در سیستم ایجاد می‌کنیم، در چندین نقطه دیگر خطاهایی پیدا می‌شود، این مورد تا آنجا ادامه پیدا می‌کند که هر تغییری در سیستم باعث سکته‌های ناقص نرم افزار می‌گردد!

این مشکل باعث می‌شود که تیم اعتبار خود را در نزد مدیران و مشتریان از دست دهد! چون به ازای هر تغییری یا فیچر جدیدی که برای تیم ارسال می‌شود نیاز به صرف زمان و هزینه زیادی جهت ایجاد تغییر، تست و ساختن محصول می‌شود و حتی باعث می‌گردد که پروژه ها دیرتر از موعد به مرحله تحویل برسند. در نهایت این امر موجب کاهش کیفیت در محصول می‌شود.


## چطور بفهمیم دچار Fragility شده ایم؟

- مجبور شوید به ازای هر تغییر، تغییرات متوالی و زیادی در دیگر بخش‌ها ایجاد کنید.
- خطاهای جدیدی در بخش‌هایی رخ دهد که هیچ ربطی با تغییرات ندارند.

- جدا کردن بخشی از سیستم، ما را مجبور کند تا بعضی بخش‌ها را دوباره از نو بازنویسی کنیم.

## دلایل ابتلا به Fragility

ابتلا به Fragility ممکن است دلایل مختلفی داشته باشد، اما مهمترین و اصلی‌ترین دلیل ابتلا به Fragility داشتن وابستگی و ایزوله نبودن کامپوننت‌ها و بخش‌های مختلف سیستم است، که با تغییر یک بخش، نیاز به تغییر بخش‌های دیگر داریم.

## عدم تحرک یا Immobility

خاصیت عدم تحرک یا Immobility اشاره به این موضوع دارد که نتوانیم آن قسمت از کد یا کامپوننت را در دیگر بخش‌های سیستم استفاده کنیم. اگر در بخشی از سیستم احتیاج به ماژولی داشتیم که مشابه آن در قسمت دیگری از سیستم وجود دارد اما امکان استفاده مجدد در بخش دیگر به دلایل مختلف (مثل وجود برخی خصوصیات جزئی مثلا همان رنگ پیش‌زمینه یا background با فونت فلان) وجود نداشت آنگاه دچار immobility شده‌ایم.

فرض کنید در بخشی از نرم افزار یک ماژول login با استفاده از username و passowrd ساخته ایم. حال تصمیم داریم آن ماژولِ لاگین را از سیستم جدا کرده و در جایی دیگر استفاده کنیم. اگر نتوانید به سرعت و آسانی آن تکه کد را جدا کرده و در سیستم یا بخش دیگر (بدون تغییر) مجددا استفاده کنید، آنگاه ماژول **Immobile** است. درواقع نمی‌تواند حرکت کند!

وقتی سیستم *immobile* باشد، روزی که بخواهیم بخشی از سیستم را که به آن احتیاج داریم جدا کرده و در جای دیگری به کار بگیریم، این قضیه آنقدر سخت و پر ریسک می‌شود که نهایتا ترجیح می‌دهیم به‌جای اینکه کد را مجددا استفاده کنیم، بنشینیم و از ابتدا کدها را برای بخش جدید بازنویسی کنیم.

هنگامی که قصد آن را داشته باشیم تا بخشی از سیستم را جدا کرده و در بخش دیگری استفاده کنیم، در حالتی که کدها *immobile* باشد، دچار مشکل خواهیم شد؛ از آنجا که این عمل (به دلیل وجود جزئیات و وابستگی‌های بیش از حد) سخت و پر ریسک است، توسعه دهنده ترجیح می‌دهد به‌جای استفاده مجدد کدها در جای مورد نظر، وقت زیادی گذاشته و از پایه دوباره کدها را بازنویسی کند. این عمل سبب دوباره کاری و داپلیکیت شدن کدها می‌شود. امیدوارم متوجه باشید که تکراری یا داپلیکیت شدن کدها ممکن است چه عواقبی داشته باشد.

## چطور بفهمیم دچار Immobility شده ایم؟

- ماژول آنقدر پیچیده و وابسته به دیگر بخش‌ها شده که قابل جدا شدن نیست.
- ماژول قابل جدا شدن باشد، اما جدا کردن و استفاده کردن از آن در محیطی غیر از محیط اصلیِ آن، بسیار سخت و پر ریسک باشد.
- نتوانید ماژول یا بخشی از کد را از سیستم جدا کرده و فورا بدون تغییر در آن، در جای دیگری استفاده کنید.

## دلایل ابتلا به Immobility

- اصلی ترین دلیل ابتلا به immobility یا عدم تحرک، پرداختن به جزئیات بیش از حد و غیر ضروری در آن بخش از کد است.
- ماژول بیش از حد به محیط و *envirement* خودش وابسته باشد.
- ماژول بیش از یک وظیفه را بر دوش داشته باشد.

برای مثال، همان ماژول لاگین را در نظر داشته باشید. این ماژول برای احراز هویت کاربر از یک *database schema* خاص در دیتابیس MySQL استفاده می‌کند. این ماژول ابتدا به دیتابیس MySQL وصل شده و سپس با توجه به آن *schema* خاص، کاربر را احراز هویت می‌کند. حال قصد داریم آن را در سرویسی دیگری استفاده کنیم که از Redis استفاده می‌کند که در ساختار دیتابیس خود اصلا از چیزی به اسم *schema* پشتیبانی نمی‌کند. *حال، تو خود حدیث مفصل بخوان ازین مجمل!*

## چسبناکی یا Viscosity

چسبناکی، ویسکوزیته یا Viscosity مقاومت در مقابل تغییر است. وقتی که ساخت مجدد و تست سیستم برای ما سخت می‌شود و ترجیح بدهیم از خیر تغییرات آن قسمت بگذریم، آنگاه کد ما viscos یا چسبناک است.

در هنگام طراحی هر موقع که نیاز به یک تغییر بود، یا آن را در نظر می‌گیریم و با پیدا کردن یک راه درست تغییر را ایجاد می‌کنیم، یا اینکه می‌توانیم یک راه سریع، کوتاه و البته کثیف برای آن پیدا کنیم. شرایط و محیط‌هایی که اغلب در آن کار می‌کنیم، ما را به سمت آن رویکرد سریع و کثیف هدایت می‌کنند.

وقتی زمانِ کامپایل شدن بیش از حد طول می‌کشد، به‌جای آنکه آن قسمت از کد که موجب این مشکل شده است را تصحیح کنیم، تصمیم میگریم جایی را تغییر دهیم که نیازی ندارد مجددا کامل کامپایل شود.

وقتی که *check-in* یا *check-out*های دیتا یا فایل‌ها زمان زیادی از سیستم بگیرد، تصمیم به تغییر جایی میگیریم که احتیاج به دسترسی به فایلهای کمتری را دارد.

اگر مستقر کردن محیط آنقدرها هم برای ما سخت نباشد، تصمیم می‌گیریم به‌جای تغییر در کد، دیتابیس را تغییر دهیم. زیرا اجرا کردن یک اسکریپت دیتابیس برای تغییرات، آسان‌تر و امن‌تر از دوباره *deploy* کردن کل اپلیکیشن است.

بنابراین کدی که خاصیت Viscosity آن بالا باشد، ما را مجبور می‌کند تا به‌جای ماژول یا آن بخش از سیستم، تغییرات را روی بخش‌هایی اجرا کنیم که کمترین مقاومت ممکن را دارند.

## چطور بفهمیم دچار Viscosity شده ایم؟

- تغییر دادن آن بخش از سیستم آنقدر سخت و هزینه بر باشد که ترجیح دهیم جهت کاهش بار سیستم، بخش دیگری که کمتر ما را درگیر می‌کند را تغییر دهیم.

## دلایل ابتلا به Viscosity

- عدم توجه توسعه دهندگان به کد یا حواله کردن بهبود و *refactoring* کد به زمان دیگر
- وابستگی بیش از حد ماژول‌ها یا بخش‌های سیستم به یکدیگر

## پیچیدگی های غیر ضروری

یکی از مباحثی که معمولا در هنگام توسعه نرم افزار مطرح می‌شود این است که درباره آینده نرم افزار چه تصمیمی باید بگیرید؟ آیا فقط نیازهای کنونی خود را باید در طراحی لحاظ کنیم؟ یا اینکه باید یک دید باند مدت داشته و تمامی نیازهای آینده نرم افزار را در نظر داشته و در طراحی اعمال کنیم؟

به بیانی دیگر، آیا لازم است دستگیره‌هایی در سیستم جهت استفاده در آینده (دور یا نزدیک) قرار دهیم؟

به ازای هر دستگیره‌ای که در طراحی لحاظ می کنید که (شاید) در آینده مورد استفاده قرار گیرد، آن را پیچیده‌تر کرده و بار اضافی جهت توسعه و نگهداری به سیستم و تیم توسعه خود افزوده اید. اگر تنها نیازهای کنونی خود را در نظر داشته و سیستم را با استفاده از تکنیک‌هایی مدیریت وابستگی - که در آینده به آنها خواهم پرداخت - ایزوله و ماژولار تهیه کنید، آنگاه هم می‌توانید از پیچیدگی های غیر ضروری جلوگیری کرده و هم هیچکدام از آن چهار خصیصه نامطبوع را در سیستم خود تجربه نخواهید کرد.

هیچگاه هیچ سیستمی از همان ابتدا بد نیست. بدی‌ها یا آن خواص نامطبوع در طول زمان پدیدار می‌شوند. تصمیم‌های بدی که در مقاطع گوناگون میگیریم که حاصل بی‌توجهی و تجربه‌های غلط ما هستند، دلایل اصلی به وجود آمدن این خصایص نامطبوع و پیچیدگی‌های غیر ضروری اند.

> *گسترده شدن پیچیدگی‌ها، پیشرفت را سخت تر می‌کند و ما را بیشتر وسوسه می‌کند تا راه‌های میانبری برای کاهش این سختی پیدا کنیم که همان‌ها نیز به افزایش وابستگی و پیچیده‌تر شدن سیستم در آینده می‌انجامد.*
>
> *Robert C Martin*

## مدیریت وابستگی ها با SOLID

مدیریت وابستگی موضوعی است که بیشترِ ما با آن سروکار داریم. هر موقع صفحه‌ی اسکرین را که پوشیده از کدهای پیچیده و بهم ریخته تماشا می‌کنیم، در حال تجربه‌ی ناخوشایندِ نتایجِ حاصل از مدیریت ضعیف در وابستگی هستیم.

مدیریت ضعیف وابستگی منجر به کدهایی می‌شود که تغییر دادن آن سخت است، شکننده است و قابل استفاده مجدد نیست. ما داریم راجع به آن ۴ بوی نامطبوع حرف می‌زنیم. از سویی دیگر، زمانی که وابستگی‌ها به خوبی مدیریت شوند کد انعطاف پذیر، قوی‌تر و قابل استفاده مجدد می‌شوند.

اصول ۵ گانه SOLID یکی از راه‌هایی است که به ما در مدیریت وابستگی‌ها کمک می‌کند. بهتر است نگاهی اجمالی به این اصول بیاندازیم:

| حرف اول | مخفف | مفهوم |
|---|---|---|
| **S** | Single Responsibility Principle | **اصل تک‌وظیفه‌ای**<br>یک کلاس باید تنها یک وظیفه داشته باشد (یک کلاس باید تنها یک دلیل برای تغییر داشته باشد و نه بیشتر) |
| **O** | Open–Closed Principle | **اصل باز ـ بسته**<br>اجزای نرم‌افزار باید نسبت به توسعه باز (یعنی پذیرای توسعه باشد) و نسبت به اصلاح بسته باشند (یعنی پذیرای اصلاح نباشد). (مثلا برای افزودن یک ویژگی جدید به نرم‌افزار نیاز نباشد که بعضی از قسمت‌های کد را بازنویسی کرد، بلکه بتوان آن ویژگی را مانند پلاگین به راحتی به نرم‌افزار افزود) |
| **L** | Liskov Substitution Principle | **اصل جانشینی لیسکاو**<br>اشیاءِ یک برنامه که از یک کلاس والد هستند، باید به راحتی و بدون نیاز به تغییر در برنامه، قابل جایگزینی با کلاس والد باشند. |
| **I** | Interface Segregation Principle | **اصل تجزیه (تفکیک) رابط**<br>استفاده از چند رابط که هر کدام، فقط یک وظیفه را بر عهده دارد بهتر از استفاده از یک رابط چند منظوره است. |
| **D** | Dependency Inversion Principle | **اصل وارونگی وابستگی**<br>بهتر است که برنامه به انتزاع یا تجرید (abstraction) وابسته باشد نه به پیاده‌سازی. |

## جمع بندی

در این فصل  با اصول پنج گانه سالید آشنایی پیدا کردیم، سپس به مفهوم مدیریت وابستگی‌ها یا همان DM پرداختیم. با ۴ بوی نامطبوع کدهایی که مدیریت وابستگی در آن رعایت نشده آشنا شدیم، که عبارت بودند از:

- سختی یا Rigidity
- شکنندگی یا Fragility
- عدم تحرک یا Immobility
- چسبناکی یا Viscosity

در ادامه هر کدام را مفصل بررسی کردیم. در نهایت به این نتیجه رسیدیم، برای اینکه کدهای ما هیچکدام از این ۴ بود را نداشته باشد، بهتر است از اصول SOLID و دیگر ابزارهای DM استفاده کنیم.

در ۵ فصل آینده، به تشریح هر کدام از این ۵ اصل خواهیم پرداخت.

# اصل اول
# Single Responsibility Principle

*«Just because you can, doesn't mean you should!»*

## مقدمه

در این بخش به برخی مفاهیم پایه SRP خواهم پرداخت، خواهیم دید کلاسی که چندین وظیفه را انجام می‌دهد چه خصوصیات و مشکلاتی دارد. در ادامه برای مثال کدهای یک سرویس فروش را بررسی و با استفاده از اصل تک‌وظیفه‌ای، مشکلات آن را رفع کرده و بازنویسی می-کنیم.

## تعریف

مفهوم Single Responsiblity به وضعیتی می‌گویند که هر شئ تنها یک وظیفه داشته و آن وظیفه در کلاس خودش کاملا کپسوله‌سازی[۱] شده باشد.

رابرت سی مارتین یا آنکِل باب[۲]، که پایه گذار این مفاهیم است، در جایی می‌گوید: *«نباید بیشتر از یک دلیل برای تغییر کلاس وجود داشته باشد.»*


فقط به این دلیل که می‌توانید، به این معنی نیست که باید انجامش بدهید.

---

[۱] کپسوله‌سازی (Encapsulation) یا لفافه‌بندی، در علم رایانه مخفی‌سازی مکانیزم داخلی و ساختار داده‌های اجزای نرم‌افزار در پشت یک رابط کاربر است. عمل مخفی‌سازی باعث می‌شود که اشیاء بدون آنکه از چگونگی کارکرد یکدیگر اطلاع داشته باشند با هم کار کنند.

[۲] Robert. C. Martin (Uncle Bob)

این تصویر به خوبی برای ما روشن می‌کند که چه اتفاقی پیش می‌آید وقتی یک ابزار وظایف زیادی دارد و چطور این چند وظیفگی باعث می‌شود تا آن ابزار عملا بلا استفاده شود. این چاقوی جیبی تعداد زیادی ابزار در خود دارد، و شما هر نیازی که داشته باشید می‌توانید با استفاده از این ابزارها برطرف کنید. نکته جالب ماجرا این است که این چاقوی جیبی ابدا در جیب شما جا نمی‌شود!

این چاقوی زیبا، هر کاری که فکرش را بکنید برای شما انجام می‌دهد. اما وای به روزی که یکی از ابزارهایش خراب شود، آنگاه مجبور می‌شوید تمام قطعات آن را جدا و قطعه خراب را تعمیر یا تعویض کرده و مجددا چاقو را سر هم کنید. البته اگر بتوانید آن را مثل اولش سر هم کنید!

همانطور که گفتم، چون می‌توانید کاری را انجام دهید، به این معنی نیست که باید آن را انجام دهید. همین مفهوم را می‌توانیم در طراحی نرم افزار و کلاس‌های خودمان تعمیم دهیم. این مطلب با دو مفهوم انسجام[۱] و جفتگری[۲] ارتباط نزدیکی دارد.

- **انسجام:** به میزان ارتباط و تمرکز وظایف مختلف در یک ماژول یا کلاس گفته می‌شود.
- **جفتگری:** یا وابستگی[۳] به میزان اتکای یک ماژول از برنامه به سایر ماژول‌ها گفته می‌شود.

تمام تلاش ما داشتن بالاترین انسجام و پایین‌ترین وابستگی است.

## مسئولیت – Responsibilty

مسئولیت‌ها یا وظایف به عنوان محور تغییر، تعریف می شوند. تغییرات مورد نیاز، معمولا رابطه‌ی یک به یک با وظایف دارند. وظایف بیشتری که یک کلاس بر عهده می‌گیرد، احتمال تغییرات را بیشتر می‌کند. داشتن وظایف چند گانه در یک کلاس، باعث جفت شدن و وابستگی بین این وظایف می‌شود؛ آنگاه زمانی که بخواهید تغییری در یک وظیفه ایجاد کنید، در دیگر وظایف

---

[۱] Cohension

[۲] Coupling

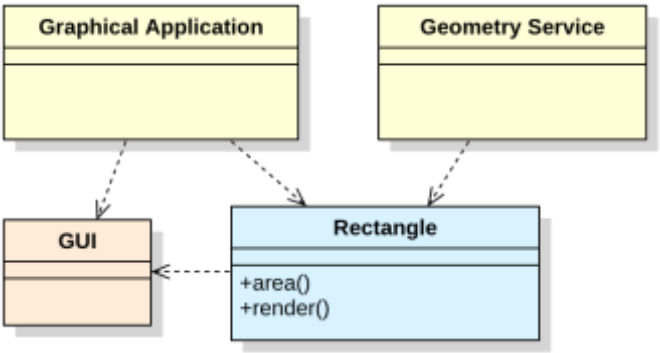
[۳] Dependancy

یا خصوصیات[۱] آن کلاس نیز تاثیر گذار خواهید بود. هرچه تاثیر تغییرات در یک کلاس بیشتر شود، احتمال تولید و مواجه شدن شما با خطا[۲] بیشتر می‌شود.

بنابراین مهم است که سعی کنیم تا کلاس‌ها را به گونه‌ای طراحی کنیم تا بخش‌هایی که بیشترین تغییر را دارند، تنها با یک وظیفه در کلاس‌هایی جداگانه کپسوله شوند.

## مثال

برای اینکه همه چیز روشن شود، می‌خواهم با مثال پیش برویم. ابتدا دیاگرام UML زیر را در نظر بگیرید.




همانطور که در این دیاگرام می‌بینیم، کلاسی داریم با نام Rectangle که شامل دو فانکشن `area()` و `render()` است. که *area* مساحت مستطیل را به ما بر می‌گرداند و *render* هم آن مستطیل یا شئ Rectangle را، روی واسط گرافیکی GUI ترسیم می‌کند.

همانطور که میبینید متد render به لایبرری GUI برای انجام عمل ترسیم، نیازمند و وابسته است. اما در همین حال، کلاس Rectanlge توسط دو ماژول دیگر مورد استفاده قرار می‌گیرد. یکی از آن‌ها سرویس Geometry Service است که تمامی فانکشن‌هایش کپسوله شده و هیچ واسط گرافیکی نیاز نداشته و تنها از متد ()area در کلاس Rectangle برای محاسبه‌ی مساحت استفاده می‌کند. ماژول دیگر اپلیکیشن Graphical Application است، که سعی می‌کند تصویر یک مستطیل را روی واسط گرافیکی ترسیم کند.

---

[۱] Features

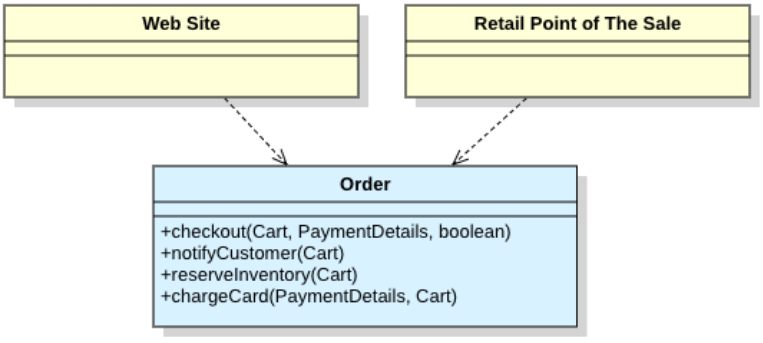
[۲] Errors

دیاگرام را یکبار دیگر اما دقیق‌تر مطالعه کنید، به جهت فلش‌های متقاطع دقت کنید، آن‌ها وابستگی‌ها را نشان می‌دهند. در این جا کلاس Geometry نیاز دارد تا از متد area در Rectangle استفاده کند، در این کلاس علاوه بر متد area، متد render هم وجود دارد که برای ترسیم شکل مستطیل به ماژول GUI وابسته است. جالب نیست؟ کلاس Geometry درحالی که هیچ نیازی برای ترسیم و استفاده از render ندارد، بدون هیچ دلیل قانع کننده‌ای تنها به واسطه وابستگی به کلاس Rectangle به کلاس GUI نیز وابستگی پیدا کرده است.

درصورتی که نیاز شود تغییری در GUI بدهیم، پس از تغییر باید آن را *recompile* کنیم، و این عمل باعث می‌شود تا Rectanlge هم به واسطه وابستگی که به GUI دارد احتیاج به compile مجدد داشته باشد و در نهایت سرویس Geometry به دلیل وابستگی که به واسطه‌ی Rectanlge به GUI دارد، احتیاج به compile پیدا می‌کند. این درحالی است که کلاس Geometry هیچ اطلاع یا دانش یا حتی احتیاجی به واسط گرافیکی یا رندر شدن ندارد. چه پیچیدگی‌هایی غیر ضروری ای!

بهترین راه برای بهبود طراحی یا دیزاین این دیاگرام این است که، کلاس Rectangle را به دو کلاس کوچک‌تر تقسیم کنیم. یکی از آنها صرفا به مسائل ریاضی و محاسبه مساحت یا چیزهایی از این دست خواهد پرداخت، و دیگری صرفا به ترسیم و رندر کردن تصویر مستطیل روی واسط گرافیکی می‌پردازد.

مثال دیگری بزنیم که با کسب و کار در ارتباط باشد. دیاگرام UML زیر را در نظر بگیرید، در این دیاگرام روند یک سفارش طراحی شده است.



یک Order (همان سفارش) در این اینجا تعریف شده است که از کاربر کارت را دریافت کرده، خرید را انجام می‌دهد، به مشتری خبر می‌دهد که در وضعیت سفارش چه اتفاقی افتاده و در نهایت محصولات سفارشی را در انبار برای آن مشتری رزرو کرده و علامت گذاری می‌کند. همانطور که در دیاگرام می‌بینید، کلاس Order برای اینکه این کارها را انجام دهد از چند متد یا عملوند استفاده می‌کند.

اجازه بدهید کدهای کلاس Order را بررسی کنیم. همانطور که می‌بینید بیشترین کارها در متد Checkout انجام می‌شود. این متد ابتدا بررسی می‌کند که آیا روش پرداخت به صورت CreditCard است؟ در صورت صحت، اقدام به Charge کردن و پرداخت آن می‌کند.
سپس آن محصولاتی که در سبد خرید وجود دارند را در انبار برای آن مشتری رزرو کرده، و در نهایت در صورتی که نیاز بود تا مشتری از وضعیت سفارش مطلع شود، برای وی پیام ارسال می‌کند.

```
public void Checkout(
    Cart cart,
    PaymentDetails paymentDetails,
    boolean notifyCustomer
    ) throws Exception
{
    if (paymentDetails.getPaymentMethod() == CreditCard)
    {
        ChargeCard(paymentDetails, cart);
    }

    ReserveInventory(cart);

    if(notifyCustomer)
    {
        NotifyCustomer(cart);
    }
}
```

برای اینکه Order بتواند وظایفی را که در بالا ذکر کردیم انجام دهد به متدهای دیگری نیز احتیاج دارد. متد NotifyCustomer که به کاربر پیام ارسال می‌کند. این متد در دل خود از سرویس‌های Properties و Session و MimeMessage جهت ارسال پیام بر بستر پروتکل SMTP استفاده می‌کند.

```
private void notifyCustomer(Cart cart)
{
    String customerEmail = cart.getCustomerEmail();
    if (!customerEmail.isEmpty())
    {
        Properties properties = System.getProperties();
        properties.setProperty("mail.smtp.host", "localhost");
        Session session = Session.getDefaultInstance(properties);
        MimeMessage message = new MimeMessage(session);

        try
        {
            message.setFrom(new InternetAddress("mail@example.com"));
            message.addRecipient(
                    Message.RecipientType.TO,
                    new InternetAddress(customerEmail)
            );

           message.setSubject(
                "Your order placed on " + new Date().toString());
            message.setText(
                "Your order details: \n " + cart.toString());

            Transport.send(message);
            System.out.println("Message sent successfully.");
        }
        catch (Exception ex)
        {
            Logger.error("Problem sending notification email", ex);
        }
    }
}
```

در متد ReserveInventory هم برای رزرو کردن سبد خرید کاربر در انبار، یک نمونه از کلاس InventorySystem ساخته و آن را استفاده می کند.

```
private void ReserveInventory(Cart cart) throws Exception
{
    for(OrderItem item : cart.getItems())
    {
        try
        {
            InventorySystem inventorySystem = new InventorySystem();
            inventorySystem.Reserve(item.getSku(),
                                    item.getQuantity());
        }
        catch (InsufficientInventoryException ex)
        {
            throw new OrderException(
            "Insufficient inventory for item " + item.getSku(), ex);
        }
        catch (Exception ex)
        {
            throw new OrderException(
                  "Problem reserving inventory", ex);
        }
    }
}
```

و در آخر متد ChargeCard که با استفاده از کلاس PaymentGateway عملیات پرداخت را انجام می‌دهد.

```
private void ChargeCard(PaymentDetails paymentDetails, Cart cart)
throws Exception
{
    PaymentGateway paymentGateway = new PaymentGateway();

    try
    {
        paymentGateway.credentials = "account credentials";
        paymentGateway.cardNumber =
                    paymentDetails.getCreditCardNumber();
        paymentGateway.expiresMonth =
                    paymentDetails.getExpiresMonth();
        paymentGateway.expiresYear =
                    paymentDetails.getExpiresYear();
        paymentGateway.nameOnCard =
                    paymentDetails.getCardholderName();
        paymentGateway.amountToCharge = cart.getTotalAmount();

        paymentGateway.Charge();
    }
    catch (AvsMismatchException ex)
    {
        throw new OrderException( "The card gateway rejected the
                    card based on the address provided.", ex);
    }
    catch (Exception ex)
    {
        throw new OrderException(
                    "There was a problem with your card.", ex);
    }
}
```

## بررسی و تحلیل مشکل

چه مشکلی در این طراحی وجود دارد؟ همانطور که در دیاگرام دیدیم، این کلاس هم در هنگام ثبت سفارش در وبسایت (Web Site) و هم در محل فروشگاه (Retail Point Of The Sale) استفاده می‌شود. حال یک مشتری را در نظر بگیرید که به‌جای روش آنلاین و پرداخت با کارت، بخواهد در محل فروشگاه و به صورت نقدی خرید خود را انجام دهد. در این صورت عملا بیشتر بخش‌های سیستم بالا استفاده خواهد بود. برای مثال:

- از آن جایی که مشتری پرداخت خود را به صورت نقدی انجام می‌دهد، هیچ احتیاجی به بررسی موجودی و پرداخت از طریق درگاه بانکی نخواهیم داشت.
- درصورتی که خرید در محل فروشگاه انجام شود، احتیاجی به رزرو کردن محصول در انبار نیست.
- موجودی انبار در همان لحظه به روز خواهد شد.
- هنگامی که مشتری از محل فروشگاه خرید می‌کند، هیچ ایمیلی برای او ارسال نمی‌شود.
- اصلا ایمیلی از مشتری دریافت نمی‌شود که بخواهد سیستم ثبت یا استفاده شود.
- مشتری همان لحظه آگاه می‌شود که خریدش با موفقیت انجام شده و لیست سفارش را تحویل می‌گیرد.

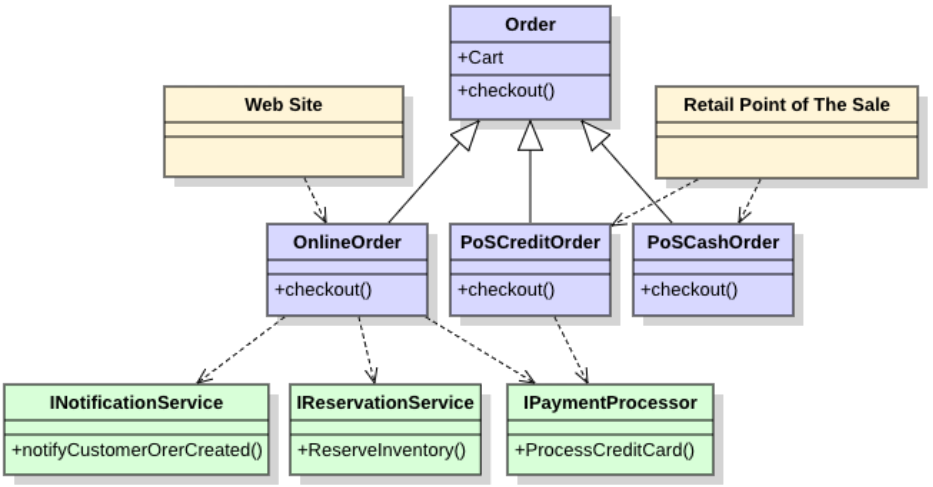
در نهایت متوجه می‌شویم که ۳ متد `NotifyCustomer`، `ReserveInventory` و `ChargeCard` عملا هیچ کارایی در این حالت نخواهند داشت.

متاسفانه با این طراحی درصورتی که بخواهیم تغییری در منطق ارسال پیام، پروسه خرید از کارت و یا مدیریت انبار ایجاد کنیم، آنگاه این تغییر روی Order تاثیر گذاشته و به واسطه‌ی آن Order روی Website و Retail Point Of The Sale هم تاثیر خواهد گذاشت. در نهایت برای تغییر جزئی در یکی از متدها، مجبور می‌شویم کل سیستم را مجدد *compile* کنیم.

## بهبود طراحی

حال چطور می‌توانیم این کد را بازسازی[۱] کرده و طراحی آن را بهبود دهیم. قبل از هر چیز دیاگرام UML زیر را مشاهده کنید.

[۱] Refactoring




ابتدا باید وظایفی که ممکن است در آینده روی آن‌ها تغییرات داشته باشیم را مشخص کنیم. همانطور که دیدیم، متد `Checkout` سه وظیفه را انجام می‌دهد؛ وظایفی مانند انجام پرداخت، ارسال پیام به مشترک و مدیریت انبار. بدین ترتیب می‌توانیم این وظایف را به **اینترفیس‌هایی**[۱] جداگانه تقسیم کرده و از کلاس `Order` خارج می‌کنیم:

- اینترفیس **IPaymentProccessor** که شامل متد `ProcessCreditCard()` است و می‌تواند روند پرداخت را انجام دهد.
- اینترفیس **IReservationService** که شامل متد `ReserveInventory()` است و سرویس مدیریت انبار و رزرو اجناس را بر عهده دارد.
- اینترفیس **INotificationService** که شامل متد `NotifyCustomerOrderCreated()` است و امکان ارسال پیام به مشتری را فراهم می‌کند.

[۱] Interface

سپس می‌توانیم کلاس `Order` را به کلاس های کوچکتری تقسیم کنیم که هرکدام وظایف کمتری دارند؛ در نهایت از اینترفیس‌های مناسب برای هرکدام از آنها استفاده کنیم.

برای مثال وقتی که یک سفارش آنلاین در سایت ایجاد شود، همانطور که در دیاگرام UML هم میبینیم، سیستم **Web Site** تنها از کلاس `Order Online` جهت پردازش روند خرید و پرداخت استفاده می‌کند. کلاس `Order Online` از هر سه اینترفیس برای پردازش روند پرداخت، رزرو در انبار و ارسال پیام به مشتری استفاده می‌کند.

این در حالی است که سیستم **Retail Point Of The Sale** از نوع دیگری از شئ `Order` استفاده می‌کند که وابستگی‌های آن نیز متفاوت است. هنگامی که مشتری بخواهد در محل فروشگاه خرید خود را انجام بدهد، این سفارش به دو صورت خواهد بود:

- **پرداخت با کارت:** که از کلاس `PosCreditOrder` استفاده می‌کند. این کلاس به اینترفیس **IPaymentProccessor** وابستگی دارد که روند بررسی موجودی و سپس کسر از حساب مشتری را انجام می‌دهد.

- **پرداخت نقدی:** که از کلاس `PosCashOrder` استفاده می‌کند. این کلاس هیچ وابستگی به هیچکدام از اینترفیس‌ها ندارد و روند خرید مشتری به راحت‌ترین شکل ممکن انجام می‌شود.

## بازسازی کد

حال که طراحی سیستم را اصلاح کردیم، بهتر است سراغ کدها برویم. در اولین گام وظایف را به اینترفیس‌هایی جداگانه می‌شکنیم.

```
public interface INotificationService {
    public void notifyCustomerOrderCreated(Cart cart);
}
```

```
public interface INotificationService {
    public void notifyCustomerOrderCreated(Cart cart);
}
```

```
public interface IReservationService {
    public void reserveInventory(Iterable<OrderItem> items)
                                                    throws Exception;
}
```

سپس به ازای هر وظیفه یک کلاس ایجاد کرده و اینترفیس ها را در آن پیاده‌سازی[۱] می-کنیم. آنگاه آن وظایفی که که در کلاس `Order` مجتمع بود را به این کلاس‌های متناظر منتقل می‌کنیم.

[۱] Implementation

```
public class ImplNotificationService implements INotificationService
{
    @Override
    public void notifyCustomerOrderCreated(Cart cart)
    {
        String customerEmail = cart.getCustomerEmail();

        if (!customerEmail.isEmpty())
        {
            Properties properties = System.getProperties();
            properties.setProperty("mail.smtp.host", "localhost");

            Session session =
                  Session.getDefaultInstance(properties);

            MimeMessage message = new MimeMessage(session);

            try
            {
                message.setFrom(new
                  InternetAddress("mail@example.com"));

                message.addRecipient(
                        Message.RecipientType.TO,
                        new InternetAddress(customerEmail)
                );

                message.setSubject(
                  "Your order placed on " + new Date().toString());

                message.setText(
                  "Your order details: \n " + cart.toString());

                Transport.send(message);
                System.out.println("Message sent successfully.");
            }
            catch (Exception ex)
            {
                Logger.error(
                  "Problem sending notification email", ex);
            }
        }
    }
}
```

```
public class ImplPaymentProcessor implements IpaymentProcessor
{
    @Override
    public void processCreditCard(
         PaymentDetails paymentDetails,
         float amount) throws Exception
    {
       PaymentGateway paymentGateway = new PaymentGateway();

        try
        {
            paymentGateway.credentials = "account credentials";
            paymentGateway.cardNumber =
                   paymentDetails.getCreditCardNumber();
            paymentGateway.expiresMonth =
                   paymentDetails.getExpiresMonth();
            paymentGateway.expiresYear =
                   paymentDetails.getExpiresYear();
            paymentGateway.nameOnCard =
                   paymentDetails.getCardholderName();
            paymentGateway.amountToCharge = amount;

            paymentGateway.Charge();
        }
        catch (AvsMismatchException ex)
        {
            throw new orderException("The card gateway rejected the
                   card based on the address provided.", ex);
        }
        catch (Exception ex)
        {
            throw new orderException(
                   "There was a problem with your card.", ex);
        }
    }
}
```

```
public class ImplReservationService implements IreservationService
{
    @Override
    public void reserveInventory(Iterable<OrderItem> items)
                                        throws Exception
    {
        for(OrderItem item : items)
        {
            try
            {
                InventorySystem inventorySystem =
                                        new InventorySystem();
                inventorySystem.reserve(item.getSku(),
                                        item.getQuantity());
            }
            catch (InsufficientInventoryException ex)
            {
                throw new orderException(
                  "Insufficient inventory for item "
                  + item.getSku(), ex);
            }
            catch (Exception ex)
            {
                throw new orderException(
                  "Problem reserving inventory", ex);
            }
        }
    }
}
```

حال که تمام وظایف ممکن را از کلاس `Order` خارج کردیم، بهتر است به این موضوع فکر کنیم که به چند نوع سفارش احتیاج داریم؟ اجازه دهید کار شما را راحت کنم، تمامی سفارشاتی که ممکن است این فروشگاه آنها را پردازش کند از این قرار است:

- سفارش آنلاین از طریق سایت با پرداخت از کارت **OnlineOrder**
- سفارش در محل با پرداخت از کارت **PosCreditOrder**
- سفارش در محل با پرداخت نقدی **PosCashOrder**

فکر می‌کنم شما هم با من موافق باشید که نیاز داریم به ازای هرکدام از این نوع سفارشات، یک شئ جدا گانه داشته باشیم که همه آنها از کلاس پدر، یعنی `Order` ارث بری می‌کنند.

قبل از هرچیزی باید کلاس `Order` را به یک کلاس انتزاعی تبدیل کنیم. می‌دانیم که در این ۳ نوع Order متد `checkout()` مشترک است. پس کلاس ابسترکت `Order` باید حاوی این متد نیز باشد.

```
public abstract class Order
{
    protected Cart cart;
    protected Order(Cart cart)
    {
        this.cart = cart;
    }
    public abstract void checkout() throws Exception;
}
```

پس از آنکه کلاس `Order` را ریفکتور کردیم، باید سه فرزند دیگر Order را نیز ایجاد کنیم. در ابتدا کلاس **OnlineOrder** را ایجاد می‌کنیم. از این کلاس هنگامی استفاده می‌کنیم که کاربر بخواهد بطور آنلاین و از طریق وبسایت سفارش خود را ثبت کرده و آن را پرداخت نماید.

```
public class OnlineOrder extends Order{
    private INotificationService notificationService;
    private PaymentDetails paymentDetails;
    private IPaymentProcessor paymentProcessor;
    private IReservationService reservationService;

    public OnlineOrder(Cart cart, PaymentDetails paymentDetails)
    {
        super(cart);
        this.paymentDetails = paymentDetails;
        this.paymentProcessor = new ImplPaymentProcessor();
        this.reservationService = new ImplReservationService();
        this.notificationService = new ImplNotificationService();
    }

    @Override
    public void checkout() throws Exception
    {
        paymentProcessor.processCreditCard(
                  paymentDetails, cart.getTotalAmount());
        reservationService.reserveInventory(cart.getItems());
        notificationService.notifyCustomerOrderCreated(cart);
        //TODO save order record on database
    }
}
```

پس از اینکه سفارش توسط وبسایت ساخته شد، در هنگام `checkout()` کردن، ابتدا با استفاده از `ImplPaymentProcessor` باقی مانده حساب کاربر بررسی شده سپس پرداخت انجام می‌شود. آنگاه با استفاده از `ImplReservationService` لیست سفارشات کاربر در انبار رزرو می‌شود، و در نهایت با استفاده از `notifyCustomerOrderCreated` یک پیام جهت اطلاع از انجام شدن فرآیندِ سفارش، به ایمیل کاربر ارسال می‌شود.

همانطور هم که قبلا دیدیم، کاربر ۲ راه برای پرداخت در اختیار دارد، یکی پرداخت نقدی و دیگری پرداخت با کارت. در پرداخت نقدی هیچ نیازی به آن ۳ وظیفه پرداخت، رزرو و اطلاع رسانی نداریم؛ در نتیجه کلاس **PosCashOrder** متد `checkout()` را تنها برای ذخیره اطلاعات سفارش در دیتابیس فراخوانی[۱] می‌کند.

---

[۱] Call

```
public class PosCashOrder extends Order
{
    public PosCashOrder(Cart cart) {
        super(cart);
    }

    @Override
    public void checkout() throws Exception {
        //TODO save order record on database
    }
}
```

اما اگر مشـتری وجه نقدی در جیبش نداشـته باشـد، ترجیح می‌دهد از طریق کارت بانکی پرداخت خود را انجام دهد. پس به کلاس PosCreditOrder احتیاج خواهیم داشت.

در ایـن کـلاس بـا اســتـفـاده `ImplPaymentProcessor` و مـتـد داخـل آن یعنی `processCreditCard()` عمل پرداخت انجام شــده و در نهایت ســفارش در دیتابیس ذخیره می‌شود.

```
public class PosCreditOrder extends Order
{
    private PaymentDetails paymentDetails;
    private IPaymentProcessor paymentProcessor;

    public PosCreditOrder(
                Cart cart,
                PaymentDetails paymentDetails
    ){
        super(cart);
        this.paymentDetails = paymentDetails;
        this.paymentProcessor = new ImplPaymentProcessor();
    }

    @Override
    public void checkout() throws Exception {
        paymentProcessor.processCreditCard(
                paymentDetails, cart.getTotalAmount());
        //TODO save order record on database
    }
}
```

در گام پایانی، اگر یادتان باشد دو سرویس **WebSite** و **RetailPointOfTheSale** داشتیم که مستقیما از کلاس `Order` استفاده می‌کردند. نیاز است تا آنها را نیز بازسازی کنیم. به همین منظور ابتدا به کلاس WebSite این بشارت را می‌دهیم که دیگر لازم نیست از کلاس Order استفاده کند، بلکه می‌تواند از کلاسِ سفارشی که مخصوص به او ساخته ایم بهره ببرد. سپس با استفاده از سبد خرید و اطلاعات پرداخت، `checkout` را انجام می‌دهیم.

```
public class WebSite
{
    public WebSite(
         Cart cart,
         PaymentDetails paymentDetails
    ) throws Exception {
        OnlineOrder order = new OnlineOrder(cart, paymentDetails);
        order.checkout();
    }
}
```

دقیقا مشابه این کار را برای کلاس RetailPointOfTheSale باید انجام دهیم. منتهی در اینجا متد پرداخت نیز مهم است و بر اساس روش پرداختی که کاربر انتخاب می‌کند، کلاس Order مخصوص به آن را ساخته و عمل `checkout` را انجام می‌دهیم.

```
public class RetailPointOfTheSale
{
    public RetailPointOfTheSale(
            Cart cart,
            PaymentDetails paymentDetails,
            PaymentDetails.PaymentMethod paymentMethod
    ) throws Exception {

        Order order;

        if(paymentMethod == Cash){
            order = new PosCashOrder(cart);
        }else{
            order = new PosCreditOrder(cart, paymentDetails);
        }

        order.checkout();
    }
}
```

خب، همه مشکلات حل شد!

## جمع‌بندی

بیایید چیزهایی که یاد گرفتیم را یکبار دیگر مرور کنیم. اصلا مسئولیت یا Responsibility چیست؟

- یک دلیل برای تغییر.
- تفاوتی بین سناریوی طراحی شده و دیدگاه مشتری. چه کسی از کدهای شما استفاده می‌کند؟ شما یا مشتری؟ (قطعا مشتری).
- اینترفیس‌های کوچک مختلفی که کمک می‌کند تا به هدف تک وظیفه‌ای برسیم.

همانطور هم که در بازسازی کدها دیدیم، وظایف را به اینترفیس‌هایی کوچک و تک وظیفه‌ای تقسیم کردیم که هرکدام به خوبی از مسئولیت خود آمده و هیچ نگرانی بابت پیاده‌سازی و دستیابی به آن‌ها نداریم. این تغییر باعث شد تا کلاس Order، تنها مسئولِ اتصالِ کلاس‌های وظایف اصلی شود، تا آن‌ها بتوانند در کنار یکدیگر و (و البته مستقل از یکدیگر) به انجام وظایف مشخص شده‌ی خود بپردازند.

بطور خلاصه، پیروی از اصل تک وظیفه‌ای می‌تواند ما را به Cohesion بالاتر و Coupling پایین‌تر هدایت کند.

کلاس‌های کوچک تک‌وظیفه‌ای با مسئولیت‌های متفاوت می‌توانند باعث می‌شوند یک نرم افزار ما منعطف و قابل توسعه باشد. اگر این اصل را دنبال کنید، متوجه می‌شوید که if/else ها، switchها و در کل عبارات منطقی به مراتب کمتری در کدهای خود ساخته اید. دلیلش بسیار ساده است، زیرا فعالیت‌هایی که رفتارهای مختلف داشته اند را به کلاس‌های کوچک‌تر منتقل و آن‌ها را ایزوله کرده اید که هرکدام از آنها فقط یک وظیفه داشته و فقط همان وظیفه را (البته به درستی) انجام می‌دهد. دیگر نیازی نیست که رفتارهای مختلف کد خود را بررسی کنید، چون شما فقط و فقط یک رفتار را از آن انتظار دارید و کلاس شما باید همان نتیجه‌ای که مد نظر دارید، در اختیار شما بگذارد.

البته کار ما همینجا تمام نمی‌شود. هنوز کلاس OrderOnline چند وظیفه بی ربط به هم را انجام می‌دهد که باید از دل آن جدا شوند. کلاس OrderOnline و PosCreditOrder هیچ چیز به ما درباره وابستگی‌هایشان نمی‌گویند. در نتیجه هنوز نیاز است اصول دیگری نیز در کدهایمان اعمال کنیم، اصولی مانند:

- Open Close Principle
- Interface Segregation Principle
- Dependency Injection Principle

هرسه‌ی این مفاهیم بخشی از مفهوم بزرگتری به نام طراحی شئ گرا یا Object Oriendet Design هستند، که در نهایت باهم مرتبط بوده با مفهوم دیگری به نام Separation Of Concern یا تفکیک دغدغه‌ها به ما در بهبود طراحی سیستم کمک می‌کنند. در فصل‌های پیش رو به تک تک این موارد خواهیم پرداخت.

از شما درخواست می‌کنم برای فهم دقیق‌تر و آشنایی با جزئیات هرکدام از مفاهیم بالا، کتاب Clean Code از Uncle Bob را مطالعه کنید.

# اصل دوم
# Open-Closed Principle

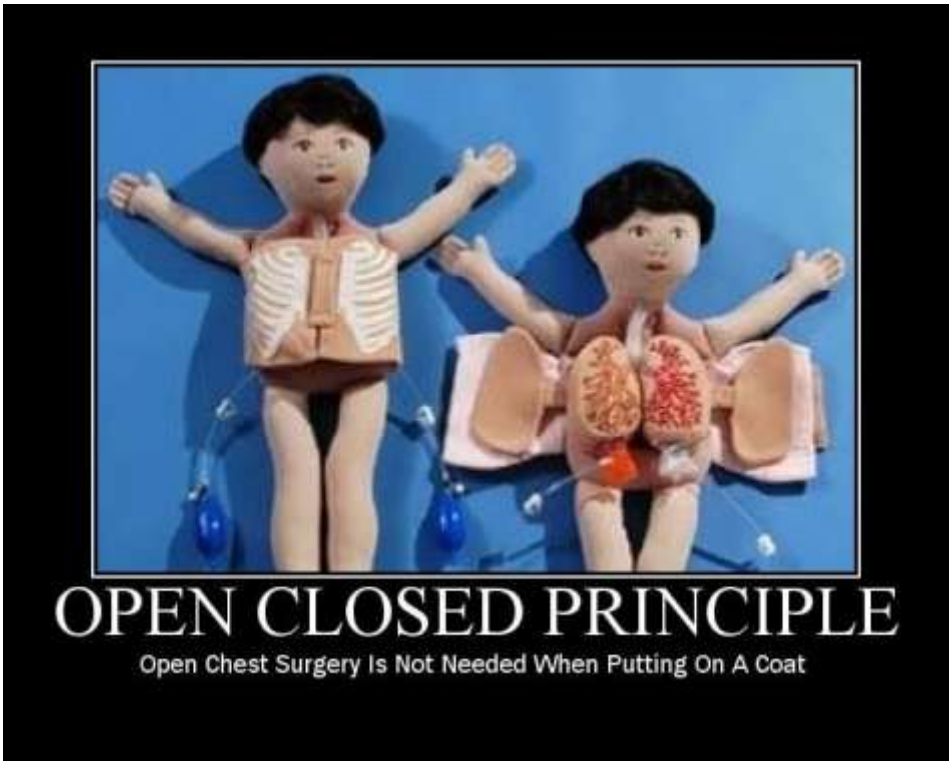
*«Open chest surgery is not needed when putting on a coat!»*

## مقدمه

در این بخش می‌خواهیم به دومین اصل SOLID یعنی **اصل باز–بسته** یا همان **Open-Closed** بپردازیم. مانند بخش قبلی، در ابتدا به مفهوم OCP خواهیم پرداخت، سپس مشکلات را تشریح کرده و برای آن یک مثال زده و سعی می‌کنیم تا آن مثال را بازنویسی کرده و اصل OCP را در آن پیاده سازی کنیم.

## تعریف

همانطور که در توضیحات بخش پیش نیازها به آن اشاره کردم، اصل باز-بسته بیان می‌کند که اجزای نرم افزار (مانند کلاس‌ها، ماژول‌ها، فانکشن‌ها و غیره) باید نسبت به توسعه باز و نسبت به تغییر/اصلاح بسته باشد. تصویر عجیب و جالبی درباره‌ی این اصل وجود دارد.



به طور تحت لفظی این‌گونه ترجمه می‌کنیم که «وقتی کت پوشیدی، احتیاجی نیست عمل جراحی قلب باز کنی». شاید درک این جمله از مفهوم OCP نیز سخت‌تر بنظر برسد، درحالی که بسیار لطیف منظور خود را رسانده و می‌گوید وقتی فرد تمام اجزای بدن خودش را تنها با یک کت پوشانده و درواقع به‌جای قفسه سینه تنها یک پالتو پوشیده است، به راحتی با کنار زدن پالتو یا کت می‌توانیم به داخل آن دسترسی پیدا کنیم (مانند این عروسک‌ها) دیگر احتیاجی به جراحی عمل قلب باز، بریدن پوست، جدا کردن قفسه سینه و همچنین عوارض پس از آن نداریم.

به همین ترتیب هنگامی که می‌خواهید نرم افزار خود را گسترش دهید، نیازی نیست کل سیستم خود را جراحی کرده و داخل آن را حفاری کنید تا بتوانید رفتار آن را تغییر دهید. بلکه باید بتوانید با افزودن یک قابلیت جدید، کلاس جدید یا حتی یک فانکشن جدید سیستم خود را گسترش دهید. دستیابی به رفتار جدید، بدون نیاز به هیچ تغییر یا اصلاحی در کلاس‌ها و توابع موجود، باید وجود داشته باشد.

دکتر برتراند مایر[۱] در کتاب «ساختار نرم‌افزار شئ گرا»[۲] قواعد اصل باز-بسته را اینطور تشریح می‌کند:

- **باز برای گسترش:** امکان افزودن رفتار جدید در آینده وجود داشته باشد.
- **بسته برای تغییر:** نیازی به تغییر کدهای منبع یا باینری نباشد.

معنای این حرف چیست؟ همانطور که قبلا هم گفتم، برای افزودن یک امکان یا رفتار جدید، نباید احتیاجی به تغییر در سورس‌کد برنامه یا کدهای باینری داشته باشید، برای رسیدن به این هدف نباید کدهایی که هم اکنون وجود دارند دوباره کامپایل شوند.

خب، حالا همه این‌ها را می‌دانیم. اما چطور این کار را انجام دهیم؟ چطور رفتار سیستم را تغییر داده یا چیزی به آن بیافزاییم، بدون آن که تغییری در کدها ایجاد کرده باشیم.

---

[۱] Dr. Bertrand Meyer

[۲] Object-Oriented Software Construction (1998)

کلید ماجرا، استفاده از انتزاع‌ها یا درواقع همان Abstractionها در برنامه است. هنگامی که این مقولات را در برنامه خود استفاده کنیم، دیگر هیچ محدودیتی در تعداد روش‌های مختلفی که می‌توانیم آن انتزاع‌ها را پیاده‌سازی کنیم نداشته، بنابر این هیچ محدودیتی نیز برای تعداد روش‌هایی که می‌توانیم رفتار کد را با استفاده از این انتزاع‌ها تغییر دهیم نخواهیم داشت.
اما منظور از انتزاع‌ها یا Abstractionها چیست؟ در اکثر زبان‌های برنامه‌نویسی که قابلیت شئ گرایی دارند، دو شئ زیر به عنوان انتزاعاتِ برنامه در نظر گرفته می‌شوند:

- Interface ها
- وکلاس‌های Abstract

البته در کدهای پروسه‌ای یا رویه‌ای[1] نیز با استفاده از پارامترها می‌توانیم تا حدی این اصل را در برنامه رعایت کنیم.

بهتر است مستقیما به سراغ یک مثال رفته تا گفته‌های بالا را حقیقتاً درک کنیم.

## مثال

در این مثال یک سبد خرید را فرض می‌کنیم، که می‌خواهیم در آن مقدارِ هزینه‌ی کل را به ازای sku[2] محصولاتی که در سبد وجود دارد محاسبه کنیم.
سبد خریدِ ما، شامل آرایه‌ای از OrderItem ها است. که هر OrderItem شامل دو پارامتر `sku` و `quanitity` است. در کد زیر این مدل را مشاهده می‌کنید.

---

[1] Procedural

[2] عبارت sku مخفف *Stock Keeping Unit* و به معنی واحد نگهداری موجود می‌باشد. sku برای یک محصول می‌تواند عدد یا حروف یا ترکیبی از این دو باشد. درواقع sku شناسه منحصر بفرد یک محصول است.

```
public class OrderItem
{
    private String sku;
    private int quantity;

    public OrderItem(String sku, int quantity)
    {
        this.sku = sku;
        this.quantity = quantity;
    }

    public String getSku() { return sku; }
    public int getQuantity() { return quantity; }
}
```

در ادامه کلاس `Cart` را داریم. در این کلاس دو متد داریم؛ یکی برای افزودن آیتم به لیست خرید `Add(OrderItem orderItem)` و دیگری برای محاسبه هزینه کل سبد خرید `totalAmount()` .

```
public class Cart
{
    private List<OrderItem> items;

    public Cart() { this.items = new ArrayList<OrderItem>(); }

    public void Add(OrderItem orderItem){ items.add(orderItem); }

    public float totalAmount(){...}
}
```

همانطور که گفتم، در داخل متدِ `totalAmount` عملیات محاسبه‌ی هزینه‌ی کل سبد خرید انجام می‌شود، اما چطور؟ هر OrderItem دارای یک **sku** خاص است که با عبارت خاصی شروع می‌شود.

برای مثال `EACH_WIDGET` که نشان می‌دهد محصولی داریم به نام **WIDGET** که به ازای هرکدام از آنها مبلغی (مثلا ۲ دلار) تعیین شده است یا `WEIGHT_PEANUTS` که به ازای وزن (گرم/کیلوگرم) مبلغ آن تعیین می‌گردد. بهتر است ابتدا به این متد نگاهی بیاندازیم.

```
public float totalAmount()
{
    float total = 0;

    for(OrderItem orderItem : items)
    {
        if (orderItem.getSku().startsWith("EACH"))
        {
            total += orderItem.getQuantity()*2;
        }
        else if (orderItem.getSku().startsWith("WEIGHT"))
        {
            // quantity is in grams, price is per kg
            total += orderItem.getQuantity()*6/1000;
        }
        else if (orderItem.getSku().startsWith("SPECIAL"))
        {
            // $0.40 each; 3 for a $1.00
            total += orderItem.getQuantity()*.4;
            int setsOfThree = orderItem.getQuantity()/3;
            total -= setsOfThree*.2;
        }
        // more rules are coming!
    }

    return total;
}
```

همانطور که می‌بینید، درحال حاضر ۳ نوع محصول یا نحوه محاسبه داریم که به ازای هر محصول (در حلقه for) با استفاده از sku (که با چه عبارتی شروع می‌شود) نوع آن را پیدا کرده و بر اساس نوعِ آن، به شرح زیر محاسبه را انجام می‌دهیم:

- اگر با `EACH` شروع شده باشد، به ازای هر quanitity (که اینجا تعداد محصول است) ضرب در ۲ دلار، قیمت آن را حساب می‌کنیم.
- اگر با `WEIGHT` شروع شده باشد، به ازای هر quanitity (که اینجا وزن محصول است، گرم به کیلوگرم) ضرب در ۶ دلار، قیمت آن را حساب می‌کنیم.
- و اگر با `SPECIAL` شروع شده باشد، به ازای هر quanitity (که اینجا تعداد محصول است) ضرب در ۰,۴ دلار (۴ سنت)، قیمت آن را محاسبه می‌کنیم. البته فروشنده در

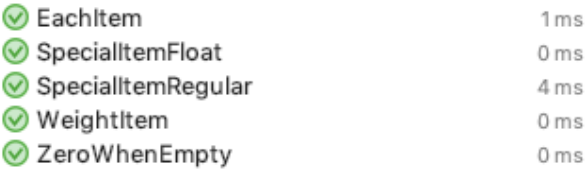
اینجا تصمیم داشته برای این محصول خاص به ازای هر ۳ عدد از محصول، ۲۰٪ تخفیف در نظر بگیرد که ما آن را نیز اعمال کردیم.

سپس برای کلاسِ `cart`، یونیت تست هم نوشتیم. اجازه بدهید آن را نیز بررسی کنیم.

```
public class CartTest
{
    private Cart cart;

    @Before
    public void Setup() throws Exception {
        cart = new Cart();
    }

    @Test
    public void ZeroWhenEmpty() {
        Assert.assertTrue(0 == cart.totalAmount());
    }

    @Test
    public void EachItem() {
        cart.Add(new OrderItem("EACH_WIDGET", 1));
        Assert.assertTrue(2 == cart.totalAmount());
    }

    @Test
    public void WeightItem() {
        cart.Add(new OrderItem("WEIGHT_PEANUTS", 500));
        Assert.assertTrue(3 == cart.totalAmount());
    }

    @Test
    public void SpecialItemRegular() {
        cart.Add(new OrderItem("SPECIAL_CANDYBAR", 6));
        Assert.assertTrue(2 == cart.totalAmount());
    }

    @Test
    public void SpecialItemFloat() {
        cart.Add(new OrderItem("SPECIAL_CANDYBAR", 2));
        Assert.assertTrue(0.8f == cart.totalAmount());
    }
}
```

ابتدا در متد `Setup()` مقدار دهی اولیه را انجام دادیم. در ادامه کل هزینه برای سبد خالی تست کرده، سپس برای هرکدام از آیتم‌های *EACH* ، *WEIGHT* و *SPECIAL* تست جداگانه‌ای نوشتیم.

همانطور که انتظار می‌رفت تمامی تست‌ها با موفقیت پاس شد.




مشکل آنجایی حاصل می‌شود که مدیر سایت از ما می‌خواهد تا محصولی جدید با یک sku جدید ایجاد کنیم. برای اینکار نیاز است تا وارد کلاس `Cart` شده و فانکشن `totalAmount()` را برای یک sku جدید تغییر دهیم. درواقع باید به انتهای `if else` های مان یک `else` دیگر اضافه کرده و نحوه محاسبه قیمت جدید را در آن قرار دهیم. اینکار دقیقا نقطه مقابل اصل OCP است! یادتان هست؟ در حین اینکه باید بتوانیم امکانات جدید را به برنامه اضافه کنیم، حق نداریم کدهای قبلی خود را تغییر داده یا آن‌ها را اصلاح کنیم.

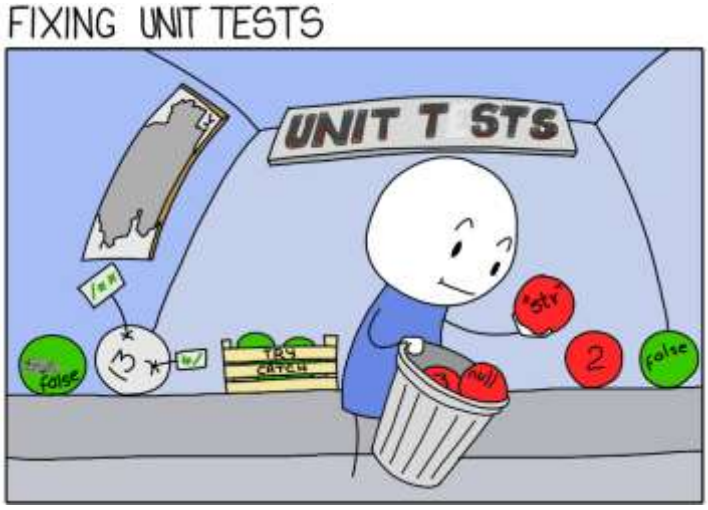
## بررسی و تحلیل مشکل

اولین مشکلی که با آن روبرو هستیم این است که، به ازای هر باری که بخواهیم یک *rule* جدید اضافه کنیم، نیاز داریم کدهای خود را تغییر دهیم. هر تغییر جدیدی که در سیستم ایجاد می‌کنیم خطاهای جدید ایجاد می‌کند و ما مجبوریم testها را دوباره انجام داده و در محاسبه‌ی قیمت تغییرات را اعمال کرده و دوباره کاری های مختلفی را پیش ببریم. یعنی به‌جای اینکه *rule* قیمت‌ها را تست کنیم، مجبوریم در `Cart` تغییرات را ایجاد کرده و به ازای هر تغییر `Cart` را دوباره تست کنیم.

در حالت ایده‌آل، می‌خواهیم از ایجاد تغییراتی که به صورت آبشاری روی دیگر ماژول‌های برنامه ما تاثیر می‌گذارد جلوگیری کنیم. در موردِ مثالی که زدیم، چیزهای مختلفی وجود دارد که به رفتار `Cart` وابسته اند، و به ازای هر تغییری که در نحوه‌ی محاسبه‌ی قیمت‌ها در `Cart` ایجاد می‌کنیم همه آن بخش‌ها نیز متاثر از رفتارِ جدیدِ `Cart`، تغییر می‌کنند.

هنگامی که قرار باشد از **OCP** پیروی کنیم، برای گسترش سیستم از کلاس‌های جدید استفاده می‌کنیم. درواقع به ازای هر تغییری که می‌خواهیم در رفتارِ سیستم ایجاد کنیم یا امکان جدیدی به آن بیافزایم، تنها با ایجاد یک کلاس جدید می‌توانیم به این هدف برسیم. این کلاس‌ها بسیار کوچک هستند و هرکدام دقیقا روی یک هدف یا یک وظیفه خاص متمرکز اند. بدین ترتیب از **اصل تک وظیفه‌ای** یا همان **SRP** نیز پیروی می‌کنیم.

ایجاد کلاس‌های جدید می‌تواند در این مورد کمک شایانی به ما بکند. از مزایای ایجاد کلاس‌های کوچک این است که می‌توانیم بخش‌های مختلف اپلیکیشن را از اهم جدا کنیم. کلاس‌های جدید، کدهایی هستند که قبلا در سیستم شما وجود نداشته و هیچ وابستگی‌ای به بخش‌های دیگر و قدیمی سیستمی شما ندارد. بدین سبب وابستگی‌های سیستم را تا حد ممکن کاهش می‌دهیم. به همین دلیل به راحتی و به سرعت می‌توانید یک کلاس جدید طراحی و آن را ایجاد کرده، سپس وظیفه‌ی جدید را برای آن تعریف کرده و به آسانی تست‌های آن را پاس کنید.




در حال حاضر ۳ روش جهت دستیابی به OCP در برنامه وجود دارد.

**استفاده از Parameterها در برنامه‌نویسی رویه‌ای:** با قرار دادن پارامترها در برنامه، کلاس یا فانکشن‌ها به مشتری اجازه می‌دهید تا از طریق خصوصیات این پارامترها، رفتار سیستم را کنترل کند. برای مثال با استفاده از ارسال یک متغییر `String` به عنوان پارامتر به یک

کلاس یا فانکشن، می‌توانید نوع رفتاری که باید نشان دهد را نیز تعیین کنید. می‌توانید این رویکرد را با `delegate` ها و یا عبارات `lambda` ترکیب کنید و آن وقت ببینید که با چه قدرتی می‌توانید رفتار یک کلاس و عملکرد آن را تغییر دهید.

برای اینکه موضوع روشن شود، درخواست می‌کنم به این مثال توجه کنید. تصور کنید یک دوربین دیجیتال دارید که در آن با زبان (`Java` یا هر چیز دیگری) برنامه‌ای نوشته شده که پس از اتصال آن به کامپیوتر، همه‌ی عکس‌های شما را بطور اتوماتیک به درایو C انتقال می‌دهد. یک روزی تصمیم می‌گیرید (یا خودم تصمیم گرفتم) که می‌خواهیم تصاویر موجود در دوربین به درایو *D* و پوشه *Sadra* منتقل شود، فرض کنید برای انجام این‌کار به صورت `hard code` تغییرات را در داخل برنامه ایجاد کرده و به صورت دستی مسیر پوشه را تغییر می‌دهم. این برنامه *برای من* به راحتی و بدون هیچ مشکلی کار می‌کند تا روزی که تصمیم می‌گیرم آن روی وبلاگ یا گیتهاب خودم منتشر کنم. اینگونه می‌شود که خیلی‌ها به وبلاگ من مراجعه کرده (شتر در خواب بیند پنبه دانه!) و تصمیم می‌گیرند از این برنامه استفاده کنند. اولین مشکلی که با آن روبرو می‌شوند این است که پس از اجرای برنامه، با خطای مواجه می‌شوند که می‌گوید فولدر *Sadra* را پیدا نمی‌کند یا اصلا به درایوی به نام D دسترسی ندارد! اگر آن‌ها نیز مانند ما برنامه‌نویس باشند، سراغ کدها رفته، محل تعیین مسیر انتقال فایل‌ها را پیدا کرده و مسیر مورد نظر خود را در آن اصلاح می‌کنند. اگر هم برنامه‌نویس نباشند، عملا نمی‌توانند از برنامه استفاده کنند! هنگامی که قرار باشد برای ایجاد هر رفتار جدیدی در سیستم، در کدهای قدیمی تغییرات ایجاد کنیم، آن‌گاه از **OCP** تخطی کرده‌ایم. حال چه باید کرد؟

به‌جای اینکه مسیر فایل انتقالی را در برنامه قرار دهیم، بهتر است امکانی فراهم کنیم تا (مثلا در کامندلاین) مسیر انتقال فایل‌ها را از کاربر درخواست کند. بدین ترتیب هربار که کاربر برنامه را اجرا می‌کند، از او مسیر انتقال را دریافت کرده و به عنوان پارامتر به کلاس مربوطه تحویل می‌دهد، در نتیجه فایل‌ها به آن مسیری که کاربر در هنگام اجرا تعریف کرده است منتقل می‌شوند.

**استفاده از ارث‌بری[۱] و الگویِ Template Method[۲]:** راه دیگری که می‌توانیم جهت پیروی از OCP طی کنیم، استفاده از ارث‌بری یا الگوی طراحیِ Template Methot در برنامه‌های شئ‌گرا است. با استفاده از الگوی Template Method می‌توانیم الگوریتم و رفتار پیش‌فرض را در یک کلاس تعریف کرده و پیاده سازیِ برخی قدم‌هایِ آن را به کلاس‌هایی که از آن ارث‌بری کرده اند محول کنیم. این الگو به زیرکلاس‌ها اجازه می‌دهد تا برخی رفتارهای کلاس بالایی را *override* کرده و بدون تغییر در رفتار کلاس بالایی، تغییرات مورد نظر خود را اعمال کنند.

**استفاده از Composite و الگوی Strategy:** که ما در این بخش دنبال این هستیم که با استفاده از این دو الگو برنامه‌ی خود را بازسازی کنیم. در این بخش می‌خواهیم علاوه بر ارث‌بری از ترکیب (Composite) استفاده کنیم که از الگوی طراحی Strategy استفاده می‌کند.

در الگوی Strategy، کدِ مشتری، کدی که آن رفتار را فراخوانی می‌کند، به انتزاع یا آن *Abstraction*هایی که قبلا گفتیم بستگی دارد. این امر به ما کمک می کند تا حالتی ماژولار به سیستم بدهیم و هرگاه نیاز داشتیم که کلاس رفتار خاصی را داشته باشد، صرفا آن انتزاع را در آن تزریق می‌کنیم.

بنابراین هدفی که دنبال می‌کنیم این است که با استفاده از وراثت، کلاس‌ها و اینترفیس‌هایی که مد نظر داریم را پیاده‌سازی کنیم. و مشتری از ترکیب (یا همان Composite) استفاده کرده و رفتار خاصی که میخواهد در کلاس داشته باشد را با استفاده از اینترفیس‌ها و کلاس‌های بالاتر در کلاس پایینی پیاده‌سازی می‌کند.

---

[۱] Inheritance

[۲] یکی از الگوهـای برنـامـه‌نویســی (Software Design Patterns) کـه از نوع الگوهـای «رفتـاری» (Behavorial Patterns) است.

می‌دانم مسائل کمی گیج کننده و بیش از حد انتزاعی شده است. بهتر است دوباره سراغ کد رفته و ببینیم همه‌ی این حرفهایی که زده ایم را چگونه می‌توانیم در آن ماژول محاسبه قیمت سبد خرید پیاده کرده و طراحی آن را بهبود دهیم. شاید با استفاده از OCP برنامه‌ی ما بهتر کار کرده و کدهای ما تمیزتر شود.

## بازسازی کد

بیایید قبل از هرچیزی معضل اصلی را حل کنیم، و آن چیزی نیست جز `rule` های مختلفی که برای محاسبه‌ی قیمت انوع محصولات داریم. همانطور که قبلا هم گفتم، بهترین روش برای پیاده سازی این امر استفاده از انتزاعات یا همان *Abstract/Interface* ها است. پس در اولین قدم، یک اینترفیس با عنوان `IPriceRule` می‌سازیم که شامل دو متد انتزاعی `isMatch()` (برای تطابق نوع *rule*) و دیگری `calculatePrice()` (برای محاسبه قیمت) است.

```
public interface IPriceRule
{
    boolean isMatch(OrderItem orderItem);
    float calculatePrice(OrderItem orderItem);
}
```

سپس با استفاده از این اینترفیس، *rule* هایی که به صورت *if else* در *Cart* وجود داشت را خارج کرده و با استفاده از کلاس‌های کوچک پیاده سازی می‌کنیم.

```
public class EachPriceRule implements IPriceRule
{
    @Override
    public boolean isMatch(OrderItem orderItem) {
        return orderItem.getSku().startsWith("EACH");
    }

    @Override
    public float calculatePrice(OrderItem orderItem) {
        return orderItem.getQuantity()*2;
    }
}
```

```
public class PerGramPriceRule implements IPriceRule
{
    @Override
    public boolean isMatch(OrderItem orderItem) {
        return orderItem.getSku().startsWith("WEIGHT");
    }

    @Override
    public float calculatePrice(OrderItem orderItem) {
        return orderItem.getQuantity()*6/1000;
    }
}
```

```
public class SpecialPriceRule implements IPriceRule
{
    @Override
    public boolean isMatch(OrderItem orderItem) {
        return orderItem.getSku().startsWith("SPECIAL");
    }

    @Override
    public float calculatePrice(OrderItem orderItem) {
        float total = orderItem.getQuantity()*.4f;
        int setsOfThree = orderItem.getQuantity()/3;
        total -= setsOfThree*.2f;
        return total;
    }
}
```

حال نگاهی به کلاس `Cart` بیاندازیم. در این کلاس یک وظیفه داریم و آن هم محاسبه هزینه کل سبد خرید است. در این مرحله باید وظیفه‌ی *محاسبه هزینه کل سبد خرید* را از به صورت یک اینترفیس از کلاس خارج کنیم.

```
public interface IPricingCalculator
{
    float CalculatePrice(OrderItem orderItem);
}
```

و سپس کلاس مربوط به محاسبه قیمت آیتم‌ها را پیاده سازی کنیم.

```
public class PricingCalculator implements IPricingCalculator
{
    private List<IPriceRule> pricingRules;

    public PricingCalculator() {
        this.pricingRules = new ArrayList<IPriceRule>();
        pricingRules.add(new EachPriceRule());
        pricingRules.add(new PerGramPriceRule());
        pricingRules.add(new SpecialPriceRule());
    }

    @Override
    public float CalculatePrice(OrderItem orderItem) {
        float total = pricingRules.stream()
                .filter(rule -> rule.isMatch(orderItem))
                .findFirst()
                .get()
                .calculatePrice(orderItem);
        return total;
    }
}
```

به `PricingCalculator` دقت کنید. در این کلاس آرایه‌ای از *rule* ها داریم به نام `pricingRules` که نگهدارنده همه‌ی *rule*های ایجاد شده توسط ما خواهد بود. در متد سازنده‌ی کلا س[1] ruleهای موجود را به این آرایه اضافه می‌کنیم. سپس در متد `CalculatePrice()` کاری که باید بکنیم این است که با استفاده از عبارات *lambda* داخل هرکدام از این *rule*ها رفته و ببینیم که نوع محصولی که به صورت پارامتر معرفی شده با کدامیک از *priceRule*ها یکی است؛ سپس محاسبه‌ی قیمت مربوط به آن را انجام می‌دهیم. به همین سادگی و به همین تمیزی!

در ادامه متد `totalAmount()` در کلاس *Cart* را با توجه به اینترفیس‌ها و کلاس‌های پیاده سازی شده‌ی جدیدی که ساخته‌ایم ویرایش می‌کنیم.

[1] Class Constructor Method

```
public class Cart
{
    private List<OrderItem> items;
    private IPricingCalculator pricingCalculator;

    public Cart()
    {
        pricingCalculator = new PricingCalculator();
        this.items = new ArrayList<OrderItem>();
    }

    public void Add(OrderItem orderItem)
    {
        items.add(orderItem);
    }

    public float totalAmount()
    {
        float total = 0;

        for(OrderItem orderItem : items)
        {
            total += pricingCalculator.CalculatePrice(orderItem);
        }

        return total;
    }
}
```

حال که این همه شگفتی آفریدیم، اجازه دهید تست‌هایی که نو شته بودیم را یکبار دیگر اجرا کرده و از پاس شدن همه‌ی آنها مطمئن شویم.


خب، بیایید یک مورد جدید به تست‌مان اضافه کنیم. مثلا محصولی داریم که میخواهیم به ازای هر ۴ عدد، ۱ عدد رایگان باشد (۴تا بخر، ۵تا ببر). اول تست آن را می‌نویسیم.

```
@Test
public void SpecialItemBuy4GetOneFree()
{
    cart.Add(new OrderItem("B4GOF_PENCIL", 5));
    Assert.assertTrue(4 == cart.totalAmount());
}
```

و تست را اجرا می‌کنیم، که قاعدتا تست ما باید *fail* شود، زیرا این rule جدید را نداریم و sku جدید قابل شناسایی نیست.

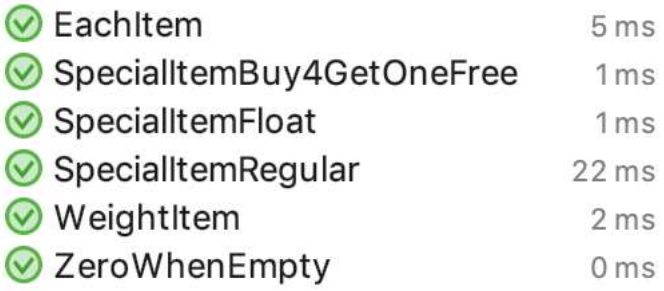
تنها کاری که باید انجام دهیم این است که یک کلاس جدید درست کنیم که اینترفیس `IPriceRule` را در خود پیاده سازی کرده باشد.

```
public class SpecialPrice4BuyGetOneFree implements IPriceRule
{
    @Override
    public boolean isMatch(OrderItem orderItem) {
        return orderItem.getSku().startsWith("B4GOF");
    }

    @Override
    public float calculatePrice(OrderItem orderItem) {
        float total = orderItem.getQuantity()*1;
        int setsOfFive = orderItem.getQuantity()/5;
        total -= setsOfFive*1;
        return total;
    }
}
```

و بعد *rule* جدید را به لیست *rule*ها در کلاس `PricingCalculator` اضافه کنیم.

```
public PricingCalculator()
{
    this.pricingRules = new ArrayList<IPriceRule>();
    pricingRules.add(new EachPriceRule());
    pricingRules.add(new PerGramPriceRule());
    pricingRules.add(new SpecialPriceRule());
    pricingRules.add(new SpecialPrice4BuyGetOneFree()); //new rule
}
```

تست‌ها را دوباره اجرا می‌کنیم و خواهیم دید که همه آن‌ها با موفقیت پاس می‌شوند.




خب، همه‌ی آن کاری که لازم بود انجام دهیم همین بود. حالا دیگر نیازی نیست کلاس بزرگ Cart را دستکاری کرده و نگران از تغییرات و ایجاد باگ‌های جدید باشیم. تنها کافیست به ازای هر *rule* جدید، کلاس مربوط به آن را بسازیم، آن را به لیست *rule*ها اضافه کنیم و در گام آخر تست‌ها را به راحتی پاس کنیم.

## جمع بندی

حقیقتا چه زمانی باید OCP را اعمال کنیم؟ خیلی مهم است تا این موضوع را بخاطر داشته باشید که بدون دلیل و فکر (به قول انگلیسی‌ها *willy-nilly* یا همان هردمبیل خودمان) در هر جایی از کد که دست‌تان می‌رسید از *Abstract Class* و *Interface* استفاده نکنید! نتیجه‌ی آن سختی بیش از حد و پیچیدگی غیر ضروری است، زیرا ادامه کار هم برای شما و هم برای کسانی که می‌خواهند مسیر شما را در توسعه‌ی کدها ادامه دهند سخت خواهد شد.

تجربه، بله تجربه اولین چیزی است که باید در این مواقع به آن رجوع کرده و ببینید تجربه‌ی شما در آن مورد خاص چه می‌گوید؟ اگر تجربه‌ی مناسبی درباره آن مشکل خاص داشته باشید، آنگاه می‌توانید بدون هیچ نگرانی آن بخش از سیستم را تغییر داده و OCP را در آن اعمال کنید.

اما و اما اگر هیچ تجربه‌ای در این مورد نداشتید و یا همیشه در پروژه‌هایی کار می‌کنید که مسائل و کانسپت‌های جدید در آن رخ می‌دهد، یاد این ضرب المثل انگلیسی بیافتید:

> *Fool me once, shame on you; Fool me twice, shame on me.*

به این معنی که «اگر یکبار مرا دست انداختی، شرم بر تو باد؛ اگر دو بار مرا دست انداختی، شرم بر من!» این ضرب المثل معادل همان ضرب المثل فارسی خودمان است که می‌گوید «آدم عاقل از یک سوراخ دو بار گزیده نمی‌شود». اما این ضرب المثل چطور به ما کمک می کند؟ ساده است، از مراحل زیر پیروی کنید:

۱. در ابتدا ابدا OCP را اعمال نکنید، برنامه را به صورت ساده بنویسید. (با همان *hard-code*ها، همان *if-else*ها و همان چیزهایی که همیشه می‌نویسیم، ساده و بی تکلف. البته مطمئن شوید که حتما تست‌ها پاس می‌شوند!)

۲. اگر یک بار تغییری در آن بخش از برنامه نیاز شد، آن تغییر را اعمال کنید. (البته طوری که برنامه هنوز کار کند و تست‌ها پاس شود.)

۳. اگر برای بار دوم هم تغییر لازم شد، برنامه را ریفکتور کرده و OCP را اعمال کنید. (اینجا دقیقا همان *سر به زنگاه* است. کلاس *Abstract* را بسازید، *Interface*ها را ایجاد کنید و آن‌ها در کلاس‌های مربوطه پیاده سازی کرده و لاجیک یا منطقی برنامه را به آن کلاس‌های کوچک که تنها یک وظیفه دارند منتقل کنید.)

این عبارت معروف انگلیسی را نیز هیچوقت فراموش نکنید:

> *There Ain't No Such Thing As A Free Lunch.*

این جمله پر کاربرد در اوایل دهه ۱۹۳۰ بشدت رواج پیدا کرده و با اختصار ***TANSTAAFL*** شناخته می‌شود با یک جستجوی ساده می‌توانید به تاریخچه جالب این عبارت دست پیدا کنید. و اما معنای آن این است که «***نمی‌شود در ازای هیچ چیزی، چیزی را طلب کنی***».

هنگامی که OCP را در برنامه خود اعمال می کنید، به روشنی آمار استفاده‌ی شما از انتزاعاتِ *Abstract* و *Interface* بالا می‌رود، که نتیجه‌ی آن افزایش پیچیدگی در برنامه است. هیچوقت نخواهید توانست طرحی داشته باشید که در مقابل تغییر مقاوم باشد. بنا بر مقتضیات همیشه احتیاج

به تغییر وجود خواهد داشت و نمی‌توانید برای همیشه با آن مقابله کنید؛ در نتیجه نیاز دارید تا این اصول را در کدهای خود اعمال کنید که ثمره آن چیزی جز پیچیدگی‌های بیشتر نیست.

بطور خلاصه، اگر در پروژه‌ای بودید که نیاز شد تا OCP را پیاده کنید، ممکن است پیچیدگی‌های برنامه بیشتر شود، اما برنامه شما منعطف‌تر، تمیز تر، قابل استفاده مجدد، قابل تست و قابل نگهداری می‌شود. سعی کنید بفهمید کجا احتیاج به پیاده سازی این اصل دارید و تا جایی که می‌توانید در مقابل ایجاد انتزاعات غیر ضروری مقاومت کنید.

مفاهیم و الگوهایی که در این بخش مرتبط با موضوع بودند عبارت اند از:

- Single Responsibility Principle
- Strategy Pattern
- Template Method Pattern

پیشنهاد می کنم کتاب «اصول، الگوها و تمرین‌های اجایل» نوشته رابرت مارتین (همان آنکل باب خودمان) و میکا مارتین[۱] را نیز در این باره مطالعه کنید.

[۱] Agile Principles, Patterns, and Practices in C# , Micah Martin & Robert Cecil Martin

# اصل سوم
# Liskov Substitution Principle

*« If it looks like a Duck, Quacks like a Duck, But needs batteries,*
*You probably have the wrong Abstraction »*

## مقدمه

در این بخش می‌خواهیم به حرف "L" از اصول ۵ گانه SOLID یعنی اصل جایگزینی لیسکاو یا به اختصار LSP بپردازیم. مانند بخش قبلی، در ابتدا مفهوم LSP را تشریح کرده، سپس مشکلی که ممکن است پیش بیاید را بررسی می‌کنیم؛ برای آن یک مثال می‌زنیم و با یکدیگر آن را بازنویسی کرده و اصل LSP را در آن پیاده سازی می‌کنیم.

## تعریف

بنا بر LSP، زیر کلاس‌ها باید بتوانند بدون هیچ تغییری با کلاس والد (پایه) جایگزین شوند. این اصل را باربارا لیسکاو(۱) در طی سخنرانی در کنفرانسی در سال ۱۹۸۷ با نام انتزاع داده ها و سلسله مراتب معرفی کرد که به همین دلیل این اصل با نام وی معروف شد. باربارا لیسکاو و ژنیت وینگ این اصل را به صورت خلاصه در مقاله ای در سال ۱۹۹۴ فرموله کردند:

> *Subtype Requirement: **Let φ(χ)** be a property provable about objects **χ** of type **T**. Then **φ(γ)** should be true for objects **γ** of type **S** where **S** is a subtype of **T**.*

یکی از معروف‌ترین تصاویری که برای شرح این مفهوم استفاده می‌شود همین این است.

قضیه از این قرار است که اگر یک وسیله‌ای دارید که شبیه اردک است، مثل اردک شنا می‌کند، مثل اردک کوئَک می‌کند، اما احتیاج به باتری دارد، احتمالا انتزاع (کلاس Asbtraction) شما مشکل دارد. درواقع شما نمی‌توانید این اردکِ اسباب بازی را با یک اردک واقعی جایگزین کنید.

برای اینکه بتوانیم جایگزینی را داشته باشیم، کلاس فرزند نباید دو خصیصه زیر را داشته باشد:

۱. رفتار کلاس والد (پدر یا پایه) حذف کند (یا تغییر دهد).

۲. ناورداهای کلاس والد را نقض کند.

و در مجموع نباید در کد چیزی را فراخوانی کنیم تا به واسطه‌ی متوجه شویم که آیا این دو کلاس فرزند (یا drive-type) و والد (base-type) شبیه به هم هستند یا خیر، باهم فرق دارند.

## ارث بری و رابطه IS-A

یکی از اولین چیزهایی که دانشجویان برنامه‌نویسی در اولین قدم‌های آشنایی با OOP یاد می‌گیرند، استفاده از رابطه‌ی ایزاِ (***IS-A***) برای توصیف ارث بری است. اینطور است که می‌گویند کلاس A ایزاِ کلاس B، مثلا یک Employee ایزاِ Person، یا Sqaure ایزاِ Shape، یا Car ایزاِ Vehicle. درواقع می‌گویند A هست B (یا همان ای، بی است).

رابطه‌ی IS-A یک ابزار بسیار معمول در شناسایی اشیا OO است. اما لیسکاو معتقد است به‌جای اینکه بگوییم یک شئ یا چیزی فلان شئ یا چیزی است، باید ببینیم آیا آن شئ یا چیز (در هر شرایط یا مکانی) قابل جایگزینی (***IS-SUBSTITUTBLE-FOR***) با آن شئ که انتظارش را داریم است یا خیر. این موضوع در مثال‌هایی که در ادامه خواهیم زد واضح‌تر خواهد شد.

## ناورداها یا Invariants

یکی از چیزهایی که هنگام صحبت کردن درباره‌ی LSP لازم است تا با آن آشنا باشیم، مفهوم ناوردا (تغییر ناپذیر/پایا) یا *Invariant* است. ناوردا یا ناوردایی یک مفهوم ریاضی/فیزیکی است که از عبارت *Variance* یا تغییرات (نوسانات) مشتق شده است. هر ویژگی یا خصیصه‌ی سیستم

که تحت یک تبدیل خاص، تغییر نکند را ناوردا، تغییر ناپذیر یا پایا می‌نامند. (البته باید توجه داشت که ویژگی‌ای که تحت یک تبدیل خاص ناورداست، لزوما تحت تبدیل‌های دیگر ناوردا نخواهد بود.)

در علوم کامپیوتر، ممکن است با مواردی روبرو شوید که در طول اجرا شدن یک برنامه (یا بخشی از یک برنامه) صادق و پایدار هستند. به این بخش‌هایی که در طول یک مرحله خاص همیشه صادق و پایدار هستند، ناوردا می‌گویند. برای مثال یک حلقه‌ای[۱] که شرط آن صادق (*true*) است.

ناوردا از یک سری مفاهیم قابل قبول از رفتارها، که توسط client یا کلاس‌های دیگری که از آن کلاس استفاده می‌کنند، تشکیل شده است. این‌ها اغلب می‌توانند به عنوان پیش‌شرط یا پس‌شرط (مثلا یک شرط while که در ابتدا یا انتهای حلقه قرار می‌گیرد) برای یک متد بیان شوند، و لزومی ندارد حتما در کدها نمایان باشند.

اغلب یونیت تست‌ها می‌توانند برای ما مشخص کنند که دقیقا انتظار چه رفتاری را از آن کلاس یا متد داریم، و البته این تست‌های واحد هنگامی که کلاس یا تایپ فرزند، آن رفتار را نقض می‌کنند، باید fail شده و شکست بخورند.

در این بین مفهومی وجود دارد به نام Design By Contract. طراحی تحت قرارداد یا همان DbC که با نام‌های برنامه‌نویسی قراردادی[۲]، برنامه‌نویسی با قرار داد[۳] و برنامه‌نویسیِ طراحی تحت قرارداد[۴] نیز شناخته می‌شود. اشاره به این دارد که با استفاده از اینترفیس‌ها برای یک برنامه ***preCondition*** و ***postCondition*** هایی تعریف کنیم تا بتوانیم رفتار آن بخش از سیستم را کنترل کنیم.

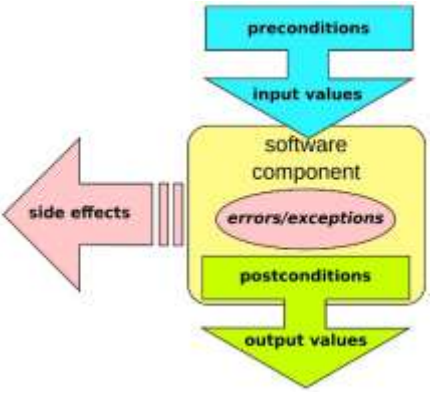
---

[۱] Loop

[۲] Contract Programming

[۳] Programming by Contract

[۴] Design-by-Contract Programming




درواقع با استفاده از interfaceها و متدهایی که در آن تعریف می‌کنیم، نوعی preCondition یا پیش‌شرط‌هایی را در نظر می‌گیریم که کلاسی که آن اینترفیس را پیاده سازی می‌کند، ملزم است حتما آن متدها یا رفتارها و همچنین ناورداها را پیاده کرده و از آن پیروی کند؛ سپس با استفاده از تست‌هایی که می‌نویسیم و پس‌شرط‌ها یا postConditionهایی که تعریف می‌کنیم به آن بخش یا ماژول می‌گوییم انتظار داریم آن رفتار را نشان دهی و مواقعی که نمی‌توانی آن رفتارها را اجرا کنی (به واسطه‌ی ورودی‌های نادرست یا مشکل در کد) یا در روند اجرا مشکلی پیش می‌آید، آن‌ها را با *expception*ها یا *error-handeling*ها مدیریت کنی. بدین وسیله می‌توانیم *side effect* های سیستم یا آن بخش از ماژول را نیز پیشبینی کنیم.

برای پیروی از اصل LSP، کلاس فرزند (*drived-class*) نباید هیچ یک از رفتارها یا محدودیت‌های وضع شده توسط client یا کلاس والد (*base-class*) را نقض کند.

در ادامه، تمامی این مفاهیمی را که تشریح کردیم، در مثال‌های عینی بررسی خواهیم کرد.

## مثال

در این‌جا، یک شئ داریم به نام `Rectangle` که دارای یک `width` و `height` است.

```
public class Rectangle
{
    private int width;
    private int height;

    public int getWidth() { return width; }

    public void setWidth(int width) { this.width = width; }

    public int getHeight() { return height; }

    public void setHeight(int height) { this.height = height; }
}
```

در ادامه شئ دیگری داریم به نام `Square` که از `Rectangle` ارث بری کرده است. با توجه به اینکه می‌دانیم طول و عرض یک مربع باهم برابر است، در نتیجه در هنگام مقدار دهی به طول یا عرض مربع هردو را برابر با همان مقدار قرار می‌دهیم.

```
public class Square extends Rectangle
{
    @Override
    public int getWidth()
    {
        return super.getWidth();
    }

    @Override
    public void setWidth(int width)
    {
        super.setWidth(width);
        super.setHeight(width);
    }

    @Override
    public int getHeight()
    {
        return super.getHeight();
    }

    @Override
    public void setHeight(int height)
    {
        super.setHeight(height);
        super.setWidth(height);
    }
}
```

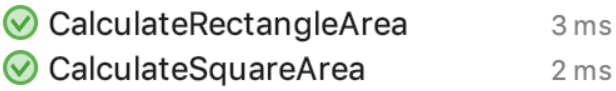
همچنین یک کلاس داریم با نام `AreaCalculator` که وظیفه آن محاسبه مساحت این اشیا است.

```
public class AreaCalculator
{
    public static int CalculateArea(Square square)     {
        return square.getHeight() * square.getHeight();
    }

    public static int CalculateArea(Rectangle rectangle)     {
        return rectangle.getHeight() * rectangle.getWidth();
    }
}
```

در بخش بعدی، برای این دو متدِ چند ریختیِ[۱] `CalculateArea` یونیت تست نوشتیم تا از صحت رفتار آنها مطمئن شویم.

```
public class AreaCalculatorTest
{
    @Test
    public void CalculateSquareArea() {
        Square square = new Square();
        square.setWidth(2);
        Assert.assertEquals(4,
                  AreaCalculator.CalculateArea(square));
    }

    @Test
    public void CalculateRectangleArea() {
        Rectangle rectangle = new Rectangle();
        rectangle.setWidth(2);
        rectangle.setHeight(3);
        Assert.assertEquals(6,
                  AreaCalculator.CalculateArea(rectangle));
    }
}
```

خب، تســـت اول، یعنی `CalculateSquareArea` یک شـــئ از `Square` ایجاد کرده، یکی از اضـلاع را مقدار دهی می‌کند و انتظار دارد که مسـاحت برابر با ۴ شـود. تسـت دوم، یعنی `CalculateRectangleArea` نیز یک شـــئ از نوع `Rectangle` ایجاد کرده و با مقدار دهی طول و عرض آن، انتظار دارد مسـاحت مربع ۶ شـود. تسـت‌ها با موفقیت پاس می‌شوند.




حال می‌خواهیم یک تست دیگر نیز اضافه کنیم. همه‌ی ما می‌دانیم که مربع یک نوع مستطیل است (قانون *IS-A* اینجا جواب می‌دهد) که طول و عرض آن باهم برابر است. از لحاظ ریاضی مساحت هردوی اینها، به یک روشِ صادق محاسبه می‌شوند. حال، آیا مربع قابل جایگزینی (*IS-SUBSTITUTBLE-FOR*) با مستطیل نیز است؟

---

[۱] Polymorphism

بگذارید با آن یونیت تستی که گفتم، این مورد را نیز بررسی کنیم.

```
@Test
public void CalculateMixtureShapeArea()
{
    Rectangle rectangle = new Square();
    rectangle.setWidth(4);
    rectangle.setHeight(5);
    Assert.assertEquals(20,
         AreaCalculator.CalculateArea(rectangle));
}
```

میخواهیم یک شئ `Rectangle` بسازیم و از آنجایی که قبلا گفتیم و اعتقاد داریم مربع یک مستطیل است، پس شئ مستطیل را با یک `Square` مقدار-دهی اولیه[1] می‌کنیم. سپس طول و عرض را به آن می‌دهیم. مطابق با چیزی که در ذهن داریم انتظار می‌رود طول ضرب در عرض برابر شود با ۲۰. بنظر شما چه اتفاقی می‌افتد؟ آیا تست با موفقیت پاس می‌شود؟ (بدیهی-ست که خیر!)

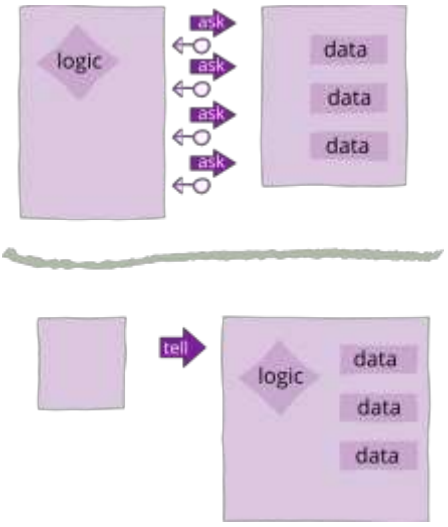

همانطور که از نتیجه تست مشخص است، تست پاس نشد! اما چرا؟ واضح است. شئ‌ای ساختیم از نوع مربع، که مستطیل است. وقتی به `width` آن مقدار ۴ می‌دهیم، ارتفاع آن نیز ۴ می‌شود؛ و وقتی به `height` آن مقدار ۵ می‌دهیم، عرض آن نیز ۵ می‌شود. انتظارِ یک مستطیلی داریم با طول و عرض‌های متفاوت، اما در حقیقت مربعی داریم که طول و عرض‌های آن برابر است با آخرین مقداری که اختصاص داده ایم. پس، نتیجه تست برابر است با `۵×۵=۲۵`، نه ۲۰! در نهایت تست ما دچار خطا می‌شود. اینجاست که در می‌یابیم **`Square` قابل جایگزینی با `Rectangle` نیست.**

در اینجا شئ مربع رفتار مستطیل را نقض کرده. زیرا در مستطیل طول و عرض جدا از هم مقدار دهی می‌شوند، اما در مربع هنگام مقدار دهی هرکدام، مقدار دیگری را نیز به آن `value`

[1] Initialization

تغییر می‌دهیم. این خصیصه مربع، رفتار مستطیل را نقض کرده و سبب به وجود آمدن مشکل می‌شود.

اما مشکل دیگری نیز در طراحی‌مان وجود دارد که باید به آن توجه کنیم. در کلاس ***AreaCalculator*** اصل `tell, don't ask` را نقض کرده‌ایم.




اصل «***دستور بده، نپرس***» را اولین بار اندی هانت و دیو توماس[۱] در مجله IEEE Software column تشریح کرده و از قول اَلک شارپ[۲] می‌نویسند:

> *Procedural code gets information then makes decisions. Object-oriented code tells objects to do things. "Alec Sharp"*

این اصل به ما کمک می‌کند تا به یاد داشته باشیم که جهت گیری اشیا باید به سمت باندل و یا بسته‌بندی کردن `data` و `function` های که روی آن داده‌ها، عملیات انجام می‌دهند باشد. درواقع به‌جای اینکه از شئ داده‌هایش را تقاضا کنیم و سپس رو آن داده‌ها عملیاتی انجام

---

[۱] Andy Hunt and "Prag" Dave Thomas

[۲] Alec Sharp

دهیم، باید از شئ بخواهیم که خود آن کار را انجام دهد. این باعث می‌شود رفتارها را به سمت خود شئ‌ای که حاوی آن داده‌هاست حرکت دهیم نه به بیرون.

در مورد مثال ما هم این مشکل وجود دارد. در کلاس ***AreaCalculator*** یک شئ از مستطیل یا مربع ساخته‌ایم، سپس طول و یا عرض آن را درخواست می‌کنیم و بعد عملیات محاسباتی را انجام می‌دهیم. درواقع به‌جای اینکه رفتارِ ***مساحت*** را به خود آن شئ نسبت دهیم و از او بخواهیم که خود مساحتش را حساب کند، جهت رفتار را به سمت کلاس AreaCalculator حرکت دادیم و باعث به وجود آمدن اشتباه محاسباتی در مساحت و در نهایت آن خطا در یونیت تست‌هایمان شدیم.

برای حل این معضل، ابتدا یک کلاس ابسترکت از `Shape` تعریف می‌کنیم.

```
public abstract class Shape
{
}
```

سپس به ازای مربع و مستطیل یک کلاس متناظر که از `Shape` ارث‌بری کرده است، ایجاد می‌کنیم.

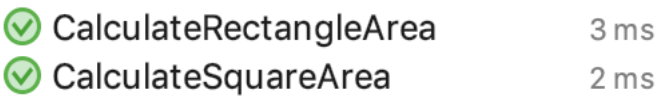
```
public class Square extends Shape
{
    private int side;

    public Square(int side) {
        this.side = side;
    }

    public int Area() {
        return side*side;
    }
}
```

```
public class Rectangle extends Shape
{
    private int width;
    private int height;

    public Rectangle(int width, int height)
    {
        this.width = width;
        this.height = height;
    }

    public int Area()
    {
        return width*height;
    }
}
```

حال، کلاس تست را نیز مطابق با تغییراتی که دادیم، به‌روز می‌کنیم.

```
public class AreaCalculatorTest
{
    @Test
    public void CalculateSquareArea()
    {
        Square square = new Square(4);
        Assert.assertEquals(16, square.Area());
    }

    @Test
    public void CalculateRectangleArea()
    {
        Rectangle rectangle = new Rectangle(4, 5);
        Assert.assertEquals(20, rectangle.Area());
    }
}
```

البته بدیهی است که تست‌ها با موفقیت پاس می‌شوند.




دوباره باید سوال Is Substitutable For را برای `Shape` و کلاس‌هایی که از آن ارث‌بری کرده اند بپرسیم. به یونیت تست زیر توجه کنید.

```
@Test
public void CalculateIsSubstitutableForArea() =
{
    List<Shape> shapes = new ArrayList<Shape>();
    shapes.add(new Square(3));
    shapes.add(new Rectangle(3, 4));

    int[] areas = new int[2];
    for(Shape shape : shapes)
    {
        if(shape instanceof Square){
            areas[0] = (((Square) shape).Area());
        }else if(shape instanceof Rectangle){
            areas[1] = (((Rectangle) shape).Area());
        }
    }

    Assert.assertEquals(areas[0], 9);
    Assert.assertEquals(areas[1], 12);
}
```

یک لیست از Shapeها درست کرده و نمونه‌های Sqaure و Rectangle را به آن اضافه می‌کنیم. سپس آرایه‌ای از intها درست می‌کنیم تا مقادیر محاسبه شده مساحت را در آن قرار دهیم. با یک حلقه‌ی for ابتدا نوع Shape را بررسی کرده و مساحت آن شئ مورد نظر را به آرایه اضافه می‌کنیم. در انتها بررسی می‌کنیم که آیا مساحت خواسته شده با آن چه در لیست قرار دارد یکی است؟ و می‌بینیم که تست‌ها با موفقیت پاس می‌شوند. (اگر این سوال برای شما پیش آمده است که چرا کد را اینقدر پیچیده کردم و رفتار را در همان کلاس ابسترکت قرار ندادم، بدانید که این کار عمدی بوده و در ادمه قصد من را متوجه خواهید شد.)


پس در می‌یابیم که کلاس‌های Shape و Rectangle قابل جایگزینی یا *IS-SUBSTITUTBLE-FOR* هستند برای Shape. حال اگر بخواهیم شئ دیگری مثل مثلث اضافه کنیم چه؟ برای اینکار ابتدا باید شئ مثلث را ایجاد کنیم.

```
public class Triangle extends Shape
{
    private int base;
    private int height;

    public Triangle(int base, int height)
    {
        this.base = base;
        this.height = height;
    }

    public double Area()
    {
        return 0.5*base*height;
    }
}
```

بعد از ساختن شئ Triangle باید تست آن را نیز بنویسیم. مشکل اینجاست اگر قرار باشد Triangle را به تست اضافه کنیم اول از همه باید نوع مقدار بازگشتی متد محاسبه مساحت را که int هست به double تبدیل کنیم. جدا از آن باید در ساختار if-then ها هم دست برده و یک else دیگر برای Triangle اضافه کنیم. به همین ترتیب به ازای هر شئ جدیدی که اضافه می‌شود باید این دور باطل را طی کنیم. اینجاست که OCP را نقض کرده‌ایم! یادتان است؟ کد باید برای گسترش باز و برای تغییر بسته باشد. حال باید دید چگونه می‌توانیم با رعایت کردن LSP از این مشکلات عبور کنیم.

## بررسی و تحلیل مشکل

با مثالی که در بالا زدیم و مشـکلی که با آن رو برو شـدیم، چند چیز را در می‌یابیم که در ادامه به آنها اشاره خواهم کرد.

۱. هرجایی که LSP رعایت نشـود، و تایپ‌هایی وجود داشـته باشـند که قابل جایگزینی نباشـند، آنگاه در اسـتفاده از *Polymorphism* دچار مشـکل خواهیم شد.

۲. دیدیم که اگر کد قابل جایگزین با کلاس والدِ خود نباشد، عملا یا بلا استفاده می‌ماند یا ما را به دردسر خواهد انداخت تا بتوانیم آن را در سیستم مورد بهره برداری قرار دهیم.
۳. حل کردن مشکل جایگزینی با استفاده از *if-then* ها یا *switch* ها شاید در آن لحظه راه‌گشا بنظر برسد؛ اما در ادامه‌ی کار، نگهداری و توسعه سیستم را به یک کابوس بدل خواهد کرد. ضمنان دیدیم که این عمل، ناقض OCP نیز است.

## ناقضان لیسکاو

در مثالی که دیدیم و تعاریفی که قبل از آن بررسی کردیم، دریافتیم که چیزهایی در کدهایمان وجود دارد که اگر آن‌ها را رعایت نکنیم، موجب نقض اصل لیسکاو خواهند شد و به واسطه‌ی آن دیگر نمی‌توانیم اشیا فرزند را با کلاس والد جایگزین کنیم. من به آن‌ها «بوی بد لیسکاو» می‌گویم. (آن ۳ بویی که در فصل اول به آن اشاره کردم را بخاطر بیاورید)

یکی از آن‌ها، پیروی از اصل `tell, don't ask` بود. به مثال زیر توجه کنید.

```
for(Employee employee : Employees)
{
    if(employe instanceof Manager)
    {
        printer.printManager( ((Manager) employee) );
    }else{
        printer.printStaff( ((Staff) employee) );
    }
}
```

همانطور که در مثال می‌بینید روی لیستی از کسانی که استخدام شده اند حلقه *for* زده ایم، و در داخل آن با استفاده از *if-then* بررسی می‌کنیم که اگر نوع مستخدم برابر با `Manager` بود به پرینتر می‌گوییم که بر اساس خصوصیات یک مدیر، و اگر چنین نبود، بر اساس خصوصیات یک کارمند، مشخصات این مستخدم را چاپ کند. حال اگر مستخدمی با نوع جدید اضافه شد چه؟ بهتر است به‌جای اینکه در جایی دیگر نوع مدل را تشخیص دهیم و بخواهیم با استفاده از داده‌ی خودِ مدل یا تایپ، اعمالی را روی آن انجام دهیم، این وظیفه را به خود آن مدل محول کنیم. بهتر است هر تایپ از `Employee` ها، با استفاده از اینترفیس یا کلاس

ابسترکتی که از آن *extend* شده، عمل پرینت یا درواقع آن عملیات روی داده خود را، به صورت کپسوله شده در داخل خود انجام دهید. بدین ترتیب دیگر نیازی به استفاده از *if-then* یا *switch* نخواهیم داشت و مدیریت، نگهداری و گسترش کدها به مراتب آسان تر خواهد شد.

مورد دیگری که باید در نظر داشته باشیم، استفاده بیش از حد و بی مورد از متدها در کلاس‌های ابسترکت است. به مثال زیر توجه کنید.

```
public abstract class Base
{
    public abstract void Method1();
    public abstract void Method2();
}

public class Child extends Base
{
    @Override
    public void Method1()
    {
        throw new NotImplementedException();
    }

    @Override
    public void Method2()
    {
        //TODO some stuff
    }
}
```

همانطور که در مثال می‌بینید کلاس `Base` دارای دو متد بوده و کلاس `Child` از آن *extend* شده است. در ادامه دو متد را به اجبار *override* کرده است. کلاس Child تنها احتیاج به متد `Method2` داشت، اما مجبور بود `Method1` یک را هم اضافه کند و نتیجتا آن را نادیده بگیرد. به همین دلیل در داخل آن یک اکسپشن را اجرا می‌کند که می‌گوید `NotImplementedException` یعنی این متد اینجا هست اما حقیقتاً پیاده سازی نشده و به ناچار از آن استفاده می‌کنم. در نهایت در یونیت تست هم باید برایش یک تست بنویسد و مطمئن شود که حتما اکسپشن `Method1` اجرا شود. هنگامی که متدهای نابجا و بی مورد را در کلاس‌های ابسترکت به‌کار می‌بریم باعث می‌شویم که یا کلاس‌های فرزند قابل

جایگزینی با کلاس والد نباشند، یا اینکه بعد از پیاده سازی مجبور شویم آنها را نادیده گرفته و نهایتا با اکسپشن‌های مختلف مدیریت کنیم. تا می‌توانید از این نوع قضایا دوری کنید. اگر چند نفر روی یک پروژه کار می‌کنید، هشیار باشید که مواردی از این دست برای دیگران بسیار آزار دهنده خواهد بود و ممکن است ندانند که شما در کدهایتان چه کرده‌اید. تا جایی که می‌توانید حتما و حتما در مواقع اینچنینی یونیت تست بنویسید تا حداقل در پاس شدن/نشدن تست ها بتوانید این را موارد را کنترل کنید.

با اینحال راه حل ما استفاده از اکسپشن یا تست نیست. بلکه باید از اینترفیس‌ها کمک بگیریم. اینکه در یک کلاس، متدهای بی مورد و غیر قابل استفاده در بعضی موارد داریم، نشان از این است که اصل SRP را رعایت نکرده و حتما تایپ و کلاس ما بیش از یک وظیفه دارد.

پس وظایف حاشیه‌ای یا چیزهای که فکر می‌کنید ممکن است همه فرزندان از آنها پیروی نکنند را به اینترفیس‌ها منتقل کرده و از قدرت آن‌ها بهره ببرید. در این باره در بخش مربوط به *Interface Segrigation*مفصل صحبت خواهم کرد.

لازم بود تا قبل از شروع بازسازی و ریفکتور کدها، شما را با این مفاهیم و مشکلات آشنا کنم. حال بهتر است به سراغ کدهای خودمان رفته و مشکلات را حل کنیم.

## بازسازی کدها

مشکل `Triangle` را بخاطر بیاورید، و همچنین اصل `tell, don't ask`. بهترین راه حل برای رفع این معضل، این است که متد Area را در داخل کلاس والد یا همان `Shape` منتقل کنیم و به اشیا بگوییم که آن کلاس را در خود *override* کنند.

تغییرات کلاس `Shape` بدین صورت خواهد بود.

```
public abstract class Shape
{
    public abstract double Area();
}
```

سپس متدهای محاسبه‌ی هرکدام از اشیا به صورت زیر تغییر می‌کنند.

```
public class Square extends Shape
{
    private int side;

    public Square(int side)
    {
        this.side = side;
    }

    @Override
    public double Area()
    {
        return side*side;
    }
}
```

```
public class Rectangle extends Shape
{
    private int width, height;

    public Rectangle(int width, int height)
    {
        this.width = width;
        this.height = height;
    }

    @Override
    public double Area()
    {
        return width*height;
    }
}
```

```
public class Triangle extends Shape
{
    private int base, height;

    public Triangle(int base, int height)
    {
        this.base = base;
        this.height = height;
    }

    @Override
    public double Area()
    {
        return 0.5*base*height;
    }
}
```

پس از اینکه متد `Area()` در کلاس‌های فرزند *override* کردیم، آنگاه لازم است تا تست را نیز بروز کنیم. بدین ترتیب دیگر نیازی به *if-then* ها در حلقه‌ی آرایه‌ی مساحت‌ها نداریم و می‌توانیم تنها با فراخوانی متد `Area` بدون نگرانی از نوعِ نمونه‌گیری شده کلاسِ فرزند، آرایه را مقدار دهی کنیم.

```
@Test
public void CalculateIsSubstitutableForArea()
{
    List<Shape> shapes = new ArrayList<Shape>();
    shapes.add(new Square(3));
    shapes.add(new Rectangle(3, 4));
    shapes.add(new Triangle(3, 5));

    ArrayList<Double> areas = new ArrayList<Double>();
    for(Shape shape : shapes)
    {
        areas.add(shape.area());
    }

    Assert.assertEquals(areas.get(0), 9, 0);
    Assert.assertEquals(areas.get(1), 12, 0);
    Assert.assertEquals(areas.get(2), 7.5, 0);
}
```

البته یادمان نرود که برای `Triangle` هم باید یونیت تست اضافه کنیم.

```
public class Triangle extends Shape
{
    private int base;
    private int height;

    public Triangle(int base, int height)
    {
        this.base = base;
        this.height = height;
    }

    @Override
    public double area()
    {
        return 0.5*base*height;
    }
}
```

و در نهایت تمامی تست‌ها با موفقیت پاس می‌شوند.


## چه موقع از LSP استفاده کنیم؟

سوال مهم این که چه موقع باید از LSP استفاده کرد؟ اگر با یکی از آن بوهای بد یا همان ناقضان لیسکاو مواجه شدید، یا جایی که با استفاده از *if-then* ها مجبور شدید متدهای چند ریختی با تایپ‌های مختلف را بررسی کنید، یا کلاس و اینترفیسی را پیاده سازی کردید که یک یا چند متد آن را استفاده نمی‌کنید، دقیقا همان جایی هستید که باید LSP اعمال شود.

اگر به یکی از این مواردی که نام بردم برخوردید و خواستید آن را رفع کنید، می‌توانید با استفاده از یک اینترفیس بهتر که تمامی متدهای آن قابل پیاده سازی است (البته با رعایت اصل SRP)، و یا تبدیل کردن کلاس پایه به کلاس دیگری (گاهی کوچکتر گاهی جامع‌تر) که قابل جایگزینی باشند، اصل جایگزینی لیسکاو را اعمال کنید.

اگر نتوانستید آن نشانه‌هایی که گفتم را پیدا کنید یا در تشخیص آن مشکل داشتید، می‌توانید با رعایت آن روشی که در OCP توضیح دادم به این مهم برسید. هرگاه که نیاز به

تغییر در دومین بار دیده شد، آنگاه کدهای خود را بازنویسی کرده و اصل OCP را اعمال کنید. آنجایی که اصل OCP را پیاده می‌کنید همزمان LSP را نیز مورد توجه قرار داده و تا می‌توانید از کلاس‌های والد/فرزندی استفاده کنید که قابل جایگزینی با یکدیگر باشند.

## نکات

این چند نکته آخر را نیز به یاد داشته باشید.

۱. اصل `tell, don't ask` را فراموش نکنید. از اشیا، مقادیر و حالت‌هایشان را پرسیده و بعد روی آن‌ها اعمال سلیقه کنید. ساده است، همانند انسان. متدهایی که روی مقادیر داخلی یک شئ تاثیر گذار هستند را به داخل آن شئ منتقل کنید. از شئ نپرسید، بلکه به او بگویید چه می‌خواهید و چه کاری باید برای شما انجام دهد. سعی کنید مقادیر و متدهای مرتبط با هم را در داخل کلاس کپسوله کنید.

۲. هرگاه متوجه شدید که کلاس فرزند قابل جایگزینی با هم نیستند، کلاس ها را به کلاس های کوچکتر بشکنید، همانند کاری که در مثال انجام دادیم. نوانستیم کلاس `Square` و `Rectangle` راه باهم جایگزین کنیم، در نتیجه آن ها را بازسازی کرده و یک کلاس `Shape` ایجاد کردیم که در نهایت هردوی آنها از آن *extend* شدند. از این پس دو شئ مربع و مستطیل داشتیم که قابل جایگزینی با کلاس پایه‌ی خود یعنی `Shape` بودند.

## جمع بندی

همانطور که در مثال‌ها و تعاریف دیدیم، استفاده از LSP به ما کمک می‌کند تا بتوانیم متدهای چندریختی بهتری ایجاد کنیم و همچنین بتوانیم کدهایی را توسعه دهیم که نگهداری و گسترش آن‌ها راحت تر است.

در این بخش با مفاهیم `IS-A` و `IS-SUBSTITUTBLE-FOR` آشنا شدیم و فهمیدیم که چگونه متوجه شویم که آیا ارث-بری مناسبی انجام دادیم یا خیر. به‌جای اینکه بگوییم A ایز-اِ B، باید بگوییم آیا A قابل جایگزینی با B است؟

در این بخش با مفاهیمی روبرو شدیم که مرتبط با این موضوع بودند:

- چند ریختی یا Polymorphism
- ارث‌بری یا Inheritance
- اصل تجزیه‌ی (تفکیک) رابط یا Interface segregation principle
- اصل باز-بسته یا Open-Close Principle

برای آگاهی بیشتر و آشنایی با مفاهیمی که در بالا مکرر راجع به آنها صحبت کردیم، مطالعه کتاب Agile Principles, Patterns, and Practices in C# نوشته رابرت مارتین (آنکل باب) و میکا مارتین را توصیه می‌کنم.

# اصل چهارم
# Interface Segregation Principle

*« You want me to plug this in, Where? »*

## مقدمه

در این بخش می‌خواهیم به حرف "I" از اصول ۵ گانه SOLID یعنی اصل تجزیه‌ی (تفکیک) رابط (یا بنا بر برخی منابعی دیگر، اصل جدایی واسط‌ها) که به اختصار ISP نامیده می‌شود، بپردازیم. مانند بخش قبلی، در ابتدا مفهوم ISP را تشریح می‌کنیم، سپس مشکلی که ممکن است پیش بیاید را بررسی کرده و برای آن یک مثال می‌زنیم. آنگاه سعی می‌کنیم تا آن مثال را بازنویسی کرده و اصل ISP را در آن پیاده سازی کنیم.

## تعریف

بنا بر اصل ISP می‌گوییم استفاده از چند رابط که هر کدام، فقط یک وظیفه را بر عهده دارد بهتر از استفاده از یک رابطِ چند منظوره است. اگر بخواهم ساده‌تر بگویم، ISP به ما می‌گوید حق نداریم Client را (منظور کلاسی که از این تایپ یا کلاس والد استفاده می‌کند) مجبور کنیم تا به متدهایی وابستگی داشته باشد که حقیقتاً هیچ استفاده ای از آن‌ها نمی‌کند. (مثال کلاس‌های Base و Child در فصل قبلی به یاد بیاورید) این تعریف از آنکل باب نقل شده است.

در نتیجه ترجیح این است که به‌جای اینترفیس‌های چاق (fat) از رابط‌های منسجمِ کوچکِ تک‌وظیفه‌ای استفاده کنید.

واقعاً هیچ تصویری بهتر از این گویای ISP نیست. همانطور که در تصویر می‌بینید «میخواهی من به کدام سوکت وصل شوم؟». در این تصویر فقط یک وابستگی داریم و آن هم کابل usb است که قرار است در نهایت به سوکت usb لپتاپ متصل شود. اما دستگاهی در این بین قرار دارد با تعداد بسیار زیادی از دکمه، سوییچ و درگاه‌های usb که نمی‌دانیم به کدامیک باید وصل شویم و با خود دستگاه چطور کار کنیم تا در نهایت کابل با موفقیت با لپتاپ ارتباط برقرار کند! این دقیقا همان چیزی است که نیاز داریم تا درباره‌اش صحبت کنیم. در اینجا فقط یک پورت usb کفایت می‌کند، در حالی که وسیله‌ای عجیب قریب با کلی چیزهای مختلفِ متصل به آن داریم که نه نمی‌دانیم چطور باید از آن استفاده کنیم؟ و حقیقتاً هیچ استفاده‌ای از آن‌ها نداریم.

در نتیجه، وقتی میخواهیم درباره اصل جدایی رابط‌ها حرف بزنیم، درواقع میخواهیم بفهمیم و به این تعریف برسیم که خودِ Interface چیست و چه مواقعی باید از آن استفاده کنیم.

در زبان Java یا C# یا هر زبان دیگری، همانطور که در ادامه هم می‌بینید، کلاس اینترفیس تشکیل شده از یک عبارت رزرو شده سیستمی به نام `inteface` که در داخل آن تعدادی `method` و یا `property` عمومی (که همراه با عبارت `public` مشخص می‌شوند) داریم که لازم است تمامی کلاس‌هایی که اینترفیس را پیاده سازی می‌کنند، این خصوصیات و رفتارها را نیز پیاده‌سازی[۱] کرده باشند.

```
public interface MyInterface

{
    public String name = "John Doe";
    public void printName();
}
```

اگر در کلاس از یک اینترفیسِ پابلیک استفاده کرده (کاری نداریم که داخل آن چه مقادیر، ویژگی[۲] یا متدهایی ایجاد شده و چه ساختاری در داخل آن اینترفیس وجود دارد) و آن را پیاده‌سازی کنیم و آن هنگام ببینیم که بخش‌هایی از آن را احتیاج نداریم درحالی که فقط بخش‌های کوچکی از آن قرار است مورد استفاده قرار گیرد، آنگاه لازم است تا به فکر یک

---

[۱] Implement

[۲] Property

طراحی بهتر با استفاده از اصل Interface Segregation یا همان تجزیه‌ی اینترفیس باشیم. ما نیاز داریم تا کلاس اینترفیس خود را به کلاس‌های کوچکتری تقسیم کنیم که تنها شامل آن چیزهای باشند که حقیقتاً تمام بخش‌های آن، مورد استفاده قرار می‌گیرند.

برای مثال، یکی از الگوهای قدیمی که اینکار را انجام می‌دهد، الگوی `Façade` است که به شما کمک می‌کند تا کلاس‌های بزرگ و پیچیده را به اینترفیس‌ها یا کلاس‌های کوچک تقسیم کرده و آن بخش‌هایی را به *client* تحویل دهید که دقیقا به آن‌ها احتیاج دارد.

فکر می‌کنم به اندازه کافی روشن شده باشد که قرار است چه کنیم، پس مستقیما برویم سراغ مثال تا ببینیم در عمل چه باید کرد.

## مثال

می‌خواهم با هم سفری به دنیای انیمیشن‌ها داشته باشیم. در این مثال از انیمیشن اسباب‌بازی‌ها یا همان Toy Story کمک گرفتم. فرض کنید می‌خواهیم داستان اسباب‌بازی‌ها را در قالب یک برنامه‌ی ساده‌ی Java درست کنیم. در این برنامه، در ابتدا تنها ۳ شخصیت را ایجاد خواهیم کرد.

Sherrif Woody

Slinky Dog

Buzz Lightyear

برای اینکه پیاده سازی راحت شود، ابتدا یک interface به نام `Toy` درست می‌کنیم که شامل قیمت و رنگ آن اسباب بازی است. اینترفیس یا رابط *Toy* تعدادی رفتار مثل `move`، `talk` و `fly` را نیز در خود دارد.

```
public interface Toy
{
    void setPrice(int price);
    void setColor(String color);
    void move();
    void fly();
    void speak();
}
```

در اولین قدم، یک مدل برای *بازلا یتیر* می‌سازیم و اینترفیس Toy را در آن، ایمپلمنت یا پیاده‌سازی می‌کنیم.

```
public class BuzzLightyear implements Toy
{
    private int price;
    private String color;

    @Override
    public void setPrice(int price) { this.price = price; }

    @Override
    public void setColor(String color) { this.color = color; }

    @Override
    public void move()
    {
        System.out.println("Toy is walking.");
    }

    @Override
    public void fly()
    {
        System.out.println("Toy is flying.");
    }

    @Override
    public void speak()
    {
        System.out.println("Toy is speaking.");
    }

    @Override
    public String toString() {
        return "The "+color
                  +" Buzz Lightyear toys's price is "+price+"$";
    }
}
```

در متد `toString()` ، تنها به ذکر کردن رنگ، نام و قیمت اسباب‌بازی بسنده می کنیم.

سپس متدِ تست آن را نیز نوشته و بررسی می کنیم که همه تست‌ها بدون مشکل پاس شوند.

```
public class ToyStoryTest
{
    BuzzLightyear buzz = new BuzzLightyear();

    @Test
    public void buildBuzz()
    {
        buzz.setPrice(250);
        buzz.setColor("white");
        buzz.speak();
        buzz.fly();
        buzz.move();
        Assert.assertEquals(
            buzz.toString(),
            "The white Buzz Lightyear toys's price is 250$"
        );
    }
}
```

و تست با موفقیت پاس می‌شود.


در گام بعدی، میخواهیم اسباب بازی *کلانتر وودی* را نیز پیاده سازی کنیم.

```
public class SheriffWoody implements Toy
{
    private int price;
    private String color;

    @Override
    public void setPrice(int price) { this.price = price; }

    @Override
    public void setColor(String color) { this.color = color; }

    @Override
    public void move()
    {
        System.out.println("Toy is walking.");
    }

    @Override
    public void fly() throws NotImplementedException
    {
        throw new NotImplementedException("The toy can't fly!");
    }

    @Override
    public void speak()
    {
        System.out.println("Toy is speaking.");
    }

    @Override
    public String toString()
    {
        return "The "+color
                  +" Sheriff Woody toys's price is "+price+"$";
    }
}
```

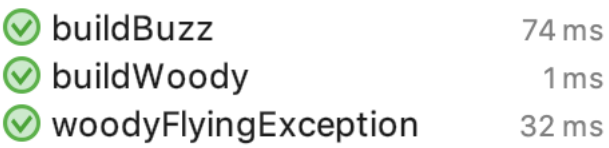
اسباب‌بازی *وودی*، هم راه می‌رود و هم حرف می‌زند؛ اما بر خلاف *باز لایتیر*، نمی‌تواند پرواز کند. متاسفانه مجبوریم متد `fly` را اضافه کنیم، اما با یک اکسپشن اشاره می‌کنیم که این متد پیاده‌سازی نشده و بلا استفاده است، درواقع مدل ما توانایی اجرای این رفتار را ندارد. وقتی این کار را انجام دهیم، مجبوریم به اینترفیس `Toy` بازگشته و به متد `fly` بگوییم ممکن است در جایی استفاده نشوی پس باید آماده اجرای اکسپشن مربوط به پیاده‌سازی نشدن متد یا `NotImplementedException` نیز باشی. (در بعضی از زبان‌ها احتیاج به این کار نیست)

```
void fly() throws NotImplementedException;
```

در حین اینکه به OCP دقت می‌کنیم، اولین تغییر را روی `Toy` انجام می‌دهیم. لازم است برای مدل *وودی* هم تست بنویسیم. جدا از اینکه از خودِ مدل تست می‌گیریم، باید در نظر داشته باشیم برای متد `fly()` که *exception* پیاده‌سازی نشدن را اجرا می‌کند نیز تست بنویسیم و مطمئن باشیم که حتما اکسپشن اجرا می‌شود. در نتیجه کلاس تست اینگونه تغییر می‌کند.

```
public class ToyStoryTest
{
    @Rule
    public ExpectedException thrown = ExpectedException.none();

    BuzzLightyear buzz = new BuzzLightyear();
    SheriffWoody woody = new SheriffWoody();

    @Test
    public void buildBuzz() { ... }

    @Test
    public void buildWoody()
    {
        woody.setPrice(235);
        woody.setColor("brown");
        woody.speak();
        woody.move();
        Assert.assertEquals(
            woody.toString(),
            "The brown Sheriff Woody toys's price is 235$"
        );
    }

    @Test
    public void woodyFlyingException()throws NotImplementedException
    {
        thrown.expect(NotImplementedException.class);
        thrown.expectMessage("The toy can't fly!");
        woody.fly();
    }
}
```

و همانطور که انتظار داریم، نتیجه تست باید سبز باشد.



حالا باید شخصیت سوم یعنی *اسلینکی سگه* را نیز اضافه کنیم. عروسک *اسلینکی* نه‌تنها که پرواز نمی‌کند، بلکه حرف هم نمی‌زند! حتما حدس می‌زنید که چه کارهایی باید انجام دهیم! ابتدا باید به کلاس `Toy` رفته و به متد `speak()` بگوییم که تو هم باید مواظب اجرای خطای پیاده‌سازی نشده باشی.

```
void speak() throws NotImplementedException;
```

سپس کلاس SlinkyDog را ایجاد کرده، و بعد اینترفیس Toy را اضافه کنیم و تمامی متدهای قابل استفاده و غیر قابل استفاده آن را پیاده سازی کنیم.

```
public class SlinkyDog implements Toy
{
    private int price;
    private String color;

    @Override
    public void setPrice(int price) { this.price = price; }

    @Override
    public void setColor(String color) { this.color = color; }

    @Override
    public void move()
    {
        System.out.println("Toy is walking.");
    }

    @Override
    public void fly() throws NotImplementedException
    {
        throw new NotImplementedException("The toy can't fly!");
    }

    @Override
    public void speak() throws NotImplementedException
    {
        throw new NotImplementedException("The toy can't speak!");
    }

    @Override
    public String toString()
    {
        return "The "+color
                  +" Slinky Dog toys's price is "+price+"$";
    }
}
```

و سپس سراغ کلاس تست رفته و تست‌های مربوط به *slinky* را نیز اضافه کنیم. توجه داشته باشید که علاوه بر `fly` باید برای `speak` هم تستِ مربوط به بررسی کردنِ اجرای خطای `NotImplementedException` را نیز اضافه کنیم.

```
public class ToyStoryTest
{
    @Rule
    public ExpectedException thrown = ExpectedException.none();

    BuzzLightyear buzz = new BuzzLightyear();
    SheriffWoody woody = new SheriffWoody();
    SlinkyDog slinky = new SlinkyDog();

    @Test
    public void buildBuzz() {...}

    @Test
    public void buildWoody() {...}

    @Test
    public void buildSlinky()
    {
        slinky.setPrice(175);
        slinky.setColor("yellow");
        slinky.move();
        Assert.assertEquals(
            slinky.toString(),
            "The yellow Slinky Dog toys's price is 175$"
        );
    }

    @Test
    public void woodyFlyingException() throws NotImplementedException{...}

    @Test
    public void slinkyFlyingException() throws NotImplementedException
    {
        thrown.expect(NotImplementedException.class);
        thrown.expectMessage("The toy can't fly!");
        slinky.fly();
    }

    @Test
    public void slinkySpeakingException() throws NotImplementedException
    {
        thrown.expect(NotImplementedException.class);
        thrown.expectMessage("The toy can't speak");
        slinky.speak();
    }
}
```

در نهایت همه تست‌ها با موفقیت پاس می‌شوند.

| | |
|---|---|
| buildBuzz | 31 ms |
| buildSlinky | 107 ms |
| buildWoody | 24 ms |
| slinkyFlyingException | 1 ms |
| slinkySpeakingException | 0 ms |
| woodyFlyingException | 81 ms |

از سبز شدن تمامی تست‌ها خوشحال نباشید. زیرا ما به بدترین شکل ممکن برنامه خود را توسعه دادیم! این برنامه هم ناقض OCP است، هم ناقض ISP! حتی SRP را در نیز رعایت نکردیم!

نگران نباشید. اینجا ISP به داد ما خواهد رسید. اجازه دهید درباره راه حل و اینکه چه باید بکنیم در گام بعد صحبت کنم.

## بررسی و تحلیل مشکل

دوباره به صفحات قبل بازگشته و برنامه‌ای که نوشتم را بررسی نمایید. خواهید دید که یک اینترفیس بزرگ و چاق داریم که تعداد زیادی رفتار مختلف را برای یک عروسک متصور شده است (خدا را شکر کنید که من به همین ۳ رفتار بسنده کردم، وگرنه خاموش/روشن شدن چراغ،

شنا کردن، به دیوار خوردن و برگشتن، مثلا صدای شلیک درآوردن، دریافت دستورات مختلف از کنترل از راه دور و خیلی چیزهای مختلف دیگری که در این اینترفیس قرار داد را هم می‌آوردم). تصورش را بکنید، اگر قرار بود همه کارهایی که اسباب بازی‌ها باید انجام می‌دادند را در این اینترفیس ذکر می‌کردیم، همین حالا هم مشکلات فراوان داریم.

پس اول از همه، اینترفیسی داریم که کارهای مختلفی انجام می‌دهد که با هم تفاوت زیادی دارند. در اینجا SRP یا اصل تک‌وظیفه‌ای را نقض کرده ایم.

جدای از آن، متدهایی داریم که کلی و ضروری نیستند؛ درواقع متدهایی در اینترفیس وجود دارند، که برای همه فرزندان قابل استفاده نیست. در نتیجه ISP نیز نقض شده.

دیدیم که برای اضافه کردن هر اسباب‌بازی جدید، مجبور شدیم هم اینترفیس را تغییر دهیم، هم ساختار تست‌ها را، در نتیجه پیچیدگی سیستم نیز افزایش پیدا کرد. یادمان نرود، سیستم ما باید برای گسترش باز و برای تغییر بسته باشد. آن تکنیکی که گفته بودم را یادتان هست؟ قرار بود در اولین درخواست برای تغییرات، بدون اغماض آن را اعمال کنیم؛ اما اگر برای بار دوم احتیاج به تغییر شد، آنگاه OCP را رعایت کرده و بخش‌های قابل تغییر را جدا کنیم.

مورد چهارمی هم وجود دارد. فرض کنید `Toy` به‌جای اینترفیس یک کلاس انتزاعی یا همان abstract بود. آنگاه می‌توانستیم *وودی* یا *اسلینکی* را با آن جایگزین کنیم؟ خیر، چون برخی از رفتارهای پدر را نقض کرده و نادیده می‌گرفت. پس در اینجا اصل جایگزینی لیسکاو را هم نقض کرده بودیم.

چند مشکل را ذکر کردیم. پیش از رفتن به بخش بعدی، راجع به آنها فکر کنید و ببینید چه راه حلی می‌توانیم ارائه دهیم که همه آن ۴ مشکلی که در بالا ذکر کردم را بدون هیچ کم و کاستی در کنار حل کنیم.

## بازسازی کدها

یادمان است که ISP درباره جدا سازی اینترفیس‌ها و تبدیل کردن آنها به اینترفیس‌های کوچک و تا حد ممکن تک وظیفه‌ای است. (وظایفی که حقیقتا به یکدیگر مرتبط و با هم منسجم هستند). پس ۳ اینترفیس جداگانه با نام‌های `Movable` برای قابلیت حرکت کردن، `Flyable` برای

قابلیت پرواز کردن، `Speakable` برای قابلیت حرف زدن (البته در حد همان *TO INFINITE AND BEYOND!*) ایجاد می کنیم.

```
public interface Movable
{
    void move();
}
```

```
public interface Flyable
{
    void fly();
}
```

```
public interface Speakable
{
    void speak();
}
```

سپس اگر شما هم موافق باشید اینترفیس `Toy` را به یک کلاس ابسترکت تبدیل کرده و تمامی آن ۳ وظیفه‌ی اضافی را از داخل آن حذف می‌کنیم. در نتیجه کلاس ابسترکت جدید `Toy` این‌گونه می‌شود.

```
public abstract class Toy
{
    public abstract void setPrice(int price);
    public abstract void setColor(String color);
}
```

همانطور حتما حدس زدید باید کلاس های اسباب‌بازی‌هایمان را نیز تغییر دهیم. ابتدا باید آنها را از کلاس Toy مشتق (extend) کرده، سپس بسته به نوع اسباب بازی، رفتارهای مناسبش را با استفاده از اینترفیس‌های مربوطه پیاده سازی کنیم.

```
public class BuzzLightyear extends Toy
         implements Movable, Flyable, Speakable
{
    private int price;
    private String color;

    @Override
    public void setPrice(int price) {
        this.price = price;
    }

    @Override
    public void setColor(String color) {
        this.color = color;
    }

    @Override
    public void move()
    {
        System.out.println("Buzz is walking.");
    }

    @Override
    public void fly()
    {
        System.out.println("Buzz is flying.");
    }

    @Override
    public void speak()
    {
        System.out.println("TO INFINITE AND BEYOND! ");
    }

    @Override
    public String toString() {
        return "The "+color
                  +" Buzz Lightyear toys's price is "+price+"$";
    }
}
```

برای *بازلا یتیرِ* که وضـــعیت تغییر چندانی نکرد. تنها از کلاس `Toy` مشـــتق یا *extend* شده، و رفتارهایی که نیاز دا شت را با ا ستفاده از اینترفیس‌های مربوطه‌اش پیاده سازی کردیم.

همین وضعیت برای *وودی* نیز صادق است، منتهی دیگر نیازی به پیاده‌سازی آن متدهایی بالا استفاده نداریم، در نتیجه هیچ احتیاجی هم به مدیریت اکسپشن‌ها نداریم و فقط رفتاری را پیاده‌سازی می‌کنیم که حقیقتا مورد نیاز است.

```
public class SheriffWoody extends Toy implements Movable, Speakable
{
    private int price;
    private String color;

    @Override
    public void setPrice(int price)
    {
        this.price = price;
    }

    @Override
    public void setColor(String color)
    {
        this.color = color;
    }

    @Override
    public void move()
    {
        System.out.println("Woody is walking.");
    }

    @Override
    public void speak()
    {
        System.out.println("Woody is speaking.");
    }

    @Override
    public String toString()
    {
        return "The "+color
                  +" Sheriff Woody toys's price is "+price+"$";
    }
}
```

کلاسی که در این بین از همه بیشتر سود کرده، کلاس مربوط به *اسلینکی* است. تنها یک رفتار دارد، و تنها همان رفتار را پیاده‌سازی می‌کند.

```
public class SlinkyDog extends Toy implements Movable
{
    private int price;
    private String color;

    @Override
    public void setPrice(int price)
    {
        this.price = price;
    }

    @Override
    public void setColor(String color)
    {
        this.color = color;
    }

    @Override
    public void move()
    {
        System.out.println("Slinky is walking.");
    }

    @Override
    public String toString()
    {
        return "The "+color
                  +" Slinky Dog toys's price is "+price+"$";
    }
}
```

حالا که اینترفیس، کلاس ابسترکت و کلاس فرزندان را بازنویسی و بازسازی کردیم، وقت آن رسیده که به سراغ یونیت تست‌ها برویم.

دیگر نیازی به آن‌همه مدیریت اکسپشن‌ها و چیزهای مختلف نیست. فقط و فقط بررسی می‌کنیم که آیا مقادیر درست مقدار دهی[۱] شده و رفتارهای بدون مشکل اجرا می‌شوند یا نه.

---

[۱] Set

```
public class ToyStoryTest
{
    BuzzLightyear buzz = new BuzzLightyear();
    SheriffWoody woody = new SheriffWoody();
    SlinkyDog slinky = new SlinkyDog();

    @Test
    public void buildBuzz()
    {
        buzz.setPrice(250);
        buzz.setColor("white");
        buzz.speak();
        buzz.fly();
        buzz.move();
        Assert.assertEquals(
            buzz.toString(),
            "The white Buzz Lightyear toys's price is 250$"
        );
    }

    @Test
    public void buildWoody()
    {
        woody.setPrice(235);
        woody.setColor("brown");
        woody.speak();
        woody.move();
        Assert.assertEquals(
            woody.toString(),
            "The brown Sheriff Woody toys's price is 235$"
        );
    }

    @Test
    public void buildSlinky()
    {
        slinky.setPrice(175);
        slinky.setColor("yellow");
        slinky.move();
        Assert.assertEquals(
            slinky.toString(),
            "The yellow Slinky Dog toys's price is 175$"
        );
    }
}
```

اگر یادتان باشد، قرار بود اصل جایگزینی لیسکاو را نیز اعمال کنیم. حال برای نمونه، *وودی* را انتخاب کرده و تستِ مربوط به آن را نیز می‌نوسیم.

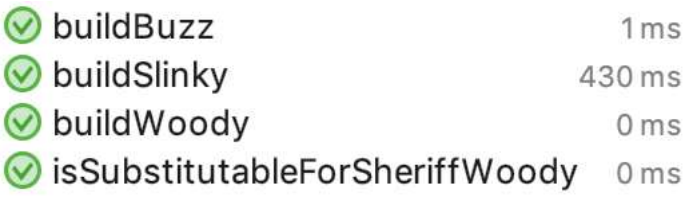
```
@Test
public void isSubstitutableForSheriffWoody()
{
    Toy woody = new SheriffWoody();
    woody.setPrice(217);
    woody.setColor("red");
    ((SheriffWoody) woody).speak();
    ((SheriffWoody) woody).move();
    Assert.assertEquals(
        woody.toString(),
        "The red Sheriff Woody toys's price is 217$"
    );
}
```

حتما می‌پرسید که چرا در هنگام صدا زدن رفتارها یکبار دیگر شئ `woody` را اینیشیال کرده‌ام؟ اینجا هم که قابل جایگزینی نیستند. آیا اصل لیسکاو را نقض کرده‌ایم؟

```
((SheriffWoody) woody).speak();
((SheriffWoody) woody).move();
```

خیر! اگر یادتان باشد گفته‌ام که کلاس فرزند نباید وظایف پدر را نقض کند و نباید هیچ چیزی کمتر از آن داشته باشد. اما اینکه وظیفه‌ای بیشتر از پدر داشته باشد هیچ اشکالی ندارد. از آنجایی که پدر این قابلیت‌ها را نداشت، مجبوریم مجددا به برنامه بگوییم دقیقا رفتارهای فرزند را برای فراخوانی مد نظر داریم و بس.

بدون هیچ نگرانی تست‌ها را اجرا می‌کنیم و از پاس شدن تمامی تست‌ها به وجد می‌آییم!



همانطور که مشاهده کردی، با استفاده از اصل ISP هم به اینترفیس‌هایی رسیدیم که از SRP پیروی می‌کردند، هم مشکل OCP را حل کردیم، هم اصل LSP را پیاده سازی کردیم. چه چیزی بهتر از این؟

حالا که از کرده خود دل‌شادیم، بهتر است نکات و روش‌هایی که در ادامه ذکر خواهم کرد را نیز به خاطر بسپاریم.

## نکات

هرگاه با کلاسی مواجه شدید که در داخل آن یک متد *override* شده وجود داشت که بلا استفاده بود، یا اینکه یک اکسپشن `NotImplementedException` را اجرا می‌کرد، باید آن بخش از کلاس یا اینترفیس را به عنوان یک اینترفیس واحد جدا کنید.

```
@Override
public void fly() throws NotImplementedException
{
    throw new NotImplementedException("The toy can't fly!");
}
```

اصل جایگزینی لیسکاو را بخاطر بیاورید. بدین ترتیب، هر جایی که اصل جایگزینی لیسکاو نقض شد، احیانا در کلاس والد، متدهایی دارید که برای فرزند بلا استفاده است؛ پس در آن بخش نیز باید ISP و در نتیجه LSP اعمال شود.

وقتی کلاس فرزند تنها به بخش خاصی از یک کلاس نیاز دارد، اما در عوض به یک کلاس چاق با تمام خواصش وابسته باشد، آنگاه سیستم پیچیده‌ای دارید که نگهداری و توسعه آن را سخت می‌کند. کلاس‌هایی با این مشخصه عموما ناقض OCP هستند.

هرگاه با کلاسی مواجه شدید که متدهایی داشت و کلاس فرزند فقط به بخشی از آن کلاسِ پایه محتاج بود، آنگاه می‌توانید با استفاده از اینترفیس‌ها، یا کلاس‌های کوچکِ تکِ وظیفه‌ای و یا با استفاده از الگوی `Façade` آن بخش‌های اضافه از کلاس را به بخش‌های مجزای قابل استفاده برای کلاس‌های فرزند تبدیل کنید.

اگر در کدهایتان اینترفیسی دارید که خود صاحب آن هستید و می‌توانید هر تغییری در آن اعمال کنید، آن را به اینترفیس‌های کوچکی تجزیه کنید؛ سپس اینترفیس چاق را نادیده گرفته

(یا حذف کرده) و اینترفیس‌های جدید کوچک را در بخش‌هایی که به آن نیاز دارید، پیاده سازی کنید. اما درصورتی که اینترفیسی دارید که صاحب آن نیستید و اجازه‌ی لازم جهت تغییر یا ویرایش آن را ندارید (مثل خیلی از اینترفیس‌هایی که به صورت پیش‌فرض در بسیاری از فریم‌ورک‌ها یا ماژول‌های سیستمی وجود دارند). ابتدا اینترفیس کوچک مورد نیاز خود را ایجاد کرده، سپس این اینترفیس را با استفاده از یک `adapter` که کل اینترفیس (آن اینترفیس سیستمی) را پیاده سازی می‌کند، پیاده سازی کنید. این کار به شما کمک می‌کند تا از آن اینترفیس کوچک و اختصاصی که از اینترفیس چاق سیستمی مشتق شده استفاده کنید؛ آن *adapter* درواقع رابط بین اینترفیس کوچک با آن اینترفیس چاق خواهد بود که از کنترل شما خارج است.

و چند نکته آخر:

- تا جایی که می‌توانید از ایجاد اینترفیس، کلاس ابسترکت و هرآنچه که انتزاعی است خودداری کنید؛ این امر فقط پیچیدگی سیستم شما را بیشتر می‌کند و به قول معروف «سری که درد نمی‌کند را دستمال نمی‌بندند».
- اینترفیس‌های خود را کوچک، منسجم و متمرکز (روی یک یا چند وظیفه مرتبط) نگه دارید.
- هرگاه خواستید اینترفیسی توسعه دهید، مطمئن باشید دقیقا همان چیزی است که client به آن نیاز دارد.
- هرگاه که شد اینترفیس و client را باهم پکیج کنید. این به شما کمک می کند تا برنامه‌ی شما ماژولار و قابل استفاده مجدد باشد.
- اگر امکانش وجود داشت، اینترفیس و کلاسی که آن را پیاده سازی کرده (و حتی خود client) (که البته همه آنها فقط به یکدیگر وابسطه اند) را در یک کلاس پکیج کنید. این امر از پیچیدگی بی مورد سیستم شما جلوگیری می‌کند و باعث می‌شود همه آنچه لازم داریم در یک فایل پکیج شده باشد. این نوع استفاده در برنامه‌هایی که با C# توسعه داده شده‌اند زیاد دیده می‌شود.

## جمع بندی

چیزهای زیادی در این بخش یاد گرفتیم که آن ها را باهم مرور می‌کنیم:

- هیچگاه client را مجبور نکنید تا از متدهایی استفاده کند که هیچ نیازی به آن‌ها ندارد.
- اینترفیس‌های خود را کوچک و متمرکز نگاه دارید.
- اینترفیس‌های چاق خود را به اینترفیس‌های کوچک و اختصاصی تجزیه کنید.

مفاهیمی که در این بخش به آن‌ها پرداختیم از قرار زیر است:

- چند ریختی یا Polymorphism
- ارث بری یا inheritance
- اصل تک وظیفه‌ای یا Single Responsibility Principle
- اصل باز-بسته یا Open-Close Principle
- اصل جایگزینی لیسکاو یا Liskov Substituble Principle
- الگوی فَصاد یا Façade Design Pattern

در نهایت، فراموش نکنید که تمرین، تمرین و تمرین شما را پادشاه هرچیزی خواهد ساخت.

# اصل پنجم

# Dependency Inversion Principle

*« Whould you solder a lamp directory*

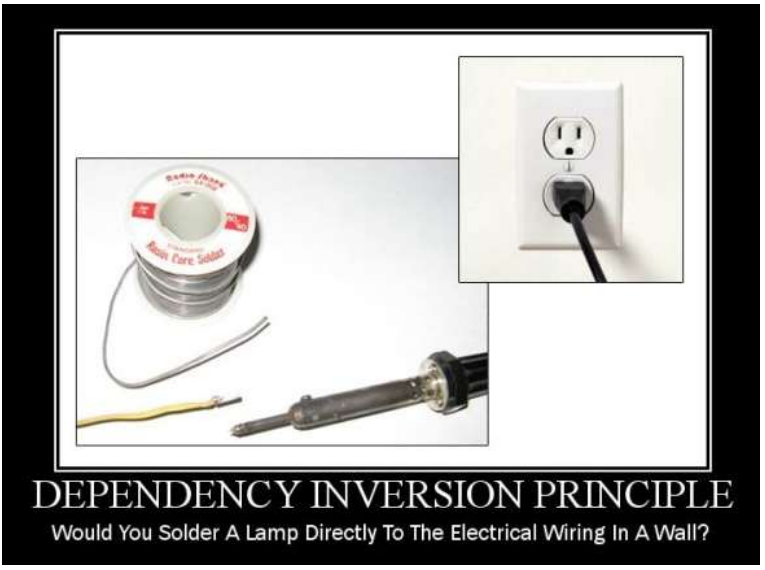
*to the electrical wiring in all wall? »*

## مقدمه

در این بخش می‌خواهیم به حرف "D" از اصول ۵ گانه‌ی SOLID، یعنی اصل وارونگی وابستگی که به اختصار DIP نامیده می‌شود بپردازیم. مانند بخش قبلی، در ابتدا مفهوم DIP را تشریح کرده، سپس به برخی روش‌های قدیمی در برنامه‌نویسی اشاره می‌کنیم و در گام بعدی به تشریح وابستگی کلاس‌های خواهیم پرداخت. در ادامه برای آن یک مثال می‌زنیم و سپس سعی می‌کنیم تا آن مثال را بازنویسی کرده و اصل DIP را پیاده سازی کنیم.

## تعریف

آنکل باب در کتابِ خود[۱] در توضیحِ اصل وارونگی وابستگی می‌آورد که ماژول‌های سطح بالا (**High-Level**) نباید به ماژول‌های سطح پایین (**Low-Level**) وابسته باشند؛ بلکه هردوی آنها باید به **انتزاعات (کلاس ابسترکشن یا اینترفیس)** وابستگی داشته باشند. درواقع، ابسترکشن نباید به جزئیات وابستگی داشته باشد، بلکه جزئیات (سطوح پایین‌تر برنامه) باید به ابسترکشن وابسته باشند.




[۱] Agile Principles, Patterns and PRactices in C#

تصویر صفحه‌ی قبل می‌گوید که «آیا شما لامپ را بطور مستقیم به سیم برق لحیم می‌کنید؟». همانطور که در تصویر می‌بینید، یک پریزِ برق روی دیوار وجود دارد، و ما به‌جای اینکه لامپ را بطور مستقیم به سیم برق لحیم کنیم، تنها با استفاده از دو شاخ برق، آن را به پریز (که یک رابط یا اینتفرفیس، ما بین لامپ و سیم برق است) وصل می‌کنیم. و البته می‌بینیم که با استفاده از اینترفیس، نه یک لامپ، بلکه می‌توانیم چندین لامپ را به پریز و در نتیجه به برق وصل کنیم. این دقیقا همان امکانی است که *Dependency Inversion* به ما می‌دهد.

وقتی کلاس‌هایمان را طوری می‌نویسیم که وابستگی آن‌ها به صورت اینترفیس (رابط) در دسترس قرار می‌گیرید (*expose* می‌شود)، آنگاه می‌توانیم این اینترفیس‌ها را در هرجایی که دلمان بخواهد پیاده سازی کرده و استفاده کنیم؛ درواقع می‌توانیم هر ماژول (دستگاه یا وسیله برقی) خاصی را که می‌خواهیم به آن (اینترفیس یا همان پریز برق) وصل کنیم تا اتصال/ارتباط بدون وابستگی برقرار گردد.

## چه چیزهایی وابستگی یا Dependepcy هستند

تا به حال به این فکر کرده‌اید که سیستم شما چه وابستگی‌هایی دارد؟ در ادامه لیستی از چیزهای که ممکن است در برنامه شما وجود داشته باشند و ما اعتقاد داریم که آن‌ها وابستگی‌های سیستم شما هستند را آورده ام.

- **چارچوب نرم‌افزاری**[1]: در دوران ما، بعید می‌دانم کسی از فریم‌ورک‌ها استفاده نکند. وقتی می‌خواهید برنامه‌ای را تحت جاوا توسعه دهید احتمالا از *Spring* استفاده می‌کنید، اگه با *C#* برنامه‌نویسی می‌کنید به احتمال زیاد درگیر فریم ورک *Net.* هستید، اگر جاوا اسکریپت استفاده می‌کنید حتما با یکی از فریم‌ورک‌های *Vue.js*، *React.js*، *Angular.js* و غیره آشنا هستید (از نام بردن باقی فریم‌ورک‌ها صرف نظر می‌کنم، وگرنه باید تمام کتاب را به مثال زدن درباره‌ی آن‌ها اختصاص دهیم). فریم ورک یکی از بزرگترین وابستگی‌هایی است که شما می‌توانید داشته باشید، چون شما را مجبور می‌کند بر اساس قوانین، چهارچوب، API و متدهایی که خودِ آن وابستگی برای شما

[1] Framework

تعیین می‌کند استفاده کنید. با این‌حال فریم‌ورک آنقدر بزرگ، گسترده و کاربردی است که احیانا تا آخر عمر برنامه هرگز آن را جایگزین چیز دیگری نخواهید کرد.

- **کتابخانه‌های شخص ثالث یا Third Party Library:** کتابخانه‌های نرم افزاری اغلب از آن بخش‌هایی است که با تغییر و پیشرفت تکنولوژی جایشان را به کتابخانه‌های جدیدتر و بهتر خواهند داد. بنابر این یکی از گزینه‌های وابستگی‌های سیستمی، استفاده از کتاب‌خانه‌های ثالث است؛ بدین ترتیب همیشه در نظر داشته باشید که جهت وابستگی‌های شما به چه صورت است و تا آن‌جا که می‌توانید ارتباط خود با کتابخانه‌های شخص ثالث را به واسطه‌ی استفاده از انتزاعات (کلاس ابسترکشن و اینترفیس) حفظ کنید تا وابستگی سیستم شما به حداقل برسد. مگر اینکه مطمئن باشید از کتابخانه‌هایی استفاده می‌کنید که در تمام عمر آن برنامه بدون تغییر مورد استفاده قرار خواهد گرفت.


- **پایگاه داده یا Database:**

پایگاه داده یکی از آن بخش‌هایی است که معمولا به عنوان وابستگی دیده نمی‌شود. عموم توسعه دهندگان تصور می‌کنند که پایگاه داده صرفا یکی از بخش‌های سیستم است که تا ابد با آن پکیج شده و قرار است کار کند. در حالی که تجربه نشان داده یکی از بخش‌هایی که بیشترین تغییرات و جایگزینی‌ها را دارد همین پایگاه داده است. سعی کنید تا جایی که می‌توانید وابستگی خود را به نوع و اسکیماهای[۱] دیتابیس کاهش دهید.

[۱] Schema

- **اتصالات خارجی:** اتصال خارجی شامل همه‌ی آن چیزهایی می‌شود که سیستم شما را به سرویس‌هایی در خارج از کدهایتان وابسته می‌کند. برای مثال *File system*، سرویس ایمیل (ارسال ایمیل یا مثلا خواندن صندوق ایمیل روی پروتکل POP3) و همچنین وب‌سرویس‌ها یا هر اتصال دیگری روی شبکه. تغییر در هرکدام از آن‌ها می‌تواند روی کل سیستم شما تاثیر گذار باشد.
- **منابع سیستمی یا System Resource:** درگیری شما با منابع سیستمی می‌توانند وابستگی شما به محیط اجرا را به حداکثر برساند. برای مثال Clock را نظر بگیرید که از آن طریق به `DateTime` وابستگی پیدا می‌کنید و مجبور خواهید بود برای بدست آوردن ساعت دقیق با استفاده موقعیت مکانیِ محیطِ اجرا، برخی بخش‌های سیستم را تغییر داده یا آن‌ها را مدیریت کنید که می‌تواند روی رفتارِ سیستم شما تاثیرگذار باشد.
- **تنظیمات یا Configurations:** یکی از وابستگی‌های برنامه است که کمتر به چشم می‌آید. فایل تنظیمات، خود یک وابستگی است و سیتم شما را به آن فایل که حاوی تنظیمات ضروری جهت راه اندازی برنامه شماست وابسته می‌کند. تصورش را بکنید، برنامه را اجرا کرده‌اید درحالی که فایل تنظیماتی وجود ندارد، یا مقادیر آن اشتباه است. از آن‌جا که فایل تنظیمات با خطای انسانی رابطه مستقیم دارد در نتیجه سیستمِ منطقی شما برای همیشه با یک خطای انسانی احتمالی درگیر خواهد بود.

- **کلمه‌ی کلیدی جدید یا The New Keyboard:** اگر در برنامه خود جهت مدیریت حالات یا رفتارهای مختلف (مثلا در switch) استفاده می‌کنید، آنگاه سیستم را به آن مجموعه کلمات کلیدی وابسته می‌کنید. در آن صورت هرگاه که بخواهید حالت یا رفتار جدیدی را کنترل کنید مجبورید کلمه کلیدی جدیدی ایجاد کرده و آن بخش‌هایی که به آن وابسته هستند را تغییر دهید. این مورد عموما درمورد کلمات کلیدی اتفاق می‌افتد که از نوع `String` ساخته شده اند. یا کلاسی دارید که دیگر بخش‌ها (یا حتی پروژه‌های خارجی) از آن کلاس استفاده می‌کنند، به ناگاه تصمیم می‌گیرید نام کلاس خود را تغییر دهید. نتیجه‌ی آن این است که دیگر بخش‌هایی که از آن کلاس

استفاده می‌کنند ، یا با خطای نا موجود بودن کلاس مواجه می‌شوند یا ناگزیرند نام کلاس را به نام جدید تغییر دهند! به همین دلیل است که می‌گوییم حتی نام کلاس و کلمات کلیدی نیز می‌توانند وابستگی برنامه شما باشند.

- **متدهای استاتیک یا Static Methods:** هرگاه که یک متد استاتیک را فراخوانی می‌کنید، درواقع آن بخش از کد خود را به آن متد وابسته کرده‌اید؛ آنگاه مدیریت و تست سیستم شما سخت می‌شود و به همین راحتی نمی‌توانید بخش‌های سیستم را از هم جدا کرده و آن‌ها را در مواقع ضروری ایزوله کنید. اگر متدهای استاتیک را در جای-جای سیستمِ خود استفاده کرده باشید، آنگاه در ویرایش و تغییر آن در تمام بخش‌های سیستم دچار مشکل بزرگی خواهید شد.
- **متد Thread.Sleep:** این متد هم یکی از وابستگی‌های برنامه است. این متد فرآیند تست کردن برنامه را مشکل و در مواقعی هم غیر قابل ممکن می‌کند.
- **متد Random:** شاید جالب باشد بدانید که متد `Random` نیز می‌تواند یکی از وابستگی‌های برنامه باشد. متد Random به شما نتایج تصادفی می‌دهد که این امر نیز تست کردن را سخت می‌کند. برای مثال در اینترفیس خود متدی دارید که مقداری تصادفی را ایجاد می کند، آنگاه چطور می‌توانید آن متد را در تست خود پیاده سازی کرده و از او انتظار داشته باشید *(expected)* فلان مقدار را تولید کند، درحال که مقداری که برمیگرداند یک مقدار تصادفی است؟ به همین علت هیچگاه نمی‌توانید از مقدار بازگشتی مطمئن باشید و سیستم خود را به احتمالات و تصادفات وابسته کرده اید.

مطمئنا این لیست کامل نبوده و احتمال دارد وابستگی‌های بیشتری نیز وجود داشته باشد، اما این‌ها نمونه‌هایی از وابستگی‌های برنامه‌های نرم افزاری هستند که ممکن است تا به امروز خیلی‌هایشان را به این چشم ندیده باشیم که همیشه باید نسبت به آن‌ها آگاه بود.

## برنامه‌نویسی سنتی

در برنامه‌نویسی سنتی، بطور معمول شاهد این هستیم که ماژول‌هایی با مرتبه‌ی بالاتر (High-Level)، ماژول‌هایی با مرتبه‌ی پایین‌تر (Low-Level) را بدون هیچ واسطی فراخوانی کرده‌اند. وقتی کلاسی را فراخوانی می‌کنید تا از متدهایش استفاده‌کنید، باید یک نمونه از آن بسازید تا بتوانید کلاس‌های داخل آن را فراخوانی کنید؛ وقتی از کلاس نمونه‌ای ساخته و در برنامه‌ی خود استفاده می‌کنید، درواقع آن بخش از برنامه را به کلاس دیگر وابسته کرده‌اید. پس شما باید یک اینترفیس داشته و به واسطه‌ی آن اینترفیس به بخشِ منطقیِ برنامه (Business Logic) دسترسی داشته باشید. برای مثال یک کلاس مشتری یا سفارش که توسط اینترفیس جدا شده و سبد خرید از طریق اینترفیس با آنها در ارتباط است.

در فرآیندِ فراخوانی متد از یک کلاس دیگر که از آن نمونه[۱] گرفته شـــده اســـت، ممکن است با یک دیتابیس سروکار داشته باشید، یا یک سرویسی که با ساختار اصلی برنامه شما تفاوت زیادی داشـــته باشـــد. آن وقت آن بخش از منطقِ برنامه‌ی شـــما یا همان بیزنس لاجیک برنامه، هنگامی که کاری انجام می‌دهد نیاز دارد تا نتیجه فعالیت یا موردی را روی دیتابیس ذخیره کند (به کامپوننت *Data Access* دسترسی داشته باشد) یا مثلا توسط ماژول *Logger* چیزی را در کنســـول چاپ کند؛ در نهایت رابط کاربری به بیزنس لاجیک، ســـاختارها یا ابزارهای خارجیِ (مثلا دیتابیس) برنامه وابسطه بوده.

علاوه بر این، معمول است در هنگام پیاده کردن الگوی `Façade` از متدهای استاتیک استفاده شود تا بتوانند از بخش‌های منطقی برنامه به شکل یک API آسان و ساده استفاده کنند. این کار باعث می‌شود به‌جای اینکه ۳۰-۴۰ خط کد بنویسیم تا بتوانیم چیزی (مثلا اطلاعات یک کاربر) را روی دیتابیس ذخیره کنیم، تنها با فراخوانی آن متدِ استاتیک به این مهم برسیم. در نگاه اول شاید کار خود را ساده‌تر کرده باشیم، اما در حقیقت وابستگی بی مورد ایجاد کرده ایم.

در آخر، اگر از کلاسی نمونه ساخته و متدهای منطقی آن را در جای-جای برنامه استفاده کردید، ناقض SRP می‌شـــوید. هنگامی که کلاســـی از یک کلاس دیگر نمونه‌ای می‌ســـازد و

---

[۱] Instance

برخی متدهای اسـتاتیک آن یا متدهایی که با کلیدهای خاصـی مدیریت می‌شـوند یا فراخوانی می‌کند، یعنی دارد علاوه بر وظیفه‌ی واقعی خود، وظایف کلاس دیگری را نیز اجرا می‌کند. لازم است که این ابیات از حکایت زاغ و کبک، سروده‌ی جامی را به خاطر داشته باشیم:

> *عاقبت از خامی خود سوخته / رهروی کبک نیاموخته*
> *کرد فرامش ره و رفتار خویش / ماند غرامت‌زده از کار خویش.*

به همین دلیل هوشیار باشید تا کلاس به‌جای اینکه وظایف خود را به درستی انجام دهد، درگیر انجام وظایف بخش دیگری از سیستمی نشود که خود آن نیز دچار مشکل است، که در نهایت این امر سبب نشود که نه تنها خودِ کلاس بلکه کل سیستم از کار بیافتد.

## وابستگی کلاس‌ها

وقتی درباره وابستگی کلاس‌ها حرف می‌زنیم، باید درباره آن صادق باشیم. هرگاه می‌خواهیم کلاسی را به کلاس، ماژول یا چیز دیگری وابسته کنیم، باید مطمئن شویم که کلاس حقیقتاً به آن احتیاج دارد.

کلاس‌هایی که در متد سازنده‌ی[۱] خود، وابستگی‌شان را واضح بیان می‌کنند را وابستگی‌های آشکار می‌نامیم؛ و در مقابل کلاس‌هایی که اینکار را انجام نمی‌دهند و صرفا در لابلای کدها یا متدها اقدام به نمونه‌سازی[۲] کلاس دیگری می‌کنند را، وابستگی پنهان[۳] می‌گوییم.

---

[۱] Constructor

[۲] Instantiate

[۳] Hidden Dependency *(antonym: Explicit Dependeny)*

این را بخاطر داشته باشید که کلاس‌های دسته دوم درباره‌ی وابستگی‌هایشان به شما دروغ می‌گویند. آن کلاس‌ها می‌گویند نگران چیزی نباش، فقط مقادیر مورد نظر خود را به این متد (متد فراخوانی شده از آن کلاسِ نمونه‌سازی شده) پاس کن؛ و بعد میبینی که هیچ چیز کار نمی‌کند! آنگاه بررسی می‌کنی و مشخص می‌شود که آن متد برای اجرا شدن احتیاج به دیتابیس داشته درحالی که دیتابیس وجود ندارد یا در دسترس نیست. آن‌ها به شما می‌گویند که این کلاس کار خواهد کرد، اما به شما اجازه نمی‌دهند که حقیقت را بدانید، به یکباره متوجه می‌شوید که اُه، نه، من واقعا به یک دیتابیس هم احتیاج دارم!

به مثالی که در ادامه می‌آید توجه کنید. کلاسی داریم به نام `HelloWorldHidden` که قرار است با فراخوانی متد `Hello` با پاس کردن نام فرد، در زمان مناسب، پیام مناسب را نمایش دهد.

```
public class HelloWorldHidden
{
    public void Hello(String name)
    {
        if(new Date().getTime().Hour<12)
        {
            return "Good Morning" + name;
        }
        if(new Date().getTime().Hour < 18)
        {
            return "Good Afternoon" + name;
        }
        return "Good Evening" + name;
    }
}
```

اگر این کلاس را از بیرون فراخوانی کنیم، متوجه نمی‌شویم که چه وابستگی‌هایی دارد، همچنین مطلع نخواهیم شد که به واسطه‌ی `Date` به `System clock` وابستگی دارد.

درواقع به ما درباره‌ی وابستگی‌هایش دروغ گفته است. با اینکه در صبح، بعد از ظهر و عصر به خوبی کار کرده و پیام مناسب را برمی‌گرداند، باز هم هیچ اطلاعی درباره وابستگی‌های آن نداریم. کلاس خوب باید اینگونه باشد:

```
public class HelloWorldExplicit
{
    private DateTime greetingDateTime;

    public HelloWorldExpecilt(Date greetingDateTime)
    {
        this.greetingDateTime = greetingDateTime;
    }

    public void Hello(String name)
    {
        if(greetingDateTime.Hour < 12)
        {
            return "Good Morning" + name;
        }
        if(greetingDateTime.Hour < 18)
        {
            return "Good Afternoon" + name;
        }
        return "Good Evening" + name;
    }
}
```

کلاس خوب و راست‌گو، وابستگی‌هایش را در متد *constructor* اعلام می‌کند. حال وقتی بخواهیم از این کلاس در جایی استفاده کنیم، در هنگام ساختن نمونه، متوجه می‌شویم که این کلاس به `Date` وابستگی دارد و باید این نیازش را برطرف کنیم. در نتیجه خودمان وابستگی آن را کنترل می‌کنیم.

جدا از آن، در تست‌ها نیز می‌دانیم که این کلاس یک وابستگی دارد، آنگاه آن نمونه را (که درواقع همین `Date` است) می‌توانیم خودمان ساخته و به عنوان ورودی به آن پاس کنیم (این وابستگی می‌تواند دیتابیس یا هرچیز دیگری هم باشد).

این کلاس دیگر درغگو نیست. البته زیاد خوشحال نباشید، هنوز هم بازه‌های زمانی را هاردکد کرده ایم. اگر برای شب هم بخواهیم پیام بفرستیم یا بازه‌های زمانی دیگری را مد نظر

قرار دهیم یا حتی پیام ها را تغییر دهیم، مجبوریم خودِ آن کد را ویرایش کنیم؛ این کلاس OCP را نقض کرده است. فعلا از این موضوع می‌گذریم.

تا اینجا راجع به مسائل مختلفی از وابستگی صحبت کردیم. بهتر از سراغ مثال رفته و ببینیم این بار قرار است با چه چیز روبرو شویم.

## مثال

مثالی که در بخش SRP زدیم را به یاد دارید؟ همان مرکز فروش که اصل تک وظیفه‌ای را نقض کرده و باهم آن‌ها را رفع کردیم. اکنون می‌خواهیم دوباره همان مورد را، اما این‌بار در استفاده از DIP بررسی و بازسازی کنیم.

در این سرویس مدل‌های `Cart`، `OrderItem` و `PaymentDetails` را داریم:

```
public class Cart
{
    private float totalAmount;
    private Iterable<OrderItem> items;
    private String customerEmail;

    public Cart()
    {
        this.items = new ArrayList<OrderItem>();
    }

    //implement getters and setters
}
```

```
public class OrderItem
{
    private String sku;
    private int quantity;

   //implement getters and setters
}
```

```
public class PaymentDetails
{
    public enum PaymentMethod
    {
        Cash,
        CreditCard
    }
   //implement getters and setters
}
```

یک مدل دیگر هم به نام *Order* داریم. سیستمِ ثبت سفارشِ این فروشگاه با ساختن یک نمونه از این کلاس، فرآیند ثبت و فروش سفارش را انجام می‌دهد.

```
public class Order
{
    public void checkout(
        Cart cart,
        PaymentDetails paymentDetails,
        boolean notifyCustomer
    ) throws Exception
    {
        if (paymentDetails.getPaymentMethod() == CreditCard)
        {
            processPayment(paymentDetails, cart);
        }

        reserveInventory(cart);

        if(notifyCustomer)
        {
            notifyCustomer(cart);
        }
    }

    private void notifyCustomer(Cart cart){...}

    private void reserveInventory(Cart cart){...}

    private void processPayment(PaymentDetails paymentDetails,
                                                Cart cart){...}
}
```

اگر به متد `notifyCustomer`، که وظیفه‌ی ارسال پیام به مشتری را دارد، نگاهی بیاندازیم خواهیم دید که این متد در دلِ خود به `Session`، `MimeMessage`، `Properties` و حتی `Date` وابستگی دارد و هیچکدام از این موارد در متد سازنده (Construct) کلاس `Order` ذکر نشده است.

```
private void notifyCustomer(Cart cart) throws Exception
{
    String customerEmail = cart.getCustomerEmail();

    if (!customerEmail.isEmpty())
    {
        Properties properties = System.getProperties();
        properties.setProperty("mail.smtp.host", "localhost");
        Session session = Session.getDefaultInstance(properties);
        MimeMessage message = new MimeMessage(session);

        try
        {
            message.setFrom(new
                   InternetAddress("mail@example.com"));

            message.addRecipient(
                    Message.RecipientType.TO,
                    new InternetAddress(customerEmail)
            );

            message.setSubject("Your order placed on "
                                    + new Date().toString());

            message.setText("Your order details: \n "
                                    + cart.toString());

            Transport.send(message);
            System.out.println("Message sent successfully.");
        }
        catch (Exception ex)
        {
            throw new Exception("Problem sending
                                    notification email"+ex);
        }
    }
}
```

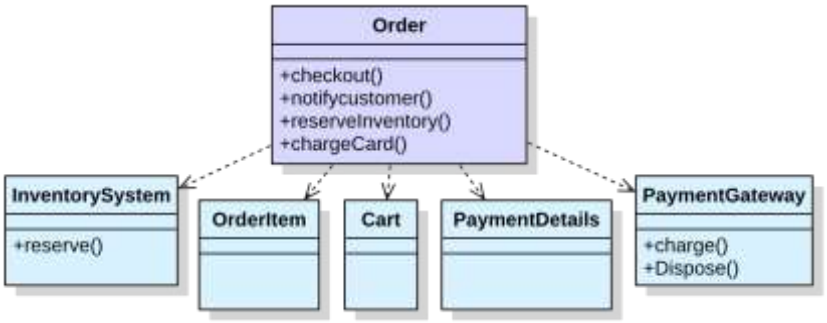
این موضـــوع در متـد `reserveInventory()` هم دیـده می‌شـــود. در اینجـا هم به `InventorySystem` وابستگی داریم.

```
private void reserveInventory(Cart cart) throws Exception
{
    for(OrderItem item : cart.getItems())
    {
        try
        {
            InventorySystem inventorySystem = new InventorySystem();

            inventorySystem.reserve(
                            item.getSku(),
                            item.getQuantity()
            );
        }
        catch (InsufficientInventoryException ex)
        {
            throw new orderException(
                   "Insufficient inventory for item "
                   + item.getSku(),
                   ex
            );
        }
        catch (Exception ex)
        {
            throw new orderException(
                   "Problem reserving inventory",
                   ex
            );
        }
    }
}
```

و همانطور که حدس می‌زنید، در متد `processPayment()` هم به کلاس `PaymentGateway` وابستگی داریم.

```
private void processPayment(PaymentDetails paymentDetails, Cart
cart) throws Exception
{
    PaymentGateway paymentGateway = new PaymentGateway();

    try
    {
        paymentGateway.credentials = "account credentials";

        paymentGateway.cardNumber =
                            paymentDetails.getCreditCardNumber();
       paymentGateway.expiresMonth =
                            paymentDetails.getExpiresMonth();
       paymentGateway.expiresYear =
                            paymentDetails.getExpiresYear();
       paymentGateway.nameOnCard =
                            paymentDetails.getCardholderName();

        paymentGateway.amountToCharge = cart.getTotalAmount();

        paymentGateway.Charge();
    }
    catch (AvsMismatchException ex)
    {
        throw new orderException(
                "The card gateway rejected the card
                  based on the address provided.", ex);
    }
    catch (Exception ex)
    {
        throw new orderException("There was a problem
                                  with your card.", ex);
    }
}
```

باهم نگاهی به دیاگرام UML این چند کلاس بیاندازیم.




برای تست گرفتنِ کلاس `Order` حقیقتاً با مشکل روبرو هستیم. درواقع بخش سخت ماجرا، ***تزریق*** یا اصطلاحا ***Inject*** کردن وابستگی‌ها به داخل کلاسِ *Order* است.

بهتر است نگاهی به کلاس تست بیاندازیم. در متد `withNoItemsNoNotificationNoCreditCard` یک `Order` ساخته، و میخواهیم ببینیم بدون اینکه CreditCard، ار سال پیام به م شتری و همچنین مح صولی در سبد خرید داشته باشیم، آیا وظیفه‌ی Order به درستی انجام می‌شود یا خیر؟

```
public class OrderTest
{
    @Test
    public void withNoItemsNoNotificationNoCreditCard()
                                                    throws Exception
    {
        Order order = new Order();

        Cart cart = new Cart();

        boolean shouldNotifyCustomer = false;

        PaymentDetails paymentDetails = new PaymentDetails();

        paymentDetails.setPaymentMethod(
            PaymentDetails.PaymentMethod.Cash
        );

        order.checkout(cart, paymentDetails, shouldNotifyCustomer);
    }
}
```

که بدون هیچ مشکلی، تست پاس می‌شود.

withNoItemsNoNotificationNoCreditCard 28 ms

حال متد تســت دیگری اضــافه می‌کنیم ولی این‌بار ایمیل مشــتری را وارد کرده و انتظار داریم برای مشتری یک پیام مبنی بر انجام شدن سفارش ارسال شود.

```
@Test
public void NotFailWithNoItemsNotificationNoCreditCard()
                                                throws Exception
{
   Order order = new Order();

   Cart cart = new Cart();
   cart.setCustomerEmail("someone@nowhere.com");
   boolean shouldNotifyCustomer = true;

   PaymentDetails paymentDetails = new PaymentDetails();

   paymentDetails.setPaymentMethod(
       paymentDetails.PaymentMethod.Cash
   );

   order.checkout(cart, paymentDetails, shouldNotifyCustomer);
}
```

تســت ما *fail* می‌شــود. خطایی که به ما داده شــده از این قرار اســت که روی *localhost* پورت ۲۵ اصلا سرور SMTP اجرا نشده است یا ماژول ارتباطی با آن ندارد و نمی‌تواند ایمیل را ارســال کند. این شــاهد همان مثالی بود که در بخش قبلی زدم. فرض کنید شــخصــی بدون اینکه از داخل این کلاس اطلاعی داشــته باشــد، از آن نمونه گرفته و ســپس با همچین خطایی روبرو می‌شود!

NotFailWithNoItemsNotificationNoCreditCard 760 ms

```
java.lang.Exception: Problem sending notification email javax.mail
.MessagingException: Could not connect to SMTP host: localhost, port: 25;
nested exception is:
  java.net.ConnectException: Connection refused (Connection refused)
```

چه راه حل‌هایی داریم؟ در حال حاضر شاید ۲ راه به ذهنمان برسد:

- مثلا بیاییم برای ارســال ایمیل یک کلاس *fake* از SMPT ایجاد کرده و این ســرویس ارسال ایمیل را به نوعی mock کرده تا بتوانیم تست را مدیریت کنیم.

- یا اینکه بیاییم یک SMTP سـرور روی دسـتگاه خود اجرا کنیم (با آن همه دردسـرها برای تنظیمات و اجرای SMPT روی دیوایس لوکال) و بعد تست را اجرا کنیم تا ایمیل ارسال شود.

بنظر شـما این عاقلانه اسـت که تنها برای یک بخشِ منطقی یا Logical در سـیسـتم، بیایم اینهمه دردسـر به خود داده و زحمت mock کردن یا اجرای یک سـرور SMTP را متحمل شویم؟ حقیقتا اینهمه پیچیدگی و سختی برای چه!؟

وقتی متد `notifyCustomer` را صـدا می‌زنیم هیچ احتیاجی به جزئیات و داخل آن نداریم. اصـلا برایمان اهمیت ندارد چطور این کار را انجام می‌دهد، حتی برایمان اهمیتی ندارد که آیا پیام را ایمیل می‌کند، SMS می‌فرستد یا پوش نوتیفیکیشن ارسال می‌کند؟ تنها چیزی که اهمیت دارد این است که وقتی این متد را صدا می‌زنیم، بدون هیچ مشکلی کلاس `Order` به سر انجام برسد.

اگر در این سیستم دیتابیس داشتیم چه؟ آن موقع هم برای تست کردن کلاس `Order`، برویم پایگاه داده را روی دیوایس شـخصـی خودمان اجرا کنیم؟ اگر پرداخت با کارت بانکی داشـتیم چه؟ سـیسـتم رزرو در انبار را چه کنیم؟ در حالی که در این کلاسِ کاملا منطقی، هیچ نیازی نه برای ار سال ایمیل، نه برای سروکله زدن با دیتا، و نه با هیچکدام از این بخش‌ها وجود ندارد!

وقتی قرار باشـد برای هر کدام از این بخش‌ها در کلاس `Order` نگران بوده و کاری بکنیم، حقیقتاً مسیر توسعه سخت و دردناک خواهد شد.

اجازه بدهید مشکلات را در بخش بعدی تحلیل و بررسی کنیم.

## تحلیل و بررسی

### مشکلات

مشکلاتی که در `Order` با آن روبرو هستیم، وابستگی‌های پنهان آن است. لیستی از این وابستگی‌ها را در ادامه آورده ام.

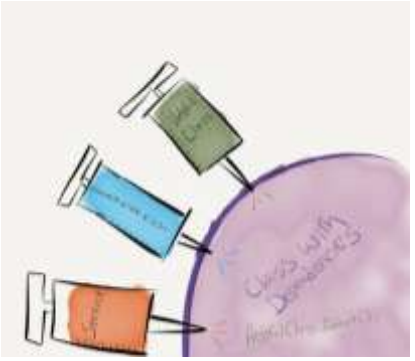
- Properties
- Session
- MimeMessage
- Date
- InventorySystem
- PaymentGateway

که نتایج زیر را در بر دارد:

- وابستگی بیش از حد کلاس `Order` به دیگر بخش های سیستم؛ که به آن Tight Coupling می گویند.
- توسعه و گسترش آن سخت می شود و مجبوریم خیلی از بخش‌ها را در داخل کد تغییر دهیم که در نتیجه OCP را نقض کرده ایم.
- تست کردن کلاس `Order` طاقت فرسا می‌شود.

### راه حل

برای اینکه این مشکلات را رفع کرده و بتوانیم DIP را به درستی پیاده کنیم، چندین تکنیک وجود دارد.



استفاده از **تزریق وابستگی** یا **Dependency Injection** است که به اختصار **DI** می‌گوییم. تزریق وابستگی تکنیکی است که به ما اجازه می‌دهد تا در هنگام ساختن نمونه از کلاس مورد نظر، وابستگی‌های خارجی آن را به داخل کدهایش تزریق کنیم، بدون آنکه نیاز باشد آن

کلاس در داخل خود این کار را انجام بدهد. درواقع متدهایی که آن کلاس برای فراخوانی نیاز دارد را ما خودمان در اختیارش خواهیم گذاشت بدون اینکه او خود زحمتی برای ساختن و آماده سازی وابستگی‌اش بکشد.

DI از اصل هالیوود[۱] یا The Hollywood Principle پیروی می‌کند. این اصل می‌گوید:

> Don't call us, we'll call you.

یعنی «تو ما را صدا نزن، ما خودمان تو را صدا خواهیم زد». درواقع وقتی کلاس `Order` میخواهد برای مشتری با استفاده از SMTP ایمیل ارسال کند، به‌جای اینکه آن را ایجاد کند، به ما می‌گوید احتیاج به متدی جهت ارسال نوتیفیکیشن دارد. خودش می‌داند که چطور با آن نمونه کار می‌کند و می‌رود با آن چیزی را که در اختیارش گذاشته‌ایم، متد مورد نظرش را فراخوانی می‌کند بدون اینکه نیاز داشته باشد خودش آن را نمونه‌ای بسازد.

در تزریق وابستگی از ۳ تکنیک اصلی استفاده می‌کند:

- Constructor Injection
- Property Injection
- Parameter Injection

## تزریق سازنده یا Constructor Injection

اولین تکنیکِ تزریق وابستگی، تکنیکِ تزریق سازنده یا Constructor Injection بوده، که خود یک نمونه از الگوی Strategy است. در این تکنیک، وابستگی‌های کلاس مستقیما به متد سازنده‌ی آن کلاس پاس می‌شوند. در این حالت، کلاس کاملا راست‌گو بوده (راست‌گویی و دروغگویی کلاس‌ها در بحث وابستگی که یادتان هست؟) و تمامی آن وابستگی‌هایی که اولا حتما از آن‌ها استفاده می‌کند و ثانیاً دقیقا به آن‌ها نیازمند است را ذکر کرده و از ما درخواست می‌کند.

---

[۱] http://wiki.c2.com/?HollywoodPrinciple

### فواید

- کلاس‌ها به صورت خودجوش داکیومنت شده‌اند، زیرا خودشان می‌گویند که به چه چیزهایی برای کار کردن نیازمندند.
- این کلاس‌ها با یا بدون Container ها به خوبی کار خودشان را انجام می‌دهند (بعدا به Container اشاره خواهم کرد).
- وقتی که ساخته می‌شوند دیگر مشکلی ندارند و در همان حالت صحیحِ خود، پایدار خواهند ماند.

### مضرات

- متد سازنده یا Constructor ممکن است پارامترها یا وابستگی‌های زیادی را درخواست کند (و بیش از حد شلوغ بنظر برسد). (بوی بد طراحی)
- بعضی امکانات (مثلا Serialization) ممکن است خودشان یک متد سازنده پیش‌فرض و جدا، نیاز داشته باشند.
- ممکن است در کلاسِ شما متدهایی وجود داشته باشد که هیچ نیازی به آن وابستگی‌هایی که در متد سازنده ذکر کردید ندارد. یعنی در کلاسِ خود، کلاس‌هایی دارید که به چیزهایی وابستگی دارند که تعدادی دیگر از متدهای همان کلاس به آن احتیاجی ندارن و اصلا استفاده نمی‌کنند. در این حالت احتمال می‌رود SRP را نقض کرده باشد، به همین سبب پیشنهاد می‌شود که کدهای خود را به کلاس‌های کوچکتری تقسیم کنید.(بوی بد طراحی)

## تزریق دارایی یا Property Injection

دومین تکنیک، تزریق دارایی یا Property Injection است. در این روش، وابستگی‌ها را به‌عنوان یک پراپرتی پاس می‌کنید. البته این تکنیک با نام Setter injection نیز شناخته می‌شود، به این علت که وابستگی‌ها را از طریق متد `setter` یا `set` در کلاسِ مربوطه تزریق می‌کنیم.

### فواید

- می‌توانید وابستگی را در طول دوره زنده بودنِ آن شئ یا کلاس تغییر دهید. درواقع هرموقع که خواستید (تا زمانی که آن object وجود دارد) با فراخوانی آن متد `set`، وابستگی جدید را مقدار دهی کنید.
- از آنجا که می‌توانیم وابستگی را در طول زنده بودن شئ تغییر دهیم، در نتیجه کلاس ما منعطف‌تر می‌شود.

### مضرات

- مابین زمانی که کلاس توسط متد `constructor` ساخته می‌شود، تا زمانی که توسط متدِ `setter` وابستگی‌اش مقدار دهی شود، در حالتی ناپایدار قرار می‌گیرد.
- مگر اینکه کلاس در متد سازنده‌ی خودش (یعنی Constructor) بیاید و متد `setter` را فراخوانی کرده و آن مقدار دهی اولیه را انجام دهد؛ که در این صورت باز ما آن شفافیت و راست‌گویی کلاس را نداریم و در یک بازه زمانی خاصی، کلاس از وابستگی استفاده می‌کند که اولا خبر از آن نداریم ثانیا آن را (بوسیله اعلام نکردن وابستگی اش در متد سازنده) داکیومنت نکرده است.

## تزریق پارامتر (مقادیر) یا Parameter Injection

و در نهایت سومین تکنیک، تزریق پارامتر (ویژگی) یا Parameter Injection است. در این تکنیک، وابستگی‌ها به‌عنوان پارامتر یا مقادیر به متدی که به آن نیاز دارد پاس می‌شود.

### فواید

- سبب می‌شود تنها وابستگی‌های مخصوصِ همان متد را برایش پاس کنیم.
- بسیار منعطف است.
- نیازی نیست در کل کلاس تغییرات ایجاد کنیم یا حواسمان باشد که تمامی اعضای کلاس از آن وابستگی استفاده می‌کنند یا نه.

### مضرات

- تغییر دادن متد (اصطلاحا می‌گویند شکستن امضای متد) خیلی سخت می‌شود. فرض کنید از این متد در بخش‌های مختلف برنامه اســتفاده می‌کنید، تصــمیم می‌گیرید که وابســتگی مورد نظرش را عوض کنید یا یک وابســتگی جدید به آن بیافزاید، آن‌وقت تغییر دادن و تزریق وابســتگی در تمامی بخش‌هایی که از این متد اســتفاده می‌کنند به شدت سخت و پر هزینه می‌شود.

اگر تنها یک متد در کلاس‌تان دارید که فقط آن متد وابستگی دارد، بهتر است به‌جای این تکنیک، از هما تکنیک تزریق وابستگی در Constructor استفاده کنید.

حال که با مشــکل آشــنا شــدیم و تکنیک‌های رفع آن را فرا گرفتیم، بهتر اســت ســراغ بازسازی کدهایمان برویم.

## بازسازی کدها

اول از همه شما را ارجاع میدهم به اولین اصل سالید، یعنی اصل تک وظیفه‌ای یا SRP. در گام اول (همانند مثال بحش SRP) باید وظایف آورده شده در `Order` را بوسیله‌ی اینترفیس‌ها از آن جدا کنیم. پس در همین مرحله احتیاج به ۳ اینترفیس داریم. اینترفیس‌های `INotification` برای ارسال پیام به مشتری، `IPaymentProcess` برای پردازش پرداخت و `IReservation` رزرو محصول در انبار. (لازم است توجه داشته باشید که در این بخش تنها پرداخت آنلاین را مد نظر داریم.)

```
public interface INotification
{
    public void notifyCustomer(Cart cart) throws Exception;
}
```

```
public interface IPaymentProcess
{
    public void processPayment(
                PaymentDetails paymentDetails,
                Cart cart
    ) throws Exception;
}
```

```
public interface IPaymentProcess
{
    public void processPayment(
                  PaymentDetails paymentDetails,
                  Cart cart
    ) throws Exception;
}
```

سپس سه کلاس با نام های `NotificationService`، `PaymentProcessService` و `ReservationService` را ساخته، و به وا سطه‌ی پیاده سازی اینترفیس‌های متناظر با آنها، وظایف چندگانه‌ی کلاسِ `Order` را به کلاس‌های مربوطه انتقال می‌دهیم.

```
public class NotificationService implements INotification
{
    @Override
    public void notifyCustomer(Cart cart) throws Exception
    {
        //notificationService body
    }
}
```

```
public class PaymentProcessService implements IPaymentProcess
{
    @Override
    public void processPayment(
                  PaymentDetails paymentDetails,
                  Cart cart
    ) throws Exception
    {
        //processPayment body
    }
}
```

```
public class ReservationService implements IReservation
{
    @Override
    public void reserveInventory(Cart cart) throws Exception
    {
        //reserveInventory body
    }
}
```

حتما یادتان هست، پس از آنکه وظایف را به واسطه‌ی اینترفیس‌ها از سیستم جدا کرده، سپس کلاس `Order` را به یک کلاس ابسترکت تبدیل نمودیم؛ اینجا هم باید همین کار را انجام دهیم. پس کلاس `Order` را به یک کلاس ابسترکت تبدیل کرده، سپس متد کانستراکتورش را ایجاد می‌کنیم. وظیفه کلاس `Order` فقط انجام فرآیند خرید و ثبت سفارش است، پس یک مدت هم برای `checkout()` نیازمندیم.

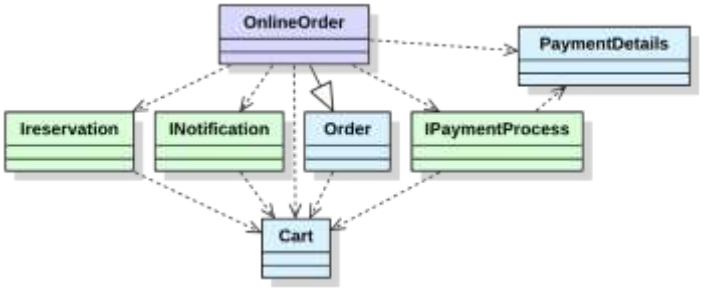
```
public abstract class Order
{
    public Cart cart;

    public Order(Cart cart)
    {
        this.cart = cart;
    }

    public abstract void checkout() throws Exception;
}
```

بعد از اینکه کلاس ابسترکت Order را ساختیم، حالا باید یک کلاس جدا گانه هم برای سفارش آنلاینی که از همان اول هدفمان انجام دادن آن بود، بسازیم.

در کانستراکتورِ `OnlineOrder` تمام چیزهایی که به آن نیاز داریم را ذکر می‌کنیم.

```
public class OnlineOrder extends Order
{
    private PaymentDetails paymentDetails;
    private IReservation reservation;
    private IPaymentProcess paymentProcess;
    private INotification notifier;

    public OnlineOrder(
            Cart cart,
            IPaymentProcess paymentProcess,
            PaymentDetails paymentDetails,
            IReservation reservation,
            INotification notifier
    ) {
        super(cart);
        this.paymentProcess = paymentProcess;
        this.paymentDetails = paymentDetails;
        this.reservation = reservation;
        this.notifier = notifier;
    }

    @Override
    public void checkout() throws Exception
    {
        paymentProcess.processPayment(paymentDetails, cart);
        reservation.reserveInventory(cart);
        notifier.notifyCustomer(cart);
    }
}
```

حالا هم SRP را رعایت کردیم، هم با استفاده از تکنیکِ تزریق سازنده یا همان *Constructor Injection* توانستیم DIP را پیاده کنیم. برای اینکه مطلب برایمان قابل درک باشد، بهتر از دوباره نگاه به دیاگرام UML بیاندازیم و ببینیم حالا وابستگی‌هایمان چطور شده است.



همانطور که می‌بینید، کلاس `OrderOnline` به‌جای آن همه وابستگی‌های پنهان و عجیب و غریب، تنهای به ۳ اینترفیس، یک کلاس ابسترک و دو مدل وابسته است که البته همه‌ی آنها، به واسطه‌ی اعلام وابستگی در متد کانتسراکتور، هم برای ما آشکار اند و هم قابل مدیریت. دیگر نگران اینکه در داخل این کلاس با چه چیز مواجه خواهیم شد نیستیم.

و از طرفی، وابستگی به ماژول‌های خارجی (مثل سرور ایمیل، دیتابیس، سرویس پرداخت و سرویس مدیریت انبار) برعکس شده و جهت وابستگی آن‌ها به سمت اینترفیس‌ها برگشته است. اینجا دقیقا همانجایی است که **وارونگی وابستگی‌ها** یا همان **Dependency Injection** رخ داده.

حال دیگر می‌توانیم با خیال راحت متد تست خود را نوشته و وابستگی‌ها را (ولو به صورت *fake*) به داخل کلاس **تزریق** یا **Inject** کنیم.

قبل از اینکه نوشتن تست را شروع کنیم، توجه داشته باشید که به کلاس‌های ارسال پیام به مشتری، پرداخت و مدیریت انبار را جهت تزریق به کلاس مورد تست‌مان احتیاج داریم. می‌توانید این کار را با ایجاد کلاس‌های *Fake* یا تقلبی انجام دهیم (هیجان زده نشوید، در ادامه دلیل این کار را خواهید فهمید). پس کلاس‌های `FakeNotificationService`، `FakePaymentProcessor` و `FakeReservationService` را به همراه پیاده سازی اینترفیس‌های متناظرشان ایجاد می‌کنیم.

```
public class FakeNotificationService implements INotification
{
    public boolean wasCalled = false;

    @Override
    public void notifyCustomer(Cart cart)
    {
        wasCalled = true;
    }
}
```

```
public class FakePaymentProcessor implements IPaymentProcess
{
    public float amountPassed = 0;
    public boolean wasCalled = false;

    @Override
    public void processPayment(
                  PaymentDetails paymentDetails,
                  Cart cart
    ){
        wasCalled = true;
        amountPassed = cart.getTotalAmount();
    }
}
```

```
public class FakeReservationService implements IReservation
{
    public boolean wasCalled = false;

    @Override
    public void reserveInventory(Cart cart)
    {
        wasCalled = true;
    }
}
```

حتما دلیل این کار را متوجه شده اید. اصلا برایمان اهمیت ندارد که پیام چگونه یا حتی اصلا به کاربر ارسال می‌شود یا نه! هیچ اهمیتی ندارد که بدانیم چگونه پرداخت انجام می‌شود. اصلا نمی‌خواهیم بدانیم که چگونه عملیات رزرو شدن محصولات در انبار صورت می‌گیرید. ما فقط می‌خواهیم همه‌ی این‌ها انجام شوند. برای ما مهم است که تنها بخش منطقی برنامه یعنی `Order` یا `OnlineOrder` به درستی کار خود را انجام دهد. نه می‌خواهیم بدانیم سرور ایمیل درحال اجراست؟ نه می‌خواهیم بدانیم که پرداخت توسط کدام درگاه انجام می‌شود. تنها و تنها `Order` باید با موفقیت به سر انجام برسد.

حال شروع می‌کنیم به ساخت کلاس یونیت تست، و در اولین گام بررسی می‌کنیم و می‌بینیم که آیا سفارش `OnlineOrder` با موفقیت ثبت شده و به سمت سرویس پرداخت پاس می‌شود یا خیر. پس متد تست `SendTotalAmountToPaymentProcessor` را

نوشته و با کمک سرویس‌های *fake*ای که ایجاد کرده‌ایم، وابستگی‌های مورد نیاز را به کلاس `OnlineOrder` تزریق می‌کنیم.

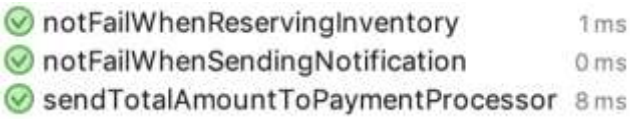
```
public class OrderTest
{
    FakePaymentProcessor paymentProcessor =
                            new FakePaymentProcessor();

    FakeReservationService reservationService =
                            new FakeReservationService();

    FakeNotificationService notificationService =
                            new FakeNotificationService();

    Cart cart= new Cart();

    PaymentDetails paymentDetails = new PaymentDetails();

    @Before
    public void setup()
    {
        paymentDetails.setPaymentMethod(
                    PaymentDetails.PaymentMethod.CreditCard
        );
    }

    @Test
    public void SendTotalAmountToPaymentProcessor() throws Exception
    {
        cart.setTotalAmount(500);

        OnlineOrder order = new OnlineOrder(
                cart,
                paymentProcessor,
                paymentDetails,
                reservationService,
                notificationService
        );

        order.checkout();
        Assert.assertTrue(paymentProcessor.wasCalled);
        Assert.assertEquals(
                    cart.getTotalAmount(),
                    paymentProcessor.amountPassed, 0
        );
    }
}
```

در انتهای این تست بررسی می‌کنیم که آیا اصلا سرویس پرداخت فراخوانی شده یا نه؟ و بعد بررسی می‌کنیم که آیا مقدارِ کلِ هزینه‌ی سبد خرید با مقدار پاس شده به سرویسِ پرداخت برابر است یا خیر؟ انتظار داریم که نتیجه‌ی تست بدون هیچ مشکلی سبز باشد.


سپس باید مطمئن شویم که Order در فراخوانی ارسال پیام و رزرو محصولات در انبار هیچ مشکلی ندارد. پس یونیت تست مربوط به آن‌ها را نیز می‌نویسیم.

```
@Test
public void notFailWhenSendingNotification() throws Exception
{
    cart.setCustomerEmail("someone@somewhere.com" );

    OnlineOrder order = new OnlineOrder(
            cart,
            paymentProcessor,
            paymentDetails,
            reservationService,
            notificationService
    );

    order.checkout();

    Assert.assertTrue(notificationService.wasCalled);
}

@Test
public void notFailWhenReservingInventory() throws Exception
{
    OnlineOrder order = new OnlineOrder(
            cart,
            paymentProcessor,
            paymentDetails,
            reservationService,
            notificationService
    );

    order.checkout();

    Assert.assertTrue(reservationService.wasCalled);
}
```

و همانطور که انتظار می‌رود، همه‌ی متدها بدون مشـکل، صـدا زده شـده و به خوبی کار می‌کنند.




همانطور که دیدید، هیچ اهمیتی نداشت که این سرویس‌ها چگونه کار می‌کنند، فقط مهم است که `Order` وظیفه خود را جهت ارسال اطلاعات و تکمیل سفارش به خوبی انجام دهد، حال می‌خواهد دیتابیس یا سرور ایمیل وجود داشته باشد یا خیر.

البته بعدا باید برای هرکدام از آن کلاس‌های سرویس نیز، تست‌های جداگانه‌ای بنویسید. این را دیگر بر عهده خودتان می‌گذارم.

### نکات

در ادامه برخی نکات را ذکر خواهم کرد که در شناخت و پیاده سازی DIP به شما کمک خواهد کرد.

## بوی بد طراحی

**استفاده کردن از کلمات کلیدی جدید**. اگر در کدتان جایی وجود دارد که بدون استفاده از اینترفیس و مستقیما یک نمونه از کلاسی را ساخته و در یک متغییر قرار داده و سپس استفاده می‌کنید، احتمالا همان‌جایی است که باید DIP را پیاده کنید. در اینجا منظور از کلمه کلیدی، همان متغییری است که با نمونه جدیدی از کلاس مقدار دهی شده است. به مثال زیر توجه کنید.

```
for(OrderItem irem : cart.getOrderItems())
{
    try
    {
        InventorySystem inventorySystem = new InventorySystem();
        inventorySystem.reserve(item.getSku(), item.getQuantity());
    }
}
```

در مثال بالا می‌بینید که از کلاس `InventorySystem` یک نمونه ساخته‌ایم. اگر سیستم مدیریت انبار از یک وابستگیِ خارجی استفاده کرده باشد، یا مثلا برای ذخیره کردن سفارش

از دیتابیس استفاده کرده و درواقع به آن وابسته باشد، آنگاه این قسمت از کد ما نیز آن وابستگی به دیتابیس را به واسطه‌ی `InventorySystem` ارث بری کرده است.

برای اینکه این وابستگی را از بین ببریم، می‌توانیم به‌جای ارث بری مستقیم از کلاس `InventorySystem` از یک کلاس ابسترکشن، یا حتی راحت تر، از یک اینترفیس استفاده کرده و به راحتی با استفاده از *Strategy Pattern* یا *Constructor injection* این وابستگی را به کد تزریق کنیم.

**استفاده کردن از متد یا مقادیر استاتیک**. همانند استفاده از کلماتِ کلیدی جدید، استفاده از متدها یا مقادیر استاتیک نیز کدهای ما را به وابستگی‌های آن کلاس وابسته می‌کند. مثال خیلی ساده آن هم، استفاده از `Date` است.

```
message.setSubject("Your order placed on " + new Date().toString());
```

در اینجا می‌بینید که کد ما به متعلقات Date وابسته شده است. شاید در نگاه اول وابستگی خاصی نبینید، اما فرض کنید برای تعیین زمان یا تاریخ بخواهیم از `Calendar` یا `DatePicker` استفاده کنید که مستقیما در ارتباط با رابط کاربری خواهد بود. آنگاه کد شما که قاعدتا باید تنها بحث منطقی و Loginc قضیه را انجام دهد، مستقیما با رابط کاربری و همچنین خود کاربر درگیر می‌شود.

اینجا با زمان و Api های `Date` سروکار داریم، ممکن است در پروژه‌ی دیگری با دیتابیس درگیر باشیم (مثلا بخواهیم اطلاعات کاربر را پس از خرید ثبت کنیم)، آنگاه در هنگام توسعه و تست بیش از پیش درگیر و سردرگم خواهیم شد. پس تا جایی که می‌توانید از متد و مقادیر استاتیک دوری کنید؛ زیرا استفاده از آنها سبب می‌شود وابستگی‌هایشان را نیز به ارث ببرید درحالی که نه احتیاجی به آن‌ها دارید نه لازم است آن‌ها را مدیریت کنید.

تنها جایی که باید از متدها یا مقادیر استاتیک استفاده کرد، آن وقتی است که مطمئن با شید آن متد به تنها چیزی که دست می‌زند و احتیاج دارد همان مقادیری ست که شما به آن پاس می‌کنید. مانند مثال پایین.

```
System.out.println( Calculator.sumTwoNumbers(5, 6) );
```

از متد استاتیک `sumTwoNumbers` استفاده کردیم، این متد تنها کاری که انجام می‌دهید این است که همان مقادیری که به آن پاس کرده‌ایم استفاده می‌کند و خود هیچ وابستگی ندارد که وابستگی کدهای ما را بیشتر کند. در غیر این صورت هم کار خود را برای توسعه سخت کرده‌اید هم برای تست گرفتن کار اضافی ایجاد کرده‌اید.

## کجا نمونه‌سازی کنیم

حال که متوجه شدیم نباید در هرجایی از سیستم و لابه‌لای کدهایمان از کلاس نمونه بگیریم، پس چه جایی مناسب این کار است؟

وقتی قرار باشد این تزریق وابستگی را انجام دهیم، آنگاه اینترفیس‌های کوچکی ایجاد می‌کنیم. این اینترفیس‌ها بخوبی همپوشانی دارند، به هم وابسته نیستند، هرکدام یک وظیفه خاصی داشته و از SRP پیروی می کنند، و صد البته اصل Interface Segregation را نیز به خوبی پیاده می‌کنند. اما خب این اینترفیس‌ها و این کلاس ها بالاخره باید در جایی ساخته شوند. برای این کار چند راه داریم.

- **متد سازنده‌ی پیش فرض:** می‌توانیم با استفاده از متدهای Constructor که توسط کلاس پیاده شده، وابستگی‌های مورد نیاز آن کلاس را ساخته و در آن تزریق کنیم. مانند مثالی که قبلا روی آن کار کردیم، در کلاس `OnlineOrder` به سرویس ارسال پیام نیاز و وابستگی داشت که نیاز خود را از طریق اینترفیس `Inotification` در متد پیش فرض سازنده‌ی خود به ما اعلام کرده بود، و ما هم از کلاس `NotificationService` یک نمونه ساخته و به آن تزریق کردیم.

گاهی به این روش `Poor man's dependency Injection` یا `Poor man's IoC` نیز می‌گویند، گویا یک آدم بیچاره‌ای احتیاج به چیزی دارد و از ما آن را درخواست می‌کند، که ما موظف‌یم تا خواسته‌ی او را اجرا کنیم؛ خواه با کلاس‌های حقیقی باشد، یا کلاس‌های دروغین و شبیه‌سازی شده. کلاس بطور خودکار وابستگی‌های مورد نیازش را دریافت کرده، و مثل سابق وظایف خود را انجام می‌دهد، بدون اینکه اطلاع داشته باشه در دل آن وابستگی‌های تزریق شده چه اتفاقی در حال رخ دادن است. دقیقا مانند کاری که در تست برای کلاس `OnlineOrder` انجام دادیم.

- **متد Main:** امکان دیگری که در اختیار ما ست، ایجاد کردن وابستگی در متد شروع کننده (Startup Methods) یا در بعضـی تکنولوژی‌ها، مثل اپلیکیشـن‌های جاوا، در متد `Main()` اسـت. در وب اپلیکیشـن‌ها (مثلا در React یا حتی Nodejs) این کار می‌تواند در شروع سرویس به صورت عمومی انجام شود.
- **IoC Container: وارونگی کنترل** یا **Inversion of Control** که به اختصار **IoC** می‌گوییم، درواقع نحوه‌ی پیاده سازی DIP را بیان می‌کند. در اینجا، کنترل به تمامی مسئولیت های اضافی یک کلاس به غیر از مسئولیت اصلی آن گفته می شود، مانند کنترل جریان برنامه، کنترل جریان ایجاد شیء، یا ایجاد شیء وابسته و **binding**. به بیان عامیانه، فرض کنید که سوار بر خودروی خود بوده و در مسیر محل کار خود در حال رانندگی هستید، که این یعنی شما در حال کنترل خودروی خود هستید. اصل IoC وارونه کردن کنترل را پیشنهاد می دهد. این یعنی شما به‌جای آنکه خودتان رانندگی کنید، یک تاکسی بگیرید و بگذارید فرد دیگری رانندگی کند. در این صورت است که کنترل از شما به راننده ی تاکسی وارونه می شود. شما دیگر مجبور

نیستید خودتان رانندگی کنید و برای آنکه بتوانید بر روی کار اصلی خود تمرکز کنید، این کار را به راننده ی تاکسی محول می کنید. با کمک این اصل، این کلاس ها قابل آزمایش و نگهداری بوده و در نهایت **توسعه** پذیر می شوند. مرجع یا همان **Inversion of Control Container**، فریم ورکی است برای انجام تزریق وابستگی‌ها. در این فریم ورک امکان تنظیم اولیه‌ی وابستگی‌های سیستم وجود دارد. برای مثال زمانی که برنامه از یک IoC Container ، نوع اینترفیس خاصی را درخواست می‌کند، این فریم ورک با توجه به تنظیمات اولیه‌اش، کلاسی مشخص را بازگشت خواهد داد. درواقع IoC همان روند مشابه را در متد *Main* یا *Startup*برنامه دارد، با این تفاوت که شامل دسته‌ای از مکانات است تا اصطلاحا نقشه‌کشی وابستگی‌ها به ساده‌ترین شکل ممکن صورت گرفته تا توسعه‌دهنده دچار سردرگمی نشود. هر تکنولوژی، متناسب با محیط توسعه خود، از فریم‌ورک IoC خاصی استفاده می‌کند. برای مثال در `.NET` از **StructureMap** استفاده می‌کنند، یا در `Java` در خود هسته‌ی فریم ورکِ **Spring** این امکان گنجانیده شده است، یا در `PHP` هم در خود **Laravel** این امکان وجود دارد. نمی‌خواهم بیش از این این مبحث را باز کنم، بهتر است متناسب با تکنولوژی خود بررسی کنید و ببینید از چه فریم‌ورکی و چگونه می‌توانید از IoC مربوط به آن استفاده کنید.

## تفاوت وارونگی کنترل و وارونگی وابستگی

اگر در باره‌ی *Inversion Control* و *Dependency Inversion* کمی گیج شده اید به مثالی که می‌زنم توجه کنید. فرض کنید قرار است به یک مهمانی بروید.

**روش سنتی:** وارد مهمانی شده و نوشیدنی مورد نظر خود را از روی میز برمی‌دارید.

**روش IoC:** وارد مهمانی شده، یک جعبه یخ می‌بینید که درون آن پر از نوشیدنی‌های مختلف است. یک نفر کنار جعبه ایستاده و خودش نوشیدنی‌ها را از داخل جعبه درآورده و به شما می‌دهد. شما فقط نوشیدنی را دریافت می‌کنید که به شما داده شده است.

**روش DI:** وارد مهمانی شده، یک بارِ نوشیدنی رایگان می‌بینید. مسئولِ بار برای شما نوشیدنی درست می‌کند، میزبان نوشیدنی را تحویل گرفته و به شما می‌دهد. شما نمی‌دانید آن نوشیدنی چیست، اما از مزه‌اش لذت می‌برید.

وارونگی وابستگی یک نوعِ جذاب و فانتزی از وارونگی کنترل است، اما خب احتیاج به یک مسئولِ بار و یک میزبان دارد.

## پیروی از DIP در ساختار برنامه

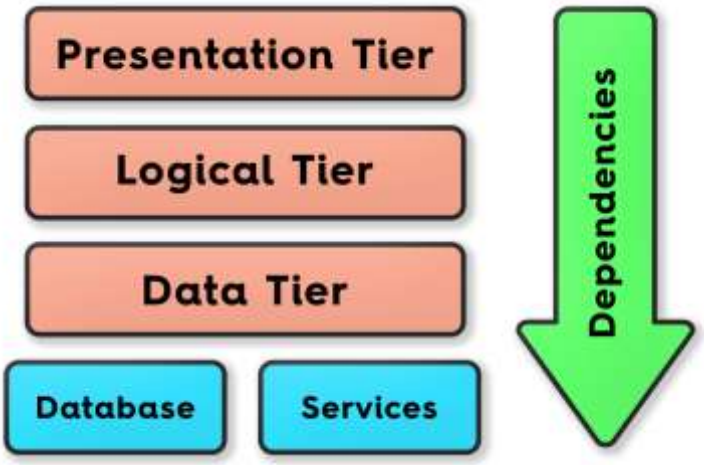
در یک برنامه نرم افزاری عموما چندین بخش منطقی (و بعضی وقت‌ها فیزیکی) داریم که در چند لایه‌ی متفاوت از هم جدا شده‌اند. بخش‌هایی نظیر:

- لایه‌ی ارائه (Peresentaion Tier)
- لایه‌ی منطقی یا درواقع هسته‌ی برنامه که معمولا با بیزنس لاجیک شناخته می‌شود (Logical Tier)
- لایه‌ی داده (Data Tier)

برنامه‌هایی که ساختارشان اینگونه است را معماری سه لایه (۳ طبقه) یا *Three-tier Architecture* می‌نامند. این لایه‌ها به خوبی کپسوله و ایزوله می‌شوند و (در حالت ایده‌آل) تنها از وجود یک لایه پایینتر خود اطلاع دارند.

برنامه‌ها ساختاری به این شکل دارند که عموما به صورت یک کامپوننت یا ماژول قابل استفاده‌ی مجدد در پروژه‌های مختلف هستند. لازم به ذکر است که در بعضی برنامه‌ها نیز، این لایه‌ها خود به پروژه‌های جداگانه و مستقل تقسیم می‌شوند.

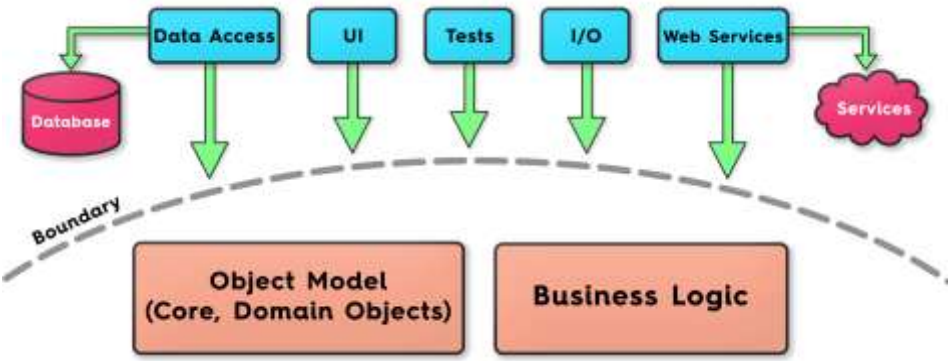
شِمای ساختار اینگونه برنامه‌ها به شکل زیر است.




همانطور که میبینید PT به LT وابستگی دارد، LT به DT و در نهایت DT به پایگاه داده یا سرویس‌های بیرونی. در این صورت شاهد هستیم که وابستگی از بالاترین سطح به پایین‌ترین سطح وجود دارد، درواقع وابستگی در این سیستم انتقالی‌است، از بالاترین لایه و مرحله به مرحله به پایین‌ترین لایه می‌رسد. در سیستم‌هایی با این ساختار، تست گرفتن و ایجاد تغییرات بسیار مشکل خواهد بود. این امر موجب پیچیده شدن نرم افزار و طاقت فرسا شدن توسعه و نگهداری از آن خواهد شد. هر لایه‌ای که شما به برنامه خود اضافه می‌کنید. پیچدگی آن را نیز بیشتر کرده‌اید.

ما در اینجا هیچ لایه‌ی انتزاعی (Abstraction Layer) را شاهد نیستیم، در نتیجه می‌بینیم که بخش‌های سیستم بلاواسطه از جزئیات یکدیگر خبر دارند. در این ساختار به‌جای اینکه بیزنس لاجیک برنامه به ابسترکشن دسترسی داشته باشد، مستقیما از نمونه‌ای که از Data Acess گرفته شده استفاده می‌کند که باعث وابستگی لایه‌ی منطقی برنامه به پایگاه داده می‌شود.

مطابق با آنچه در این فصل یاد گرفته‌ایم، بهتر است جهت وابستگی و نحوه ارتباط لایه‌ها و بخش‌ها را تغییر دهیم. ساختار مناسبی که برای این برنامه می‌توان متصور شد، شبیه چیزی است که در ادامه آورده ام.




در این صورت، بخش بیزنس لاجیک و مدل‌ها در انتهای برنامه قرار می‌گیرند (در یک حیطه مجزا در یک مرز یا *Boundary* با دیگر بخش‌ها) که البته می‌توان آن‌ها را به‌عنوان یک پکیج یا کامپوننت جدا نیز در نظر گرفت. توجه کنید که در این معماری، تمامی وابستگی‌ها از طریق انتزاعات (کلاس ابسترکشن و اینترفیس‌ها) مدیریت می‌شوند.

دیگر بخش‌های سیستم مانند *Data Access*، *UI*، تست‌ها، *I/O*، وبسرویس‌ها و غیره در بالاترین سطح برنامه قرار گرفته و جهت وابستگی آنها به سمت هسته برنامه است. و در نهایت چیزهایی که همیشه نگران آن‌ها خواهیم بود (مانند دیتابیس یا سرویس‌های خارجی) در خارجی‌ترین سطح برنامه قرار خواهند گرفت.

حال توانستیم ساختار برنامه را با اعمال DIP به گونه‌ای تغییر دهیم که به راحتی بتوانیم تمام بخش‌های منطقی سیستم را بدون نگرانی از وابستگی‌های و مشکلات احتمالی آن‌ها، مورد تست، توسعه و نگهداری مداوم قرار دهیم.

## جمع بندی

با موارد زیادی در این فصل آشنا شدیم. بطور خلاصه:

- سعی کنید به‌جای استفاده‌ی مستقیم از وابستگی‌ها، از اینترفیس یا کلاس‌های ابسترکشن استفاده کنید.
- هیچگاه کلاس‌های سطح بالای سیستم را، مجبور نکنید تا به واسطه‌ی استفاده از نمونه‌سازی درون کد یا متدهای استاتیک، به کلاس‌های سطح پایین یا جزئیات وابستگی پیدا کنند.
- وابستگی‌های کلاس را به صورت شفاف در متد سازنده‌ی آن کلاس ذکر کرده و از آن‌جا آن‌ها را دریافت کنید.
- می‌توانید وابستگی‌ها را در متد کانستراکتور یا به‌عنوان پارامتر به متدهای داخلی یک کلاس تزریق کنید.

در این فصل از مفاهیم زیر نیز استفاده کردیم:

- اصل تک‌وظیفه‌ای یا Single Responsiblity Principle
- اصل جدا سازی اینترفیس‌ها یا Interface Segeragation Principle
- الگوی فَصاد یا Façade Pattern
- مرجع وارونگی کنترل یا Inversion of Control Containers

سخت‌ترین و احیانا پیچیده‌ترین اصل سالید، همین وارونگی وابستگی یا Dependency Inversion بوده که مستلزم مطالعه و تمرین مداوم است.

# بیشتر بدانیم

*« Repetition is the root of all software evil. »*

## مقدمه

حال که با همه‌ی این ۵ اصل آشنا شده و تکنیک‌های پیاده سازی آن را فرا گرفته‌اید، وقت آن است که تمرین‌های عملی خود را شروع کنید. می‌خواهم در ادامه منابع و مباحثی را معرفی کنم، که مهارت‌های شما را در این مسیر بیشتر کرده و روند یادگیری شما را تسریع خواهد بخشید.

## مفهوم DRY

**خودت را تکرار نکن** یا **Don't Repeat Yourself** اصلی در برنامه‌نویسی رایانه‌ای است که به‌وسیله‌ی عدم تکرار یک یا چند خط کد در مکان‌های مختلف برنامه رعایت می‌شود. درواقع با این کار، برای اصلاح بخشی از برنامه نیاز به اصلاح قسمت‌های مختلف و جداگانه‌ای از کد نیست. این اصل برای اولین بار در کتاب *The Pragmatic Programmer* نوشته‌ی اندی هانت[۱] و دیو توماس[۲] معرفی شد.


شاید این اصل در ظاهر خیلی ساده به نظر برسد، اما بیش از حد گسترده است. در این کتاب، DRY اینگونه توصیف می‌شود:

> *هر بخشی از دانش (Knowledge) داخل یک سیستم باید یک نمود یکتا، نامبهم و معتبر داشته باشد.*

برای رعایت این اصل، معمولاً برنامه‌نویسان کد خود را داخل یک تابع یا کلاس قرار داده و در موارد مورد نیاز تابع را فراخوانی کرده یا یک شئ جدید از کلاس می‌سازند. بر طبق این اصل، هر برنامه‌نویس دقایقی پس از نوشتن چند خط اول، مراحل نگهداری یا پشتیبانی از کد خود را آغاز می‌کند.

---

[۱] Andy Hunt

[۲] Dave Tomas

## مفهوم KISS

*ساده نگهش دار احمق*، انجام کار به ساده‌ترین و بدیهی‌ترین روش یا اصل کیس (Kiss Principle)، عنوان یک مفهوم است که توسط کلی جانسون [۱]معرفی شد. صورت‌های مختلفی که برای این سرواژه ذکر شده است شامل عبارات زیر می‌باشد:

- Keep it simple, stupid
- Keep it short and simple
- Keep it simple and straitforward
- Keep it simple sir
- Keep it simple or be stupid
- Keep it simple and sincere

این اصـل بیان می‌کند که اکثر سـیسـتم‌ها چنانچه سـاده و به‌دور از پیچیدگی بمانند، عملکرد بهتری خواهند داشت؛ بنابراین، سادگی باید هدف اصلیِ طراحی سیستم‌ها باشد و از پیچیدگی‌های بیهوده اجتناب کرد. این سـرواژه مخفف هرچه که باشـد یک معنی بیشـتر نمی‌دهد و آن هم اینکه سعی کنید تا حد ممکن چیزهایی که در اختیار سایرین قرار می‌دهید را ساده طراحی کرده تا ایشان برای استفاده از آن مجبور به فکر کردن نباشند.

## مفهوم YAGNI

شعار «You aren't gonna need it» یا به اختصار *YAGNI* که در برنامه‌نویسی اجایل به شدت اسـتفاده می‌شـود، دلالت بر این موضـوع دارد که الزامن چون فکر می‌کنیم برخی از امکانات ممکن اسـت در آینده در نرم افزارمان نیاز شـود، نباید آن را پیاده کنیم، این شـعار می‌گوید *تو الان به آن احتیاجی نداری، تا زمانی که نیاز نشده کدی را ننویس!*

ران جفریس [۲] از موسسان متد XP [۳] در اجایل، می‌گوید:

> *Always implement things when you actually need them, never when you just foresee that you need them.*

---

[۱] Kelly Johnson

[۲] Ron Jeffries

[۳] Extreme Programming

چیزهایی را که واقعا الان لازم دارید پیاده کنید، نه چیزهایی که پیش‌بینی می‌کنید ممکن است در آینده مورد نیازتان واقع شود. دلیل مهمی که برای *YAGNI* آورده می‌شود این است که کدی که شما با پیش‌بینی آینده می‌نویسید به احتمال زیاد در آینده برای شما مفید نخواهد بود! چرایی این مساله هم خیلی روشن است: شما امروز و بر اساس تفکرات امروز اقدام به طراحی و پیاده‌سازی کد می‌کنید. اگر به همین صورتِ مساله (مثلاً ۶ ماه دیگر) نگاه بکنید ممکن است روش متفاوتی را برای پیاده‌سازی آن در نظر بگیرید.

علاوه بر این باید در نظر گرفت که زمان و در نتیجه هزینه‌ی کلی نوشتن کدهای اضافی برای یک کتابخانه فقط شامل هزینه نوشتن کد نیست، بلکه دیباگ و تست و احتمالاً توسعه آن در آینده (به دلیل عدم شناخت کافی از مساله‌ای که امروز آن را در اختیار نداریم!) نیز شامل صرف زمان و هزینه‌ی بیشتر است.

## مفهوم SoC

تفکیک دغدغه‌ها یا Separation of Concerns که به اختصار **SoC** نیز می‌گوییم، یک قاعده‌ی طراحی با روش تفکیک برنامه به بخش‌های متمایز است، به گونه‌ای که عملکرد هر بخش، یک دغدغه‌ی متمایز را پاسخ می‌دهد. دغدغه، مجموعه اطلاعاتی است که کد یک برنامه‌ی کامپیوتری را تحت تأثیر قرار می‌دهد. این دغدغه ممکن است خیلی کلّی باشد مثلاً جزئیات سخت افزاری که کد برنامه را برای اجرا روی آن بهینه می‌کنیم، یا دغدغه ممکن است خیلی مشخص و جزئی باشد مثلاً نام کلاسی که از آن اشیاء ساخته می‌شوند.

برنامه‌ای که تفکیک دغدغه‌ها را به کار می‌گیرد، برنامه‌ی پودمانی، قطعه‌ای[۱] یا ماژولار می‌گویند. قطعه‌بندی یا پودمان[۲]، و در نتیجه‌ی آن، تفکیک دغدغه‌ها از طریق کپسوله‌سازی یا بسته‌بندی[۳] اطلاعات در بخشی از کد که دارای یک رابط[۴] مناسب می‌باشد، حاصل می‌شود.

---

۱ Modular

۲ Modularity

۳ Encapsulation

۴ Interface

طراحی لایه‌ای[۱] در سامانه‌های اطلاعاتی، نمونه‌ی دیگری از به کارگیری تفکیک دغدغه‌هاست. (مثلاً لایه‌ی نمایش یا ارائه[۲] یا لایه‌ی منطق کار[۳] و لایه‌ی دسترسی به داده[۴]، لایه‌ی تداوم اطلاعات[۵])

اهمیت تفکیک دغدغه‌ها در آسان‌سازی و تسهیل فرایند توسعه و نگهداری و توسعه‌پذیری برنامه‌های کامپیوتری می‌باشد. وقتی دغدغه‌ها به صورت مناسب تفکیک شده باشند، هر بخش از برنامه را می‌توان به صورت کاملاً جداگانه در برنامه‌ی دیگر مورد استفاده قرار داد و همینطور، آن را توسعه داد و به روز رسانی کرد. یکی از فواید مهم تفکیک دغدغه‌ها این است که می‌توان بعد از پایان پروژه، بخش‌هایی از کد را بدون نیاز به دانستن جزئیات بخش‌های دیگر (بخش‌های وابسته) و بدون نیاز به اِعمال تغییرات متناظر در بخش‌های دیگر، اصلاح کرد یا آن‌ها را بهبود بخشید.

این مفهوم همان موردی بود که در اصل آخر، یعنی *Dependency Inversion*، به آن اشاره کردیم.

## شش کار احمقانه یا S.T.U.P.I.D

در ادامه با ۶ موضوع آشنا می‌شوید که همیشه فکر می‌کردید درست است، اما کاملا در مسیر اشتباه قرار دارید. آن‌ها را باهم بررسی خواهیم کرد.

### Singelton

الگوی *Singelton* احتمالا معروف‌ترین الگوی در بین *Design Pattern*های برنامه‌نویسی است، و صد البته بیشترین برداشت اشتباه نیز در این الگو رخ داده است. تا به حال چیزی در باره سندرم *Singelton* شنیده‌اید؟ خیلی از برنامه‌نویسان فکر می‌کنند این الگو را باید در بسیاری از

---

[۱] Layered Design

[۲] Presentation Layer

[۳] Business logic Layer

[۴] Data access Layer

[۵] Persistence Layer

موارد استفاده کنند، و این ماجرا باعث می‌شود شما در هرجایی از سیستم با Singeltonها مواجه شوید.

سینگلتون‌ها بحث برانگیزند و همچنین در بیشتر مواقع به عنوان ضد الگو[۱] نیز شناخته می‌شوند، که البته از شما انتظار می‌رود نسبت به آن‌ها آگاه بوده و از آن‌ها دوری کنید. استفاده از خود Singelton مشکل نیست، بلکه استفاده بیش از حد از سینگلتون باعث مشکلات می‌شود، مواردی مثل:

- برنامه‌های که از stateهای گلوبال استفاده می‌کنند، برای تست کردن بسیار سخت است.
- برنامه‌هایی که به stateهای گلوبال وابسته‌اند، وابستگی‌های خود را پنهان می‌کنند.

اما آیا باید از همیشه از Singleton حذر کرد؟ باید بگویم بله، همیشه می‌توانید سینگلتون را با چیز بهتری جایگزین کنید. خودداری از چیزهای static به دلیل جلوگیری از چیزی که Tight Coupling می‌نامیم ضروری است.

## Tight Coupling

اتصال سخت یا Tight Coupling که Strong Coupling نیز گفته می‌شود، اصلی‌ترین مشکل حاصل از Singelton است. Coupling درواقع درجه‌ای است که نشان دهنده‌ی تکیه‌ی هریک از ماژول‌های برنامه بر ماژول دیگری‌ست.

هرگاه تغییر در یک ماژول از برنامه، شما را مجبور می‌کند تا در ماژول دیگری نیز تغییراتی را اعمال کنید، آنگاه می‌فهمیم که در برنامه‌ی شما Coupling وجود دارد. برای مثال، به‌جای اینکه یک نمونه از کلاس را به متد Contstructor پاس کنید، مسقیما در خود متد Constructor نمونه‌ای از آن کلاس می‌سازید؛ این کار اشتباه است، زیرا به شما اجازه نمی‌دهد که دیگر تغییری در آن بخش از برنامه دهید. فرض کنید بخواهید به‌جای آن کلاس، از یک SubClass دیگری استفاده کنید، یا حتی در تست آن‌ها را mock کنید.

وجود Tight Coupling در برنامه، استفاده مجدد از کدها را ناممکن و همچنین تست گرفتن را سخت می‌کند.

---

[۱] Anti Pattern

## Untestability

یونیت تست مانند نفس کشیدن، برای زنده ماندن یک برنامه ضروری است. اگر کدهای خود را تست نکنید، شما ملزم به قبول مسئولیت باگ‌ها و مشکلات مختلف نرم افزاری در برنامه خود خواهید بود. حقیقتا درک نمی‌کنم، چرا هنوز هم برنامه‌نویسانی دیده می‌شوند که کدهای خود را با تست‌های کافی پوشش نمی‌دهند[۱]. حقیقتاً چرا؟

به نظر من، انجام فرآیند تست نباید سخت باشد! هرگاه که با بهانه‌ی نداشتن وقت کافی ، از نوشتن تست‌ها شانه خالی می‌کنید، به این معنی است که کدهای شما پیچیده و اسپاگی بوده و بد نوشته شده اند. قطعا Tight Coupling مسبب تمام این پیچدگی و سختی‌ها در تست کردن است. اگر کدهای شما خوب بود، قطعا بدون وقفه در ابتدا تست‌ها را نوشته سپس سراغ توسعه کدها می‌رفتید. تنها با کدهای بد است که نوشتن یونیت تست سخت می‌شود. به یاد بیاورید که در فصل‌های قبل چقدر راحت و بدون دردسر تست‌ها را نوشته و تمام بخش‌های سیستم را با یونیت تست‌های مختلف پوشش دادیم.

## Premature Optimization

دونالد کنت در جایی می‌گویید:

> *Premature optimization is the root of all evil. There is **only cost**, and **no benefit**.*

او اعتقاد داشت که بهینه سازی پیش از موعد، ریشه‌ی همه پلیدی‌هاست و بدون اینکه فایده‌ای داشته باشد، مدام بر هزینه‌ها و پیچیدگی ساختار برنامه‌ها می‌افزاید. بهینه سازی سیستم خیلی پیچیده‌تر از دوباره نویسیِ یک *loop* ساده یا تغییر یک *pre-condition* در حلقه‌ی *while* به یک *post-cindition* است.

میراثی که پس از یک بهینه سازی ناقص از شما به‌جا می‌ماند، چیزی جز پیچیدگی‌های بیشتر و کدهای ناخوانا نیست! به همین دلیل است که بهینه سازی پیش از موعد، اغلب به عنوان یک ضد-الگو[۲] نیز شناخته می‌شود.

---

[۱] Test Coverage

[۲] Anti Pattern

به قول دوستی، فقط ۲ قانون برای بهینه سازی یک برنامه وجود دارد:

- اینکار رو انجام نده.
- (برای آن‌هایی که سنیور هستند) لطفا الان انجامش نده!

## Indescriptive Naming

این موضوع خیلی واضح است، اما خب لازم است بگویم که: لطفا و خواهشا برای کلاس‌ها، متدها، مقادیر و اَتریبیوت‌ها نام درستی انتخاب کنید. و صد البته، از نام‌های کوتاه و مخفف دوری کنید.


شما کدها را برای آدم‌ها می‌نویسید، نه برای کامپیوتر (به هر حال کامپیور اصلا نمی‌داند شما چه چیز نوشته اید). کامپیوتر فقط ۰ و ۱ را می‌فهمد. زبان‌های برنامه‌نویسی، برای انسان‌ها ساخته شده‌اند وگرنه همان زبان‌های سطوح پایین به اندازه کافی مناسب و کارا بودند.

این سخن از مارتین فاولر[۱] را بخاطر داشته باشید:

> *Any fool can write code that a computer can understand.*
> *Good programmers write code that humans can understand.*

[۱] Martin Fowler

## Duplication

تکرار در برنامه بد است، پس خواهشا خودت را تکرار نکن (DRY)، و همچنین همیشه آن را ساده و احمقانه نگه دار (KISS). فرض کنید تنبل ترین آدم روی زمین هستید، یکبار کد بنویسید و همه جا آن را دوباره استفاده کنید.

خب همه‌ی این‌ها ۶ مورد از یک برنامه‌نویسی احمقانه یا همان STUPID بودند. ممکن است فکر کنید پس کدهای شما هم STUPID است. اهمیتی ندارد (البته فقط الان). آرامش خود را حفظ کرده، بر خود مسلط باشید و سعی کنید تا اصول SOLID را در برنامه خود پیاده کنید. آنگاه خواهید فهمید که بطور خودکار، کدهای شما دیگر STUPID نخواهند بود.

## اصول طراحی شئ گرا یاObject Oriented Design

وبسایت رابرت سی مارتین با آدرس butunclebob.com یکی از بهترین منابعی است که می توانید در آن اصول طراحی شئ گرا را بیاموزید .چیزی که در این کتاب به آن پرداختیم تنها ۵ اصل معروف از اصول طراحی شئ گرا هستند .در ادامه به ۶ اصل دیگر اشاره خواهم کرد.

۳ اصل زیر به انسجام و پیوستگی پکیج‌های نرم افزاری می‌پردازد و به ما می‌گوید که در دل این پکج‌ها چه چیزی است.

| | | |
|---|---|---|
| **REP** | The Release Reuse Equivalency Principle | The granule of reuse is the granule of release. |
| **CCP** | The Common Closure Principle | Classes that change together are packaged together. |
| **CRP** | The Common Reuse Principle | Classes that are used together are packaged together. |

و ۳ اصل زیر نیز درباره ارتباط و جفتی‌گری پیکیج‌ها صحبت می‌کند و به ما معیارهای برای ارزیابی ساختار پکیج‌ها در برنامه می‌دهد.

| | | |
|---|---|---|
| **ADP** | The Acyclic Dependencies Principle | The dependency graph of packages must have no cycles. |
| **SDP** | The Stable Dependencies Principle | Depend in the direction of stability. |
| **SAP** | The Stable Abstractions Principle | Abstractness increases with stability. |

برای مطالعه‌ی این ۶ اصل به همراه ۵ اصل سالید به لینک زیر مراجعه کنید:

http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

## ویدئوهای آموزشی طراحی شئ گرایی

یکی از منابع بسیار خوب دیگری که می‌توانید از آنها جهت یاد گرفتن مبانی برنامه‌نویسی و طراحی شئ گرایی بهره ببرید، وبسایت ویدئویی آنکل باب در همین موضوع است .مباحثی که رابرت سی مارتین در این وبسایت به آن پرداخته شامل موارد زیر است:

- مبانی برنامه‌نویسی تمیز
- اصولSOLID
- طراحیComponent
- توسعه‌ی آزمون‌محور یاTest-Driven Development
- الگوهای طراحی یاDesign Patterns
- اصول اجایل در برنامه‌نویسی
- برنامه‌نویسی تابعی یاFunctional programming
- هزینه فنی یاTechnical debt
- و مواردی که بطور Case-Study به آن‌ها پرداخته

برای دیدن و استفاده از این ویدئوها به لینک زیر مراجعه کنید:

https://cleancoders.com/videos

## کتاب

کتاب‌های زیادی وجود دارند که می‌توانند به‌عنوان مرجع و الگو مورد استفاده شما قرار بگیرند که تنها به صورت موردی به آن‌ها اشاره می‌کنم:

- The Clean Coder نوشته‌ی Robert Cecil Martin
- The Pragmatic Programmer نوشته‌ی Andy Hunt و Dave Thomas
- Design Patterns: Elements of Reusable Object-Oriented Software نوشته‌ی Erich Gamma, John Vlissides, Ralph Johnson و Richard Helm
- Refactoring نوشته‌ی Kent Beck و Martin Fowler
- Agile Principles, Patterns, and Practices in C# نوشته‌ی Micah Martin و Robert Cecil Martin
- Working Effectively with Legacy Code اثر Michael C. Feathers
- Agile Software Development نوشته‌ی Robert Cecil Martin
- Head First Object-Oriented Analysis and Design نوشته‌ی Brett McLaughlin از سری Head-First
- Bottels of OOP ۹۹ نوشته‌ی Katrina Owen و Sandi Metz
- Practical Object-Oriented Design in Ruby نوشته‌ی Sandi Metz
- Test-Driven Development by Example نوشته‌ی Kent Beck

کتاب‌هایی که در بالا ذکر کردم، به شما کمک می‌کند تا دید بهتری نسبت به طراحی و توسعه برنامه‌نویسی پیدا کنید، شما را راهنمایی می‌کند تا چگونه بهتر و تمیز تر کد بنویسید، به شما می‌آموزد تا برنامه‌ای بنویسید که قابل تست و نگهداری باشد .همه‌ی این‌ها به شما کمک می‌کند تا برنامه‌ای بنویسید که واقعا کار کند.

امید است همه این‌ها نتیجه‌ی فردایی روشن‌تر و پیشرفته‌تر برای شما داشته باشد.

**دوستانی که در اصلاح و بهتر شدن این کتاب لطف داشته‌اند:**

**نسخه ۱,۰,۳**

- امید راد

**نسخه ۱,۰,۴**

- محمد عسکری

**نسخه ۱,۰,۵**

- پانیز علی پور
- سید مهدیار زارع پور

# پنج اصل
# SOLID

نوشته‌ی
رحمت اله (صدرا) عیسی پناه املشی

اگر احساس می کنید در نوشتن یک برنامه گیر کرده اید، این نکته را در نظر بگیرید که آموزش سنتی متداول، ممکن است شما را برای ورود به دنیای توسعه نرم افزار و برنامه نویسی آماده نکرده باشد. کن مازاکا می‌گوید:

*" از آنجا که برخلاف مراکز آکادمیک، شکست و تقلب در دنیای برنامه نویسی خوب تلقی می شود. بنابراین هرگز نمی توانید دنیای برنامه نویسی را به طور کامل درک کنید، بنابراین به جای تمرکز بر روی ریزه کاری ها، بر روی مفاهیم کلی متمرکز می شوید."*

ممکن است با سال‌ها برنامه‌نویسی و تمرین با موارد و سوژه‌های مختلف آشنا شوید، اما بدون دانستن اصول، قوائد و مفاهیم اصلی، هرگز نخواهید توانست آن بینش لازم جهت توسعه‌ی یک نرم افزار مطلوب را بدست آورید.
یکی از آن مفاهیم اصولی و ضروری در طراحی شئ گرا، پنج اصل SOLID است.
این کتاب به همراه توضیحات، نکات و مثال‌های فراوان کمک می‌کند تا با اصول SOLID آشنا شده و آن بینش لازم جهت توسعه‌ی اصولی برنامه‌های نرم افزاری را کسب کنید.

ISBN:978-600-473-162-1
چاپ و انتشار:
انتشارات ناقوس