

ریفکتورینگ

بهسازی کدهای کامپیوتری با مثالهای سی‌شارپ

آرین ابراهیم پور

Refactoring - Improving the design of existing code

(With C# Examples)

<https://0xaryan.github.io/cv>

فهرست محتوا

۱- ریفکتورینگ	۴
۲- ابزار های ریفکتورینگ برای تکنولوژی های مختلف	۵
۳- کد نویسی تمیز	۶
۴- قرض فنی	۷
۵- دلایل قرض فنی	۷
۶- چه موقع باید کد ها را بازسازی کنیم؟	۹
۷- چگونه کد ها را بازسازی کنیم؟	۱۰
۸- بو های کد	۱۱
۹- عظیم الجنه ها	۱۱
۱۰- متدهای بلند	۱۱
۱۱- کلاس های بزرگ	۱۶
۱۲- وسوس اولیه	۱۸
۱۳- پارامتر های زیاد	۲۰
۱۴- توده های داده	۲۱
۱۵- سو استفاده کنندگان شی گرایی	۲۲
۱۶- کلاس های جایگزین با رابط های متفاوت	۲۲
۱۷- میراث به درد نخور	۲۴
۱۸- اعلان های Switch	۲۴
۱۹- فیلد های موقت	۲۸
۲۰- ضد تغییر ها	۲۸
۲۱- تغییرات واگرا	۲۸
۲۲- سلسه ورانت موازی	۲۹
۲۳- جراحی انفجاری	۳۰
۲۴- بی فایده ها	۳۱

۳۱ کامنت ها	۷-۴-۱
۳۲ کد های تکراری	۷-۴-۲
۳۸ کلاس های داده	۷-۴-۳
۴۰ کد مرده	۷-۴-۴
۴۱ کلاس های تبل	۷-۴-۵
۴۱ عمومیت های خطرناک	۷-۴-۶
۴۲ جفت گر ها	۷-۵-۵
۴۲ حسادت به ویژگی	۷-۵-۱
۴۲ صمیمیت نامناسب	۷-۵-۲
۴۴ کتابخانه ناکامل	۷-۵-۳
۴۵ زنجیره پیام	۷-۵-۴
۴۵ مرد میانی	۷-۵-۵
۴۶ منابع	۸

۱- ریفکتورینگ

به فرآیند قابل کنترل بهبدود کد ها، بدون این که به محصول یا نرم افزار قابلیت جدیدی اضافه کنیم، ریفکتورینگ یا بهسازی گفته می شود. این فرآیند کد های ناخوانا، اضافات و پیچیدگی ها را به کد هایی با طراحی ساده تر و قابل فهم تر تبدیل میکند. لذا فرآیند ریفکتورینگ، فرآیندی غیر عملیاتی از ویژگی های یک نرم افزار است.

پس از ریفکتورینگ درست، پیچیدگی کد ها کمتر و خوانایی آن ها بیشتر خواهد شد. همچنین، نگهداری این کد ها آسان تر، قابلیت ساخت مدل ها و ساختار های شی ای گویا تر و در نهایت قابلیت توسعه کد های شما بیشتر خواهد شد. به طور معمول، ریفکتورینگ مجموعه ای از میکرو-ریفکتورینگ های استاندارد را روی کد شما اعمال میکند.

میکرو-ریفکتورینگ ها تغییرات ساده ای هستند که میتوانند روی کد اعمال شوند، به طوری که حتی تغییر مکان دو خط کد یک میکرو-ریفکتورینگ محسوب می شود. به عنوان مثال به تفاوت دو کد زیر توجه کنید:

```
class Person {
    string LastName { get; set; }
    string FirstName { get; set; }
    int Age {get; set; }

    Person(string lastName, string firstName, int age){
        LastName = lastName;
        FirstName = firstName;
        Age = age;
    }
}
```

نمونه کد شماره ۱ - در این کد FirstName همواره بعد از LastName قرار گرفته است

```

class Person {
    string FirstName { get; set; }
    string LastName { get; set; }
    int Age { get; set; }

    Person(string firstName, string lastName, int age){
        FirstName = firstName;
        LastName = lastName;
        Age = age;
    }
}

```

نمونه کد شماره ۲ - شکل تصحیح شده نمونه کد اول

همان طور که در مثال بالا مشاهده می کنید تفاوت کد شماره ۱ و شماره ۲ در این است که خطوط FirstName و LastName در دو کد جا به جا شده اند. برنامه نویس معتقد است که بهتر است نام شخص زود تر از نام خانوادگی او نوشته شود.

۲- ابزار های ریفکتورینگ برای تکنولوژی های مختلف

- برنامه نویسان C، C++ و همچنین Objective-C میتوانند از نرم افزار JetBrains AppCode برای بهسازی کد های خود استفاده کنند.
- ابزار های بازسازی مختلفی را برای زبان های C#, C++, Software Reengineering Toolkit DMS، Java، کوبول، SQL، HTML، Verilog و ... توسعه داده است.
- Eclipse و محیط های توسعه مشتق شده از آن مانند PyDev برای پایتون، و یا همچنین افزونه Photran برای فورترن در این محیط های توسعه دارای ابزار هایی برای بازسازی کد هستند.
- IntelliJ IDEA و محیط های توسعه مشتق شده از آن مانند PyCharm برای پایتون، WebStorm برای جawa، Android Studio برای اندروید و ... ابزار هایی را برای به سازی کد ها توسعه داده اند.
- برنامه نویسان جawa همچنین میتوانند از محیط های توسعه NetBeans و یا JDeveloper استفاده کنند.

○ ابزار های بسیار زیادی برای .Net Framework و زبان های های آن مانند C#، VB.Net و F# وجود دارد. از مهم ترین آنها میتوان به محیط توسعه Visual Studio، محیط توسعه Rider، افزونه CodeRush، افزونه Resharper اشاره کرد. این ابزار ها برای C++ نیز در دسترس Visual Assist هستند.

۳- کد نویسی تمیز

هدف از ریفکتورینگ مبارزه با «فرض های فنی» و تبدیل کد های کثیف به کد های تمیز است. درباره اصطلاح «فرض فنی» در بخش های بعد صحبت میکنیم، اما قبل از آن باید از خود پرسیم که تعریف از «کد تمیز» چیست؟

یک «کد تمیز» باید ویژگی های زیر را داشته باشد.

۳-۱- کد تمیز برای سایر برنامه نویسان خواناست.

منظور ما از خوانایی کد برای سایر برنامه نویسان فقط کد ها و الگوریتم های خیلی پیچیده نیست. نام گذاری بد برای متغیر ها، کلاس های حجیم و متند ها و اعداد جادویی (اعدادی که فقط برای برنامه نویس معنا دارند) باعث پیچیدگی فهم کدها، حتی برای الگوریتم های ساده می شوند.

۳-۲- کد های تمیز، بخش های تکراری ندارند.

در کد های تکراری، با هر تغییر در کدتان مجبور می شوید که بخش های دیگری از کد را به ازای هر نمونه تغییر دهید. این عملیات باعث کاهش سرعت شما در پیشروی پروژه می شود.

۳-۳- کد های تمیز، از لحاظ منطقی کمترین تعداد کلاس های ممکن را دارند.

هر چقدر کد شما کمتر باشد، چیز های کمتری را مجبورید به خاطر بسپارید، نگهداری از کد ساده تر است و همچنین با باگ های کمتری مواجه خواهید شد. فراموش نکنید که کد ها مسئولیت دارند، بنابراین آن ها را کوتاه و ساده نگه دارید.

۳-۴- کد های تمیز همه آزمون های نرم افزار را پاس می کنند.

وقتی کد شما ۹۵ درصد از آزمون ها را پاس کند متوجه می شوید که کدتان تمیز نیست. وقتی این مقدار به صفر درصد نزدیک شود احتمالاً با یک خراب کاری بزرگ مواجه اید.

۳-۵- کد های تمیز قابلیت نگهداری آسان تر و ارزان تری دارند.

۴- قرض فنی

تقریباً تمامی افراد از ابتدای شروع پروژه سعی میکنند که کدهای عالی و بی نقصی بنویسند و احتمالاً هیچ برنامه نویسی وجود ندارد که از قصد بخواهد با نوشتن کدهای کثیف به پروژه ضرر برساند. پس این سوال مطرح می‌شود که دقیقاً از کدام مرحله کدها شروع به کثیف شدن می‌کنند؟

اینجاست که اصطلاح قرض فنی مطرح می‌شود. قرض فنی همانند گرفتن وام از بانک است. با گرفتن وام میتوانید خرید هایتان را سریع‌تر انجام دهید، اما نکته اینجاست که علاوه بر پرداخت مقدار وام، موظفید که سود اضافه‌ای را به بانک پرداخت کنید. این سود به مرور افزایش پیدا میکند و در نهایت زمانی خواهد رسید که پرداخت وام ناممکن می‌شود.

مشابه این اتفاق دقیقاً در دنیای کدنویسی رخ میدهد. شما موقتاً میتوانید از نوشتن تست برای ویژگی‌های جدید نرم افزار خود خودداری کنید، اما این اتفاق به مرور سرعت تان را کنترل خواهد کرد تا نهایتاً با نوشتن تست‌ها قرض خود را پردازید.

۴-۱- دلایل قرض فنی

در این بخش به بررسی اینکه چرا قرض فنی اتفاق میفتند میپردازیم.

۴-۱-۱- فشار کاری

گاهی اوقات فشار‌های کاری موجب میشود که شما ویژگی‌های نرم افزار را قبل از این که کامل شوند منتشر کنید. در این حالات، وصله‌هایی به پروژه زده می‌شود تا نقض‌های آن دیده نشود.

۴-۱-۲- کمبود دانش از عواقب قرض فنی

گاهی اوقات کارفرمای شما دانشی از اینکه در قرض فنی نیاز به پرداخت سود اضافه است، و اینکه قرض فنی به مرور زمان سرعت پیشروی پروژه را کند میکند ندارد. در این حالت، بسیار دشوار است تا وقت تیم را به ریفکتورینگ کد اختصاص دهیم، چرا که مدیریت ارزش آن را نمیبینند.

۴-۱-۳- نبود سازگاری شدید بین کامپوننت‌ها

این اتفاق زمانی رخ میدهد که پروژه به جای اینکه ارتباطی قوی بین کامپوننت های نرم افزار باشد، یک مونولیت بدون اجزاست. در این حالت، هر تغییر در بخشی از پروژه بر بخش های دیگر تاثیر مستقیم میگذارد. کار تیمی در این شرایط به شدت سخت و حتی غیر ممکن است، چرا که نمیتوان به افراد هر تیم، وظیفه ای مستقل منتب کرد.

۴-۱-۴- نبود تست

به تعویق انداختن تست نویسی عملی برای سرعت بخشیدن به کار، اما پر ریسک است. این عمل میتواند مصیبت بار باشد. به عنوان مثال یک رفع باگ ساده میتواند خود باعث باگی شود که به تمامی افراد ثبت شده در پایگاه داده شما ایمیل ارسال کند و یا حتی داده های مشتری را پاک کند.

۴-۱-۵- نبود مستندات

نبود مستندات، معرفی افراد جدید به پروژه را سخت کرده و حتی در صورت خروج افراد کلیدی از پروژه میتواند پروژه را دچار بحران کند.

۴-۱-۶- نبود ارتباط بین اعضای تیم

اگر پایگاه دانش بین اعضای شرکت توزیع نشود، افراد با دانش قدیمی از فرآیند ها و اطلاعات پروژه وظایف خود را انجام میدهند. این اتفاق زمانی رخ میدهد که اعضای جوان تیم توسط مریبان درست آموزش ندیده باشند.

۴-۱-۷- توسعه همزمان بلند مدت در شاخه های مختلف کد

توسعه همزمان در شاخه های مختلف کد (Branch) تاثیر مستقیم در انباشته شدن قرض فنی دارد، که در حین ترکیب (Merge) بیشتر خواهد شد. هر چه تغییرات بیشتری در حالت ایزوله رخ دهد، قرض فنی بیشتر خواهد بود.

۴-۱-۸- به تعویق انداختن فرآیند ریفکتورینگ

نیازمندی های پروژه دائما در حال تغییر است، و در نقطه ای از پروژه مشخص میشود که بخشی از کد قدیمی، غیر قابل استفاده (Obsolete) و سنگین است. در این حالت نیاز است تا کدها مجددا طراحی شود و با نیاز های جدید سازگار گردد.

از طرف دیگر، برنامه نویسان پروژه که هر روز در حال نوشتن کد های جدیدی هستند، از کد های قدیمی نیز استفاده میکنند. بنابراین هر چه ریفکتورینگ به تعویق بیفتند، کد های وابسته بیشتری نیاز به بازنگری دارند.

۴-۱-۹- مانیتور انطباق فکری

زمانی که اعضای پروژه هر طور که صلاح بیینند کد نویسی کنند، بدون اینکه تفکرات با سایر اعضای تیم انطباق داشته باشد
قرص فنی رخ میدهد. (برنامه نویسان معمولاً به شیوه آخرین پروژه قبلی خود کد نویسی میکنند.)

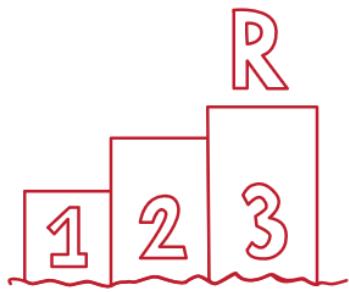
۴-۱-بی کفایتی

این حالت زمانی رخ میدهد که برنامه نویسان کد نویسی تمیز را بلد نباشند.

۵-چه موقع باید کد ها را بازسازی کنیم؟

برای تعیین زمان بازسازی کد ها چند حالت وجود دارد که به بررسی آنها میپردازیم.

۵-۱-قانون سه



وقتی کاری را برای اولین بار انجام میدهید، بدون دغدغه آن را بنویسید. وقتی برای بار دوم کار مشابه به قبل را انجام میدهید، با این تکرار جدید نیز مدارا کرده و کدنویسی را انجام دهید، اما آن را به خاطر بسپارید. ولی اگر برای بار سوم عملی تکراری انجام میدهید، وقت ریفکتور کردن کد های شما فرارسیده است.

۵-۲-هنگام افزودن ویژگی جدید



ریفکتور کردن باعث میشود که از کد های دیگران سر در بیاورید. اگر قرار است که با کد های کشیف دیگران دست و پنجه نرم کنید، ابتدا آن را ریفکتور کنید، چرا که کد های تمیز قابل فهم تر هستند. با این کار، نه تنها کد را برای خود تمیز و قابل فهم کرده اید، بلکه فهم کد برای افراد بعد از شما هم ساده تر خواهد شد. این فرآیند، باعث می شود که راحت تر بتوانید به نرم افزار ویژگی های جدید تر اضافه کنید، چرا که تغییر در کد های تمیز آسان تر است.

۵-۳-هنگام رفع یک باگ



باگ ها در کد همانند باگ ها (سوسک ها) در زندگی واقعی هستند. آنها در تاریک ترین و کشیف ترین بخش های کد زندگی میکنند. کد های خود را تمیز کنید و خواهید دید که خطاهای کم کم آشکار خواهند شد.

۶- چگونه کد ها را بازسازی کنیم؟

فرآیند ریفکتورینگ باید مجموعه ای از تغییرات کوچکی باشد که کد را قدری بهتر میکنند، در حالی که مشکلات نرم افزاری را همچنان در کد باقی خواهد گذاشت. در فرآیند ریفکتورینگ، این نکات را به خاطر داشته باشید:

۶-۱- کد شما باید تمیز تر شود.

اگر کد شما پس از ریفکتورینگ همچنان کثیف باشد، شما تنها چند ساعت از وقت خود را تلف کرده اید. در این حالت باید بگردید و مشکل را پیدا کنید.

این مشکل عموماً موقعي اتفاق میفتد که شما به جای تغییرات کوچک، تغییرات بزرگی در کد میدهید. در این حالت خیلی راحت ممکن است تمرکز خود را از دست بدید، مخصوصاً اگر از نظر زمانی تحت فشار باشید.

همچنین ممکن است با کد بسیار کثیفی رو به رو باشید و هر چقدر هم که سعی کنید آن را بهبود دهید، کد کثیف باقی میماند. در این حالت باید به نوشتن مجدد بخش هایی از کد پردازید، اما قبل از آن باید برای آنها تست نوشته باشید و وقت کافی بگذارید. در غیر این صورت نتیجه همان پاراگراف اول خواهد بود.

۶-۲- ویژگی های جدید نباید در هنگام ریفکتورینگ ایجاد شوند.

فرآیند ریفکتورینگ را با توسعه مستقیم ویژگی های جدید تر کیب نکنید. حداقل سعی کنید که این عملیات در کامیت (Commit) های جداگانه ای باشند.

۶-۳- همه تست ها باید بعد از ریفکتورینگ پاس شوند.

در دو حالت تست ها ممکن است بعد از ریفکتورینگ پاس نشوند.

- شما در حین ریفکتورینگ اشتباهی مرتکب شده اید. راه حل ساده است، برگردید و آن را درست کنید.

- تست های شما بسیار سطح پایین بودند. به عنوان مثال، شما در حال تست کردن متدهای Private کلاس‌ها بودید.
- در این حالت، تست های شما زیر سوال است. باید تست‌ها را ریفکتور کنید و یا تست‌های سطح بالا بنویسید. یکی از بهترین روش‌های جلوگیری از این نوع خطاهای نوشتن تست‌ها به فرم BDD-Style است.

۷- بو های کد

بو‌ها در کد، اصطلاحاً علائمی در سورس کد یک نرم افزار هستند که یک مشکل عمیق را به ما نشان میدهند. به تعریفی دیگر، «بو‌ها در کد، ساختار‌هایی در کد هستند که عبور از مبانی قواعد طراحی را نمایش میدهند و بر کیفیت طراحی آن تاثیر منفی میگذارند».

در اینجا چندین گروه از بو‌های کد، و مثال‌هایی از هر گروه را مشاهده میکنیم.

۷-۱- عظیم الجثه ها

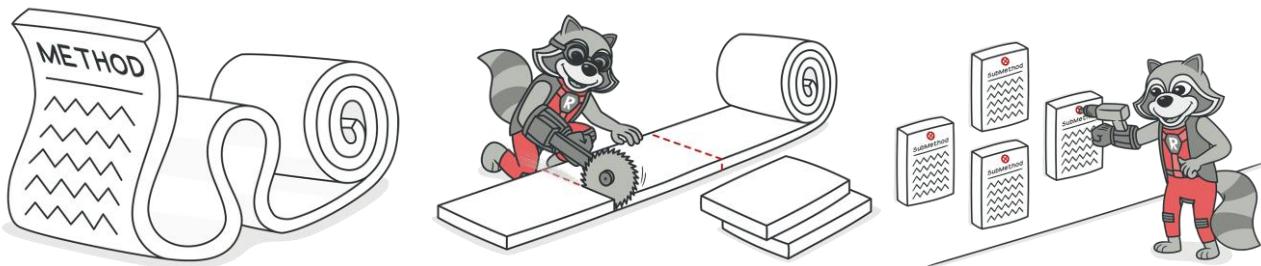


عظیم الجثه‌ها کد‌ها، متدها و کلاس‌هایی هستند که به بخش‌های عظیمی تبدیل شده‌اند و کار با آنها سخت است. این بوها به راحتی قابل تصحیح نیستند، و در عوض به مرور زمان تصحیح میشوند. معمولاً برنامه نویسان به ریفکتور کردن این گروه‌ها رغبت نشان نمیدهند.

۷-۱-۱- متدهای بلند

متدهای بلند، متدهایی هستند که دارای تعداد خطوط زیادی باشند. معمولاً هر متدهایی که بیشتر از ۱۰ خط داشته باشد باید برای شما مشکوک باشد. برای رفع این نوع بو میتوان از روش استخراج متد (Extract Method) استفاده کرد.

از نظر کارایی، در بیشتر مواقع تبدیل مجموعه خطوط به توابع آنقدر ناچیز است که نیازی به نگرانی ندارد.



روش استخراج متدها:

```
void PrintOwing(){
    PrintBanner();

    // Print Details
    Console.WriteLine($"name: {name}");
    Console.WriteLine($"outstanding: {GetOutstanding()}");
}
```

متدها قبل از استخراج

```
void PrintOwing(){
    PrintBanner();
    PrintDetails(outstanding: GetOutstanding());
}

void PrintDetails(double outstanding){
    Console.WriteLine($"name: {name}");
    Console.WriteLine($"outstanding: {outstanding}");
}
```

متدها بعد از استخراج

در این نمونه کد، اطلاعات مربوط به چاپ جزئیات جدید شده و در متدهای دیگری قرار گرفته اند.

روش جایگزینی مقدار موقت با یک Query

```
double CalculateTotal(){
    double basePrice = quantity * itemPrice;

    return (basePrice > 1000) ?
        basePrice * 0.95 :
        basePrice * 0.98;
}
```

متدها قبل از جایگزینی

```
double BasePrice => quantity * itemPrice;

double CalculateTotal(){
    return (BasePrice > 1000) ?
        BasePrice * 0.95 :
        BasePrice * 0.98;
}
```

متدها بعد از جایگزینی

در این روش، مقدار موقت basePrice از متدهای `CalculateTotal()` جدا شده و خارج از تابع به شکل یک Query در آمده است.

روش پارامتر با Object

Customer	Customer
amountInvoicedIn (start : Date, end : Date)	amountInvoicedIn (date : DateRange)
amountReceivedIn (start : Date, end : Date)	amountReceivedIn (date : DateRange)
amountOverdueIn (start : Date, end : Date)	amountOverdueIn (date : DateRange)

در این روش به جای دریافت پارامترهای جدا، آنها را به صورت یک شی دریافت میکنم. به عنوان مثال به جای دریافت دو تاریخ یکی برای شروع و دیگری برای پایان، از یک ساختمان تحت عنوان DateRange استفاده میکنیم، تا چکها و منطق مربوط به این دو تاریخ در همان ساختمان انجام شود و کدهای توابعی که از این منطق استفاده میکنند طولانی نشوند.

روش حفظ کل Object

```
void Method(){
    int low = daysRange.GetLow();
    int high = daysRange.GetHigh();

    bool withinPlan = plan.WithinRange(low, high);
}
```

استخراج ویژگی های شی و پاس دادن آن به یک تابع

```
void Method(){
    bool withinPlan = plan.WithingRange(daysRange);
}
```

حفظ کل Object

در این روش به جای تجزیه مقادیر یک کلاس و پاس دادن آنها به تابعی دیگر، خود آن کلاس را پاس میدهیم.

روش تجزیه عبارات شرطی:

```
if(data < SummerStart || data > SummerEnd)
{
    charge = quantity * winterRate + winterServiceCharge;
}
else
{
    charge = quantity * summerRate;
}
```

عبارت شرطی قبل از تجزیه

```
charge = (NotSummer(date)) ?
    WinterCharge(quantity) :
    SummerCharge(quantity);
```

عبارت شرطی تجزیه شده

در این روش، به جای چک کردن صریح تاریخ، از متدهای NotSummer کمک گرفتیم، و محاسبات پلن‌های مختلف را درون توابع مخصوص به خودشان انجام دادیم.

روش جایگزینی MethodObject با Method

```
public class Order {
    // ...
    public double Price() {
        double primaryBasePrice;
        double secondaryBasePrice;
        double tertiaryBasePrice;
        // Long computation...
        // ...
    }
}
```

انجام محاسبات سنگین در خود کلاس Order

```
public class Order {
    // ...
    public double Price() {
        return new PriceCalculator(this).Compute();
    }
}

public struct PriceCalculator {
    private double primaryBasePrice;
    private double secondaryBasePrice;
    private double tertiaryBasePrice;

    public PriceCalculator(Order order) { ... }

    public double Compute() {
        // Long computation
        // ...
    }
}
```

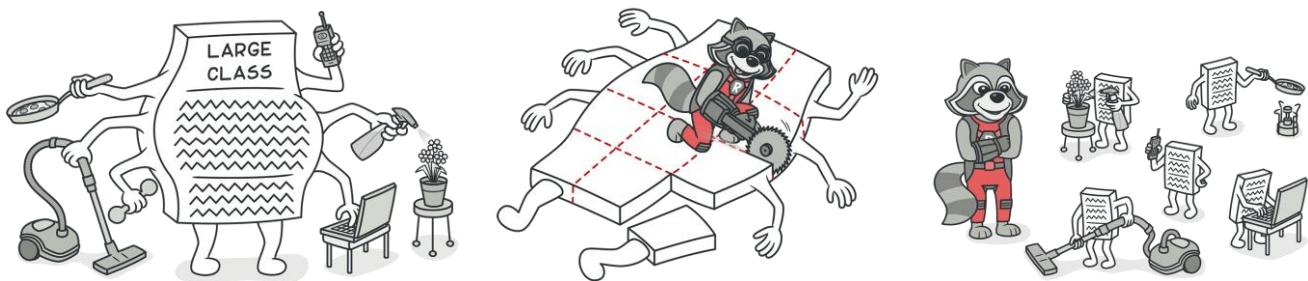
انتقال محاسبات به کلاس PriceCalculator

همان طور که در بالا مشاهده می کنید، محاسبات مربوط به مقدار Price از خود کلاس Order، به کلاس (و یا struct) PriceCalculator منتقل شده است.

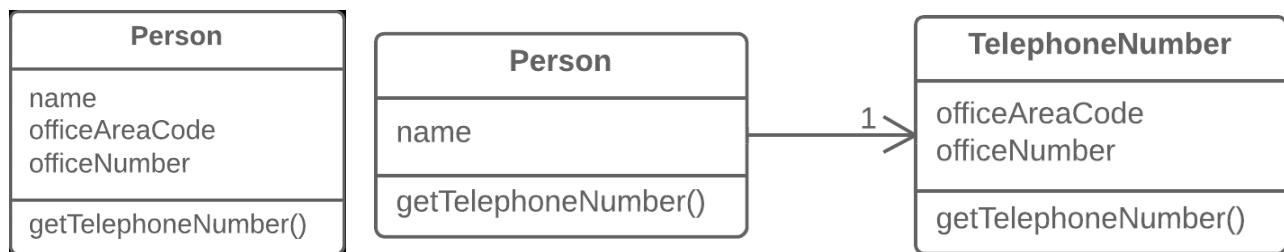
۲-۱-۷ - کلاس های بزرگ

کلاس های بزرگ کلاس هایی هستند که فیلد ها، ویژگی ها، متدها و در کل خطوط کد زیادی داشته باشند. کلاس ها عموما با اندازه های کوچک شروع می شوند اما به مرور زمان بزرگ و بزرگ تر می شوند.

همانند متدهای بلند، برنامه نویسان عموما در فکر شان این طور تصور میکنند که قرار دادن ویژگی های جدید در یک کلاسی که قبل از آن نوشته شده است، هزینه کمتری نسبت به ساخت کلاس جدید برای ویژگی مورد نظر دارد.

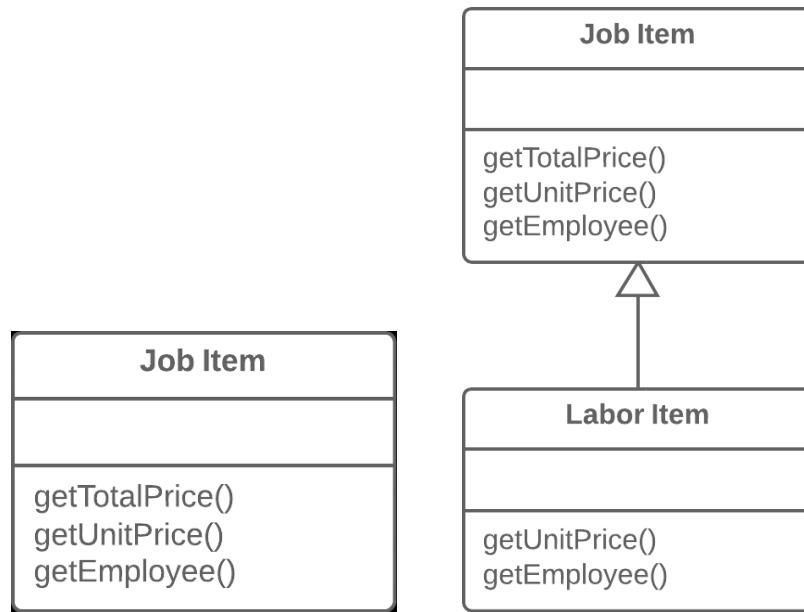


روش تجزیه کلاس:



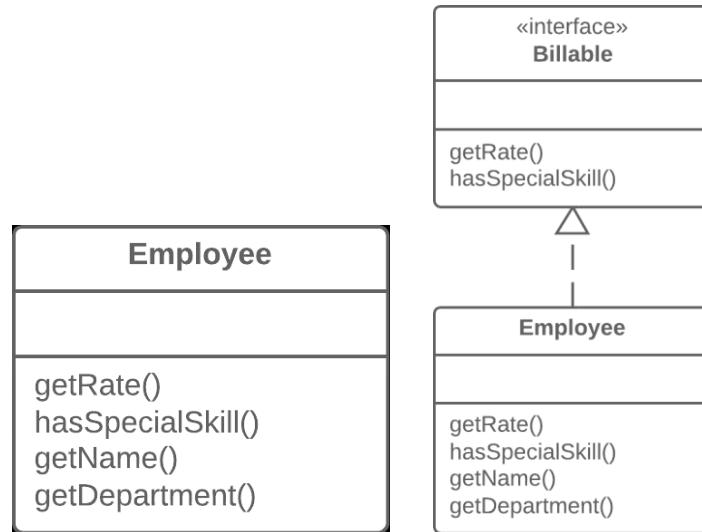
همان طور که ملاحظه میکنید، در بالا فیلد های مربوط به شماره تلفن به ساختمان جدیدی تحت عنوان TelephoneNumber منتقل شده است.

روش تجزیه زیر کلاس:



در این روش، ویژگی های یک کلاس جداسده، و به پس از ساخت یک کلاس پدر، به آن کلاس منتقل می شود.

روش استخراج رابط (Interface):



در این روش، برخی از ویژگی های کلاس استخراج شده، و پس از ساخت رابط مناسب به آن منتقل می شود.

۳-۱-۷- سوساس اولیه

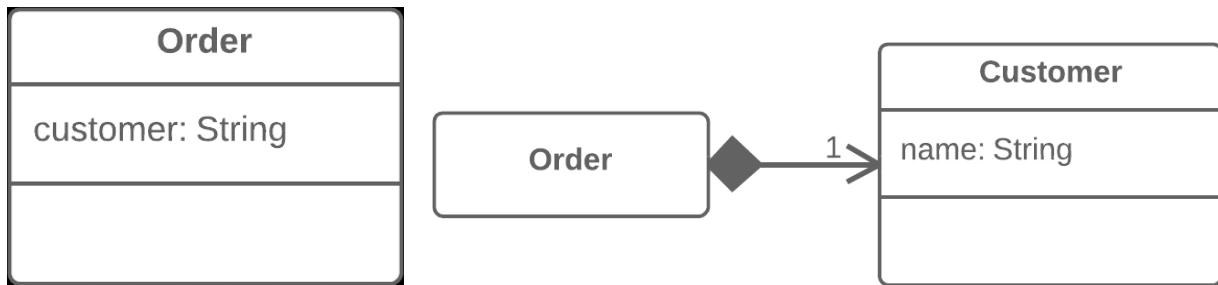
به استفاده از متغیر های Primitive به جای تعریف یک ساختمان یا کلاس برای کارهای کوچک سوساس اولیه گفته می شود. به عنوان مثال در متغیر های نگه دارنده قیمت، بازه ها، رشته های خاص، شماره تلفن و ...

برنامه نویسانی که تازه باشی گرایی آشنا شده اند عموما از تعریف کلاس ها یا ساختمان های جدید اجتناب میکنند. به جای آن، ثابت هایی مثل `USER_ADMIN_ROLE = 1` تعریف میکنند تا کاربران با دسترسی مدیر را مشخص کنند، و یا از رشته های ثابت در آرایه های داده، به عنوان نام فیلد استفاده میکنند.

در پارامتر توابع میتوان از روش هایی مانند حفظ کل `Object` و یا استفاده از `Object` در پارامتر توابع که در قبل توضیح داده شد استفاده کرد.

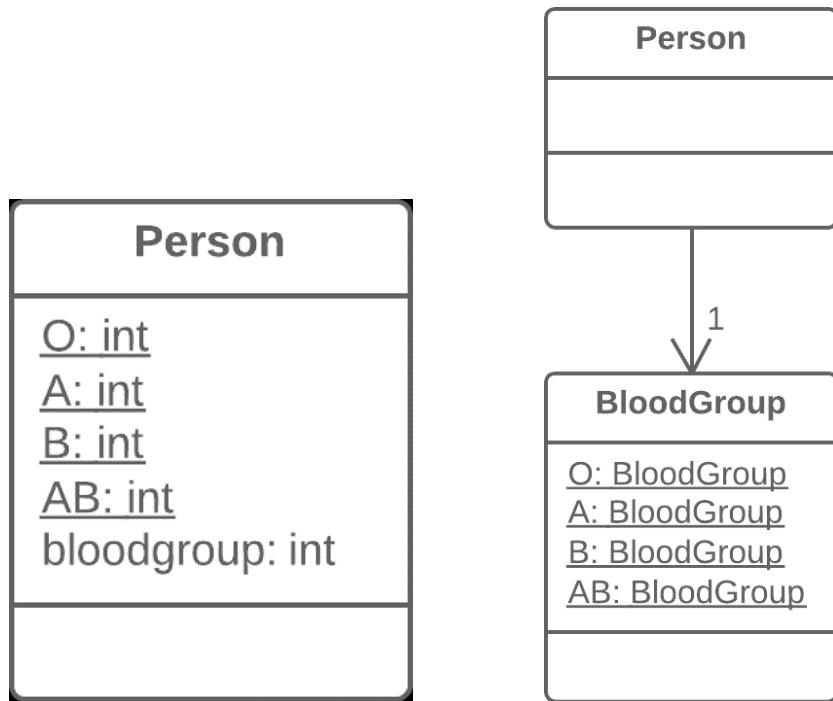


روش جایگزینی مقدار داده با یک `Object`



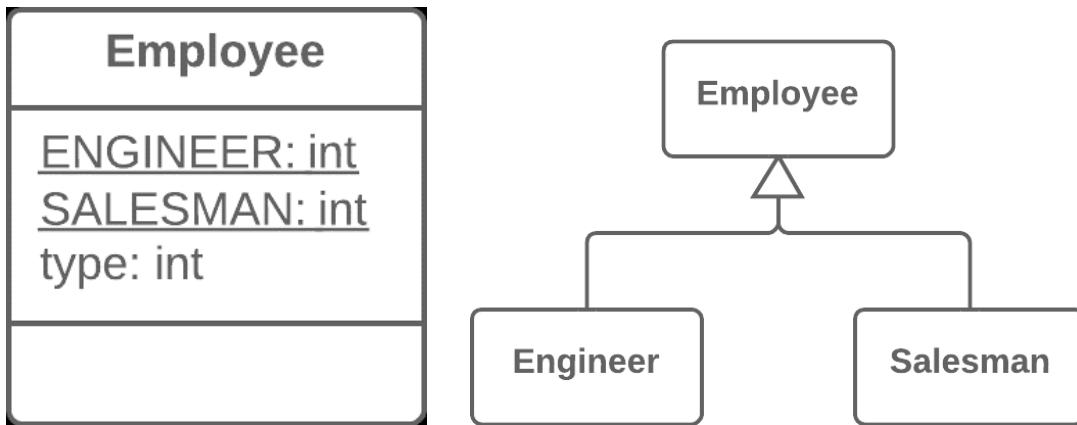
در این روش، یک مقدار داده ای با یک شی جایگزین می شود. همان طور که مشاهده میکنید نام کاربر با نوع `String` با کلاسی با نام `Customer` که در آن فیلد نام با نوع `String` دارد جایگزین شده است.

روش جایگزینی *Type Code* با کلاس ها و یا شمارنده ها:



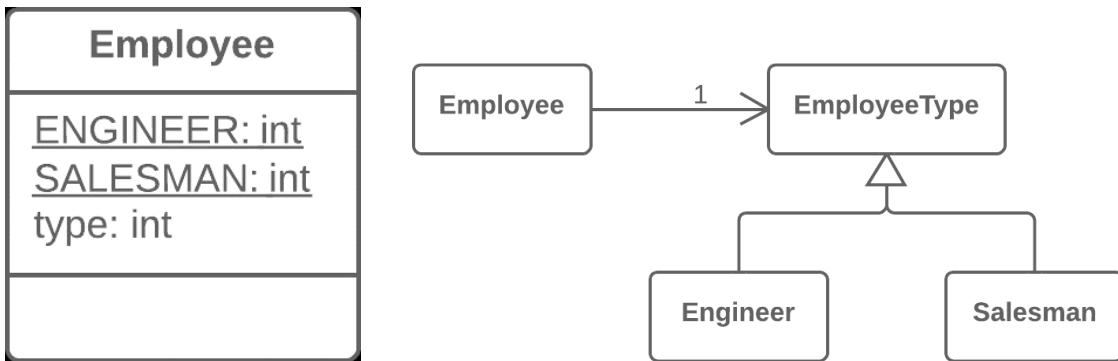
در مثال بالا، فیلد های مربوط به گروه خونی شخص به جای اعلان در خود کلاس، به یک *Enum* منتقل شده اند که کلاس *Person* به آن اشاره میکند.

روش جایگزینی *Type Code* با زیر کلاس ها:



در مثال بالا، نوع یک کارمند به جای اینکه وابسته به یک فیلد نوع و چند متغیر عددی باشد، با استفاده از زیر کلاس ها توصیف میشود.

روش جایگزینی State/Strategy & Type Code



روش جایگزینی آرایه با Object

```

string[] row = new string[2];
row[0] = "Liverpool";
row[1] = 15;
  
```

تعریف کردن نمایش نتیجه یک تیم فوتبال با استفاده از آرایه

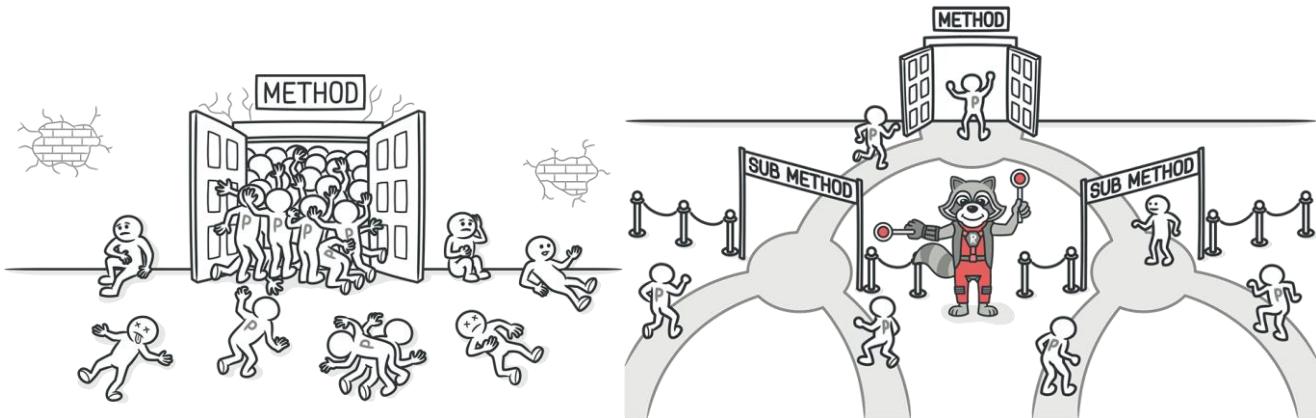
```

var row = new Performance
{
    Name = "Liverpool",
    Wins = 15
}
  
```

استفاده از یک ساختمان برای تعریف نتیجه یک تیم فوتبال

۴-۱-۷- پارامتر های زیاد

هر گاه متدهای شما بیشتر از سه تا چهار پارامتر داشته باشند، کد شما دارای بوی پارامتر های زیاد است. متدها با پارامتر های کم خوانایی بیشتر و کد های کمتری دارند. ریفکتور کردن این متدها میتواند کدهای تکراری که متوجه آنها نشده بودید را آشکار کند. در صورتی که حذف پارامتر های توابع باعث میشود که بین کلاس ها وابستگی های ناخواسته ایجاد شود، از حذف پارامتر ها خودداری کنید. در این بونیز میتوانید از روش های حفظ کل Object و استفاده از شی در پارامتر که قبلاً توضیح داده شد استفاده کنید.



روش جایگزینی پارامتر با صدا زدن تابع:

```
int basePrice = quantity * itemPrice;
double seasonDiscount = this.GetSeasonalDiscount();
double fees = this.GetFees();
double finalPrice = DiscountedPrice(basePrice, seasonDiscount, fees);
```

تابعی با پارامتر های زیاد

```
int basePrice = quantity * itemPrice;
double finalPrice = DiscounterPrice(basePrice);
```

ساده سازی لیست پارامتر ها با استفاده از صدا زدن درونی تابع

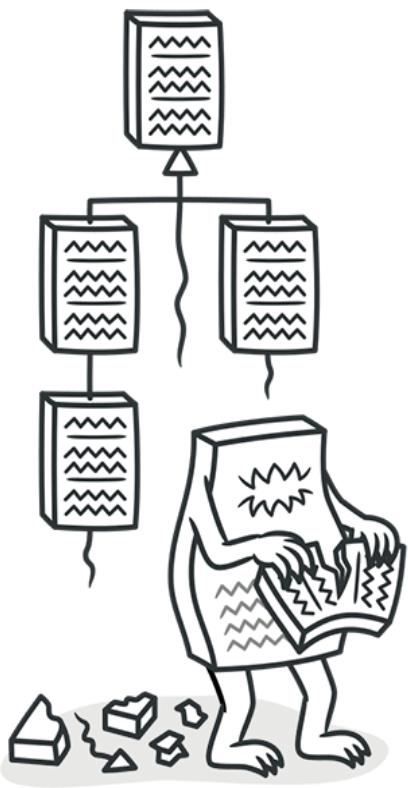
در مثال بالا، دو پارامتر Fees و SeasonalDiscount را حذف کردیم، چرا که دو متغیر پاس داده شده به تابع DiscountPrice قابل دسترسی هستند. کافی است که در بدنه تابع دو تابع GetSeasonalDiscount() و GetFees() صدا زده شوند.

۱-۵-۵- توده داده ها

در این بو، بخش هایی از کد دارای گروهی از متغیر های تکراری است. (به عنوان مثال، متغیر های مربوط به ارتباط با پایگاه داده). این گروه داده ها باید به کلاس ها و ساختمان های مربوط به خود تبدیل شوند. در این نوع بو میتوانید از روش های استخراج کلاس، استفاده از *Object* در پارامتر و حفظ کل استفاده کنید.

۲-۷- سو استفاده کنندگان از شی گرایی

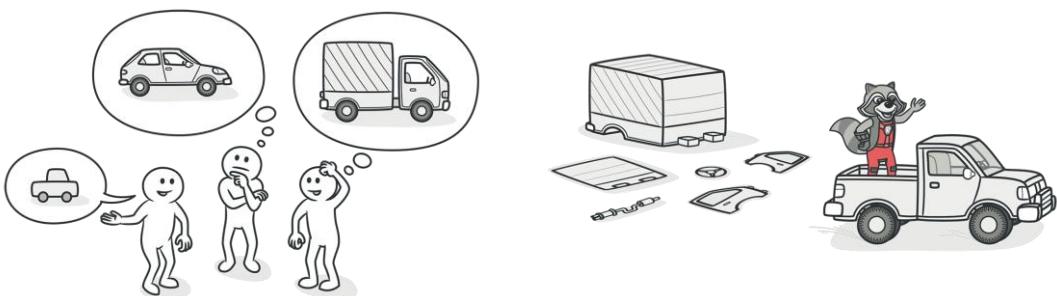
بو های موجود در این گروه، همگی استفاده های نادرست و یا ناکامل از قواعد و قوانین برنامه نویسی شی گرا هستند.



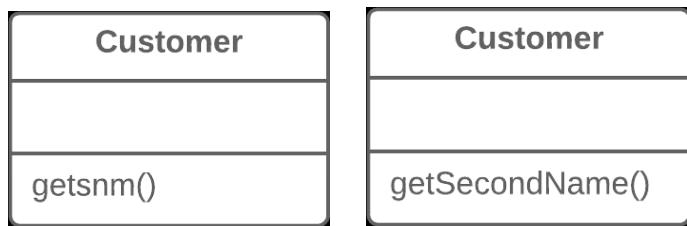
۱-۲-۷- کلاس های جایگزین با رابط های متفاوت

در این نوع بو، کلاس هایی وجود دارند که عملیات یکسانی را انجام میدهند، اما نام متدهای آن ها با یکدیگر فرق میکند. در اینجا شما میتوانید که یکی از این کلاس ها را حذف کنید.

میتوانید از روش تغییر نام متدها استفاده کنید تا نام همه متدها در کلاس های جایگزین یکسان شود. متدها را بین کلاس ها جا به جا کنید، پارامتری به متدبیفزايد و یا متدها را پارامتریک کنید تا ا مضای توابع (لیست و نوع پارامتر هایشان) یکسان سازی شود. همچنین میتوانید کلاس پدری استخراج کنید و کلاس های فعلی زیر مجموعه آن شوند.

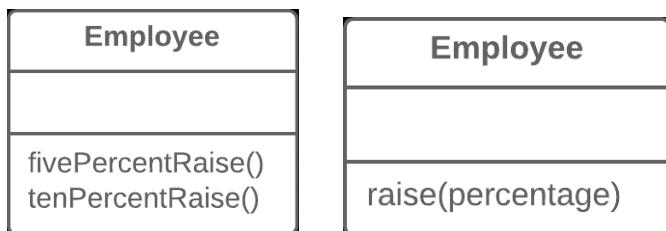


روش تغییر نام متده:



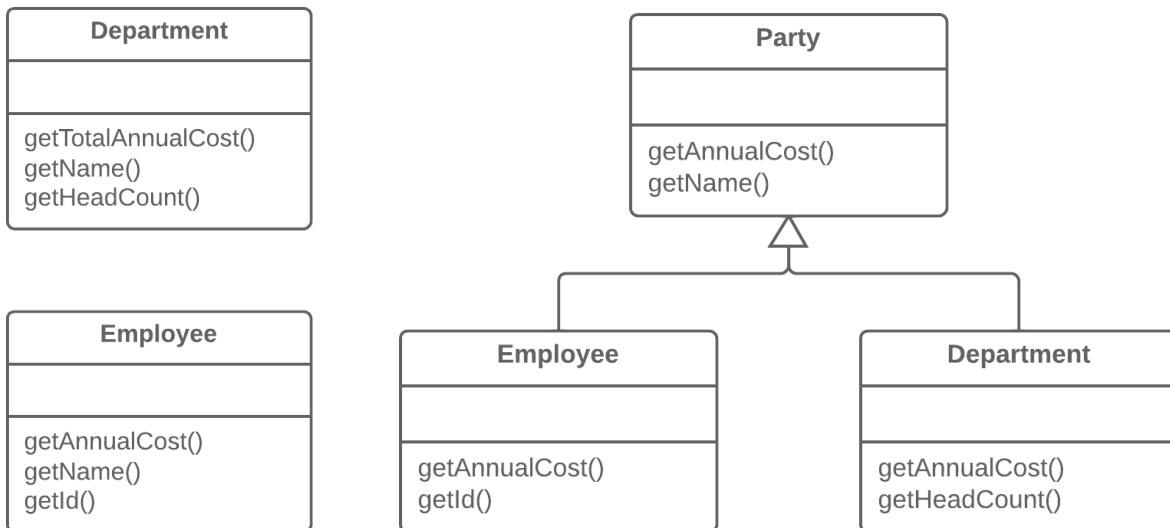
در این روش، نام تابع از GetSNM() به نام خواناتر GetSecondName() تغییر یافته است.

روش پارامتریک کردن متده:



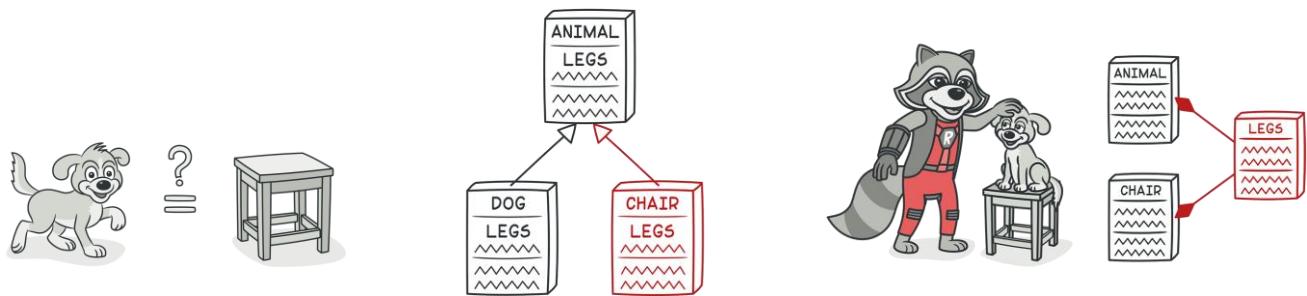
در این روش، به جای اینکه برای هر حالت تابعی تعریف کنیم، یک تابع کلی با پارامتری که به آن شکل دهد تعریف می‌کنیم.

روش استخراج کلاس پدر:

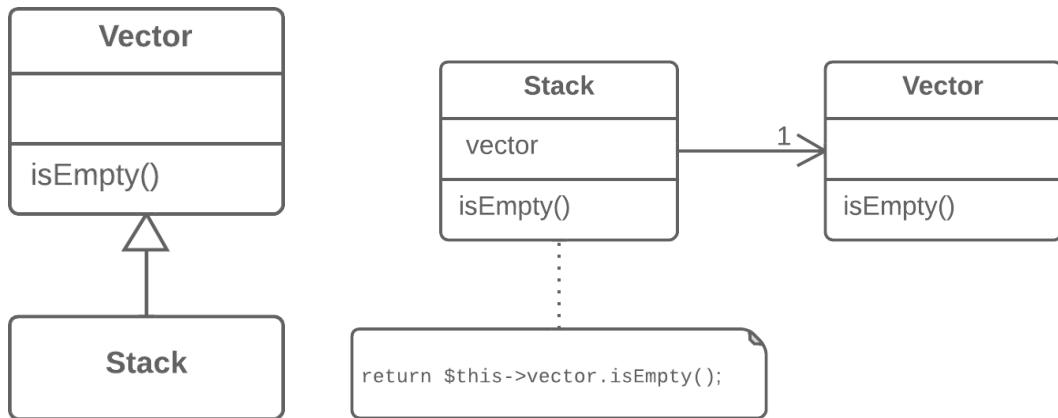


۲-۲-۷- میراث به درد نخور

اگر یک زیر کلاس تنها بخشی از ویژگی ها و متدهای والد خودش را به ارث میرد، این سلسله مراتب وراثتی ناکارآمد است. اگر وراثت در اینجا معنی ندارد، از روش جایگزینی وراثت با موکل استفاده کنید. اگر وراثت مناسب است، یک کلاس پدر استخراج کنید.



روش جایگزینی وراثت با موکل:



همان طور که ملاحظه میکنید، کلاس Stack به جای اینکه خود نوعی وکتور باشد (حالت وراثتی)، به کلاسی تبدیل شده است که درون خود یک نمونه وکتور دارد و روی آن عملیاتی انجام میدهد (حالت وکالت و یا Delegation).

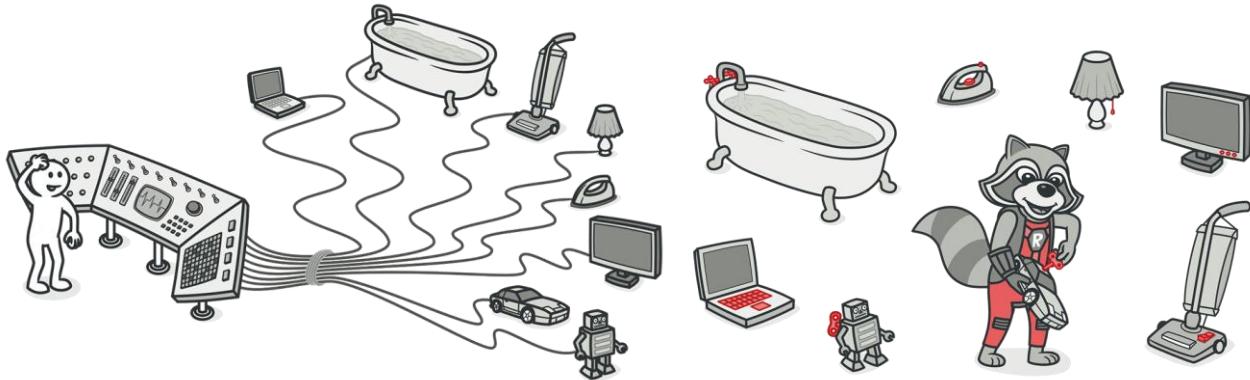
۳-۲-۷- اعلان های Switch

در این بو، شما یک اعلان Switch پیچیده و یا مجموعه ای از اعلان های if متوالی دارید.

برای قرار دادن سویچ ها در کلاس درست میتوانید از روش های استخراج متده و سپس انتقال متده استفاده کنید.

اگر این سویچ ها وابسته به Type Code ها (Enumerations) هستند، از روش جایگزینی Type Code ها با زیر کلاس و یا جایگزینی State/Strategy ها با Type Code که پیشتر توضیح داده شد استفاده کنید.

بعد از تعیین وراثت نیز میتوان از روش جایگزینی شروط با چند ریختی استفاده کرد.



روش جایگزینی پارامتر با ویژگی ها:

```
void SetValue(string propName, int value){
    if(propName == "height") {
        height = value; return;
    }
    if(propName == "width") {
        width = value; return;
    }
    Assert.Fail();
}
```

تعیین مقدار ویژگی ها با استفاده از علان های If

```
public int Height { get; set; }
public int Width { get; set; }
```

اعلان ویژگی ها با استفاده از Properties

```
if (customer == null) {
    plan = BillingPlan.BasicPlan();
}
else {
    plan = customer.Plan;
}
```

چک کردن Null و انتساب مقادیر از پیش تعیین شده

```
public sealed class NullCustomer : Customer {
    public override bool IsNull {
        get { return true; }
    }

    public override Plan GetPlan {
        get { return new NullPlan(); }
    }
    // Some other null functionality
}

// Replace null values with null-objects
customer = order.Customer ?? new NullCustomer();

// Use null-objects as if it's a normal class
plan = customer.GetPlan();
```

تعریف یک کلاس Null

روش جایگزینی شروط با چند ریختی:

```
public class Bird : Animal
{
    BirdType Type { get; set; }
    // ...
    public double GetSpeed() {
        switch (Type) {
            case BirdType.European: return GetBaseSpeed();
            case BirdType.African:
                return GetBaseSpeed() - GetLoadFactor()
                    * numberOfCoconuts;
            case BirdType.NorwegianBlue:
                return IsNailed ? 0 : GetBaseSpeed(voltage);
            default:
                throw new Exception("should be unreachable");
        }
    }
}
```

استفاده از Switch برای تعیین سرعت پرندۀ ها

```
public abstract class Bird : Animal {
    // ...
    public abstract double GetSpeed();
}

class NorwegianBlue : Bird {
    public override double GetSpeed() {
        return IsNailed ? 0 : GetBaseSpeed(voltage);
    }
}

// Somewhere in client code
speed = bird.GetSpeed();
```

تعریف یک کلاس چکیده پرندۀ و ساخت زیر کلاس برای هر نوع پرندۀ

۴-۲-۴- فیلد های موقت

فیلد های موقت، فیلد هایی هستند که تنها تحت شرایط خاصی مقدار میگیرند و خارج از این شرایط، مقدار و کارایی ندارند. به عنوان مثال الگوریتمی در کلاس وجود دارد که نیاز به متغیر های زیادی دارد. مادامی که از این الگوریتم استفاده نشود، این متغیر ها بیکار خواهند بود.

میتوانید این فیلد ها را با روش استخراج کلاس وارد کلاس جدیدی کنید. در واقع شما در حال ساخت یک `MethodObject` هستید.

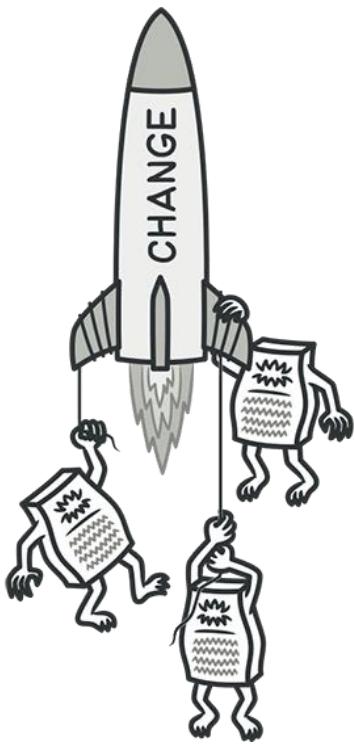
همچنین میتوانید از روش جایگزینی متدها با `MethodObject` استفاده کنید، و یا با استفاده از روش `NullObject` مقدار فیلد های موقت را برای وجود یا عدم وجودشان بررسی کنید.



۳-۳- ضد تغییر ها

ضد تغییر به این معناست که در صورت تغییر چیزی در یک بخش از کد، مجبور به تغییر کد در مکان های دیگری نیز خواهد بود. در این حالت، توسعه نرم افزار بسیار پیچیده و پر هزینه خواهد بود.

۳-۱- تغییرات واگرا



در این حالت، با یک تغییر در یک کلاس مجبور به تغییرات زیادی در همان کلاس خواهد بود. به عنوان مثال، هنگامی که هر بار یک نوع جدید از محصول ایجاد میکنید، مجبور میشوید که متدهای مربوط به پیدا کردن، نمایش دادن و سفارش محصولات را تغییر دهید.

برای رفع مشکل، میتوانید رفتار کلاس را با روش استخراج کلاس جداسازی کنید. همچنین اگر کلاس ها رفتار یکسانی دارند، کلاس ها را با وراثت ترکیب کنید.



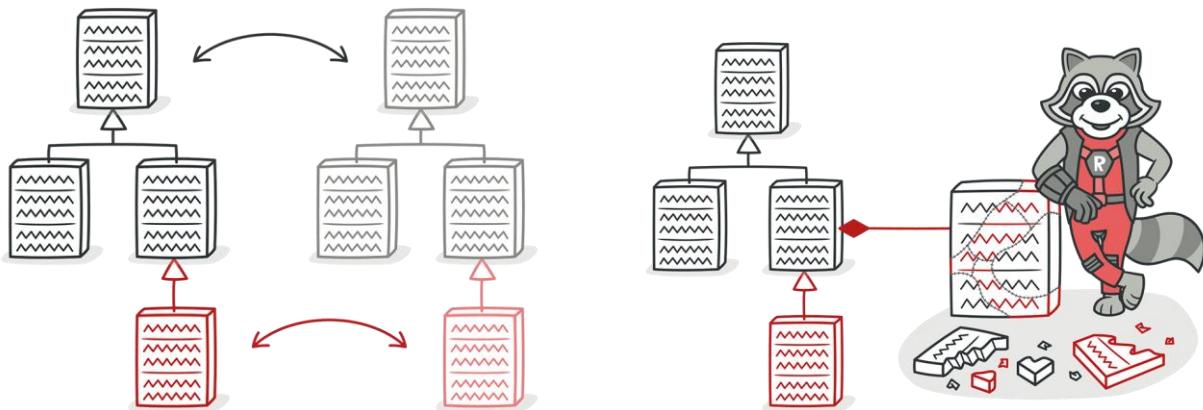
۴-۳-۷- سلسله وراثت موازی

در این نوع بو، هرگاه برای کلاسی یک زیرکلاس بسازید، متوجه میشوید که نیاز است تا برای یک کلاس دیگر نیز زیرکلاس ایجاد کنید.

تکرار این وراثت را میتوانید با این دو گام حذف کنید:

۱- کاری کنید تا نمونه های یک سلسله مراتب به نمونه های یک سلسله مراتب دیگر اشاره کند.

۲- سلسله مراتب کلاس اشاره کننده را با جایی متند ها و فیلد ها حذف کنید.



۳-۳-۷- جراحی انفجاری

در این حالت، با تغییر کوچکی در یک کلاس، مجبور می شوید تا تغییرات کوچکی را در کلاس های متفاوت ایجاد کنید، چرا که یک مسئولیت واحد میان تعداد زیادی از کلاس ها تقسیم شده است. این اتفاق میتواند از زیاده روی در تغییرات واگرا رخ دهد.

برای بازگرداندن رفتار کلاس های موجود به یک کلاس یگانه از روش های جا به جایی متند و جا به جایی فیلک استفاده کنید.

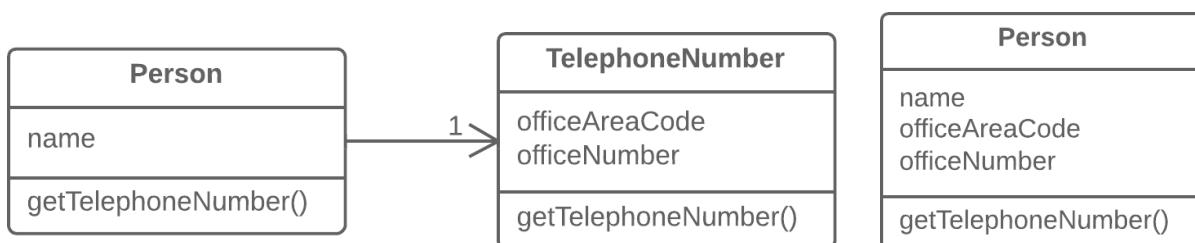
اگر هیچ کلاس مناسبی برای این کار وجود نداشته باشد، یک کلاس جدید ایجاد کنید.

اگر جا به جایی کد ها به کلاس فعلی، کلاس اصلی را تقریبا خالی میکند، سعی کنید با استفاده از کلاس های *Inline* از کلاس های اضافی خلاص شوید.



روش کلاس های *Inline*

این کلاس ها تقریبا کاری انجام نمیدهند و یا مسئولیتی برای هیچ کاری ندارند. همچنین هیچ مسئولیتی نیز در آینده برای آنها برنامه ریزی نشده است.



۴-۴- بی فایده ها



بی فایده ها چیزهایی بی هدف و غیرضروری هستند که نبودشان کد را خوانا تر، تمیز تر، کارا تر و ساده تر میکنند.

۴-۱- کامنت ها

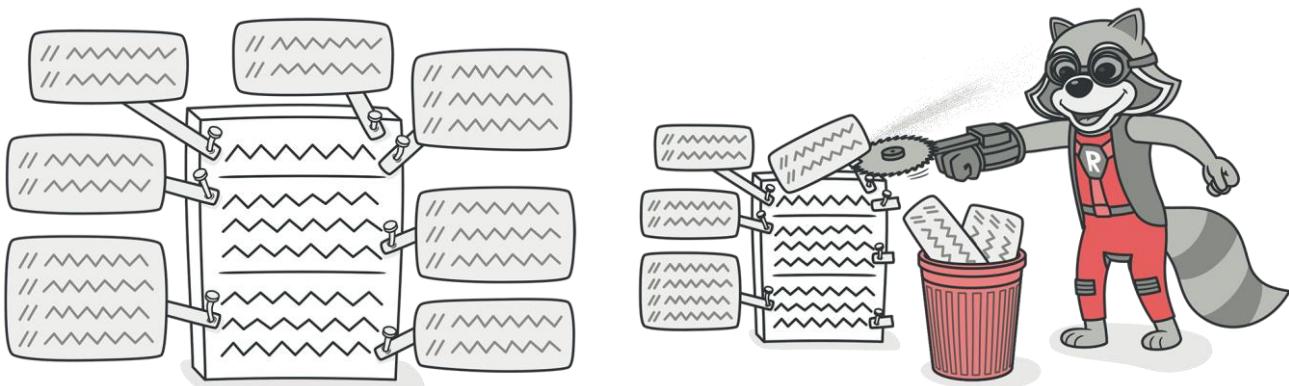
در این بو، متدها با کامنت های توضیحی زیادی پر شده اند. فراموش نکنید که بهترین کامنت، نامی خوب برای متدها یا کلاس است.

اگر کامنت قصد دارد تا یک عبارت پیچیده را توضیح دهد، این عبارات باید به زیر عبارت های قابل فهمی تبدیل شوند. برای این تبدیل میتوانید از روش استخراج متغیر استفاده کنید.

اگر کامنت قرار است بخشی از کد را توضیح دهد، این کد میتواند با روش استخراج متدهای یک متدهای جدایانه تبدیل شود.

اگر متدهای قبل استخراج شده اما همچنان نیاز به توضیح دارد، احتمالاً نام درستی برای آن انتخاب نشده است. سعی کنید با روش تغییرنام متدها، نامی گویا تر برای آن انتخاب کنید.

اگر نیاز است تا در کدتان ادعاهایی (Assert) را درباره یک حالت را بررسی کنید و این بررسی های برای کارکردن سیستم الزامی هستند، از روش استفاده از ادعاهای (Assertion) استفاده کنید.



روش استخراج متغیر:

```
// Too much comment explaining the code
if((Platform.ToUpper().Contains("MAC")) &&
(Browser.ToUpper().Contains("IE")) &&
WasInitialized() &&
resize > 0)
{
    // do something
}
```

حالت بدون متغیر و توضیحات کد

```
bool IsMacOs      => Platform.ToUpper().Contains("MAC");
bool IsIE          => Browser.ToUpper().Contains("IE");
bool WasResized   => resize > 0;

if(IsMacOS && IsIE && WasInitialized() && WasResized){
    // do something
}
```

جایگزینی کامنت ها با متغیر

روش استفاده از ادعاهای:

```
double GetExpenseLimit() {
    // should have either expense limit or a primary project
    return (expenseLimit != (double)ExpenseType.NullExpense) ?
        expenseLimit :
        primaryProject.GetMemberExpenseLimit();
}
```

استفاده از کامنت به جای ادعاهای

```

double GetExpenseLimit() {
    Assert.IsTrue(
        expenseLimit != (double)ExpenseType.NullExpense || 
        primaryProject != null;
    );

    return (expenseLimit != (double)ExpenseType.NullExpense) ? 
        expenseLimit : 
        primaryProject.GetMemberExpenseLimit();
}

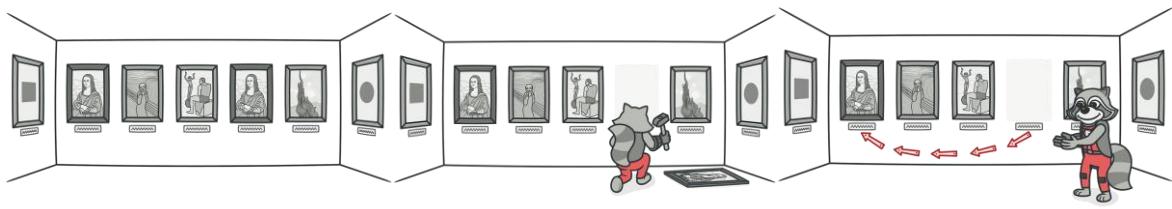
```

استفاده از Assertion

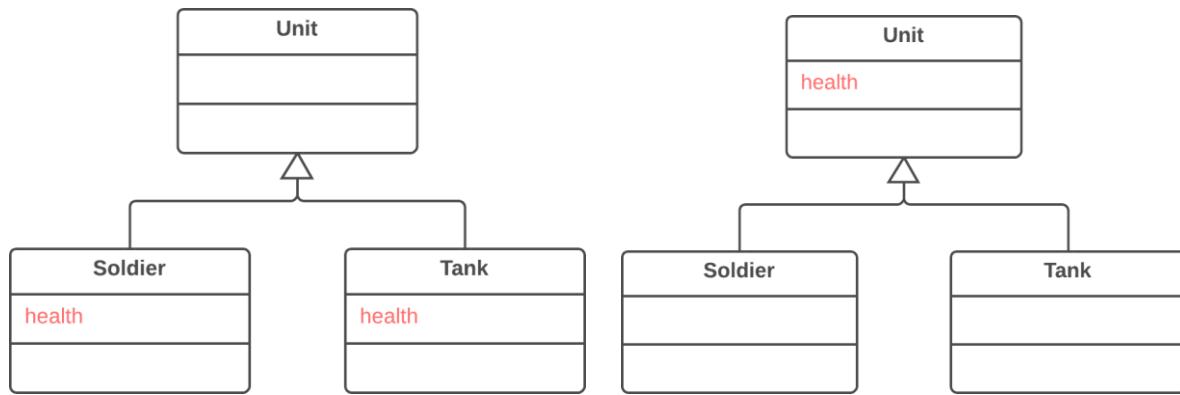
۴-۲- کد های تکراری

در این حالت، قسمت های زیادی از کد با یکدیگر یکسان هستند. با توجه به شرایط زیر میتوانید تصمیم بگیرید که چه کاری انجام دهید.

- وقتی که دو بخش تکراری، دو زیر کلاس در یک سطح باشند.
- در حالت کلی، ابتدا بخش تکراری را به شکل متدهای استخراج کنید و پس از آن از روش بالا کشیدن فیلد استفاده کنید.
- اگر کد تکراری داخل کانسٹراکتور کلاس وجود داشته باشد، از روش بالا کشیدن بدنه کانسٹراکتور استفاده کنید.
- اگر کد شبیه است، اما کاملاً تکراری نیست از روش شکل دهی به متدهای قالب استفاده کنید.
- اگر کد ها عملی یکسان را انجام میدهد اما الگوریتم های آن متفاوت است، الگوریتم بهینه تر را انتخاب کنید و سپس از روش الگوریتم جایگزین استفاده کنید.
- وقتی کد ها در دو کلاس متفاوت باشند.
- سعی کنید که زیر کلاس یا کلاس والد استخراج کنید.
- وقتی که عبارات شرطی بسیار زیادی در حال انجام اعمال تکراری باشند.
- از روش استخراج متدهای همچنین روش یکی کردن عبارات شرطی استفاده کنید.
- وقتی که بعضی از کد ها در تمامی شاخه های یک عبارت شرطی دیده شوند.
- از روش یکی کردن بخش های تکراری عبارات شرطی استفاده کنید.



روش بالا کشیدن فیلدها:



روش بالا کشیدن کانسٹرکتور:

```

public class Manager : Employee {
    public Manager(string name, string id, int grade)
    {
        this.Name = name;
        this.Id = id;
        this.Grade = grade;
    }
    // ...
}
  
```

کد های تکراری در سازنده

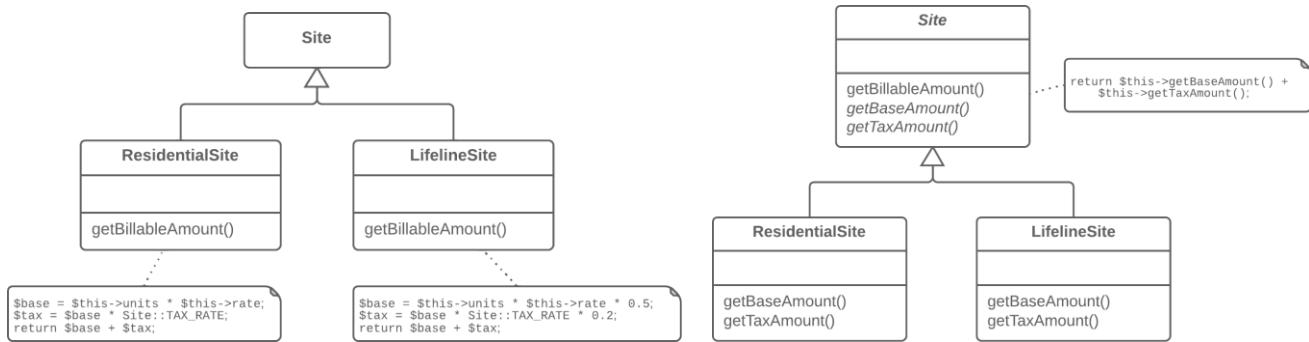
```

public class Manager : Employee {
    public Manager(string name, string id, int grade) : base(name, id)
    {
        this.Name = name;
    }
    // ...
}

```

بالا کشیدن کد های تکراری با استفاده از کلید واژه `base`

روش شکل دهی به متدهای قابل:



این روش زمانی استفاده میشود که کد های شبیه اما نه کاملاً تکراری وجود داشته باشد. همان طور که در شکل دیده میشود تابع `GetBillableAmount()` در دو کلاس `ResidentialSite` و `LifelineSite`، هر دو در حال بازگرداندن حاصل جمع مقدار `Base` و `Tax` هستند. برای ریفکتور، تابع `GetBillableAmount()` را به والد منتقل میکنیم و کد بدنه آن را به بازگرداندن حاصل جمع دو مقدار `Base` و `Tax` تغییر میدهیم. در این حالت، تنها کافی ست برای کلاس های فرزند این دو مقدار تعیین شود.

روش الگوریتم جایگزین:

```
string FoundPerson(string[] people) {
    for(int i = 0; i < people.Length; i++) {
        if(people[i].Equals("Don")){
            return "Don";
        }
        if(people[i].Equals("John")){
            return "John";
        }
        if(people[i].Equals("Kent")){
            return "Don";
        }
    }
    return string.Empty;
}
```

الگوریتمی نامناسب برای یافتن افراد خاص در یک آرایه

```
string FoundPerson(string[] people) {
    var candidates = new List<string>() {
        "Don", "John", "Kent"
    };

    for(int i = 0; i < people.Length; i++) {
        if(candidates.Contains(people[i])) return people[i];
    }

    return string.Empty;
}
```

جایگزین کردن الگوریتم با یک الگوریتم جایگزین

در الگوریتم اول، به ازای هر فرد باید یک شرط دیگر به کد اضافه شود، در حالی که در الگوریتم جایگزین تنها کافیست به لیست آیتم اضافه کنیم.

روش یکی کردن عبارات شرطی:

```
double DisabilityAmount(){
    if(seniority < 2){
        return 0;
    }
    if(monthsDisabled > 12){
        return 0;
    }
    if(isPartTime) {
        return 0;
    }
    // compute the disability amount
    // ...
}
```

تابع با مجموعه ای از شرط های اولیه

```
double DisabilityAmount(){
    if(!IsEligibleForDisability()) return 0;

    // compute the disability amount
    // ...

    returned calculatedAmount;
}
```

یکی کردن مجموعه ای از شرط ها

روش یکی کردن بخش های تکراری عبارات شرطی:

```
if(IsSpecialDeal()){
    total = price * 0.95;
    Send();
}
else {
    total = price * 0.98;
    Send();
}
```

بخش تکراری در بدنه شروط

```
if(IsSpecialDeal())
    total = price * 0.95;
else
    total = price * 0.98;

Send();
```

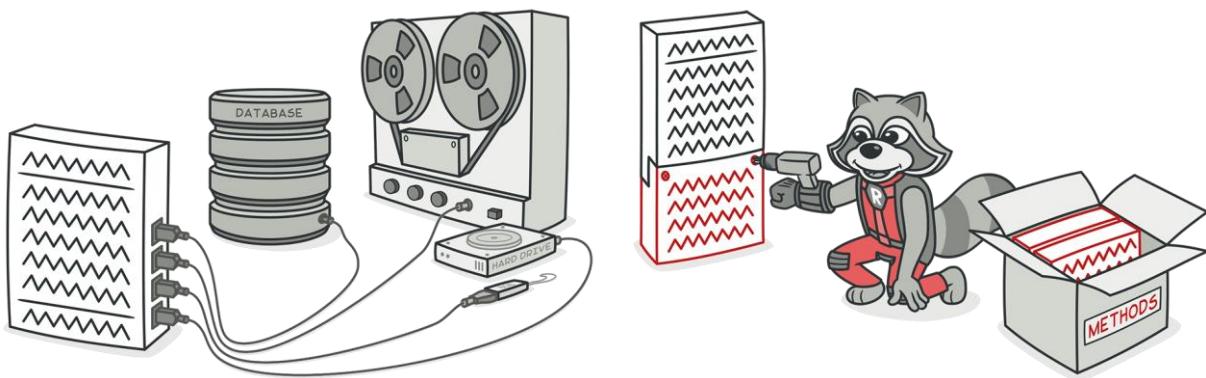
یکی کردن بخش های تکراری خارج از شرط

۳-۴-۵- کلاس های داده

این کلاس ها تنها دارای فیلد ها و ویژگی های داده، و متدهایی برای گرفتن و یا تنظیم کردن این داده ها هستند. در این حالت به جای استفاده از فیلد های Public، با روش کپسوله سازی فیلد، آن ها را با Property ها جایگزین کنید.

در صورتی که یک کارایی یا متدهای خودش جا بگیرد، با استفاده از روش استخراج متدها و جا به جایی متدها، آن را به کلاس های مناسب داده خود ارسال کنید.

همچنین متدهای قدیمی را که دسترسی بی حد و حصری به داده ها میدهند را با روش حذف تنظیم کننده و یا پنهان سازی متدهای دسترسی خارج کنید.



روش کپسوله سازی فیلد ها:

```
class Person {
    public string name;
}
```

فیلد به عنوان ویژگی کلاس

```
class Person {
    public string Name { get; set; }
}
```

کپسوله سازی فیلد

روش پنهان سازی متدها:

```
class Person {
    public void BroadDataAccess(){ ... }
}
```

دسترسی آزاد به اطلاعات به وسیله یک متدهای Public

```
class Person {
    private void BroadDataAccess(){ ... }
}
```

پنهان سازی متدهای

۴-۴-۷- کد مرده

کد های مرده، متغیر ها، پارامتر ها، متد ها، فیلد ها، یا کلاس هایی هستند که دیگر کارایی ندارند (چرا که دیگر منسون شده اند). بهترین روش برای یافتن این کد ها استفاده از یک محیط توسعه یکپارچه یا IDE است. برای خلاص شدن از این کد های منسون آن ها را پاک کنید، و یا اگر فکر میکنید که هنوز افرادی هستند که این تغییر ناگهانی، یعنی پاک شدن توابع بر روی کار آنها تاثیر میگذارد، ابتدا آن متد ها را به عنوان Obsolete یا منسون مارک کنید و پس از یک تغییر کلی (Breaking Change) این کد ها از دسترس خارج کنید. در این نوع بونیز میتوانید از روش *Inline* کردن کلاس، حذف سلسله وراثت و یا حذف پارامتر از تابع استفاده کنید.



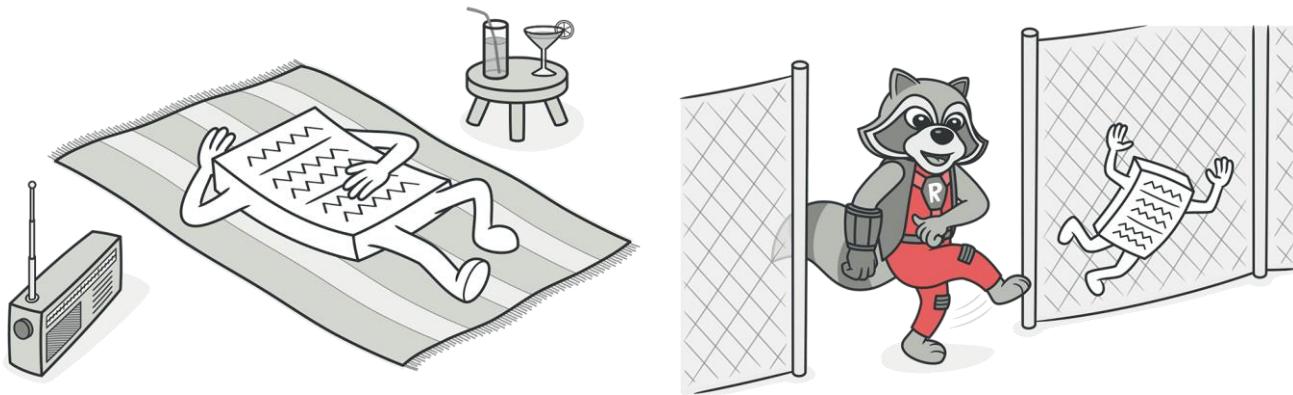
روش مارک کردن کد منسون:

```
[Obsolete("This method is Old, use NewMethod instead", error: false)]
public void DeadMethod() {
}

DeadMethod();
'DeadMethod()' is obsolete: 'This method is Old, use NewMethod instead'
```

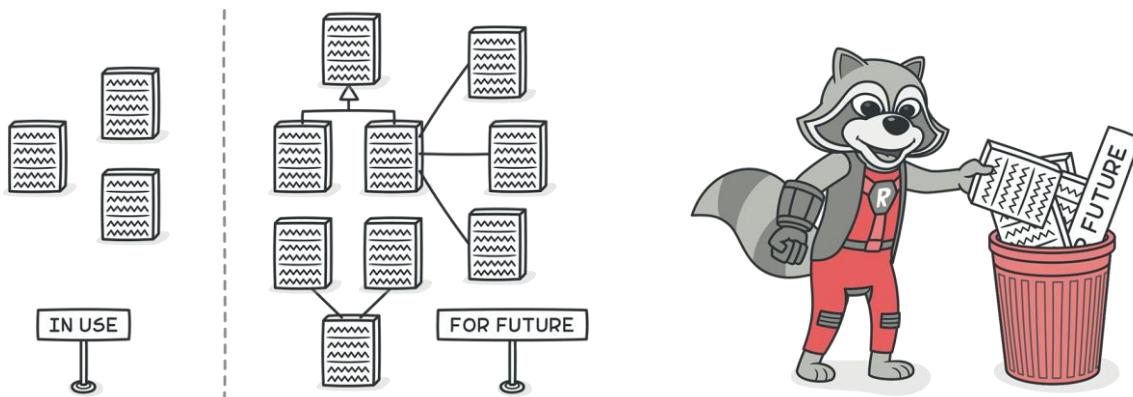
۴-۵- کلاس های تبل

نگهداری و فهم کلاس ها همواره هزینه مالی و زمانی دارد. هر گاه کلاسی آنقدر مهم نیست که توجه شما را جلب کند باید آن را حذف کنید. در اینجا نیز میتواند از روش *Inline* کردن کلاس و حذف سلسله مراتب وراثت استفاده کنید.



۴-۶- عمومیت خطرناک

در این حالت، کلاس، متدها، فیلد یا پارامتری از یک متدهای استفاده است. این کدها برای پشتیبانی از ویژگی‌های احتمالی آینده نوشته شده اند در حالتی که هرگز پیاده سازی نشده‌اند. فیلد‌های بدون کاربرد را به راحتی حذف کنید و برای سایر موارد از روش‌های *Inline* کردن کلاس، حذف سلسله مراتب وراثت، *Inline* کردن متدهای استفاده کنید.



۷-۵- جفتگر ها



تمامی بو های این گروه سعی بر اتصال کلاس ها به یکدیگر دارند و یا نشان میدهند که در صورت جایگزین شدن اتصالات با موکل ها چه اتفاقی می افتد.

۷-۵-۱- حسادت به ویژگی

هر گاه کلاسی به داده های یک کلاس دیگر، بیشتر از داده های خودش نیازمند است دچار حسادت ویژگی شده ایم. اگر مشخصا نیاز است تا یک متد به کلاسی دیگر منتقل شود، با استفاده از روش انتقال متد این کار را انجام دهید.

اگر تنها بخشی از کد به داده های کلاس دیگری رجوع میکند، با استفاده از روش استخراج متد، آن را جدا کرده و به محل مناسب منتقل کنید.



۷-۵-۲- صمیمیت نامناسب

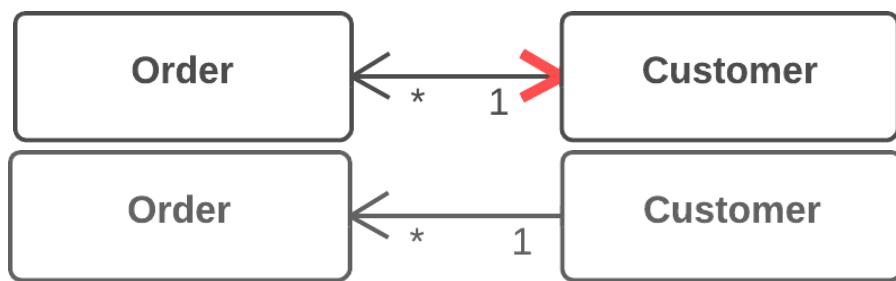
صمیمیت نامناسب زمانی رخ میدهد که یک کلاس از فیلد های درونی و متدهای یک کلاس دیگر استفاده کند. راحت ترین راه حل انتقال متد و یا انتقال فیلد است.

اگر کلاس ها متقابلا ناوابسته باشند، از روش تغییر وابستگی دو طرفه به یک طرفه استفاده کنید.

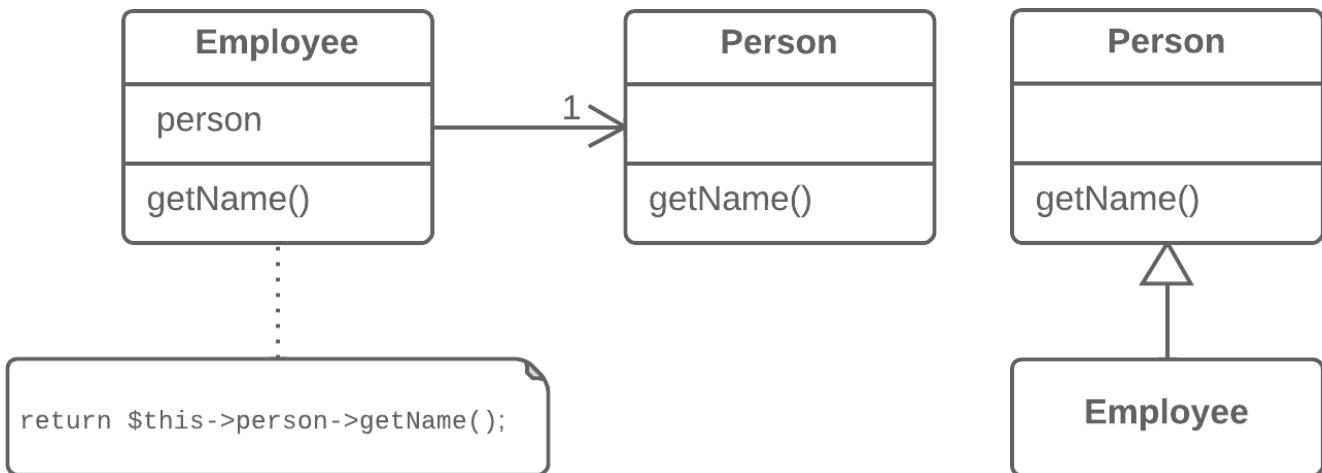
اگر این «صمیمیت» بین کلاس پدر و فرزند است، از روش جایگزینی و کالت با وراثت استفاده کنید.



روش تبدیل وابستگی دو طرفه به وابستگی یک طرفه:



روش جایگزینی و کالت با وراثت:



۳-۵-۷- کتاب خانه ناکامل

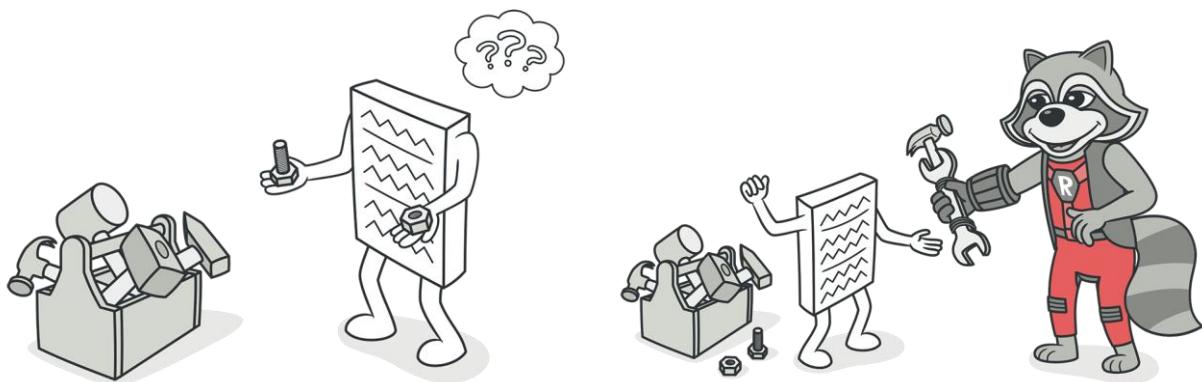
دیر یا زود کتاب خانه ها از نیاز های کاربر فاصله میگیرند. تنها راه حل این مشکل تغییر در کتاب خانه است که در بسیاری از موقع این عمل امکان پذیر نیست، چرا که کتاب خانه ها فقط-خواندنی هستند. برای معرفی کردن متدها و قابلیت های جدید به کتابخانه موجود میتوانید از روش متدهای خارجی استفاده کنید. این کار در زبان سی شارپ با استفاده از قابلیت Extension Methods انجام می شود. برای انجام تغییرات بزرگ در یک کتاب خانه کلاس، از روش استفاده از افزونه های محلی کمک بگیرید.

روش استفاده از متدهای خارجی و افزونه ها:

در سی شارپ میتوان با تعریف یک کلاس static و مارک کردن اولین پارامتر تابع با کلمه کلیدی this آن را به عنوان یک متاد افزونه تعریف کرد.

```
public static class DateTimeExtensions {
    public static DateTime Yesterday(this DateTime dt){
        return dt.AddDays(-1);
    }
    public static DateTime Tomorrow(this DateTime dt){
        return dt.AddDays(+1);
    }
}
```

تعریف یک متاد افزونه



۴-۵-۷- زنجیره پیام

اگر در کدتان زنجیره از صدا زدن های تابع به شکل زیر را دیدید علامتی از بُوی زنجیره پیام است:

CallA().CallB().CallC(...).CallD(...).CallE()...

برای از بین بردن زنجیره از روش پنهان سازی موکل استفاده کنید. همچنین از خودتان پرسید که چرا شی آخر در حال استفاده شدن است؟ کارایی آن را پیدا کنید و پس از استخراج متده مورد نظر، با استفاده از جا به جایی متده آن را به ابتدای زنجیر منتقل کنید.



۵-۵-۷- مردمیانی

اگر کلاسی تنها یک کار انجام میدهد، و آن هم این است که وظیفه خود را به کلاس دیگری محول کند، چه نیازی به وجود آن است؟ این بو در صورت سو استفاده در رفع زنجیره پیام به وجود می آید.



۸- منابع

- ❖ Fowler, Martin (1999). Refactoring. Improving the Design of Existing Code. Addison-Wesley. ISBN 0-201-48567-2.
- ❖ Refactoring Guru
- ❖ Arsenovski, Danijel (2009). Professional Refactoring in C# and ASP.NET. Wrox. ISBN 978-0-470-43452-9.
- ❖ Ritchie, Peter (2010). Refactoring with Visual Studio 2010. Packt. ISBN 978-1-84968-010-3.