

# الگوهای طراحی



REFACTORING  
· GURU ·

مترجم: پیروز آذرباد

منبع : [refactoring.guru/design-patterns](http://refactoring.guru/design-patterns)

مترجم: پیروز آذرباد

مشخصات نشر : تهران ۱۴۰۲

مشخصات ظاهري: ديجيتال، ۲۰۰ ص، مصور

موضوع: الگوهای طراحی

# الگوهای طراحی را بهتر بشناسیم

## هر کدام از آنها چه کاربردی دارد

منبع : [refactoring.guru/design-patterns](http://refactoring.guru/design-patterns)

مترجم : پیروز آذرباد

## فهرست

9.....	پیش گفتار.....
10.....	دربارهی کتاب.....
11.....	انواع الگوهای طراحی.....
12.....	<b>Creational</b>
13.....	..... <b>Factory Method</b>
13.....	طرح مسئله.....
14.....	راه حل.....
16.....	ساختار.....
17.....	مثال.....
20.....	چه زمانی باید از این الگو استفاده کنیم؟.....
21.....	معایب و مزایا.....
22.....	..... <b>Abstract factory</b>
22.....	طرح مسئله.....
23.....	راه حل.....
25.....	ساختار.....
26.....	مثال.....
29.....	چه زمانی باید از این الگو استفاده کنیم؟.....
29.....	معایب و مزایا.....
30.....	..... <b>Builder</b>
30.....	طرح مسئله.....
32.....	راه حل.....
34.....	ساختار.....
35.....	مثال.....
39.....	چه زمانی باید از این الگو استفاده کنیم؟.....
40.....	معایب و مزایا.....
41.....	..... <b>Prototype</b>
41.....	طرح مسئله.....
42.....	راه حل.....
43.....	مقایسه با دنیای واقعی.....
44.....	ساختار.....
45.....	پیاده‌سازی نمونه اولیه ..... Prototype
46.....	مثال.....
48.....	چه زمانی باید از این الگو استفاده کنیم؟.....

49.....	معایب و مزایا .....
<b>50.....</b>	<b>Singleton</b>
50.....	طرح مسئله .....
51.....	راه حل .....
52.....	مقایسه با دنیای واقعی .....
52.....	ساختار .....
53.....	مثال .....
54.....	چه زمانی باید از این الگو استفاده کنیم؟ .....
54.....	معایب و مزایا .....
<b>56.....</b>	<b>Structural</b>
<b>57.....</b>	<b>Adapter</b>
57.....	طرح مسئله .....
58.....	راه حل .....
59.....	مقایسه با دنیای واقعی .....
60.....	ساختار .....
61.....	مثال .....
63.....	چه زمانی باید از این الگو استفاده کنیم؟ .....
63.....	معایب و مزایا .....
<b>64.....</b>	<b>Bridge</b>
64.....	طرح مسئله .....
65.....	راه حل .....
66.....	ساختار .....
66.....	مثال .....
69.....	چه زمانی باید از این الگو استفاده کنیم؟ .....
69.....	معایب و مزایا .....
<b>70.....</b>	<b>Composite</b>
70.....	طرح مسئله .....
71.....	راه حل .....
72.....	مقایسه با دنیای واقعی .....
73.....	ساختار .....
74.....	مثال .....
76.....	چه زمانی باید از این الگو استفاده کنیم؟ .....
76.....	معایب و مزایا .....
<b>77.....</b>	<b>Decorator</b>

77.....	طرح مسئله.....
79.....	راه حل.....
81.....	مقایسه با دنیای واقعی.....
82.....	ساختار.....
82.....	مثال.....
86.....	چه زمانی باید از این الگو استفاده کنیم؟.....
86.....	معایب و مزایا.....
<b>87.....</b>	<b>Facade</b>
87.....	طرح مسئله.....
87.....	راه حل.....
88.....	مقایسه با دنیای واقعی.....
88.....	ساختار.....
89.....	مثال.....
91.....	چه زمانی باید از این الگو استفاده کنیم؟.....
91.....	معایب و مزایا.....
<b>92.....</b>	<b>Flyweight</b>
92.....	طرح مسئله.....
93.....	راه حل.....
97.....	ساختار.....
98.....	مثال.....
100.....	چه زمانی باید از این الگو استفاده کنیم؟.....
100.....	معایب و مزایا.....
<b>101.....</b>	<b>Proxy</b>
101.....	طرح مسئله.....
102.....	راه حل.....
103.....	مقایسه با دنیای واقعی.....
103.....	ساختار.....
104.....	مثال.....
107.....	چه زمانی باید از این الگو استفاده کنیم؟.....
108.....	معایب و مزایا.....
<b>109.....</b>	<b>Behavioral</b>
<b>110.....</b>	<b>Chain Of Responsibility</b>
110.....	طرح مسئله.....
112.....	راه حل.....

113.....	مقایسه با دنیای واقعی
114.....	ساختار
115.....	مثال
118.....	چه زمانی باید از این الگو استفاده کنیم؟
119.....	معایب و مزایا
<b>119.....</b>	<b>Command</b>
120.....	طرح مسئله
121.....	راه حل
123.....	مقایسه با دنیای واقعی
124.....	ساختار
125.....	مثال
129.....	چه زمانی باید از این الگو استفاده کنیم؟
130.....	معایب و مزایا
<b>131.....</b>	<b>Iterator</b>
131.....	طرح مسئله
132.....	راه حل
133.....	مقایسه با دنیای واقعی
134.....	ساختار
135.....	مثال
138.....	چه زمانی باید از این الگو استفاده کنیم؟
139.....	معایب و مزایا
<b>140.....</b>	<b>Mediator</b>
140.....	طرح مسئله
141.....	راه حل
142.....	مقایسه با دنیای واقعی
143.....	ساختار
144.....	مثال
146.....	چه زمانی باید از این الگو استفاده کنیم؟
147.....	معایب و مزایا
<b>148.....</b>	<b>Memento</b>
148.....	طرح مسئله
150.....	راه حل
152.....	ساختار
155.....	مثال

157.....	چه زمانی باید از این الگو استفاده کنیم؟
157.....	معایب و مزایا
<b>158.....</b>	<b>Observer</b>
158.....	طرح مسئله
159.....	راه حل
161.....	مقایسه با دنیای واقعی
161.....	ساختار
163.....	مثال
165.....	چه زمانی باید از این الگو استفاده کنیم؟
166.....	معایب و مزایا
<b>167.....</b>	<b>State</b>
167.....	طرح مسئله
169.....	راه حل
170.....	مقایسه با دنیای واقعی
171.....	ساختار
172.....	مثال
175.....	چه زمانی باید از این الگو استفاده کنیم؟
175.....	معایب و مزایا
<b>176.....</b>	<b>Strategy</b>
176.....	طرح مسئله
178.....	راه حل
179.....	مقایسه با دنیای واقعی
179.....	ساختار
180.....	مثال
181.....	چه زمانی باید از این الگو استفاده کنیم؟
182.....	معایب و مزایا
<b>183.....</b>	<b>Template Method</b>
183.....	طرح مسئله
184.....	راه حل
186.....	مقایسه با دنیای واقعی
187.....	ساختار
188.....	مثال
190.....	چه زمانی باید از این الگو استفاده کنیم؟
190.....	معایب و مزایا

191.....	Visitor
191.....	طرح مسئله
192.....	راه حل
194.....	مقایسه با دنیای واقعی
195.....	ساختار
196.....	مثال
198.....	چه زمانی باید از این الگو استفاده کنیم؟
199.....	معایب و مزایا

## پیش گفتار

هنگامی که به دنبال پیشرفت و حرفة‌ای شدن در دنیای برنامه نویسی هستیم، مسیری پیچیده و متنوع در پیش داریم. اغلب با چالش‌ها و وظایفی روبرو می‌شویم که نیاز به راه حل‌های کارآمد دارند. در چنین شرایطی، الگوهای طراحی به عنوان ابزارهای مؤثر می‌توانند به ما در بهبود فرآیند کدنویسی و حل مسائل کمک کنند. همه‌ی ما در حین کار کردن با مشکلاتی مواجه می‌شویم که راه حلی برای آن‌ها نداریم و دچار سر در گمی می‌شویم. ایده‌ی ترجمه‌ی این کتاب هم از همین مسائل و سر در گمی‌ها به وجود آمد. الگوهای طراحی راه حل‌هایی برای مسائل دنیای نرم افزار هستند که مانند تجربه‌های طلایی به کمک ما می‌آیند تا کارمان را برای مدیریت و توسعه‌ی کدهای پیچیده و حجمی راحت‌تر کنند.

حتی اگر یک بار هم فکر یادگیری الگوهای طراحی به ذهنتان رسیده باشد با اولین جستجویی که انجام دهید وب سایت [refactoring.guru](#) را در اولین نگاه پیدا خواهید کرد. این سایت همواره به عنوان یک منبع خوب برای یادگیری الگوهای طراحی بوده است. از این رو تصمیم گرفتم تا تمامی مطالب مربوط به الگوهای طراحی موجود در این سایت را به صورت یک کتاب ترجمه کنم.

در این کتاب، به بررسی الگوهای طراحی خواهیم پرداخت و خواهیم دید چگونه می‌توانند به بهبود کیفیت کد، افزایش قابلیت نگهداری، و افزایش توانایی توسعه‌پذیری برنامه‌ها کمک کنند. امیدوارم این مطالب به شما در برداشتن گامی موفق به سمت حرفة‌ای شدن در دنیای برنامه نویسی کمک کند.

## درباره‌ی کتاب

تمام تلاشم را کرده‌ام تا این کتاب به صورت ساده و روان ترجمه شود. برای خواندن این کتاب نیاز به دانش متوسطی از برنامه‌نویسی و چالش‌هایی که ممکن است با آن‌ها رو برو شوید دارید. سعی کرده‌ام تا مثال‌های ساده‌تری اضافه بر مثال‌های خود سایت بیاورم تا کمی به فهم راحت‌تر مفاهیم کمک کرده باشم. بعضی از الگوهای طراحی با توجه به تاثیراتی که بر روی برنامه‌ها می‌گذارند از پیچیدگی بیشتری برخوردارند. پیشنهاد می‌کنم در صورتی که در حین خواندن مسائل و راه حل‌های آنها با مشکل مواجه شدید و دچار گنگی شدید، مثال‌ها و قسمت مقایسه با دنیای واقعی را حتماً مطالعه کنید.

پیروز باشید.

## انواع الگوهای طراحی

در تعریف اولیه الگوهای طراحی راه حل هایی برای مشکلات دنیای برنامه نویسی می باشند. به طور کلی ۳ دسته الگوی طراحی داریم که هر کدام از آنها بر روی یک قسمتی از برنامه هایمان تاثیر می گذارند و باعث بهبود بخشیدن عملکرد برنامه هایمان می شوند. باید بدانیم که الگوهای طراحی یک قطعه کد نیستند که آنها را کپی کنیم و در برنامه هایمان استفاده کنیم و همینطور به هیچ زبان برنامه نویسی خاصی وابسته نیستند. این الگوها صرفا در نقش یک راهنمای راه حل ظاهر می شوند.

انواع الگوهای طراحی عبارتند از:

- Creational
- Structural
- Behavioral

در ادامه درباره هر کدام از این الگوها به طور مفصل صحبت خواهیم کرد.

# Design Patterns

## Creational

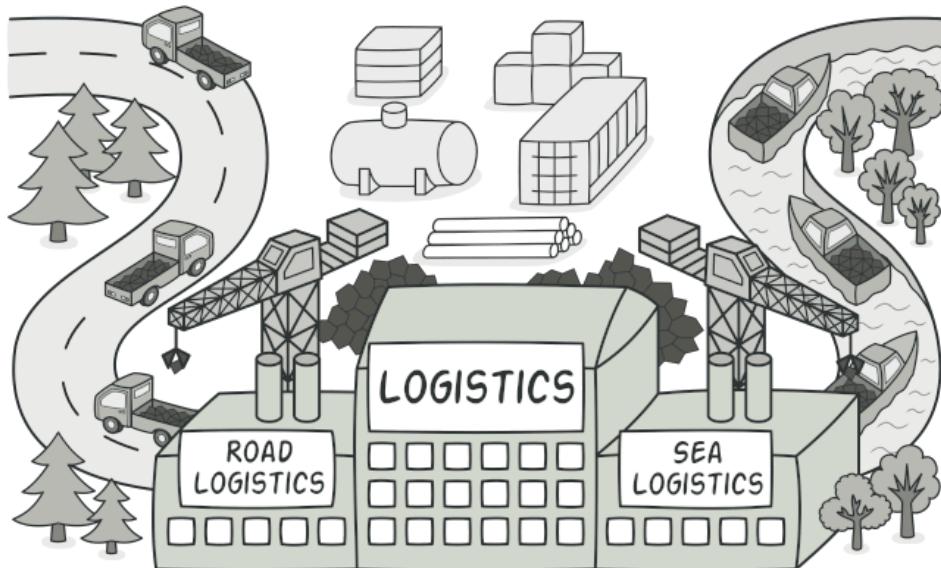
این نوع از الگوهای طراحی درباره‌ی تنوع مکانیسم‌های ایجاد Object‌ها صحبت می‌کنند که باعث بالا رفتن انعطاف کد شده و از دوباره کاری‌ها جلوگیری می‌کند. در ادامه انواع آنها را نام می‌بریم و هر کدام را توضیح می‌دهیم.

انواع Creational Design pattern‌ها :

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

## Factory Method

در این الگوی طراحی، Object ها در کلاس پدر ساخته می‌شوند و کلاسهای فرزند می‌توانند بنا به نیاز خود نوع Object هایی که باید ایجاد شوند را تغییر دهند.



### طرح مسئله

فرض کنید که یک برنامه برای مدیریت لجستیک می‌نویسید. ورژن اولیه برنامه‌تان فقط برای حمل و نقل بار توسط کامیون نوشته شده است. بنابراین کلاسی تحت عنوان `Truck` در نظر می‌گیرید و تمامی منطق برنامه را در این کلاس پیاده می‌کنید.

بعد از مدتی که برنامه بسیار محبوب شد، درخواستهایی برای اضافه کردن قابلیت حمل کالاهای توسط کشتی نیز به شما داده می‌شود.

خبر خیلی خوبی است، اما به این فکر کردید که برای کدهایمان چه اتفاقی می‌افتد؟؟

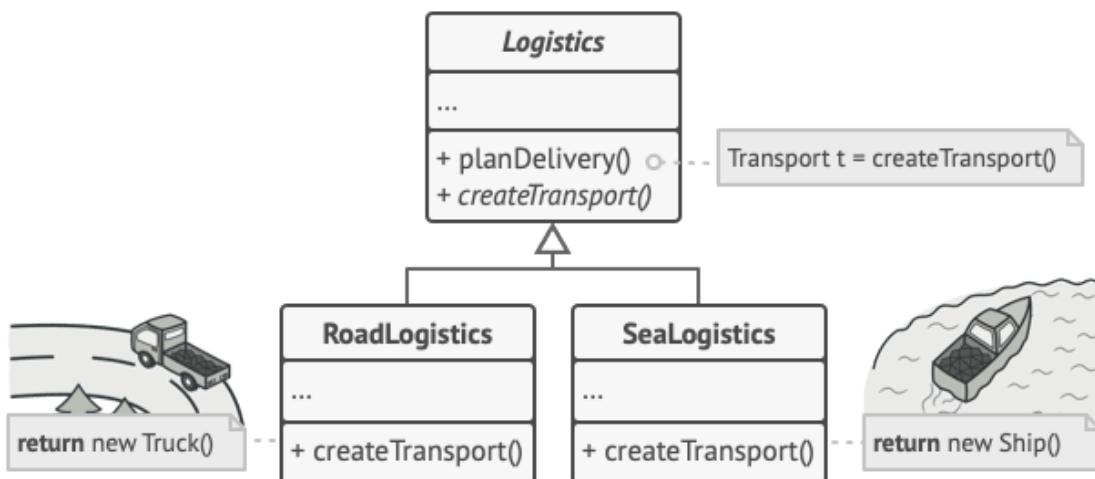


با توجه به کدی که نوشتیم، تمام منطق برنامه‌مان در جهتی است که فقط کارهای مربوط به حمل بار با کامیون را می‌تواند انجام دهد و حالا باید کل کد را تغییر دهیم تا قابلیتی برای حمل بار با کشتی هم اضافه شود. این مورد را هم باید در نظر بگیریم که ممکن است بعد ها درخواست دیگری برای اضافه کردن نوع جدیدی از حمل و نقل داده شود.

اگر این موارد را در نظر نگیریم کدی کثیف خواهیم داشت و نگهداری از آن هم برایمان سخت خواهد شد. اما راه حل چیست؟

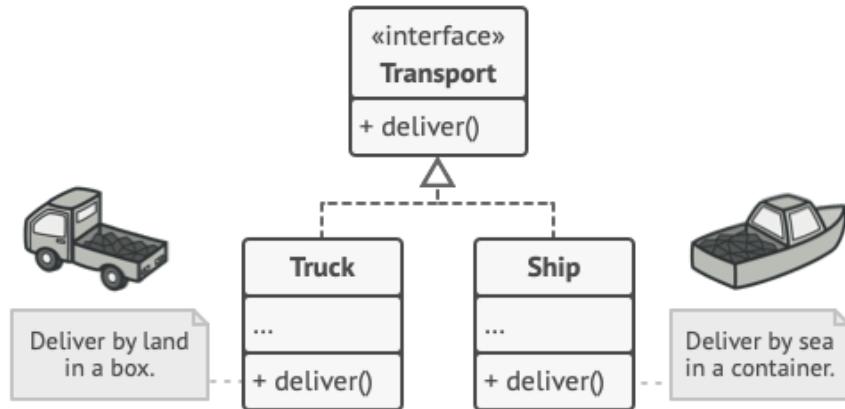
## راه حل

برای حل این مشکل از الگوی Factory method استفاده می‌کنیم. این الگوی طراحی وظیفه‌ی ساخت Object‌ها را برعهده می‌گیرد و شما بطور مستقیم شیء ای را new نمی‌کنید. در این الگو اشیائی که برگردانده می‌شوند را محصول یا product می‌نامیم.



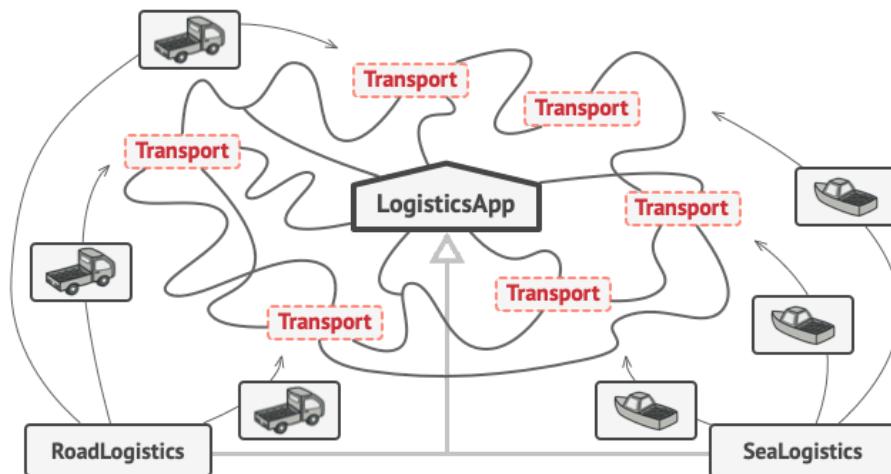
در نگاه اول ممکن است که این کار بیهوده بنظر برسد، زیرا که فقط فراخوانی سازنده را از یکجای برنامه به جای دیگری منتقل کردہ‌ایم! اما تفاوتی که وجود دارد این است که در کلاس فرزند می‌توانیم نوع شیء ای که قرار است ایجاد شود را تغییر دهیم.

البته یک محدودیتی هم وجود دارد، اشیائی که می‌خواهیم در زیر کلاسها ایجاد کنیم باید ویژگی‌های مشترکی که در کلاس پدر تعریف شده‌اند را داشته باشند.



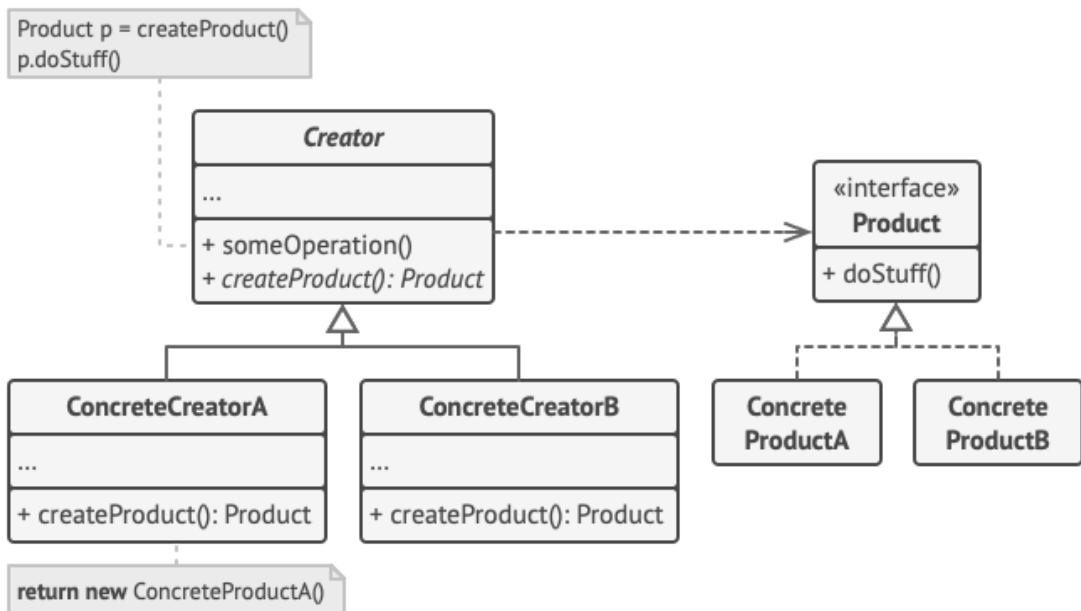
به طور مثال برای مشکلی که داشتیم، هر دو کلاس Truck و Ship باید واسطه Transport را پیاده‌سازی کنند. در این واسطه متده تخت عنوان deliver() وجود دارد که هر زیر-کلاسی این متده را به صورت متفاوتی پیاده‌سازی خواهد کرد. بطور مثال کامیون بصورت زمینی حمل و نقل را انجام می‌دهد و کشتی هم از طریق دریا این کار را انجام می‌دهد.

SeaLogistics مورد نظر در کلاس RoadLogistics شیء Truck را برمی‌گرداند و در کلاس SeaLogistics شیء Ship را برمی‌گرداند.



کدی که از Factory method استفاده می‌کند (به آن client می‌گوییم) اینکه بقیه کلاس‌ها چگونه پیاده‌سازی شده اند را نمی‌بیند و فقط یک دید کلی نسبت به حمل و نقل دارد. Client فقط می‌داند که تمامی اشیائی که باید فرایند حمل و نقلی انجام دهند، متده به نام deliver() را در خود دارند ولی چگونگی حمل و نقل برایش مهم نیست!

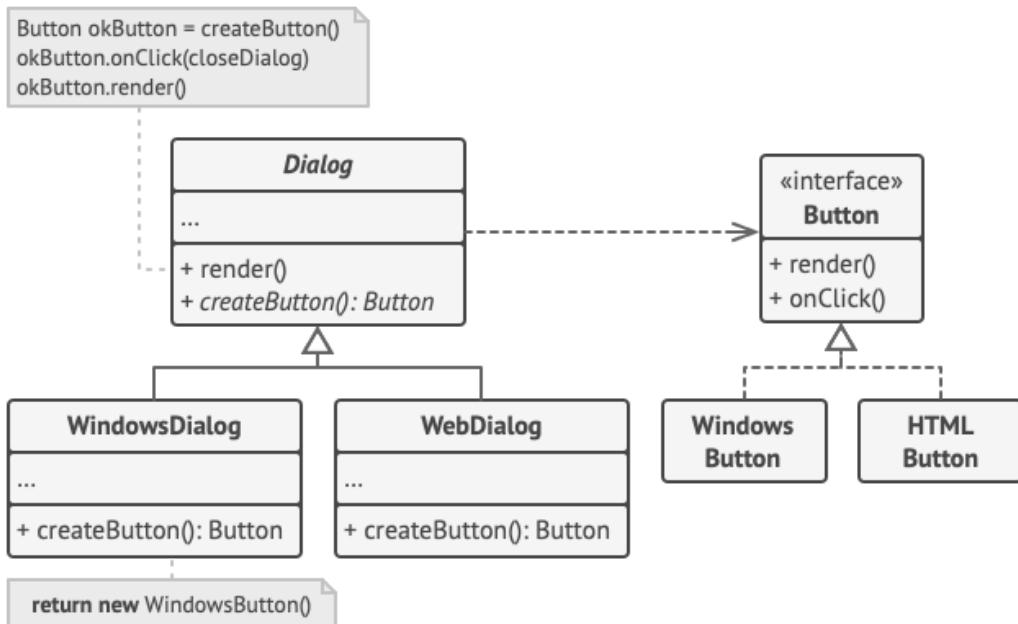
## ساختار



- **واسطی** است که بین object هایی که کلاس Creator و تمام زیرکلاس‌هایش تولید می‌کنند مشترک است.
- **پیاده‌سازی های متفاوتی از واسط Product هستند.**
- **کلاس Creator تصویر بالا،**-ای که وظیفه‌ی ایجاد اشیاء جدید برعهده‌ی آن است را در خود دارد. نکته‌ی بسیار مهمی که باید به آن توجه کرد این است که نوع برگشتی این متدهای برابر با واسط Product باشد.
- برخلاف اسم این کلاس که تولید Product وظیفه‌ی اصلی این کلاس نیست. این کلاس بعضی از منطق‌های بیزینس برنامه که مرتبط با Product می‌باشد را در خود نگه می‌دارد.
- الگوی طراحی Factory method به ما کمک می‌کند تا منطق برنامه را جدای از اینکه چه Product ای داریم در نظر بگیریم.
- بطور مثال یک شرکت بزرگ توسعه نرم‌افزار می‌تواند بخشی از سازمان خود را برای آموزش توسعه‌دهندگان خود داشته باشد در صورتی که وظیفه‌ی اصلی شرکت توسعه‌ی نرم افزار و نوشتن کد می‌باشد نه تولید برنامه‌نویس!
- **Concrete Creators** : این کلاسها متدهای اصلی Factory method را پیاده‌سازی می‌کنند و می‌توانند مقدار برگشتی‌های متفاوتی از جنس product داشته باشند.

## مثال

در اینجا مثالی برای بهتر متوجه شدن این الگو داریم، در این مثال میبینیم که الگوی Factory method میتواند برای ایجاد یک رابط کاربری cross-platform به ما کمک کند تا وابستگی بین کدهای چگونه و UI را کم کنیم.



در این مثال کلاس Dialog از اینمانهای متفاوتی برای خروجی گرفتن از صفحه اش استفاده میکند. این اینمانها باید در هر سیستم به طور متفاوتی نمایش داده شوند ولی باید رفتارهای ثابتی داشته باشند. یک دکمه چه در ویندوز و یا چه در لینوکس یک دکمه است!

وقتی که الگوی Factory method به میدان میآید دیگر لازم نیست تا Dialog را برای هر سیستم عاملی پیاده‌سازی کنیم. میتوانیم یک Factory method برای کلاس Dialog ایجاد کنیم و بعد ها با ایجاد زیر-کلاسهایی از آن بر اساس سیستم عامل مورد نظرمان استایل‌های متفاوتی به دکمه‌مان بدھیم. زیر-کلاس‌ها بخش زیادی از کد اصلی‌مان را به ارث می‌برند و به لطف factory method میتوانیم برای صفحه مورد نظرمان خروجی بگیریم.

برای اینکه الگوی Factory method کار کند، کلاس پایه‌مان یعنی کلاس Dialog باید دکمه‌های انتزاعی داشته باشد. یعنی دکمه‌هایی که در این کلاس قرار می‌گیرند باید به صورت کلی باشند و زیر-کلاسها در صورت نیاز پیاده‌سازی های خود را داشته باشند.

البته که از این رویکرد میتوان در اینمانهای دیگری هم استفاده کرد و میتوانید هر چقدر که میخواهید اضافه کنید. با این کار به الگوی Abstract Factory نزدیک می‌شوید که بعدا در مورد آن صحبت خواهیم کرد.

در ادامه شبه کدی از این مثال را خواهیم داشت :

```
// The creator class declares the factory method that must
// return an object of a product class. The creator's subclasses
// usually provide the implementation of this method.
class Dialog is
    // The creator may also provide some default implementation
    // of the factory method.
    abstract method createButton():Button

    // Note that, despite its name, the creator's primary
    // responsibility isn't creating products. It usually
    // contains some core business logic that relies on product
    // objects returned by the factory method. Subclasses can
    // indirectly change that business logic by overriding the
    // factory method and returning a different type of product
    // from it.
    method render() is
        // Call the factory method to create a product object.
        Button okButton = createButton()
        // Now use the product.
        okButton.onClick(closeDialog)
        okButton.render()

// Concrete creators override the factory method to change the
// resulting product's type.
class WindowsDialog extends Dialog is
    method createButton():Button is
        return new WindowsButton()

class WebDialog extends Dialog is
    method createButton():Button is
        return new HTMLButton()

// The product interface declares the operations that all
// concrete products must implement.
interface Button is
    method render()
    method onClick(f)

// Concrete products provide various implementations of the
// product interface.
class WindowsButton implements Button is
    method render(a, b) is
        // Render a button in Windows style.
    method onClick(f) is
```

```
// Bind a native OS click event.

class HTMLButton implements Button is
    method render(a, b) is
        // Return an HTML representation of a button.
    method onClick(f) is
        // Bind a web browser click event.

class Application is
    field dialog: Dialog

    // The application picks a creator's type depending on the
    // current configuration or environment settings.
    method initialize() is
        config = readApplicationConfigFile()

        if (config.OS == "Windows") then
            dialog = new WindowsDialog()
        else if (config.OS == "Web") then
            dialog = new WebDialog()
        else
            throw new Exception("Error! Unknown operating
system.")

    // The client code works with an instance of a concrete
    // creator, albeit through its base interface. As long as
    // the client keeps working with the creator via the base
    // interface, you can pass it any creator's subclass.
    method main() is
        this.initialize()
        dialog.render()
```

## چه زمانی باید از این الگو استفاده کنیم؟

- زمانی که از قبل اطلاعات کافی در رابطه با وابستگی‌هایی که ممکن است اشیاء‌مان با هم در آینده داشته باشند نداریم.

این الگو فرایند ایجاد یک product را از فرایند چگونگی استفاده از آن جدا می‌کند. به همین دلیل دستمنان برای توسعه‌ی برنامه در آینده باز است. بطور مثال برای اضافه کردن یک Product به برنامه، فقط کافی است که یک زیر-کلاس برای creator ایجاد کنیم و factory method ها را پیاده‌سازی کنیم.

- زمانی از این الگو استفاده کنید که میخواهید کاربران بتوانند مولفه‌های داخلی framework یا library ای که توسعه دادید را گسترش دهند.

ارث بری ساده‌ترین راه برای گسترش یک framework یا library می‌باشد. اما یک framework چگونه می‌فهمد که زیر-کلاستان هم می‌تواند به جای یک مولفه‌ی استاندارد استفاده شود؟ راه حلی که می‌توان به آن اشاره کرد این است که کدی را که component های مختلف را در framework مان می‌سازد، به یک factory method کاهش دهیم و به همه اجازه دهیم تا بتوانند این component ها را پیاده‌سازی کنند و خودشان آنها را گسترش دهند.

- زمانی که می‌خواهیم در استفاده از منابع سیستم صرفه جویی کنیم. به اینگونه که به جای دوباره کاری و ایجاد اشیاء، از اشیاء موجود استفاده کنیم و دوباره آنها را نسازیم.

به طور معمول این نیاز را هنگام برخورد با اشیاء بزرگ و پر مصرف مانند متصل شدن به دیتابیس، فایلها و ... تجربه می‌کنیم.

ببینیم که چگونه می‌توان از یک شیء موجود دوباره استفاده کرد:

- در ابتدا به این نیاز داریم تا فضایی برای پیگیری این اشیاء بسازیم.

زمانی که یک نفر درخواستی برای یک شیء ارسال کرد، برنامه باید در این فضا به دنبال آن

شیء بگردد و ببیند که آن شیء آزاد است یا خیر.

در مرحله‌ی بعدی آن شیء را به client برگرداند.

اگر شیء آزادی وجود نداشته باشد، برنامه باید یک شیء جدید بسازد و آن را به فضای ذخیره‌سازی اضافه کند

همانطور که می‌بینید مقدار زیادی کد باید نوشته شود تا این فرایند انجام شود که این کار

اصلًا تمیز نیست!

به نظر بهترین جا برای این کار سازنده‌ی کلاسی است که قرار است اشیاء آن را دوباره استفاده کنیم. اما با این کار هر بار شیء جدیدی برگردانده می‌شود و نمی‌توان از اشیاء موجود استفاده کرد.

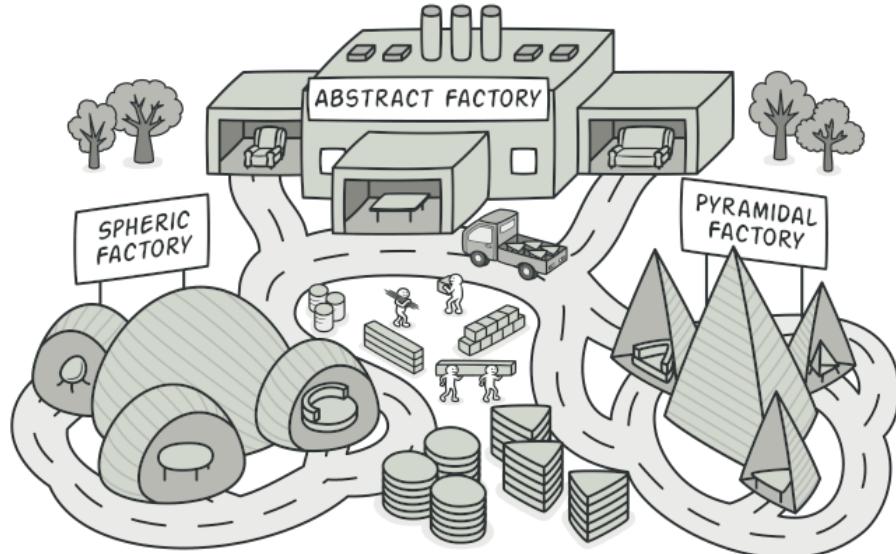
بنابراین به یک متده منظم نیاز داریم تا این کار را برایمان انجام دهد. که این خیلی شبیه می‌باشد!

## معایب و مزایا

- ❖ وابستگی بین **product** و **creator** های اصلی کم می‌شود.
- ❖ با این کار اصل اول SOLID یعنی Single Responsibility رعایت می‌شود. مراحل ایجاد **product** یک جا قرار می‌گیرد و نگهداری و پشتیبانی از آن راحت‌تر می‌شود.
- ❖ با استفاده از این الگو اصل دوم SOLID یعنی Open/closed رعایت می‌شود. با استفاده از این الگو می‌توانیم انواع مختلفی از **product** ها را بدون اینکه اشیاء موجود را تغییر دهیم به برنامه اضافه کنیم.
- از معایب این الگو این است که کدهای زیادی باید نوشته شود. زیر-کلاس‌های زیادی باید ایجاد شوند تا این الگو پیاده‌سازی شود. بهترین سناریو برای این الگو زمانی است که سلسله مراتب مشخصی از کلاسها و زیر-کلاس‌هایمان داشته باشیم.

## Abstract factory

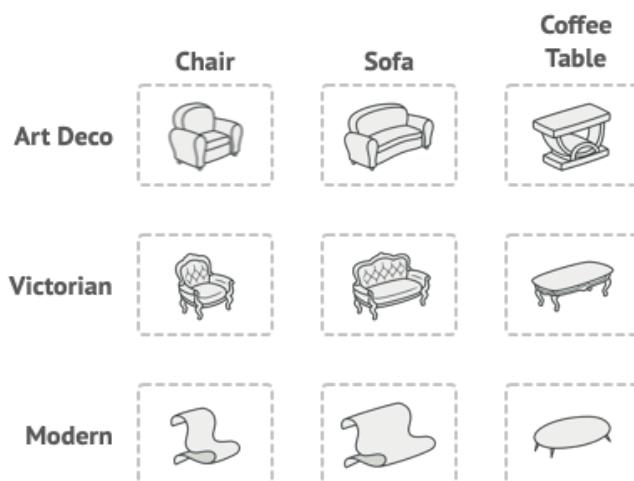
این الگوی طراحی به ما کمک می‌کند تا اشیائی که به یکدیگر مرتبط هستند را بدون اینکه اجزای داخلی آنها را بشناسیم بهم متصل کنیم.



### طرح مسئله

فرض کنید که می‌خواهید یک برنامه شبیه ساز برای فروشگاه مبلمان بنویسید. کد شما شامل یک سری کلاس می‌باشد که به شرح زیر است :

- دسته‌ای از محصولات مرتبط به هم مانند کاناپه، صندلی و میزی برای صرف قهوه!
- چندین مدل از همین محصولات و یا محصولات دیگر. به طور مثال مدل‌های Victoria , Modern و ArtDeco



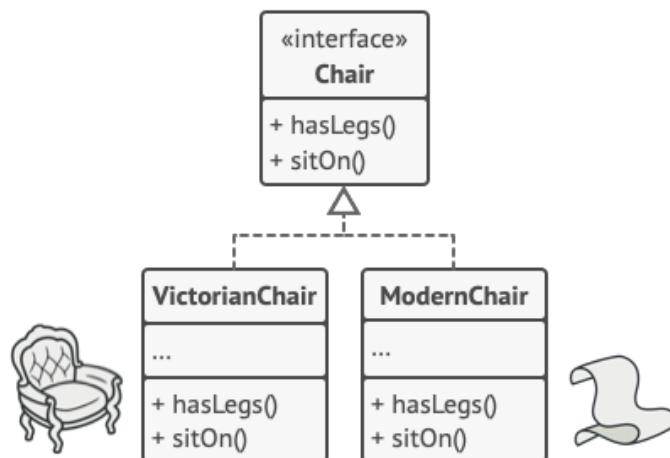
شما به راهی نیاز دارید تا بتوانید هر کدام از اشیاء مبلمان خود را به تنها یی ایجاد کنید و آن شیء باید با دیگر اشیاء خانواده خود همخوانی داشته باشد. مشتریان شما اگر محصولاتی از یک خانواده را دریافت کنند و بین آنها محصولی خارج از عرف و یا از مدلی دیگر ببینند گیج می‌شوند و این اصلاً خوب نیست.



از طرفی هم دلتان نمی‌خواهد که برای اضافه کردن یک وسیله‌ی جدید، به کدی که نوشتشید دست بزنید و آن را تغییر دهید. از آنجایی هم که فروشنده‌گان زود به زود فهرست محصولات خود را بروز می‌کنند شما باید هر دفعه کدتان را تغییر دهید !! و اما راه حل چیست ؟

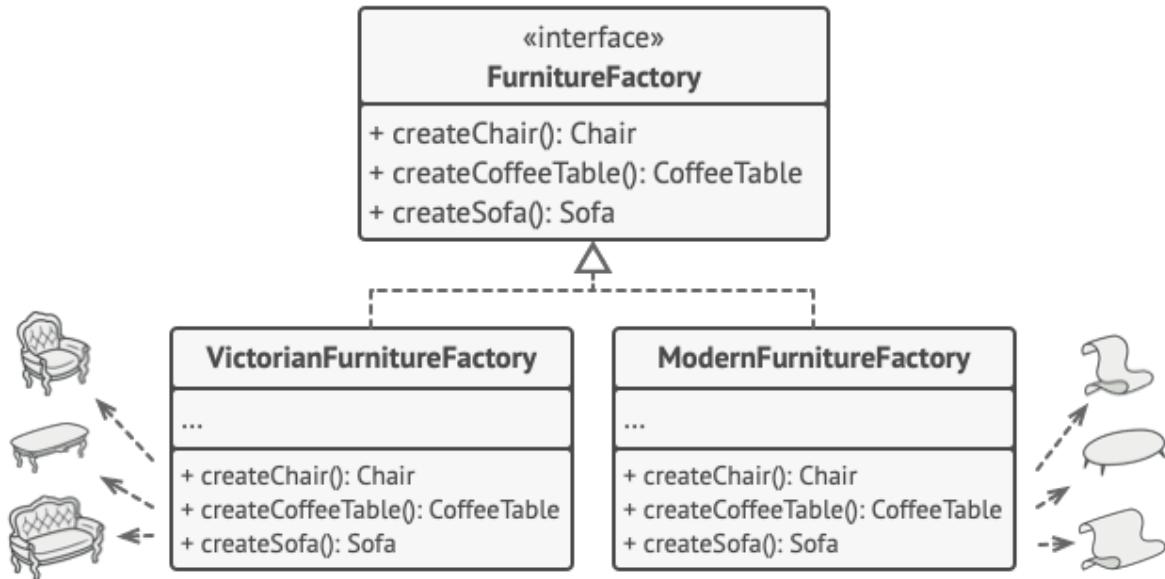
## راه حل

اولین پیشنهادی که الگوی Abstract factory می‌دهد این است که برای هر کدام از محصولاتی که در یک خانواده هستند، بصورت جداگانه واسطه‌ایی در نظر گرفته شود(مانند ... , chair, sofa, coffee table) با این کار هر گونه وسیله‌ای می‌تواند از واسطه‌ای خود پیروی کند. بطور مثال تمام مدل‌های صندلی واسطه Chair را پیاده‌سازی می‌کنند.



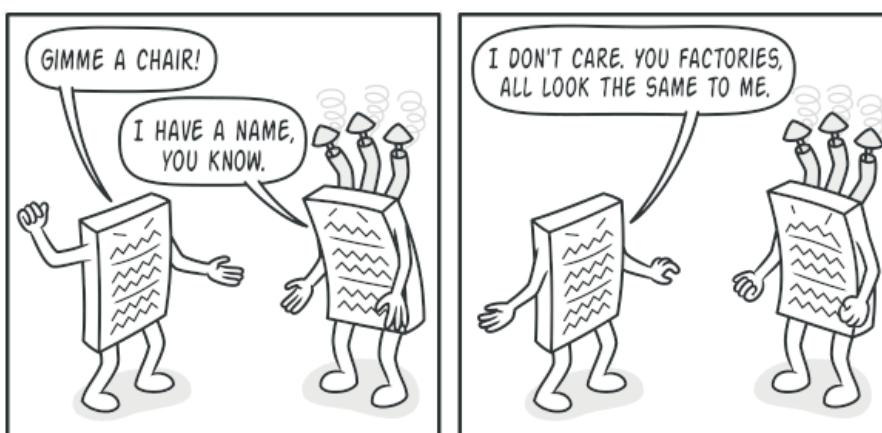
در گام بعدی باید یک واسطه Abstract factory ایجاد کنیم. این واسطه (Interface) شامل لیستی از متدهایی است که انواع محصولات در یک Category یا خانواده را می‌سازند. (بطور مثال createChair() و

. این متدها باید category-ای از نوع همان createCoffeTable() و createSofa() را برگردانند . CoffeeTable و Chair , Sofa یعنی واسطهای



و اما در رابطه با مدل‌های مختلف چه؟؟ برای هر کدام از مدل‌های مختلف یک category یک کلاس جدا بر اساس واسط Abstract factory-مان می‌سازیم. در این الگو، به کلاسی که محصولاتی از نوع خاص را برگرداند، یک factory گفته می‌شود. بطور مثال ModernFurnitureFactory فقط می‌تواند اشیائی از کدهای Client هم با factory و هم با product هایمان توسط واسطه‌ای Abstract ای که ایجاد کرده‌ایم کار می‌کنند.

با این کار می‌توانیم نوع factory ای که باید به client برگردانیم را بدون ایجاد تغییری در کد، مطابق نوع product مورد نظر تغییر دهیم.



فرض کنید client مان درخواست تولید یک صندلی داشته باشد. Client نسبت به چگونگی تولید این صندلی‌ها آگاه نیست و برایش اهمیتی ندارد که چه نوع صندلی ای باشد. مدرن باشد یا victoria style.

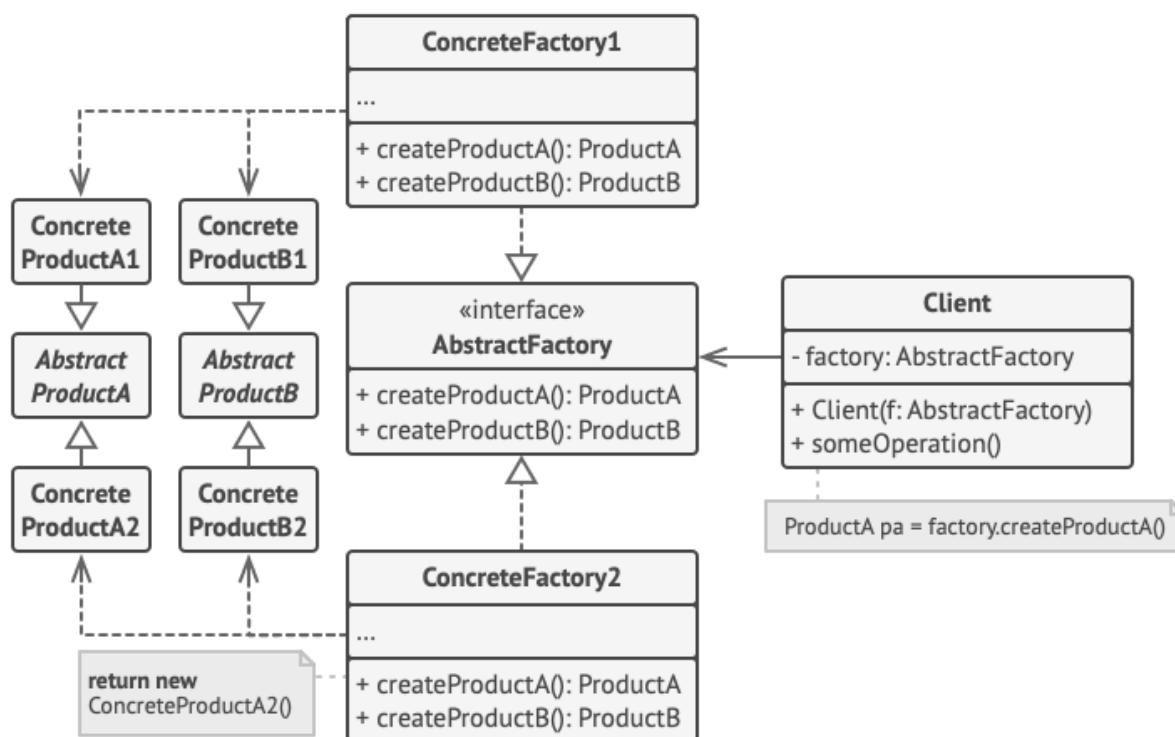
یک صندلی می‌خواهد. می‌دانیم که باید با تمام وسیله‌هایی که از خانواده Chair هستند به صورت یکسان برخورد شود.

با این رویکرد، تنها چیزی که client می‌داند این است که تمامی وسایلی که هم‌خانواده‌ی Chair هستند باید متدها sitOn() را پیاده‌سازی کنند. همین روند برای دیگر مدل‌ها هم تکرار می‌شود.

یک نکته باقی می‌ماند. این که اگر client بصورت مستقیم از واسطه Abstract factory استفاده کند، چه چیزی اشیاء factory مان را می‌سازد؟!

به طور معمول برنامه در گام اول یک شیء factory اولیه می‌سازد. قبل از این کار، برنامه باید با توجه به تنظیمات محیطی (environment settings) یا فایل‌های configuration ای که به آن می‌دهیم، نوع factory را انتخاب کند. بهتر است ساختار این الگو را ببینیم تا درک بهتری پیدا کنیم.

## ساختار

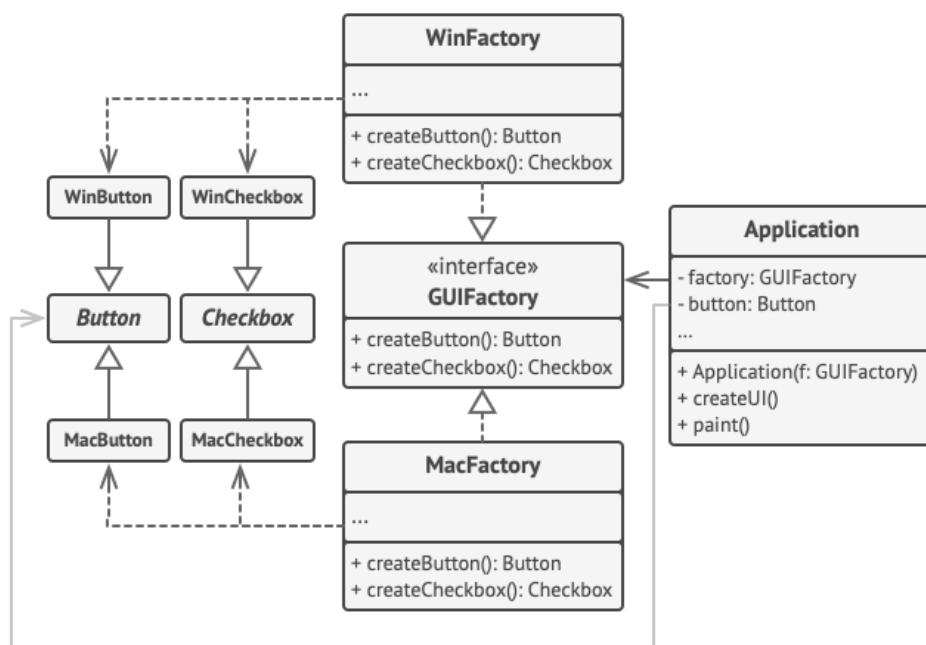


- Abstract Products** •  
یک دسته یا خانواده به حساب می‌آیند.
- Concrete Products** •  
این کلاس‌ها پیاده‌سازی انواع مختلف Abstract Product هایمان می‌باشند  
که براساس category شان دسته‌بندی می‌شوند. هر کدام از Abstract Product ها (مانند Chair, ... , Sofa) باید برای همه‌ی مدل‌ها پیاده‌سازی شوند (Victoria/Modern).
- Abstract Factory** •  
واسطه‌ی است که شامل لیستی از متدها برای ایجاد هر کدام از Abstract Factory هایمان می‌باشد.

• **Concrete Factories**: این کلاس متدهای واسط Abstract Factory را پیاده‌سازی می‌کند. هر کدام از این کلاس‌ها مطابق با یک مدل از Product هستند و فقط وظیفه‌ی ایجاد همان نوع را دارند.

## مثال

در اینجا مثالی برای بهتر متوجه شدن این الگو داریم، در این مثال می‌بینیم که الگوی چگونه می‌تواند برای ایجاد یک رابط کاربری cross-platform به ما کمک کند تا وابستگی بین کدهای Client و UI را کم کنیم.



همانطور که می‌دانیم انتظار می‌رود تا اینمانهای UI در یک برنامه‌ی cross-platform، رفتارهای مشابه داشته باشند. ولی بباید کمی متفاوت به قضیه نگاه کنیم و به این موضوع فکر کنیم که هر کدام از این المانها در سیستم عامل‌های مختلف به چه شکلی باید نمایش داده شوند.

وظیفه‌ی ما این است که المانها مطابق با سیستم عامل کاربر نمایش داده شوند. ما نمی‌خواهیم کسی که برنامه‌مان را روی ویندوز اجرا می‌کند المانهای macOS را ببیند!

واسط Abstract Factory مجموعه‌ای از متدهای مختلفی در خود تعریف می‌کند تا پاسخگوی نیازهای client در شرایط متفاوت باشد. Factory هایی که در برنامه ایجاد کرده ایم مطابق با هر کدام از سیستم عامل‌ها می‌باشند و المان‌های برنامه را مطابق با سیستم عامل کاربر تولید می‌کنند.

برنامه به این شکل کار می‌کند: زمانی که برنامه اجرا می‌شود، در ابتدا نوع سیستم‌عامل کاربر را بررسی می‌کند و از این اطلاعات استفاده می‌کند تا کارخانه‌ای از اشیاء کلاس‌هایی که با سیستم عامل کاربر همخوانی دارند

ایجاد کند. بقیهی کدها از این کارخانه برای ایجاد المان‌های Al استفاده می‌کنند و این باعث جلوگیری از بروز اشتباه در کدمان می‌شود.

با این رویکرد، دیگر کدهای client به کلاس‌های factory و نوع رابط کاربری‌مان وابسته نیستند و این کار باعث می‌شود تا دستمنان برای گسترش برنامه و اضافه کردن المان‌ها در آینده باز باشد.

در نتیجه شما با هر بار اضافه کردن یک المان نیاز به تغییر کدهای client ندارید. فقط یک factory می‌کنید و المان‌های جدید را تولید می‌کنید.

در ادامه شبه کدی از این مثال را خواهیم داشت :

```
// The abstract factory interface declares a set of methods that
// return different abstract products. These products are called
// a family and are related by a high-level theme or concept.
// Products of one family are usually able to collaborate among
// themselves. A family of products may have several variants,
// but the products of one variant are incompatible with the
// products of another variant.

interface GUIFactory {
    method createButton() : Button
    method createCheckbox() : Checkbox

    // Concrete factories produce a family of products that belong
    // to a single variant. The factory guarantees that the
    // resulting products are compatible. Signatures of the concrete
    // factory's methods return an abstract product, while inside
    // the method a concrete product is instantiated.

class WinFactory implements GUIFactory {
    method createButton() : Button is
        return new WinButton()
    method createCheckbox() : Checkbox is
        return new WinCheckbox()

    // Each concrete factory has a corresponding product variant.

class MacFactory implements GUIFactory {
    method createButton() : Button is
        return new MacButton()
    method createCheckbox() : Checkbox is
        return new MacCheckbox()

    // Each distinct product of a product family should have a base
    // interface. All variants of the product must implement this
    // interface.

interface Button is
    method paint()
```

```

// Concrete products are created by corresponding concrete
// factories.
class WinButton implements Button is
    method paint() is
        // Render a button in Windows style.

class MacButton implements Button is
    method paint() is
        // Render a button in macOS style.

// Here's the base interface of another product. All products
// can interact with each other, but proper interaction is
// possible only between products of the same concrete variant.
interface Checkbox is
    method paint()

class WinCheckbox implements Checkbox is
    method paint() is
        // Render a checkbox in Windows style.

class MacCheckbox implements Checkbox is
    method paint() is
        // Render a checkbox in macOS style.

// The client code works with factories and products only
// through abstract types: GUIFactory, Button and Checkbox. This
// lets you pass any factory or product subclass to the client
// code without breaking it.
class Application is
    private field factory: GUIFactory
    private field button: Button
    constructor Application(factory: GUIFactory) is
        this.factory = factory
    method createUI() is
        this.button = factory.createButton()
    method paint() is
        button.paint()

// The application picks the factory type depending on the
// current configuration or environment settings and creates it
// at runtime (usually at the initialization stage).
class ApplicationConfigurator is
    method main() is
        config = readApplicationConfigFile()

        if (config.OS == "Windows") then

```

```

        factory = new WinFactory()
else if (config.OS == "Mac") then
    factory = new MacFactory()
else
    throw new Exception("Error! Unknown operating
system.")

Application app = new Application(factory)

```

## چه زمانی باید از این الگو استفاده کنیم؟

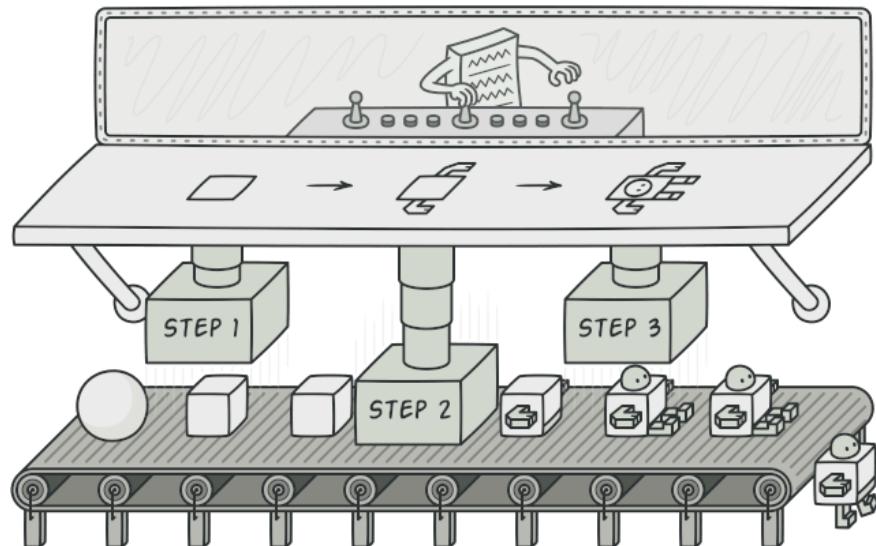
- زمانی که کدتان باید با انواع مختلفی از اشیاء که در category های مختلفی هستند سر و کار داشته باشد و نمی‌خواهید که به کلاس‌های مشخصی از آن دسته وابستگی داشته باشد. ممکن است که ندانید این اشیاء بعدها جزو چه دسته‌ای قرار می‌گیرند و می‌خواهید با این کار قابلیت گسترش‌پذیری برنامه‌تان را بالا ببرید.  
این الگو واسطی در اختیارمان قرار می‌دهد تا از هر کلاس در هر دسته‌ای از اشیاء دلخواه‌مان را بسازیم. تا زمانی که این واسط وظیفه‌ی ایجاد اشیاء را بر عهده دارد دیگر نگران بروز اشتباهات احتمالی در ساخت اشیاء نخواهید بود.
- زمانیکه کلاسی شامل لیستی از Factory method ها داریم که باعث کمرنگ شدن مسئولیت اصل کلاس‌مان می‌شود.  
در برنامه‌ای با طراحی خوب، هر کلاسی وظیفه‌ی انجام یک کار را دارد. زمانی که کلاسی تعداد زیادی از product ها را در خود داشته باشد، اینکه هر کدام از factory method های آن را در کلاسی مجزا پیاده‌سازی کنیم ایده‌ی بهتری است.

## معایب و مزایا

- ❖ این تضمین وجود دارد که اشیائی که از یک factory می‌گیریم همگی با هم سازگارند.
- ❖ وابستگی بین کد client و product های اصلی‌مان را کم می‌کند.
- ❖ با این کار اصل اول SOLID یعنی Single Responsibility رعایت می‌شود. چرا که مراحل ایجاد product یک جا قرار می‌گیرد و نگهداری و پشتیبانی آن راحت‌تر می‌شود.
- ❖ با استفاده از این الگو اصل دوم SOLID یعنی Open/Closed نیز رعایت می‌شود. چرا که با استفاده از این الگو می‌توانیم انواع مختلفی از product ها را بدون اینکه اشیاء موجود در برنامه را تغییر دهیم به برنامه اضافه کنیم.
- از معایب این الگو این است که کدهای زیادی باید نوشته شود. زیر-کلاس‌های زیادی باید ایجاد شوند تا این الگو پیاده‌سازی شود. بهترین سناریو برای این الگو زمانی است که سلسله مراتب مشخصی از کلاسها و زیر-کلاس‌هایمان داشته باشیم.

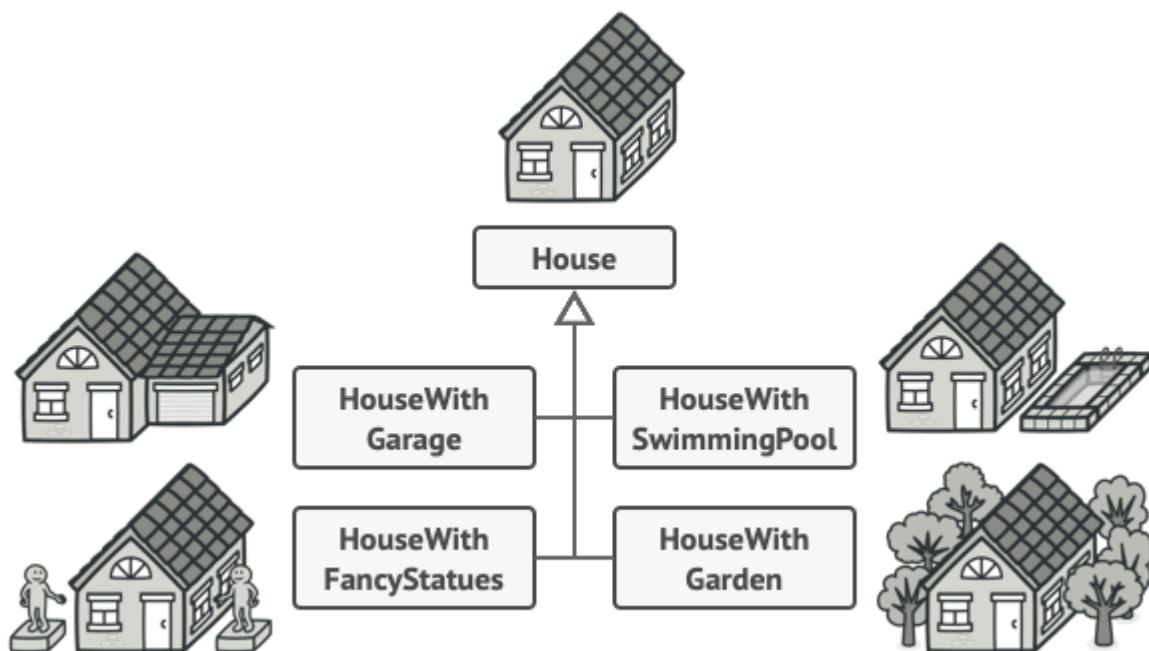
## Builder

این الگو به ما کمک می‌کند تا اشیاء پیچیده را مرحله به مرحله بسازیم و این اجازه را می‌دهد تا انواع مختلفی از اشیاء را، با کدهایی با ساختارهای یکسان ایجاد کنیم.



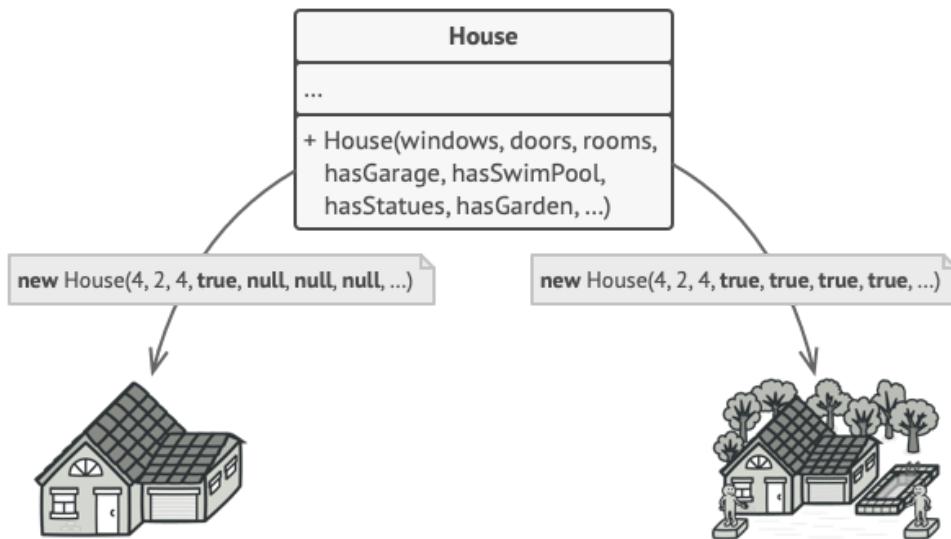
### طرح مسئله

مراحل زیاد و پر زحمتی که برای ایجاد یک Object که فیلدهای زیادی دارد را تصور کنید. برای نوشتن کدهای مربوط به این object constructor های تو در تو با پارامترهای زیادی بنویسید. به این فکر کنید که در بدترین حالت این کدها در همهی قسمت‌های کد سمت client مان پخش شده باشد !



باید راجع به اینکه چگونه یک Object خانه (House) بسازیم فکر کنیم. برای ساختن یک خانه‌ی ساده، نیاز به ساختن چهار دیوار، نصب در، گذاشتن دو پنجره و یک سقف دارید. اما اگر بخواهیم خانه‌ای بزرگتر و روشن‌تر با حیاط‌خلوت و دیگر امکاناتی مثل سیستم گرمایشی، لوله‌کشی، سیم‌کشی برق و ... داشته باشیم چه باید بکنیم؟

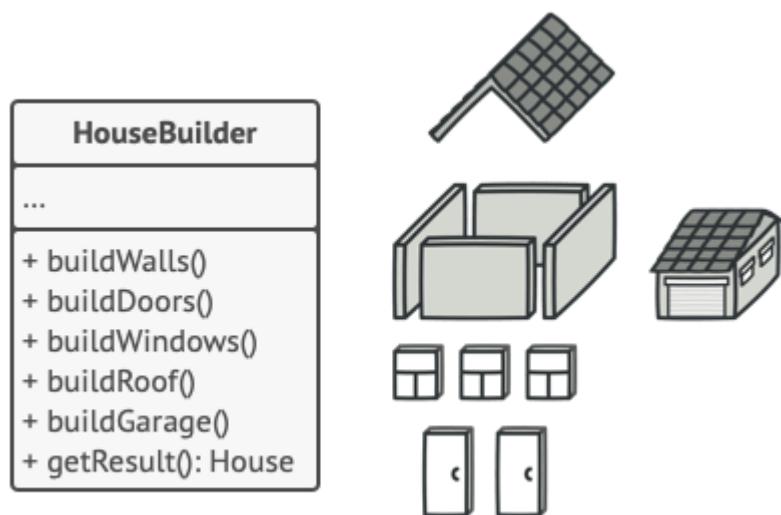
ساده‌ترین کاری که می‌توان انجام داد این است که کلاسی بسازیم تا کلاس House را extend کند و همه‌ی پارامترهای مورد نیاز را برآورده کند. اما در نهایت با تعداد قابل توجهی از زیر-کلاس موافق خواهد شد! اضافه کردن هر پارامتری مانند اضافه کردن ایوان به خانه، باعث بیشتر شدن این سلسله مراتب می‌شود. رویکرد دیگری هم وجود دارد که باعث زیاد کردن تعداد زیر-کلاس‌ها نمی‌شود. می‌توانیم یک کلاس پایه‌ی غولپیکر بسازیم و در آن Constructor ای قرار دهیم که تمامی پارامترهای احتمالی در نظر گرفته شده باشد تا یک شیء House ساخته شود.



در بیشتر مواقع تعداد زیادی از این پارامترها بلا استفاده خواهند بود و فراخوانی سازنده بسیار کار بیهوده‌ای خواهد بود! بطور مثال خانه‌های بسیار کمی دارای استخر هستند، با این حساب پارامتر مربوط به استخر از هر ۱۰ بار ۹ بار بلا استفاده خواهد بود! و اما راه حل؟

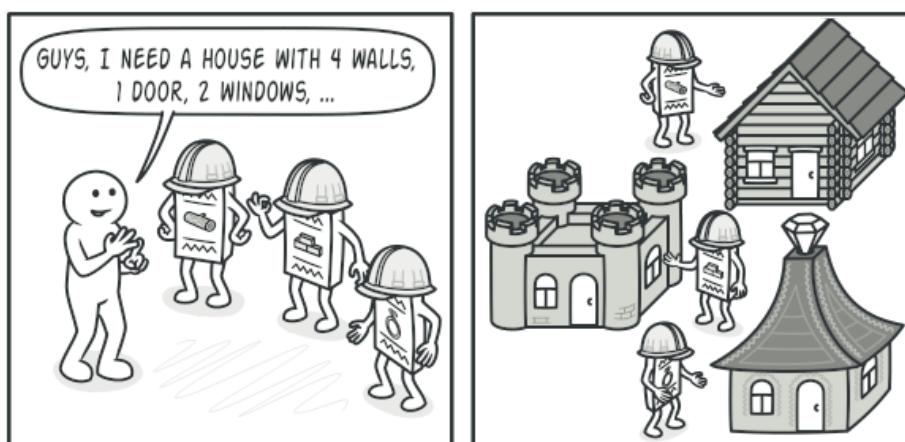
## راه حل

راه حلی که الگوی Builder پیشنهاد می‌دهد این است که فرایند ساختن اشیاء را به کلاس مختص خود بسپاریم و به اشیاء جدا از همی به اسم builders انتقال دهیم.



این الگو فرایند ساخته شدن اشیاء را به مجموعه‌ای از مراحل مانند buildWalls , buildDoor , buildWindow ... سازماندهی می‌کند. برای ایجاد یک شیء یک سری از این مراحل بر روی شیء Builder اجرا خواهد شد. مهمترین قسمت کار این است که شما مجبور به فراخوانی همه‌ی مراحل نیستید و می‌توانید فقط مراحلی که نیاز دارید را فراخوانی کنید و یک رویکرد به خصوص را برای ایجاد یک شیء اجرا کنید.

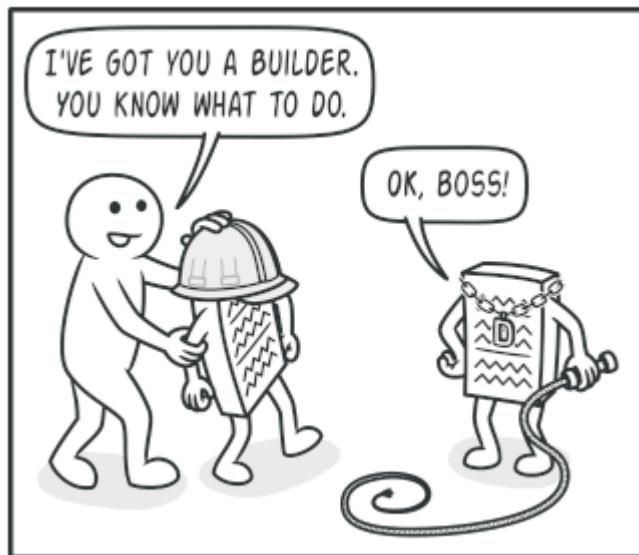
بعضی از مراحل ساختن اشیاء ممکن است که به پیاده‌سازی‌های متفاوتی نیاز داشته باشند. بطور مثال دیوارهای یک کابین می‌توانند از چوب ساخته شوند ولی دیوارهای یک قلعه باید از سنگ ساخته شوند! برای حل این مورد می‌توانیم هر مقداری که نیاز داریم کلاس های builder متفاوتی بسازیم و مجموعه‌ای از مراحل building را در آنها قرار دهیم و در طی مراحل ساخت یک شیء از آنها استفاده کنیم.



بطور مثال فرض کنید که سازنده‌ی اول فقط همه چیز را با چوب و شیشه، سازنده‌ی دوم همه چیز را با سنگ و آهن و سازنده‌ی سوم همه چیز را با طلا و الماس می‌سازد. با فراخوانی مجموعه‌ای از این مراحل، یک خانه‌ی ساخته شده از چوب از سازنده‌ی اول، یک قلعه ساخته شده از سنگ و آهن از سازنده‌ی دوم و یک کاخ ساخته شده از طلا و الماس از سازنده‌ی سوم می‌گیرید. این فرایندها زمانی امکان پذیر است که کدهای ما بتوانند توسط یک واسطه با builder‌ها ارتباط برقرار کنند.

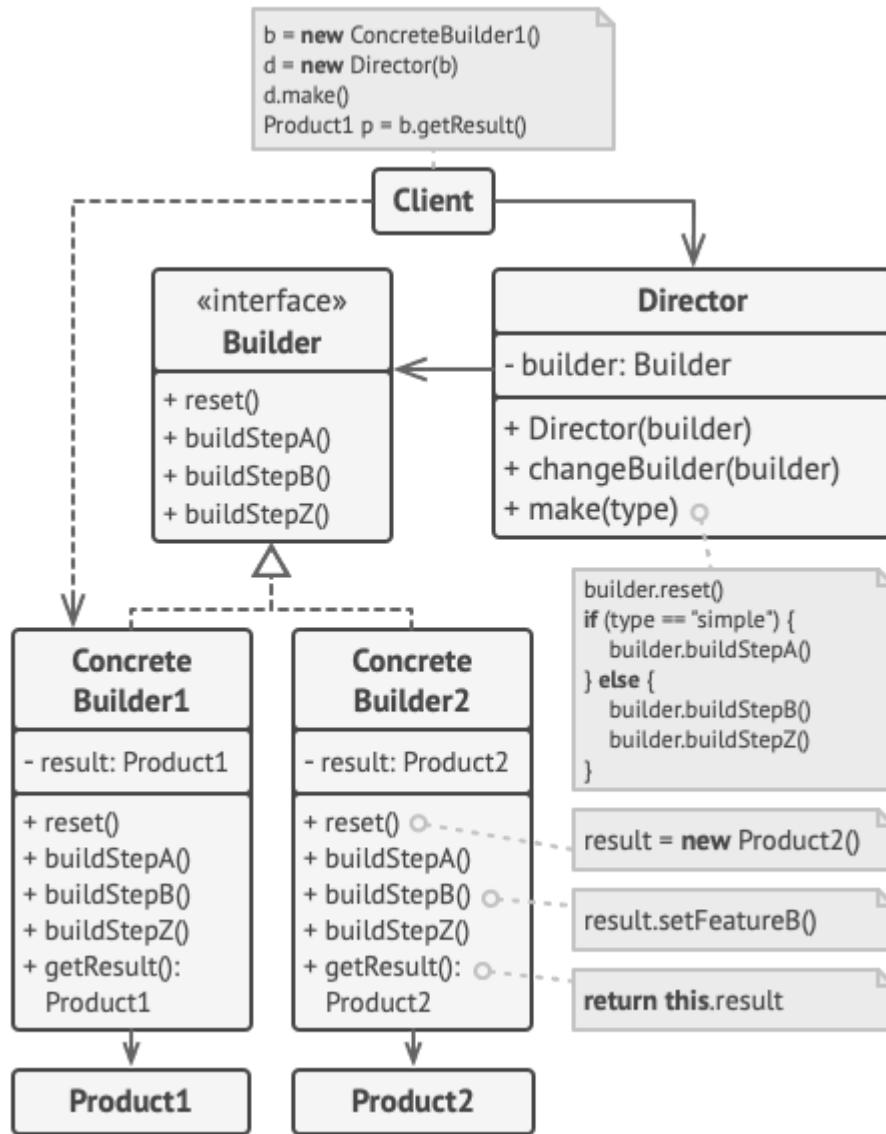
### Director

با رویکردی دیگر می‌توان یک سری از مراحل ساخت اشیاء را در کلاسی تحت عنوان Director قرار داد. این کلاس یک ترتیب برای اجرای مراحل در نظر می‌گیرد و builder‌نحوه‌ی پیاده‌سازی را مشخص می‌کند.



البته که داشتن یک کلاس director در برنامه تان کاملا ضروری نیست. همیشه می‌توان مراحل ساختن اشیاء را با ترتیب‌های دلخواه از client فراخوانی کرد. در هر حال کلاس director مکان خوبی برای قرار دادن مراحل ساخت انواع اشیاء می‌باشد تا بتوان در هر جای برنامه از آنها استفاده کرد.

علاوه بر این کلاس director به خوبی جزئیات مراحل ساخت محصولات را از دید کدهای client پنهان می‌کند. Client فقط نیاز دارد تا builder و director را به هم مرتبط کند. با استفاده از director مراحل ساختن محصول را انجام می‌دهد و با استفاده از builder نتیجه را دریافت می‌کند.

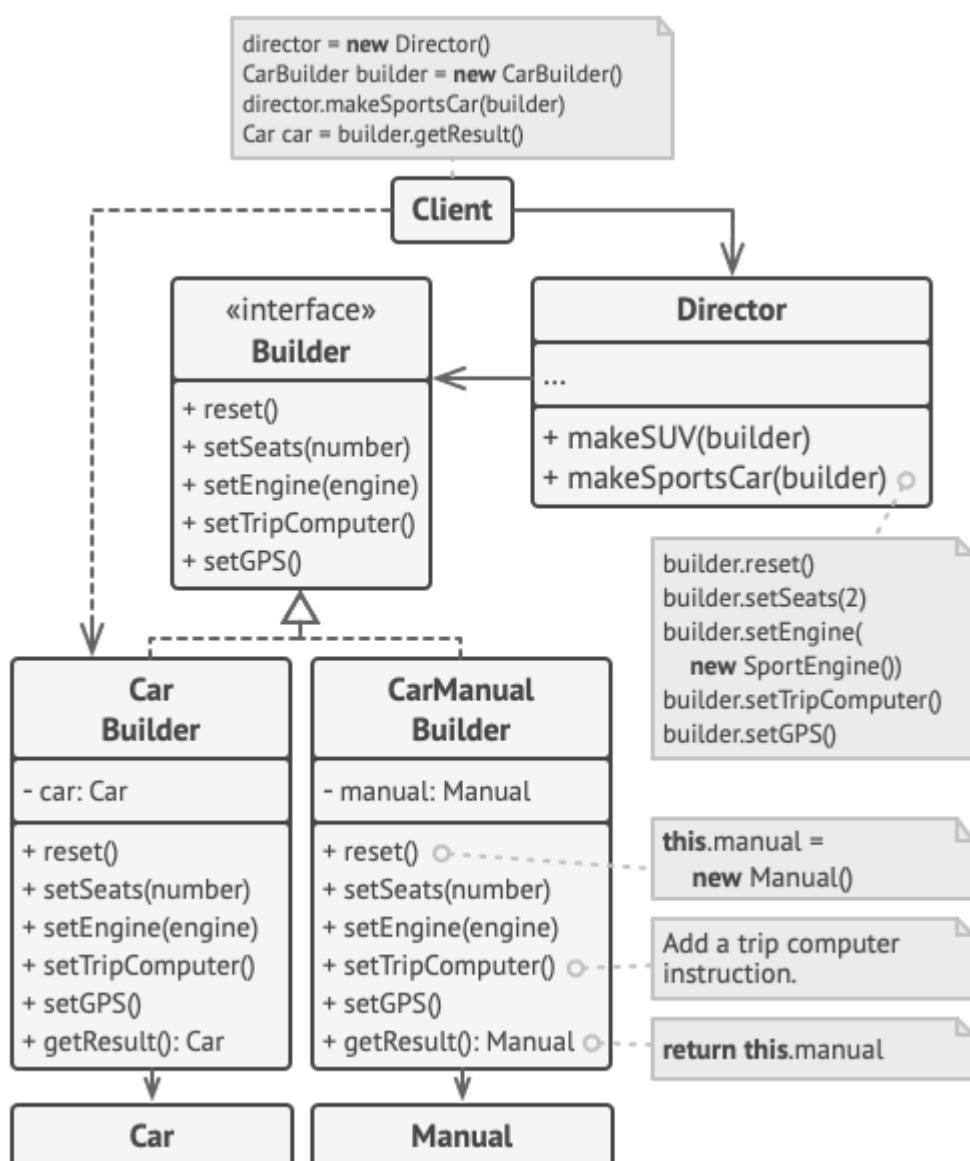


- **builder:** این واسط مراحل ساخت یک محصول که بین همه محصولات مشترک است را نمایش می‌دهد.
- **Concrete Builders:** پیاده‌سازی‌های متفاوت هر محصول در این کلاس انجام می‌شود. این کلاس ممکن است محصولاتی تولید کند که از Interface عمومی تبعیت نکنند.
- **Products:** این کلاسها محصولات نهایی هستند که توسط builder‌های متفاوتی ایجاد شده‌اند و متعلق به هیچ سلسله مرتب یا واسطی نیستند.
- **Director:** ترتیب مراحل فرآخوانی ساخت اشیاء را تعریف می‌کند.
- **Client:** این کلاس باید یکی از اشیاء builder را با director ارتباط دهد. این کار فقط یکبار استفاده از سازنده‌ی director انجام می‌شود. سپس director از آن شیء builder برای ساختن

اشیاء استفاده می‌کند. هر چند نگاهی متفاوت هم وجود دارد آن هم زمانی است که client شیء director را به متدهد director در production پاس می‌دهد. با این کار می‌توانید هر زمان که با استفاده از یک director ایجاد می‌کنید از یک builder متفاوت استفاده کنید.

## مثال

در این مثال می‌بینیم که چگونه از یک روند به خصوص برای ایجاد انواع مختلفی از اشیاء استفاده می‌کنیم. در ادامه روند ایجاد یک ماشین به همراه دفترچه راهنمای آن را پیاده سازی خواهیم کرد:



می‌دانیم که ماشین می‌تواند یک شیء پیچیده باشد و می‌توان با هزاران روش آن را پیاده‌سازی کرد. بجای اینکه کلاس ماشین را با سازنده‌های غول پیکر و حجمی پر کنیم، مراحل ایجاد یک ماشین را در کلاسی جدا تحت عنوان `builder` می‌نویسیم. این کلاس شامل متدهایی برای درست کردن هر کدام از بخش‌های ماشین می‌باشد. هر زمان `client` تصمیم بگیرد که قابلیت جدیدی به ماشین اضافه شود، می‌تواند بطور مستقیم با کلاس `builder` در تماس باشد و تغییرات موردنظر خود را اعمال کند. از طرفی `client` می‌تواند مراحل ایجاد

شیء را به کلاس `director` که می‌داند چگونه ماشین‌هایی با مدل‌های متنوع بسازد واگذار کند. شاید باور نکنید، ولی هر ماشینی به یک دفترچه راهنمای نیاز دارد(!). این دفترچه راهنمای تمامی قابلیتهای ماشین را توضیح می‌دهد. بنابراین جزئیات این دفترچه برای هر ماشین متفاوت است.

چیزی که مشخص است این است که فرایند ایجاد یک ماشین با فرایند ایجاد یک دفترچه راهنمای متفاوت است! و اینجا همان جایی است که به یک کلاس `builder` دیگر نیاز پیدا می‌کنید تا بصورت خاص بر روی دفترچه راهنمای کار کند. این کلاس کارهایی مشابه کلاس `builder` که برای ماشین است انجام می‌دهد ولی به جای ساخت قطعات خودرو، آنها را توصیف می‌کند.

با ارسال هر دوی این `builder`‌ها به کلاس `director` می‌توانیم یک ماشین به همراه دفترچه راهنمایش را بسازیم. در پایان هم کافیست تا اطلاعات را دریافت کنیم. ماشینی از جنس فلز و دفترچه راهنمایی از جنس کاغذ در عین حالی که خیلی به هم شبیه اند، به همان اندازه هم با هم متفاوت اند.

برای درک بیشتر موضوع بهتر است به مثال زیر توجه کنید :

```
// Using the Builder pattern makes sense only when your products
// are quite complex and require extensive configuration. The
// following two products are related, although they don't have
// a common interface.

class Car is
    // A car can have a GPS, trip computer and some number of
    // seats. Different models of cars (sports car, SUV,
    // cabriolet) might have different features installed or
    // enabled.

class Manual is
    // Each car should have a user manual that corresponds to
    // the car's configuration and describes all its features.

// The builder interface specifies methods for creating the
// different parts of the product objects.

interface Builder is
    method reset()
    method setSeats(...)
    method setEngine(...)
    method setTripComputer(...)
    method setGPS(...)

// The concrete builder classes follow the builder interface and
```

```

// provide specific implementations of the building steps. Your
// program may have several variations of builders, each
// implemented differently.
class CarBuilder implements Builder {
    private field car:Car

    // A fresh builder instance should contain a blank product
    // object which it uses in further assembly.
    constructor CarBuilder() {
        this.reset()
    }

    // The reset method clears the object being built.
    method reset() {
        this.car = new Car()
    }

    // All production steps work with the same product instance.
    method setSeats(...) {
        // Set the number of seats in the car.
    }

    method setEngine(...) {
        // Install a given engine.
    }

    method setTripComputer(...) {
        // Install a trip computer.
    }

    method setGPS(...) {
        // Install a global positioning system.
    }

    // Concrete builders are supposed to provide their own
    // methods for retrieving results. That's because various
    // types of builders may create entirely different products
    // that don't all follow the same interface. Therefore such
    // methods can't be declared in the builder interface (at
    // least not in a statically-typed programming language).
    //

    // Usually, after returning the end result to the client, a
    // builder instance is expected to be ready to start
    // producing another product. That's why it's a usual
    // practice to call the reset method at the end of the
    // `getProduct` method body. However, this behavior isn't
    // mandatory, and you can make your builder wait for an
    // explicit reset call from the client code before disposing
    // of the previous result.
    method getProduct():Car {
        product = this.car
        this.reset()
        return product
    }
}

```

```

// Unlike other creational patterns, builder lets you construct
// products that don't follow the common interface.
class CarManualBuilder implements Builder is
    private field manual:Manual

    constructor CarManualBuilder() is
        this.reset()

    method reset() is
        this.manual = new Manual()

    method setSeats(...) is
        // Document car seat features.

    method setEngine(...) is
        // Add engine instructions.

    method setTripComputer(...) is
        // Add trip computer instructions.

    method setGPS(...) is
        // Add GPS instructions.

    method getProduct():Manual is
        // Return the manual and reset the builder.

// The director is only responsible for executing the building
// steps in a particular sequence. It's helpful when producing
// products according to a specific order or configuration.
// Strictly speaking, the director class is optional, since the
// client can control builders directly.
class Director is
    // The director works with any builder instance that the
    // client code passes to it. This way, the client code may
    // alter the final type of the newly assembled product.
    // The director can construct several product variations
    // using the same building steps.
    method constructSportsCar(builder: Builder) is
        builder.reset()
        builder.setSeats(2)
        builder.setEngine(new SportEngine())
        builder.setTripComputer(true)
        builder.setGPS(true)

    method constructSUV(builder: Builder) is
        // ...

```

```

// The client code creates a builder object, passes it to the
// director and then initiates the construction process. The end
// result is retrieved from the builder object.
class Application is

    method makeCar() is
        director = new Director()

        CarBuilder builder = new CarBuilder()
        director.constructSportsCar(builder)
        Car car = builder.getProduct()

        CarManualBuilder builder = new CarManualBuilder()
        director.constructSportsCar(builder)

        // The final product is often retrieved from a builder
        // object since the director isn't aware of and not
        // dependent on concrete builders and products.
        Manual manual = builder.getProduct()

```

چه زمانی باید از این الگو استفاده کنیم؟

- از این الگو برای خلاص شدن از شرّ سازنده‌های تلسکوپی و تو در تو استفاده می‌کنیم.
- فرض کنید سازنده‌ایی با ده پارامتر اختیاری داریم. فراخوانی همچین سازنده‌ایی به شدت اذیت کننده است. به همین دلیل این سازنده را overload می‌کنیم و سازنده‌های کوتاه‌تری می‌سازیم. همه‌ی این سازنده‌ها همچنان به سازنده اصلی ارجاع می‌دهند.

```

class Pizza {
    Pizza(int size) { ... }
    Pizza(int size, boolean cheese) { ... }
    Pizza(int size, boolean cheese, boolean pepperoni) {
    ...
    // ...
}

```

الگوی طراحی builder این قابلیت را در اختیارمان قرار می‌دهد تا اشیاء مان را مرحله به مرحله و فقط با استفاده از مواردی که نیاز داریم بسازیم. بعد از پیاده‌سازی این الگوریتم دیگر نیازی نیست سازنده‌ایی با پارامترهای فراوان داشته باشد!

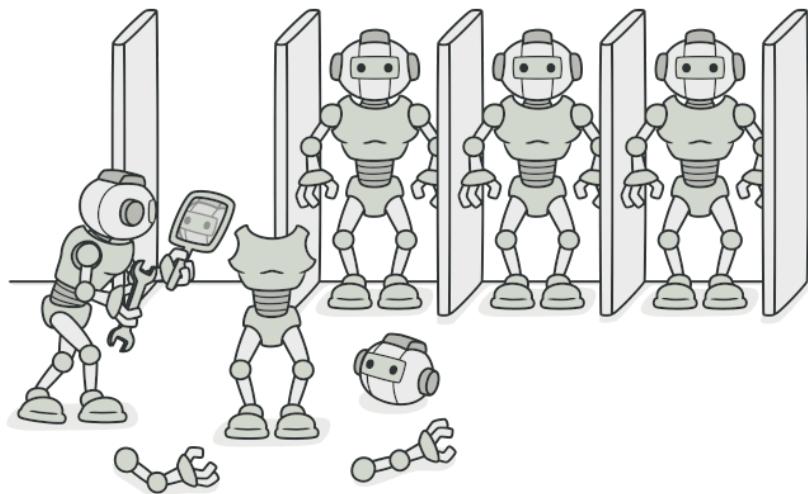
- از این الگو زمانی استفاده کنید که می‌خواهید کدتان این قابلیت را داشته باشد تا بتواند حالتها و نسخه‌های مختلفی از یک محصول را بسازد. در مثال ساخت خانه مشاهده کردیم که یک خانه می‌تواند از مواد اولیه با جنس‌های متفاوت مثل چوب و شیشه و فلز ساخته شود.
- وقتی از این الگو استفاده می‌کنیم که مراحل ساخت یکسان است اما جزئیات متفاوت است.
- این الگو تا زمانی که یک محصول به طور کامل ایجاد نشود آن را برای client نمی‌فرستد. بنابراین هیچ‌گاه با یک محصول ناقص در سمت client مواجه نخواهیم شد.

### معایب و مزایا

- ❖ می‌تواند اشیاء خود را مرحله به مرحله ایجاد کنید یا مراحل را به صورت بازگشتنی پیاده سازی کنید.
  - ❖ می‌توانید برای ساخت انواع مختلف اشیاء از یک سازنده استفاده کنید.
  - ❖ با این کار اصل اول SOLID یعنی Single Responsibility می‌شود. در این الگو کدهای پیچیده‌ی سازنده‌ها از منطق برنامه برای هر محصول جدا می‌شوند.
- پیچیدگی کلی کد افزایش پیدا می‌کند، زیرا در این الگو کلاس‌های زیادی ساخته می‌شود.

## Prototype

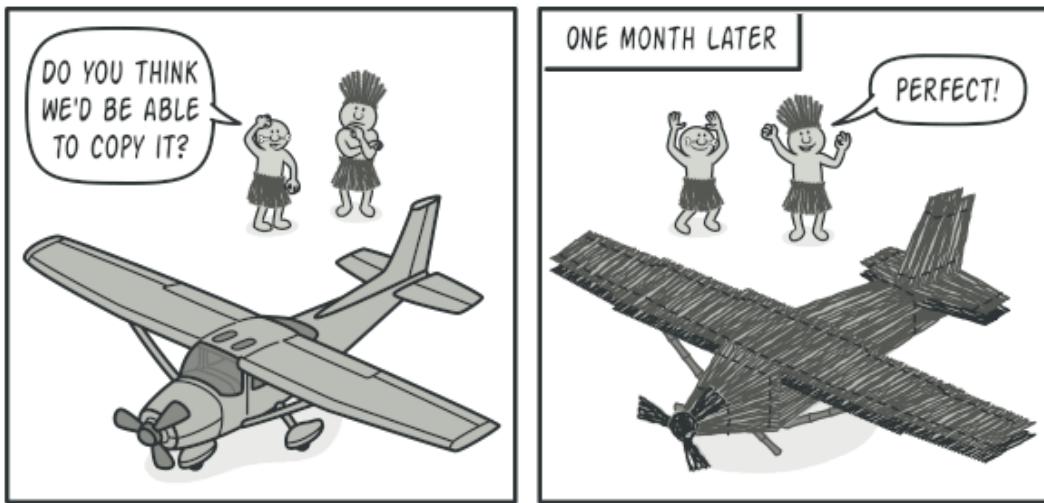
این الگوی طراحی به ما کمک می‌کند تا بدون اینکه به کلاس‌های اصلی وابستگی ای داشته باشیم، از آن‌ها کپی بگیریم!



### طرح مسئله

فرض کنید یک Object و می‌خواهید یک کاملا مشابه آن داشته باشید. چگونه این کار را انجام می‌دهید؟ در ابتدا باید یک Object از شی مورد نظر بسازید و سپس مقدار تمام فیلدهایش را در شیء جدید کپی کنید.

خیلی هم عالی! اما یک مشکل وجود دارد. اینکه تمامی Object‌ها به همین سادگی قابل کپی شدن نیستند. بعضی از Object‌ها فیلدهایی با سطح دسترسی private دارند که از خارج آن کلاس قابل مشاهده نیستند.



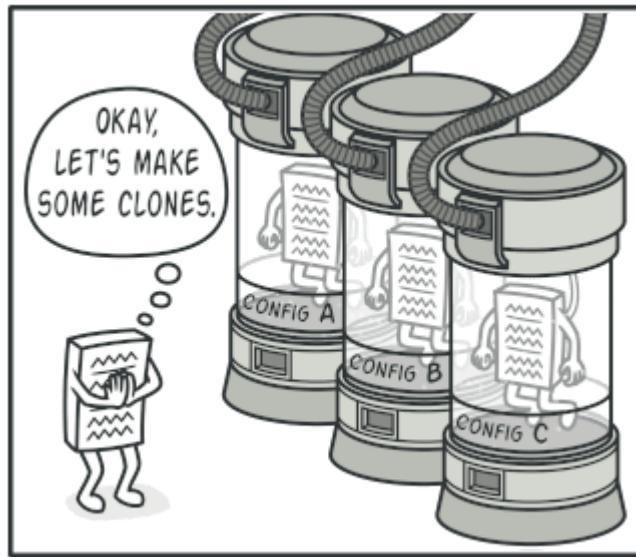
یک مشکل دیگر هم وجود دارد. اینکه در هر باری که نیازمند یک کپی از یک کلاس باشید، کد شما به آن کلاس وابسته می‌شود. اگر این مشکل شما را نمی‌ترساند، یک مشکل دیگر هم وجود دارد! گاهی اوقات شما فقط یک interface را می‌شناسید که Object مورد نظر آن را پیاده سازی می‌کند ولی اطلاعات دقیقی از اینکه آن Object چگونه است ندارید. بطور مثال در ورودی یکی از متدهایتان یک Interface پاس داده می‌شود. بنابراین این متدهر Object ای که از جنس آن interface باشد را به عنوان آرگومان ورودی خودش قبول می‌کند. شما در داخل آن متده به صورت خاص نمی‌دانید که Object ورودی چه دیتایی دارد فقط می‌دانید از چه جنسی است. برای حل این مشکل‌ها چه باید کرد؟

## راه حل

الگوی طراحی prototype فرایند کپی گرفتن را به خود اشیاء اصلی واگذار می‌کند! این الگو یک Interface برای تمامی Object هایی که قابلیت کپی شدن را دارند ایجاد می‌کند. این الگو این امکان را می‌دهد تا بدون اینکه به کلاس Object مورد نظر وابستگی‌ای ایجاد شود، از آن یک کپی بگیرید. معمولاً این Interface یک متده تحت عنوان `clone()` دارد.

پیاده‌سازی این متده تقریباً در تمامی کلاس‌ها مشابه یکدیگر است. این متده یک Object از کلاس فعلی می‌سازد و مقدار تمامی فیلدهای کلاس را در شیء جدید قرار می‌دهد. حتی می‌تواند فیلدهای با سطح دسترسی `private` را نیز کپی کنید.

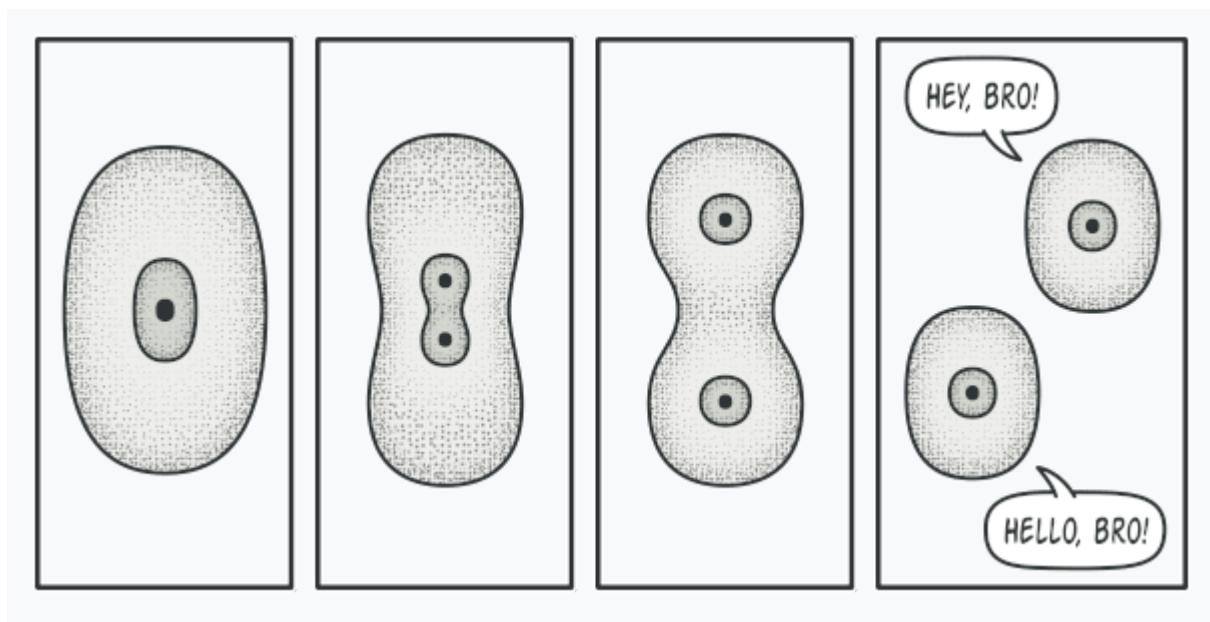
به Object ای که قابلیت cloning داشته باشد، prototype می‌گوییم. زمانی که کلاس‌هایتان دارای صدھا فیلد باشد، کپی گرفتن از آن راهکار بهتری نسبت به ایجاد زیر-کلاس از آن می‌باشد. بجای اینکه یک کلاس جدید بسازید، می‌توانید از شیء موجود یک کپی بگیرید و آن کپی همهی اطلاعات کلاس اولیه را دارد.



نحوه‌ی کار: مجموعه‌ای از Object ها را با انواع Configuration ها ایجاد می‌کنید. زمانی که به هر کدام از این Object ها نیاز پیدا کردید، به جای اینکه از آنها یک شیء جدید بسازید، از آنها یک کپی می‌گیرید.

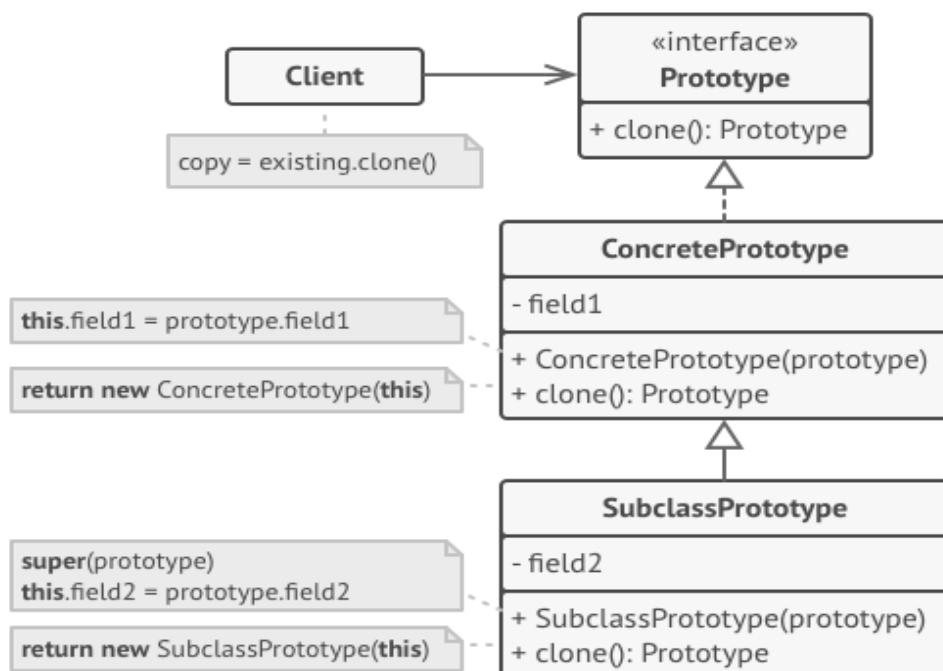
### مقایسه با دنیای واقعی

قبل از اینکه محصولی به تولید انبوه برسد در ابتدا از یک سری نمونه‌های آزمایشی برای تست آن محصول استفاده می‌شود. این محصول های تستی واقعی نیستند و در واقع یک کپی از محصول اصلی اند. محصول اصلی هیچ نقشی در این آزمایش ایفا نخواهد کرد. با این کار در فرایند تست هیچ آسیبی به آن نخواهد رسید.



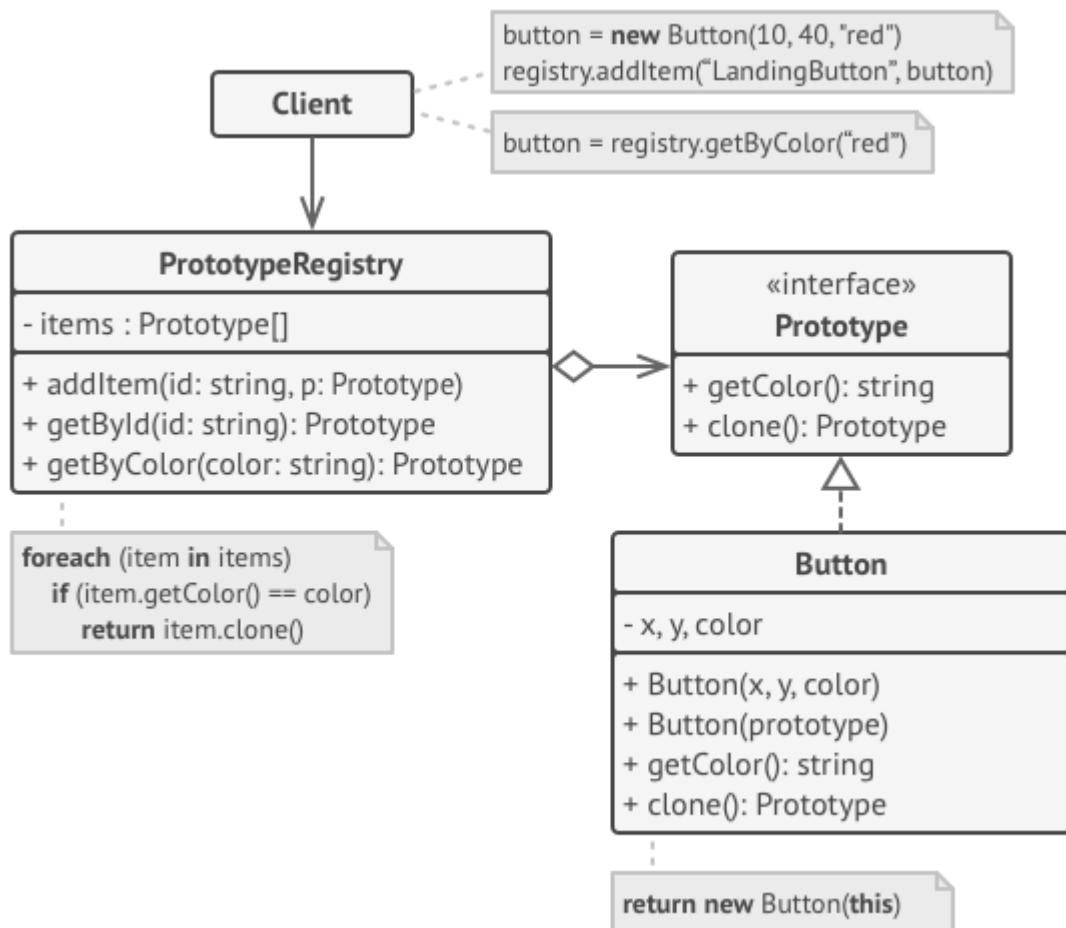
در دنیای واقعی نمونه‌ها واقعاً خودشان را کپی نمی‌کنند! اگر بخواهیم این الگو را با دنیای واقعی مقایسه کنیم، همانند تقسیم میتوز در سلول‌ها می‌ماند! (امیدوارم زیست‌شناسی را به یاد داشته باشید) پس از تقسیم میتوزی، یک جفت سلول یکسان تشکیل می‌شود. سلول اصلی به عنوان یک نمونه اولیه عمل می‌کند و نقش فعالی در ایجاد کپی دارد.

## ساختار



- در این **interface** متدها نوشته می‌شوند. که معمولاً هم دارای یک متد با نام `clone` هستند.
- این کلاس متد `clone` را پیاده‌سازی می‌کند. علاوه بر کپی کردن داده‌های اصلی، بعضی فرایندهای جانبی برای کپی کردن را هم در خود پیاده‌سازی می‌کند.
- این کلاس می‌تواند از هر کلاسی که واسط **Prototype** را پیاده‌سازی می‌کند یک کپی ایجاد کند.

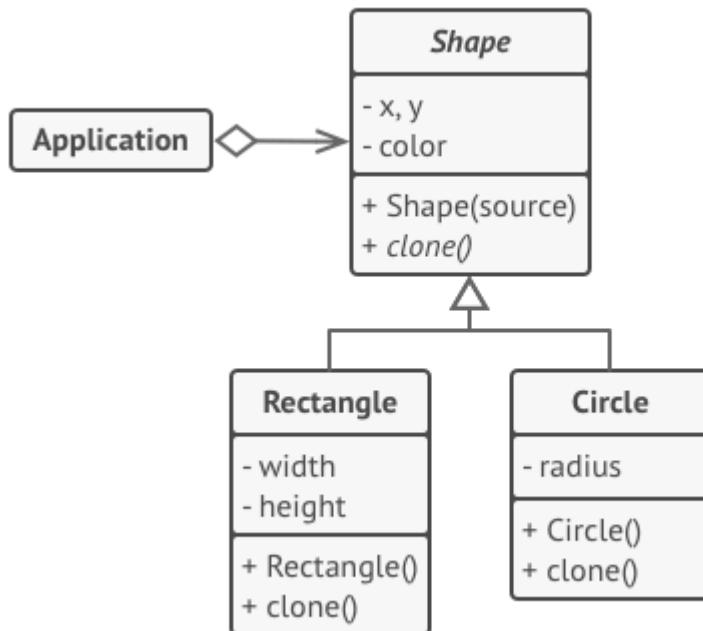
## پیاده‌سازی نمونه اولیه Prototype



این کلاس دسترسی به Prototype هایی که زیاد از آنها استفاده می‌شود را آسان می‌کند. تعدادی از اشیاء از پیش ساخته شده را در خود نگه می‌دارد که همیشه آماده‌ی کپی برداری هستند.

## مثال

در این مثال الگوی طراحی Prototype این قابلیت را به ما می‌دهد تا بدون این که به کلاس اشکال هندسی وابسته باشیم، از آنها یک کپی بگیریم.



تمام شکل‌ها یک interface را پیاده‌سازی می‌کنند، که آن دارای متد clone می‌باشد.

```

// Base prototype.
abstract class Shape is
    field X: int
    field Y: int
    field color: string

    // A regular constructor.
    constructor Shape() is
        // ...

    // The prototype constructor. A fresh object is initialized
    // with values from the existing object.
    constructor Shape(source: Shape) is
        this()
        this.X = source.X
        this.Y = source.Y
        this.color = source.color

    // The clone operation returns one of the Shape subclasses.

```

```

abstract method clone():Shape

// Concrete prototype. The cloning method creates a new object
// in one go by calling the constructor of the current class and
// passing the current object as the constructor's argument.
// Performing all the actual copying in the constructor helps to
// keep the result consistent: the constructor will not return a
// result until the new object is fully built; thus, no object
// can have a reference to a partially-built clone.
class Rectangle extends Shape is
    field width: int
    field height: int

    constructor Rectangle(source: Rectangle) is
        // A parent constructor call is needed to copy private
        // fields defined in the parent class.
        super(source)
        this.width = source.width
        this.height = source.height

    method clone():Shape is
        return new Rectangle(this)

class Circle extends Shape is
    field radius: int

    constructor Circle(source: Circle) is
        super(source)
        this.radius = source.radius

    method clone():Shape is
        return new Circle(this)

// Somewhere in the client code.
class Application is
    field shapes: array of Shape

    constructor Application() is
        Circle circle = new Circle()
        circle.X = 10
        circle.Y = 10
        circle.radius = 20
        shapes.add(circle)

        Circle anotherCircle = circle.clone()

```

```

shapes.add(anotherCircle)
// The `anotherCircle` variable contains an exact copy
// of the `circle` object.

Rectangle rectangle = new Rectangle()
rectangle.width = 10
rectangle.height = 20
shapes.add(rectangle)

method businessLogic() is
// Prototype rocks because it lets you produce a copy of
// an object without knowing anything about its type.
Array shapesCopy = new Array of Shapes.

// For instance, we don't know the exact elements in the
// shapes array. All we know is that they are all
// shapes. But thanks to polymorphism, when we call the
// `clone` method on a shape the program checks its real
// class and runs the appropriate clone method defined
// in that class. That's why we get proper clones
// instead of a set of simple Shape objects.
foreach (s in shapes) do
    shapesCopy.add(s.clone())

// The `shapesCopy` array contains exact copies of the
// `shape` array's children.

```

## چه زمانی باید از این الگو استفاده کنیم؟

- زمانی از این الگو استفاده کنید که نمیخواهید کدی که نوشته‌ید وابستگی‌ای به کلاس‌هایی داشته باشد که قصد کپی گرفتن از آنها را دارید!
- بطور مثال این مورد زمانی پیش می‌آید که شما Object ای از سمت یک 3d-party دریافت می‌کنید و اطلاعاتی از زیر-کلاس‌های آن ندارید، پس نمیخواهید که وابستگی‌ای به زیر-کلاس‌هایش داشته باشد.
- الگوی prototype یک interface کلی را در اختیار client قرار می‌دهد تا با استفاده از آن با تمام object هایی که قابلیت clone شدن دارند ارتباط برقرار کند. این interface باعث می‌شود تا کد client نیازی به اطلاع از جزئیات دقیق اشیاء نداشته باشد و مستقل از کلاس‌های خاص اشیاء باشد.

- زمانی از این الگو استفاده کنید که می‌خواهید تعداد زیر-کلاس‌هایی که فقط در مقدار دهی اشیاء متفاوت هستند را کاهش دهید.

◦ فرض کنید کلاس پیچیده‌ای دارید که برای استفاده از آن باید configuration های زیادی انجام دهید. راههای زیادی برای config کردن این کلاس وجود دارد. این کلاس در جای جای برنامه شما پراکنده شده است و مورد استفاده قرار گرفته شده است. برای اینکه از دوباره کاری جلوگیری کنید، تعدادی زیر-کلاس می‌سازید و config های مشترک بین آنها را در سازنده‌های آنها قرار می‌دهید. با این کار شما مشکل دوباره‌کاری را حل کرده‌اید ولی الان تعداد زیادی زیر-کلاس بی‌معنی دارید که تفاوت زیادی با یکدیگر ندارند!

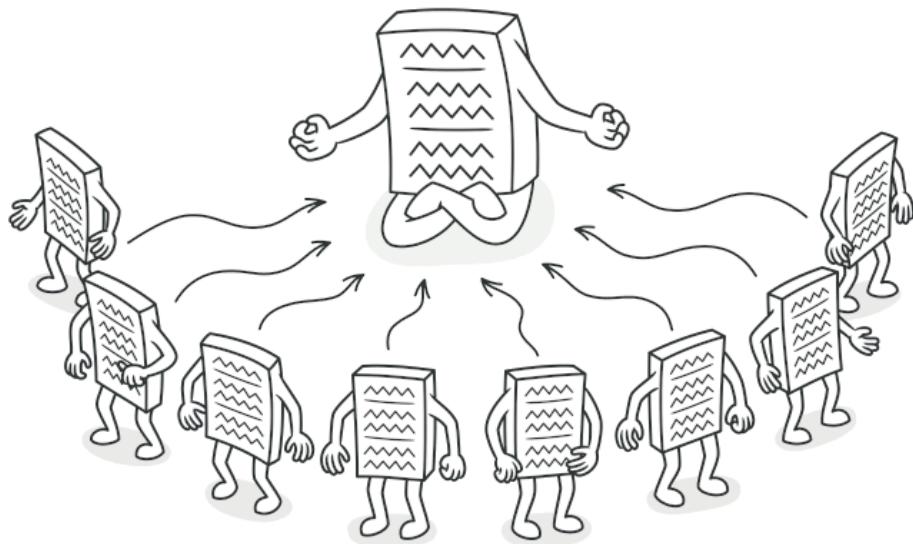
الگوی طراحی Prototype به ما این امکان را می‌دهد تا از تعدادی Object از پیش ساخته شده که با تنظیمات مختلفی ایجاد شده‌اند استفاده کنیم. بجای اینکه برای هر کدام از config هاییمان یک زیر-کلاس ایجاد کنیم، به راحتی می‌توانیم از prototype آن استفاده کنیم و آن را clone کنیم.

## معایب و مزایا

- ❖ بدون وابستگی به کلاس‌ها می‌توانید از اشیاء کپی بگیرید.
  - ❖ به جای ساخت اشیاء تکراری، می‌توانید یک سری اشیاء پیش ساخته بسازید و آنها را کپی کنید.
  - ❖ به راحتی می‌توانید Object های پیچیده بسازید.
  - ❖ زمانی که با اشیاء پیچیده سر و کار دارید، استفاده از این الگوی طراحی به جای ارث‌بری می‌تواند گزینه‌ی خوبی باشد.
- کردن اشیائی که در یک چرخه باهم در ارتباطند در این الگو کار دشواری است.

## Singleton

این الگو این قابلیت را در اختیارمان قرار می‌دهد تا مطمئن باشیم فقط یک شیء از کلاسمان ساخته می‌شود.

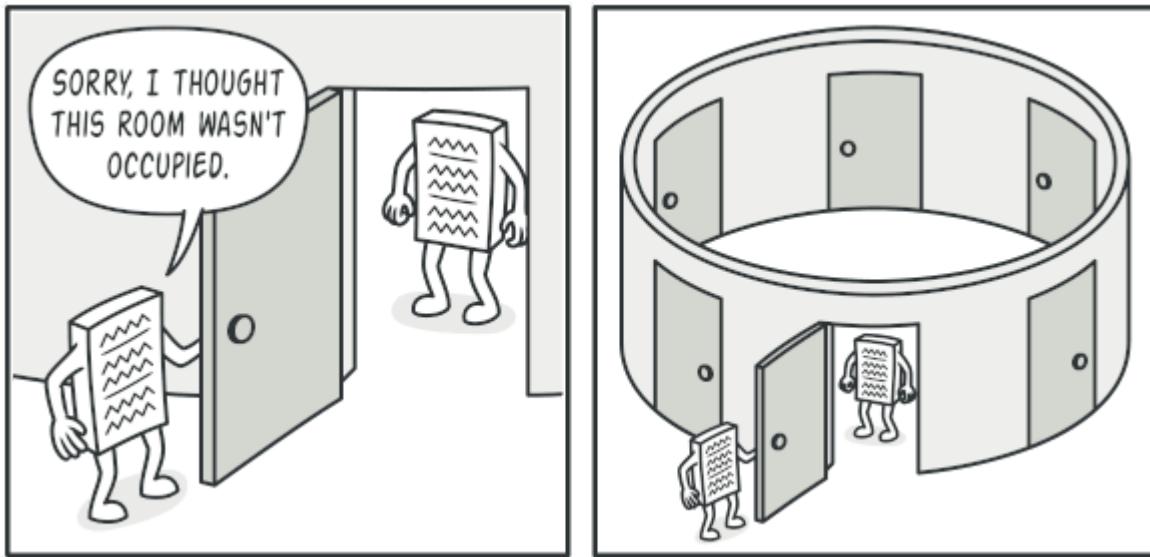


### طرح مسئله

الگوی طراحی Singleton دو مشکل را همزمان حل می‌کند اما با این حال قانون Single responsibility نقض می‌کند. مواردزیر را در نظر بگیرید :

1. اطمینان از اینکه فقط یک شیء از کلاسمان وجود دارد: چرا باید دانست که چند شیء یا نمونه از کلاسمان وجود دارد؟ از دلایل اصلی آن این است که بتوانیم منابع مشترک را مدیریت کنیم. مثل دیتابیس یا یک فایل .

فرض کنید که Object ای ساختید و بعد از مدتی می‌خواهید Object دیگری بسازید، بجای اینکه یک Object جدید دریافت کنید، همان قبلی در اختیار شما قرار بگیرد. به این نکته هم باید توجه کرد که انجام این کار با یک Constructor عادی امکان پذیر نیست. چرا که یک سازنده‌ی عادی همیشه باید یک Instance جدید برگرداند.



2. دسترسی به یک نمونه در سراسر برنامه: فرض کنید که یک شیء خاصی دارید که اشتراک‌گذاری منابع پایگاه داده را مدیریت می‌کند و می‌خواهید در هر نقطه از برنامه‌تان به این شیء دسترسی داشته باشید. همینطور می‌خواهید که اشیاء دیگر نتوانند تغییری در منابع ایجاد کنند.

برای حل موارد بالا چه راه حلی به ذهنتان می‌رسد؟؟

## راه حل

به طور معمول تمام پیاده‌سازی‌های Singleton این دو مورد را در خود دارند:

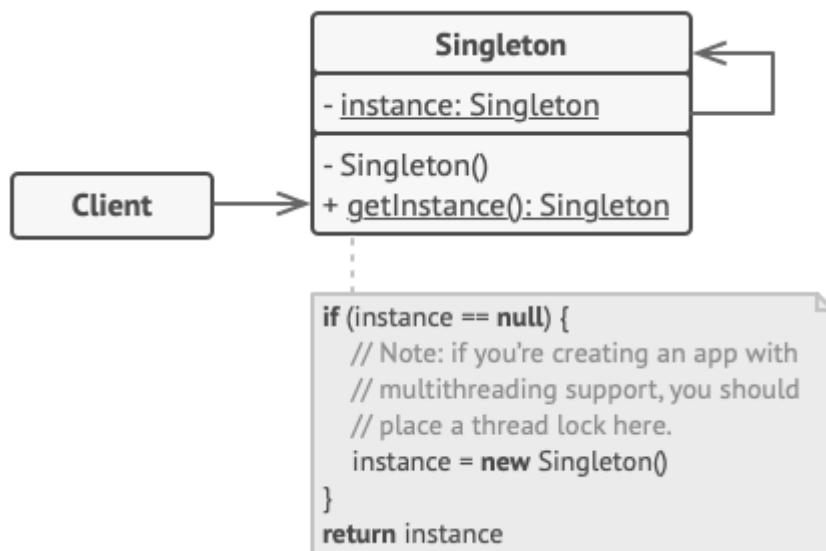
1. ایجاد سازنده‌ی private ، تا از ایجاد یک نمونه‌ی جدید جلوگیری شود.
2. ایجاد یک متدهای static تا فرایند ایجاد نمونه و یا دریافت نمونه قبلی را با آن مدیریت کنیم. این متدهای خصوصی را فراخوانی می‌کند و یک شیء جدید می‌سازد. سپس شیء ساخته شده را در یک متغیر static ذخیره می‌کند و در فراخوانی‌های بعدی این فیلد static را به عنوان نتیجه برمی‌گرداند.

اگر در هرجایی از کدتان به این کلاس که آن را Singleton کرده‌اید دسترسی دارید، پس به آن متدهای static هم دسترسی دارید و همیشه می‌توانید از یک نمونه استفاده کنید.

## مقایسه با دنیای واقعی

دولت بهترین مثال برای الگوی طراحی Singleton می‌باشد! یک کشور می‌تواند فقط یک دولت رسمی داشته باشد. صرف نظر از هویت اشخاصی که یک دولت را تشکیل می‌دهند، عنوان "دولت X" یک نقطه‌ی دسترسی در سطح جهان است که گروهی از افراد مسئول را مشخص می‌کند.

## ساختار



- کلاس Singleton متد static getInstance() ای با نام () دارد که در هر بار فراخوانی یک نمونه از آن کلاس را برمی‌گرداند.
- سازنده‌ی این کلاس باید از دید client مخفی باشد (یعنی آن را private تعریف کنیم) و تنها راه ارتباطی باید متد getInstance() باشد.

## مثال

در این مثال کلاس اتصال به دیتابیس همانند یک کلاس Singleton عمل می‌کند. این کلاس هیچ سازنده‌ای public-ای ندارد و تنها راه فراخوانی Object این کلاس، متده است () getInstance(). این متده اولین بار یک نمونه‌ی جدید می‌سازد و آن را ذخیره می‌کند و در فراخوانی‌های بعدی همان نمونه را برگرداند.

```
// The Database class defines the `getInstance` method that lets
// clients access the same instance of a database connection
// throughout the program.
class Database is
    // The field for storing the singleton instance should be
    // declared static.
    private static field instance: Database

    // The singleton's constructor should always be private to
    // prevent direct construction calls with the `new`
    // operator.
    private constructor Database() is
        // Some initialization code, such as the actual
        // connection to a database server.
        // ...

    // The static method that controls access to the singleton
    // instance.
    public static method getInstance() is
        if (Database.instance == null) then
            acquireThreadLock() and then
                // Ensure that the instance hasn't yet been
                // initialized by another thread while this one
                // has been waiting for the lock's release.
            if (Database.instance == null) then
                Database.instance = new Database()
        return Database.instance

    // Finally, any singleton should define some business logic
    // which can be executed on its instance.
    public method query(sql) is
        // For instance, all database queries of an app go
        // through this method. Therefore, you can place
        // throttling or caching logic here.
        // ...

class Application is
    method main() is
        Database foo = Database.getInstance()
```

```

foo.query("SELECT ...")
// ...
Database bar = Database.getInstance()
bar.query("SELECT ...")
// The variable `bar` will contain the same object as
// the variable `foo`.

```

## چه زمانی باید از این الگو استفاده کنیم؟

- زمانی از این الگو استفاده کنید که می‌خواهید فقط از همان یک نمونه از کلاس‌تان در همه جای برنامه استفاده شود؛ بطور مثال اشتراک گذاری منابع دیتابیس که در همه جای برنامه مشترک است.

الگوی Singleton تمامی فرایندهای دیگر برای ایجاد نمونه‌ها را غیر فعال می‌کند و تنها از یک متدهای ایجاد نمونه استفاده می‌کند که این متدهای ایجاد نمونه جدید می‌سازد و یا نمونه‌ای که از قبل موجود است را برمی‌گرداند.

- زمانی از این الگو استفاده کنید که نیاز دارید تا کنترل دقیق‌تری بر روی متغیرهای سراسری برنامه‌تان داشته باشید.

برخلاف متغیرهای سراسری، الگوی Singleton تضمین می‌کند که تنها یک نمونه از کلاس وجود دارد و هیچ چیز به جز خود کلاس Singleton نمی‌تواند نمونه‌ای که در حافظه ذخیره شده است را برگرداند.

البته توجه داشته باشید که هر زمان که خواستید می‌توانید این محدودیت‌ها را تنظیم کنید و اجازه دهید هر تعداد نمونه‌ای از کلاس‌تان ایجاد شود. تنها قسمت از کدی که نیاز به تغییر دارد متدهای `getInstance()` می‌باشد.

## معایب و مزایا

- ❖ از این موضوع اطمینان دارید که فقط یک نمونه از کلاس‌تان در دسترس است.
- ❖ در همه جای برنامه می‌توان از نمونه‌ی ایجاد شده استفاده کرد.
- ❖ نمونه‌ی Singleton تنها زمانی که درخواست ایجاد شی فراخوانی شود ایجاد می‌شود.

- این الگو Single responsibility را نقض می‌کند زیرا در یک زمان دو مشکل را حل می‌کند!
- این الگو زمانی که اجزای برنامه اطلاعات زیادی از یکدیگر داشته باشند می‌تواند منجر به یک طراحی نادرست شود و این طراحی در پس این الگو پنهان می‌شود.
- برای استفاده از این الگو در محیط‌های multi thread باید حتماً از رویه‌ی خاصی استفاده کرد. چرا که باید اطمینان حاصل شود که چند thread یک شیء Singleton را چندین بار ایجاد نکنند.

Framework که از این الگو استفاده شود، کار برای نوشتن unit test کمی سخت می‌شود. → زمانی های زیادی برای ایجاد Object های mock از ارثبری استفاده می‌کنند. این در حالی است که کلاس دارای سازنده‌ی خصوصی است. و در اکثر زبانها امکان ارثبری از کلاس خصوصی و singleton کردن متدهای static override امکان پذیر نیست. بنابراین برای نوشتن تست باید یک راه خلاقانه پیدا کرد تا یک mock object از یک کلاس Singleton ایجاد کرد.

## Structural

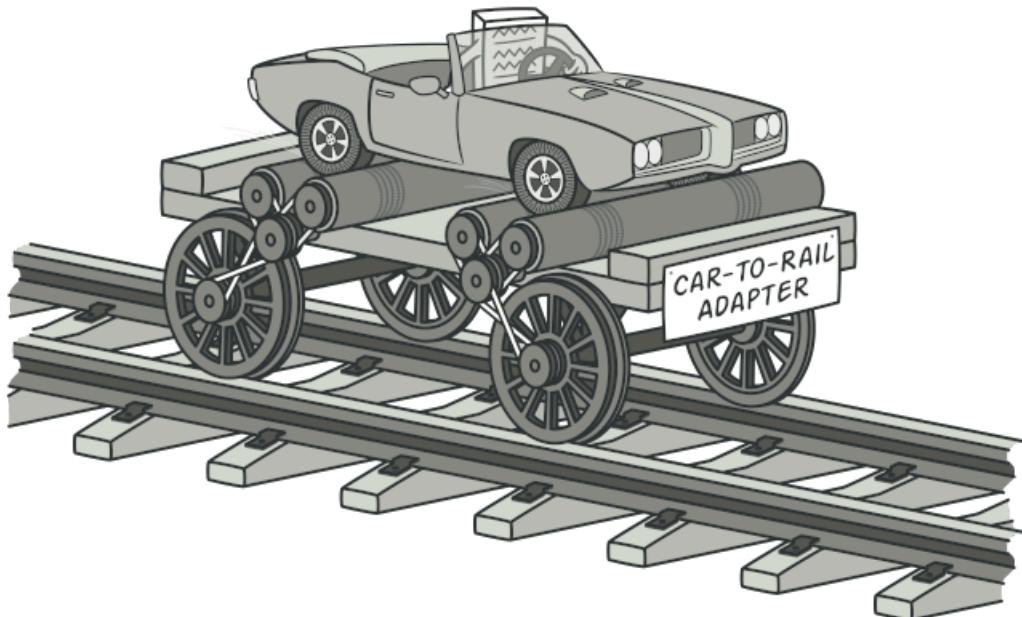
این نوع از الگوهای طراحی، راههایی برای ایجاد Object ها و کلاس‌هایی با ساختارهای بزرگتر را در اختیارمان قرار میدهد و در عین حال ساختارهایی انعطاف‌پذیر و کارآمد ارائه می‌دهد.

انواع الگوهای طراحی structural عبارتند از :

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

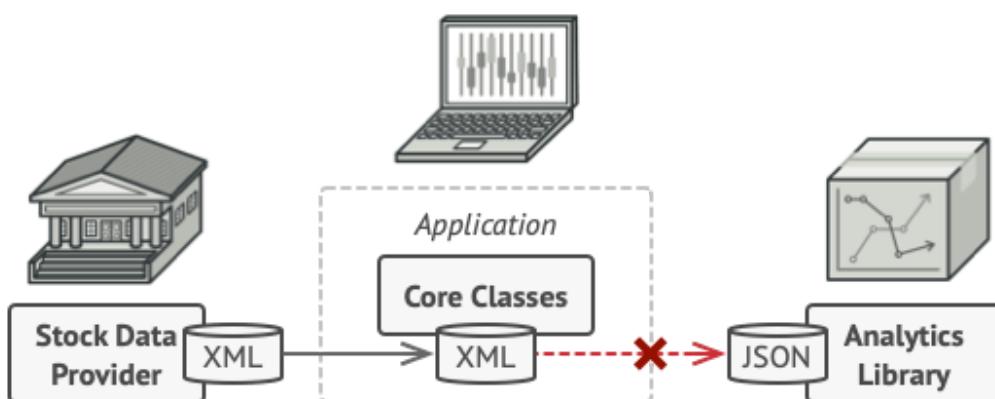
## Adapter

این الگوی طراحی همانند یک مبدل در کدهایمان عمل می‌کند و باعث می‌شود تا Object ها با واسطه‌هایی که با آنها سازگاری ندارند نیز توانایی برقراری ارتباط داشته باشند.



### طرح مسئله

فرض کنید می‌خواهید برنامه‌ای برای تحلیل بازار سهام بسازید. این برنامه داده‌های سهام را از چندین منبع در قالب XML دریافت می‌کند و سپس نمودارهای آن را به کاربران نمایش می‌دهد. در گام‌های بعدی می‌خواهید یک سری اطلاعات آنالیز سهام را از یک کتابخانه 3rd-party 3d-party دریافت کنید. اما یک مشکل داریم، اینکه کتابخانه با داده‌هایی از نوع JSON کار می‌کند.



می‌توانید کتابخانه را برای کار با XML تغییر دهید. ولی این کار ممکن است قسمتهایی از کد که به آن کتابخانه وابسته هستند را با مشکل مواجه کند و یا حتی ممکن است مشکل سخت‌تر باشد و اصلاً به کتابخانه‌ی مورد نظر دسترسی نداشته باشید که در این حالت دیگر کاری از دستتان ساخته نیست.  
برای حل این مشکل‌ها چه راه حلی پیشنهاد می‌کنید؟

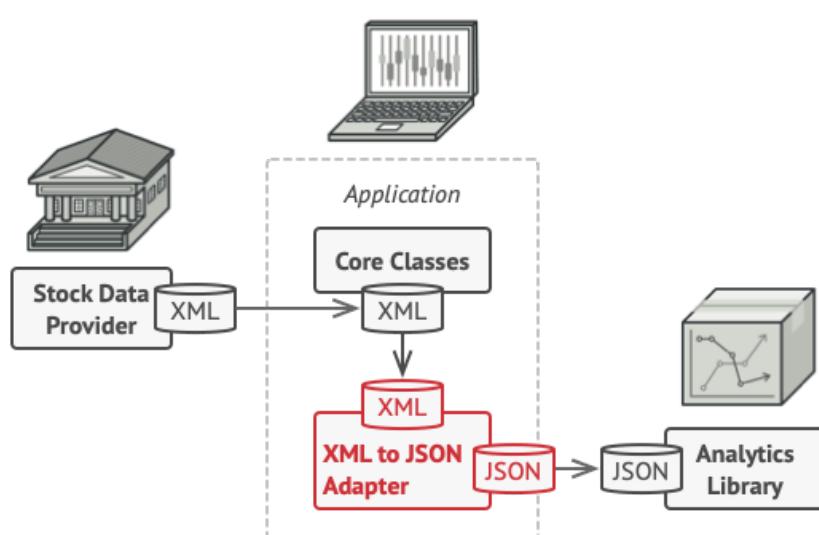
## راه حل

شما می‌توانید یک Adapter ایجاد کنید! یک Object Adapter خاص است که کار تبدیل این Object‌ها به یکدیگر را برعهده دارد.

Adapter به عنوان یک واسط در پشت صحنه عمل می‌کند و مانع از پیچیدگی‌های کد می‌شود. به طور مثال فرض کنید Object‌هایی داریم که با واحدهای متر و کیلومتر کار می‌کنند. حالا می‌خواهیم از این Object‌ها در قسمت دیگری از برنامه استفاده کنیم که با واحدهایی مانند مایل و فوت سازگار است. اینجاست که Adapter وارد عمل می‌شود. Adapter یک لایه‌ی واسط است که این دو را باهم تطبیق می‌دهد. Adapter‌ها علاوه بر اینکه می‌توانند داده‌ها با فرمتهای متفاوت را به یکدیگر تبدیل کنند، بلکه می‌توانند کاری کنند تا Object‌ها با Interface‌هایی که با آنها سازگار نیستند هم توانایی برقراری ارتباط را داشته باشند.

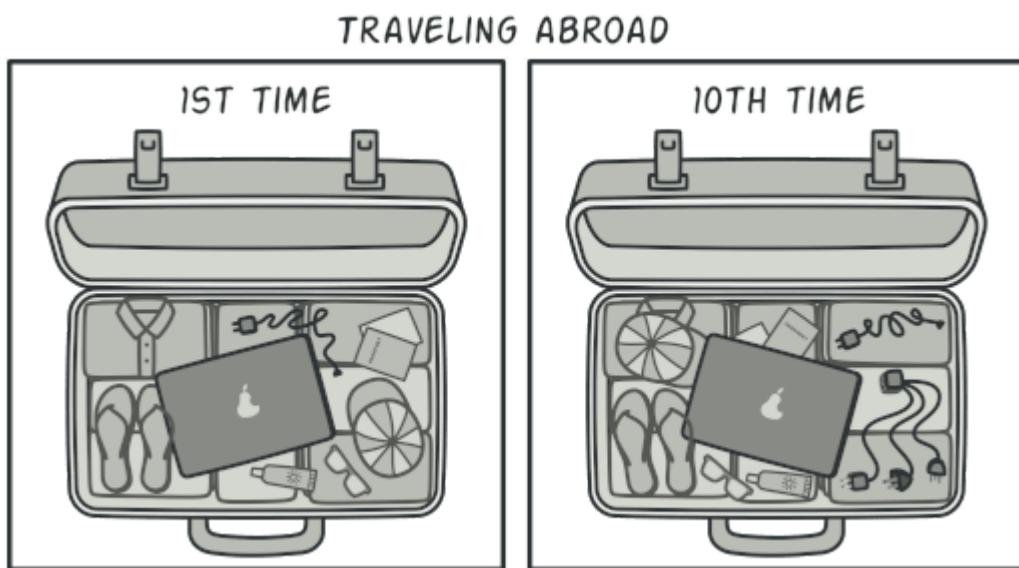
1. Adapter، واسطی که یکی از Object‌ها با آن سازگار است را دریافت می‌کند.
2. با استفاده از این واسط شی مورد نظر به راحتی متدهای Adapter را فراخوانی می‌کند.
3. در همین حین، Adapter درخواستی مطابق با فرمت Object دوم را ارسال می‌کند.

در واقع Adapter به عنوان یک مبدل عمل می‌کند و به عنوان واسط بین دو Object که با هم سازگار نیستند ایفای نقش می‌کند.



در مثال تحلیل‌گر سهام، برای اینکه مشکل ناسازگاری کتابخانه حل شود، می‌توانیم یک Adapter با عنوان XMLToJSON برای کلاس‌هایی که به صورت مستقیم با آن کتابخانه کار می‌کنند بسازیم و سپس برنامه را طوری می‌نویسیم که از طریق این Adapter‌ها ارتباطات خود را برقرار کنند. در این مثال هر زمان که فراخوانی شود و داده‌ایی از نوع XML دریافت کند، آن را به JSON تبدیل می‌کند و نتیجه را به متدهای مناسب ارسال می‌کند.

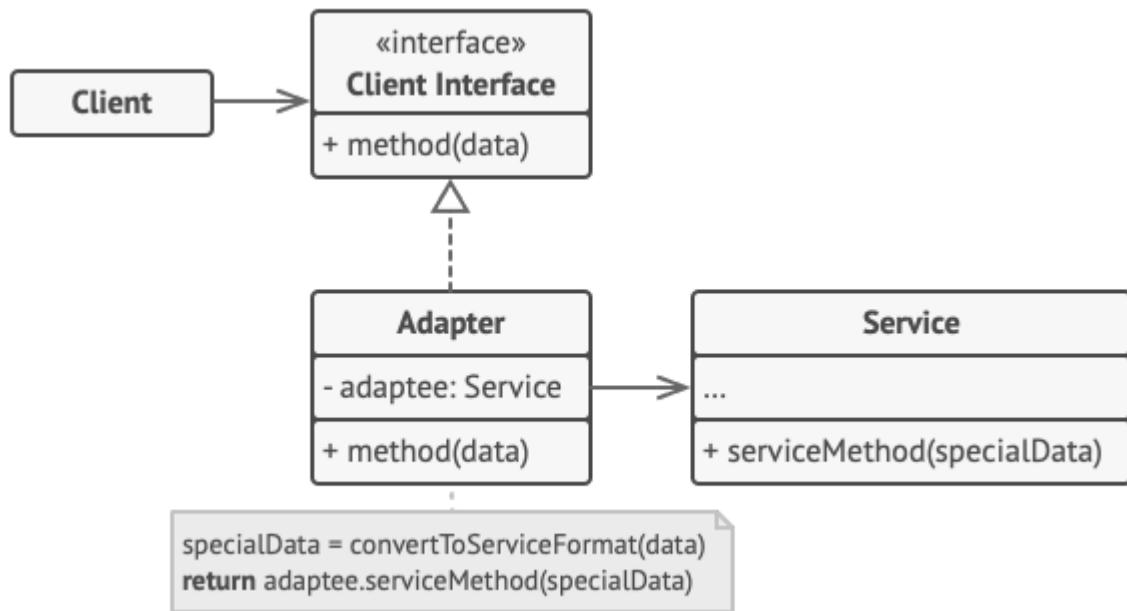
### مقایسه با دنیای واقعی



زمانی که برای اولین بار از آمریکا به اروپا سفر می‌کنید، از اینکه چگونه باید لپ تاپتان را شارژ کنید سورپرایز می‌شوید! استانداردهای دوشاخه و پریز در کشورهای مختلف متفاوت است. به همین دلیل، دوشاخه‌ی آمریکایی شما در پریزهای آلمانی وارد نمی‌شود. این مشکل را می‌توان با یک آداتور حل کرد، آداتوری که سوکت(socket) آمریکایی و دوشاخه‌ی آلمانی داشته باشد.

### Object Adapter ❖

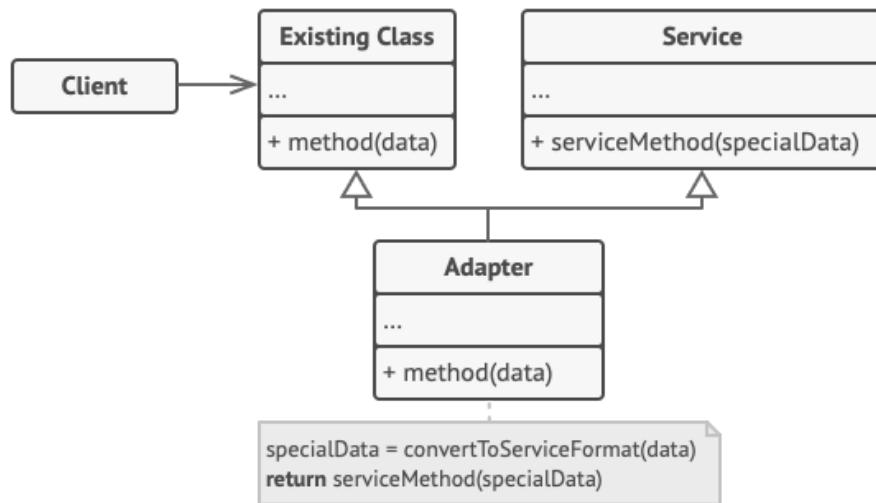
این پیاده‌سازی از اصل Object composition استفاده می‌کند. Adapter یک واسط را پیاده‌سازی می‌کند و اشیاء را به یکدیگر تبدیل می‌کند.



- کلاسی است که شامل منطق برنامه‌مان می‌شود.
- واسطی است که یک سری پروتکل تعریف می‌کند و بقیه کلاسها برای ارتباط با Client باید از این واسط پیروی کنند.
- کلاسی است که با خواسته‌های واسط Client سازگار نیست و وظیفه‌ی Adapter ایجاد سازگاری بین آن ها می‌باشد.
- کلاسی است که هم با Client و هم با Service در ارتباط است. موارد موجود در Client-interface را برای تبدیل به فرمت موارد موجود در service پیاده‌سازی می‌کند. Adapter از سمت Client با استفاده از Client interface فراخوانی می‌شود و درخواستهای client را به درخواستهای سازگار با service تبدیل می‌کند.

### Class Adapter ❖

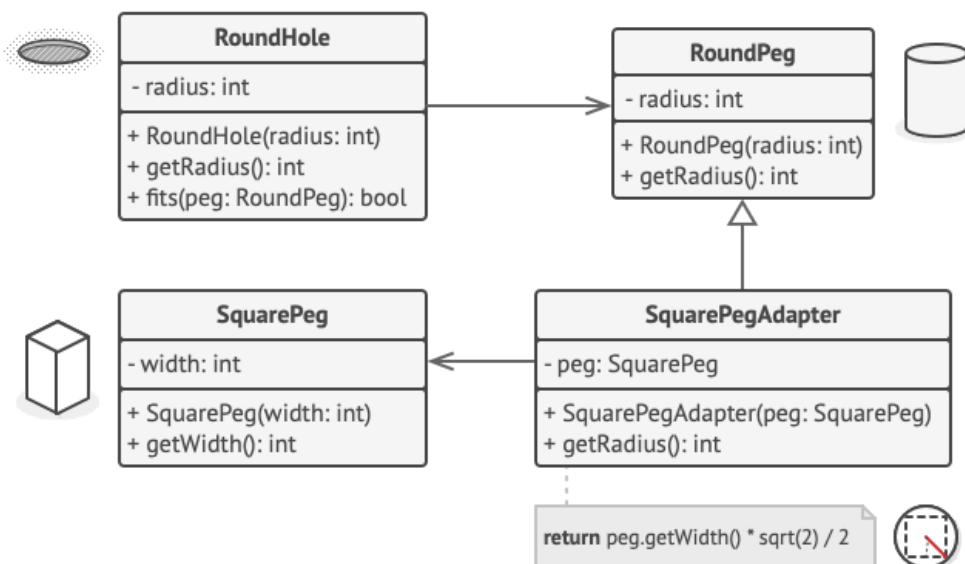
این پیاده‌سازی از اصل ارث‌بری استفاده می‌کند. در این پیاده‌سازی Adapter بصورت همزمان از واسطه‌ای هر دو Object ارث‌بری می‌کند. توجه داشته باشید که این امر در زبان‌هایی که از ارث‌بری چندگانه پشتیبانی می‌کنند قابل اجرا می‌باشد.



- این کلاس نیازی به تبدیل فرمتهای ندارد، چرا که هم رفتارهای client و هم Rftarhای Service را به ارث میبرد. عمل اصلی Adapting در واقع در متدهایی که شدهاند اتفاق میافتد.

### مثال

برای درک بهتر این الگو به سراغ مثال کلاسیک میخ‌های مربعی و حفره‌های گرد میرویم.



در این مثال، Adapter نقش میخ گردی را بازی میکند که شعاع آن برابر با نصف قطر یک مربع است. یا به عبارت دیگر، شعاع آن به اندازه‌ی شعاع کوچکترین دایره‌ای است که میتواند یک میخ مربعی را در خود جای دهد.

```

// Say you have two classes with compatible interfaces:
// RoundHole and RoundPeg.
class RoundHole is
    constructor RoundHole(radius) { ... }

    method getRadius() is
        // Return the radius of the hole.

    method fits(peg: RoundPeg) is
        return this.getRadius() >= peg.getRadius()

class RoundPeg is
    constructor RoundPeg(radius) { ... }

    method getRadius() is
        // Return the radius of the peg.

// But there's an incompatible class: SquarePeg.
class SquarePeg is
    constructor SquarePeg(width) { ... }

    method getWidth() is
        // Return the square peg width.

// An adapter class lets you fit square pegs into round holes.
// It extends the RoundPeg class to let the adapter objects act
// as round pegs.
class SquarePegAdapter extends RoundPeg is
    // In reality, the adapter contains an instance of the
    // SquarePeg class.
    private field peg: SquarePeg

    constructor SquarePegAdapter(peg: SquarePeg) is
        this.peg = peg

    method getRadius() is
        // The adapter pretends that it's a round peg with a
        // radius that could fit the square peg that the adapter
        // actually wraps.
        return peg.getWidth() * Math.sqrt(2) / 2

// Somewhere in client code.
hole = new RoundHole(5)
rpeg = new RoundPeg(5)
hole.fits(rpeg) // true

```

```

small_sqpeg = new SquarePeg(5)
large_sqpeg = new SquarePeg(10)
hole.fits(small_sqpeg) // this won't compile (incompatible types)

small_sqpeg_adapter = new SquarePegAdapter(small_sqpeg)
large_sqpeg_adapter = new SquarePegAdapter(large_sqpeg)
hole.fits(small_sqpeg_adapter) // true
hole.fits(large_sqpeg_adapter) // false

```

## چه زمانی باید از این الگو استفاده کنیم؟

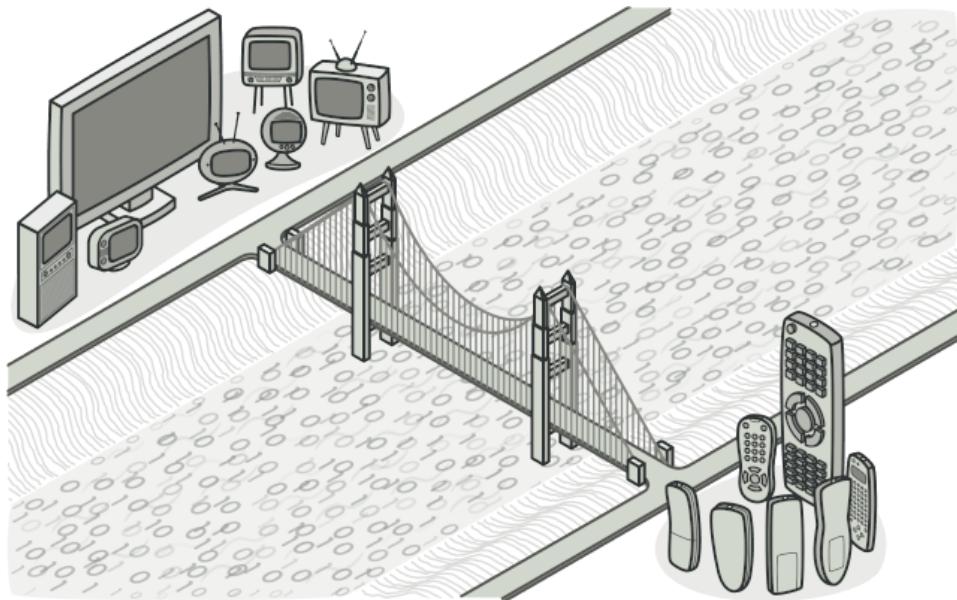
- زمانی از این الگو استفاده کنید که میخواهید از یک سری از کدهای موجود در برنامه‌تان استفاده کنید اما آن کدها با بخش‌های دیگری از برنامه‌تان سازگاری ندارند.
  - الگوی Adapter یه شما این اجازه را می‌دهد تا یک لایه‌ی میانی که نقش یک مترجم را بازی می‌کند بین کدهای قدیمی و کدهای جدیدتر داشته باشد.
  - از این الگو زمانی استفاده کنید که میخواهید از تعداد زیادی زیر-کلاس که برخی از ویژگی‌های مشترک با superClass را ندارند استفاده کنید.
- شما می‌توانید به هر تعدادی که میخواهید زیر-کلاس داشته باشید و functionality هایی که پیاده‌سازی نشده اند را در آنها به کار ببرید. در عین حال مجبوری بخش زیادی از کدهای‌تان را دوباره در آنها تکرار کنید و این اصلاً خوب نیست.
- راه حل بهتر این است که این functionality ها را در کلاس‌های Adapter بگذارید. با این کار از این پس تمام مشکل ها را در کلاس Adapter حل می‌کنید. برای این کار تمامی این کلاسها به همراه کلاس adapter باید از یک interface مشترک پیروی کنند.
- این رویکرد شباهت زیادی به الگوی طراحی Decorator دارد که در بخش‌های بعدی با آن آشنا خواهیم شد.

## معایب و مزایا

- ❖ با استفاده از این الگوی اصل Single responsibility رعایت می‌شود. به طوری که شما می‌توانید منطق اصلی برنامه را از واسطه‌هایی که به عنوان مبدل عمل می‌کنند جدا کنید.
- ❖ با استفاده از این الگو اصل Open/Closed رعایت می‌شود. چرا که با استفاده از این الگو می‌توانید در آینده Adapter های جدیدی بدون اینکه تغییر در کدهای Client ایجاد کنید بسازید.
- ➔ با استفاده از این الگو در حالت کلی اندکی به پیچیدگی کدهای شما اضافه خواهد شد، چرا که باید کلاسها و واسطه‌های زیادی تعریف کنید.

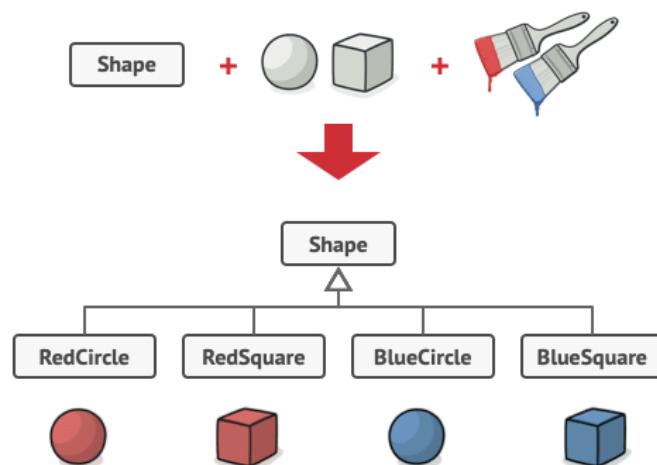
## Bridge

این الگوی طراحی به شما این اجازه را می‌دهد تا کلاس‌های بزرگ و یا مجموعه کلاس‌هایی که خیلی به هم مرتبط هستند را به سلسه مراتب های مجزا تقسیم کنید تا هر کدام از آن ها به طور مستقل توسعه بیابند.



### طرح مسئله

آیا این‌ها ترسناک به نظر می‌رسند؟ باید یک مثال ساده بزنیم. فرض کنید که کلاسی برای اشکال هندسی به نام Shape دارید که دو زیر کلاس با نام‌های Circle و Square دارد. حالا فکر کنید که می‌خواهید ترکیب رنگی را نیز به این سلسه مراتب اضافه کنید، ترکیب رنگ‌های قرمز و آبی. بنابراین تصمیم می‌گیرید تا زیر-کلاس‌هایی با این ترکیب‌های رنگی بسازید. با اینکه شما دو زیرکلاس دارید، اما حال باید چهار زیرکلاس دیگر مانند RedSquare و BlueCircle برای این ترکیب‌های رنگی بسازید.



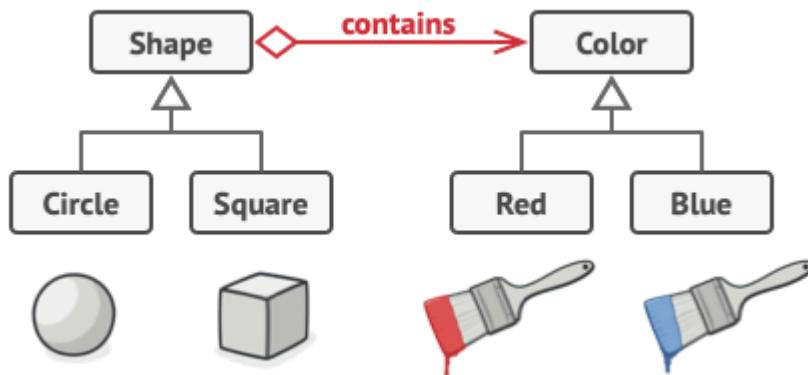
بر این اساس در صورتی که بخواهیم شکل جدیدی با رنگ جدید اضافه کنیم این سلسه مراتب افزایش پیدا می‌کند.

به طور مثال برای اضافه کردن مثلث باید دو زیرکلاس دیگر برای هر کدام از رنگها ایجاد کنید و در ادامه اگر بخواهیم رنگ جدیدی اضافه کنیم، باید سه زیر-کلاس دیگر نیز به آنها اضافه کنیم. این راهی که در پیش گرفته ایم به بدترین شکل ممکن پایان خواهد یافت ... اما راه حل چیست؟

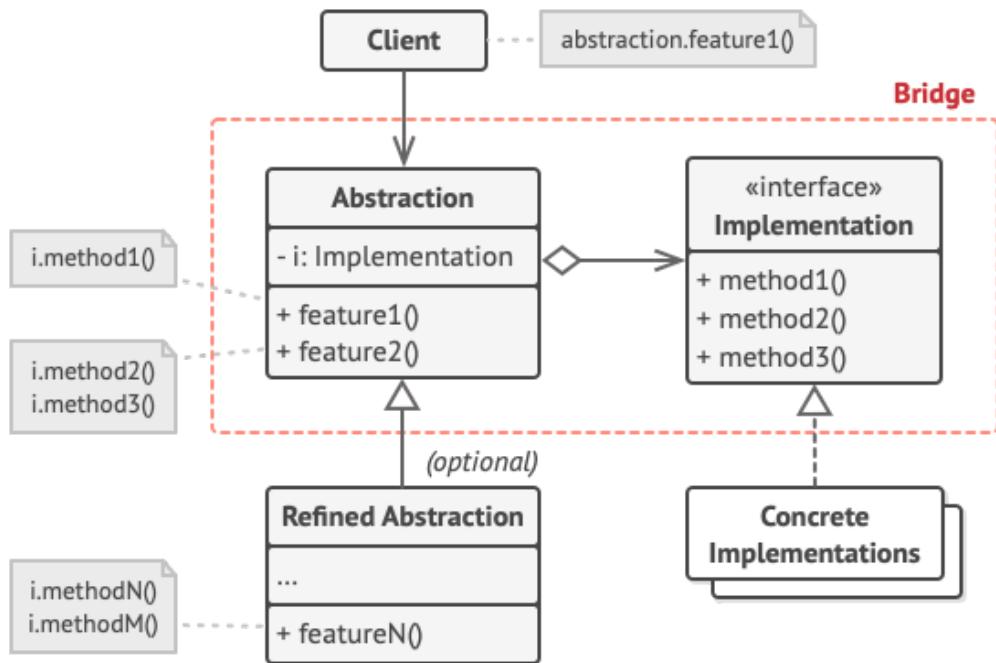
## راه حل

این مشکل به این دلیل رخ می‌دهد که ما سعی می‌کنیم تا کلاس‌های Shape را در دو بعد مستقل از هم گسترش دهیم. اولین بُعد بر اساس فرم و دومین بُعد هم بر اساس رنگ. و این یک مشکل بسیار رایجی در ارث‌بری کلاس‌ها می‌باشد.

الگوی طراحی Bridge برای حل این مشکل به جای ارث‌بری از ترکیب اشیاء استفاده می‌کند. با استفاده از این الگو، یک بُعد از اشیاء مدنظرمان را جدا می‌کنیم و آن را به یک سلسله مراتب مجزا تبدیل می‌کنیم. در واقع کلاس‌های اصلی به جای اینکه همه‌ی موارد را در خود نگه دارند، از یک شیء که دارای سلسله مراتب مستقل است استفاده می‌کنند.



با این رویکرد، می‌توانیم کدهایی که به رنگ مربوط می‌شوند را در مسیرهای مستقل خود پیاده‌سازی کنیم. به طور مثال به دو زیر کلاس Red و Blue نیاز داریم که هر کدام وظیفه‌ی خود را برعهده می‌گیرند. کلاس Shape هم یک رفرنس از Color را در خود نگه می‌دارد که به شیء Color اشاره می‌کند. در این حالت کلاس Shape هر کاری که به رنگ مربوط باشد را به شیء Color واگذار می‌کند. این رفرنس، همانند یک پل میان Shape و Color عمل می‌کند. بنابراین هر زمان که خواستیم رنگ جدید اضافه کنیم، نیازی به تغییر سلسله مراتب کلاس Shape نداریم.

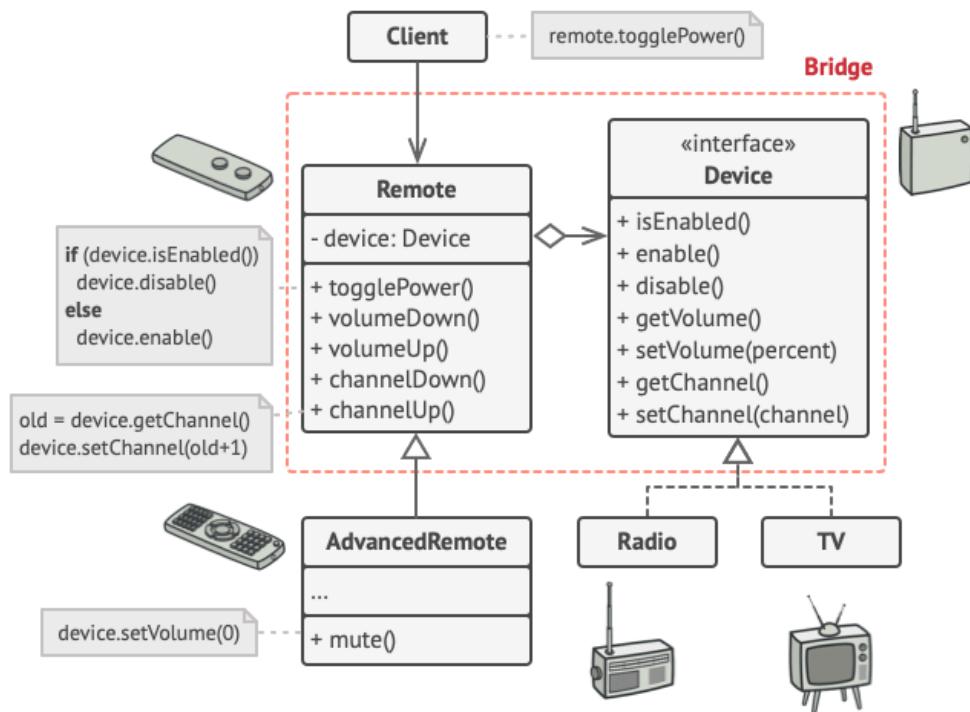


- **Abstraction**: یک لایهی سطح بالا برای کنترل منطق برنامه است. برای اینکه کارهای واقعی در سطوح پایین‌تر را انجام دهد به لایهی Implementation وابسته است.
- **Implementation**: واسطهایی که بین همهی زیر-کلاس‌ها مشترک است را تعریف می‌کند. فقط با استفاده از متدهایی که در این لایه تعریف می‌شوند می‌تواند با Abstraction ارتباط برقرار کند. نیز باید متدهای مشابه این لایه را تعریف کند، ولی معمولاً رفتارهای پیجیده‌ای که باید قبل از Implementation باشد را نیز در خود داشته باشد.
- **Concrete Implementation**: کدهای منتظر با هر پلتفرم در این قسمت قرار می‌گیرند.
- **Refined Abstraction**: گونه‌های مختلفی از کنترل منطق برنامه در این فایل قرار می‌گیرد. این لایه نیز مانند Parent اش، با Implementation های متفاوتی سروکار دارد.
- **Client**: به طور معمول Client بخشی است که به کار کردن با Abstraction علاقه دارد. هر چند که این وظیفه‌ی Client نیست.

## مثال

در این مثال می‌بینیم که الگوی طراحی Bridge چگونه به ما کمک می‌کند تا یک کد یکپارچه مربوط به برنامه‌ای که رابطه‌ی بین یک سری device و remote control شان کنترل می‌کند را به چند لایه تقسیم کنیم.

در این مثال Device نقش لایه Implementation را ایفا می‌کند و remote نقش لایه Abstraction را بازی می‌کند.



کلاس Remote یک رفرنس برای اتصال به کلاس Device را در خود نگه می‌دارد. تمامی ها با استفاده از یک واسط کلی Device با آنها ارتباط برقرار می‌کنند که این کار باعث می‌شود Remote ها توانایی پشتیبانی از چند دستگاه را داشته باشند.

ما می‌توانیم کلاس remote را بصورت جداگانه از کلاس Device توسعه دهیم. تمام چیزی که برای این کار نیاز داریم این است که یک زیر-کلاس از Remote بسازیم.

بطور مثال یک remote معمولی دو دکمه دارد، ولی شما می‌توانید قابلیتهای آن را افزایش دهید بطور مثال باتری آن را افزایش دهید یا یک صفحه‌ی لمسی برای آن تعییه کنید. remote با استفاده از سازنده‌ی Remote دستگاه مربوط به هر Client را به هم متصل می‌کند.

```

// The "abstraction" defines the interface for the "control"
// part of the two class hierarchies. It maintains a reference
// to an object of the "implementation" hierarchy and delegates
// all of the real work to this object.
class RemoteControl is
    protected field device: Device
    constructor RemoteControl(device: Device) is
        this.device = device
    method togglePower() is
        if (device.isEnabled()) then
            device.disable()
        else
            device.enable()
    method volumeDown() is
        device.setVolume(device.getVolume() - 10)

```

```

method volumeUp() is
    device.setVolume(device.getVolume() + 10)
method channelDown() is
    device.setChannel(device.getChannel() - 1)
method channelUp() is
    device.setChannel(device.getChannel() + 1)

// You can extend classes from the abstraction hierarchy
// independently from device classes.
class AdvancedRemoteControl extends RemoteControl is
    method mute() is
        device.setVolume(0)

// The "implementation" interface declares methods common to all
// concrete implementation classes. It doesn't have to match the
// abstraction's interface. In fact, the two interfaces can be
// entirely different. Typically the implementation interface
// provides only primitive operations, while the abstraction
// defines higher-level operations based on those primitives.
interface Device is
    method isEnabled()
    method enable()
    method disable()
    method getVolume()
    method setVolume(percent)
    method getChannel()
    method setChannel(channel)

// All devices follow the same interface.
class Tv implements Device is
    // ...

class Radio implements Device is
    // ...

// Somewhere in client code.
tv = new Tv()
remote = new RemoteControl(tv)
remote.togglePower()

radio = new Radio()
remote = new AdvancedRemoteControl(radio)

```

## چه زمانی باید از این الگو استفاده کنیم؟

- از این الگو زمانی استفاده کنید که می‌خواهید یک برنامه‌ی یکپارچه که دارای **functionality** هایی از گونه‌های متفاوت هستند را به چند لایه تقسیم کنید (بطور مثال کلاسی که می‌خواهد با انواع مختلفی از **database server** ها کار کند)

هر چه یک کلاس بزرگتر باشد، درک نحوه کارکرد آن سخت‌تر می‌شود و ایجاد تغییرات در آن زمان بیشتری می‌خواهد. اگر بخواهید در یکی از **functionality** ها تغییر جزئی ای ایجاد کنید مجبور می‌شوید که کل کلاس را نیز تغییر دهید که این کار اغلب موجب بروز خطاها و عدم رسیدگی به یک سری از عوارض جانبی مهم می‌شود.

- الگوی طراحی **Bridge** این اجازه را می‌خواهد تا یک برنامه‌ی یکپارچه را به چند سلسله کلاس تقسیم کنید. شما می‌توانید هر کلاس را در سلسله مراتب جداگانه خود مستقل از دیگر کلاس‌ها تغییر دهید. این رویکرد نگهداری کد را ساده می‌کند و خطرات و ریسک‌های احتمالی را به حداقل می‌رساند.
- زمانی که نیاز دارید تا یک کلاس را در چندین بُعد مستقل از هم گسترش دهید از این الگو استفاده کنید.

این الگو پیشنهاد می‌کند که برای هر بُعد از کلاس هایتان یک سلسله مراتب جداگانه‌ایی در نظر بگیرید. کلاس اصلی به جای اینکه همه‌ی کارها را به تنها یکی انجام دهد، کارهای مرتبط را به اشیاء آن سلسله مراتب محول می‌کند.

- از این الگو زمانی استفاده کنید که می‌خواهید در زمان اجرا بین چند **Implementation** جابجا شوید.

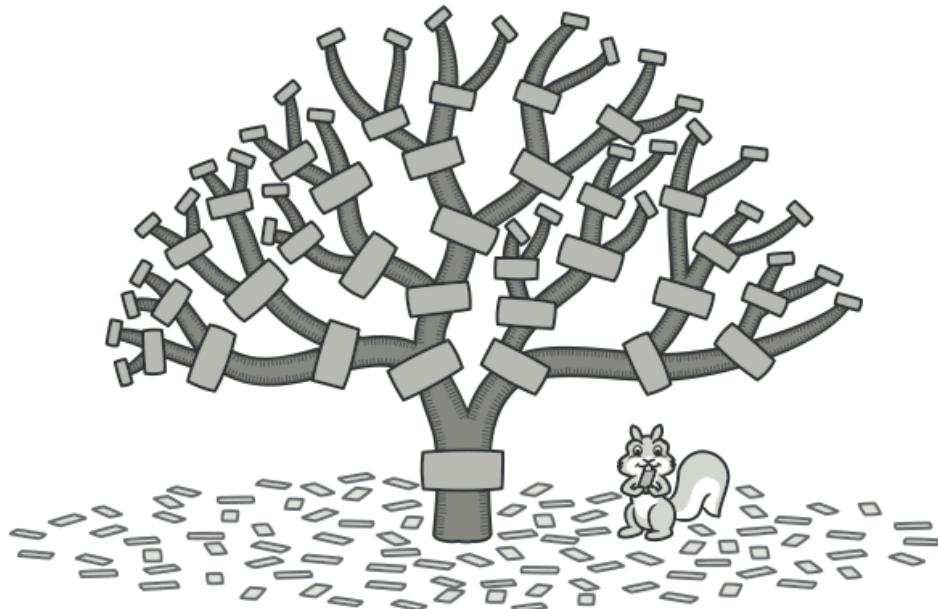
هر چند که این موضوع اختیاری است اما الگوی **Bridge** این اجازه را می‌دهد تا **Implementation** را در درون **Object abstraction** جایگزین کنید و این کار را به آسانی با مقداردهی یک فیلد می‌توانید انجام دهید.

## معایب و مزايا

- ❖ با استفاده از این الگو، می‌توانید برنامه‌هایی بر اساس پلتفرم دلخواهتان بسازید.
- ❖ کدهای **Client** با لایه‌ی سطح بالای **abstraction** ارتباط برقرار می‌کنند و در جریان جزئیات ریز پلتفرم نیستند.
- ❖ با استفاده از این الگو اصل **Open/Closed** رعایت می‌شود چرا که می‌توانید هر **Absraction** و **Implementation** ای را مستقل از هم بسازید.
- ❖ با استفاده از این الگو اصل **Single Responsibility** رعایت می‌شود چرا که شما می‌توانید بر روی منطق اصلی برنامه‌تان در **abstraction** تمرکز کنید و جزئیات **platform** را در **Implementation** مدیریت کنید.
- ➔ ممکن است که با اعمال این الگو بر روی یک کلاس، آن را پیچیده‌تر کنید.

## Composite

با استفاده از این الگوی طراحی می‌توانیم اشیاء را در ساختاری درختی بنویسیم و با آنها به نحوی کار کنیم که گویا یک شیء جداگانه هستند.

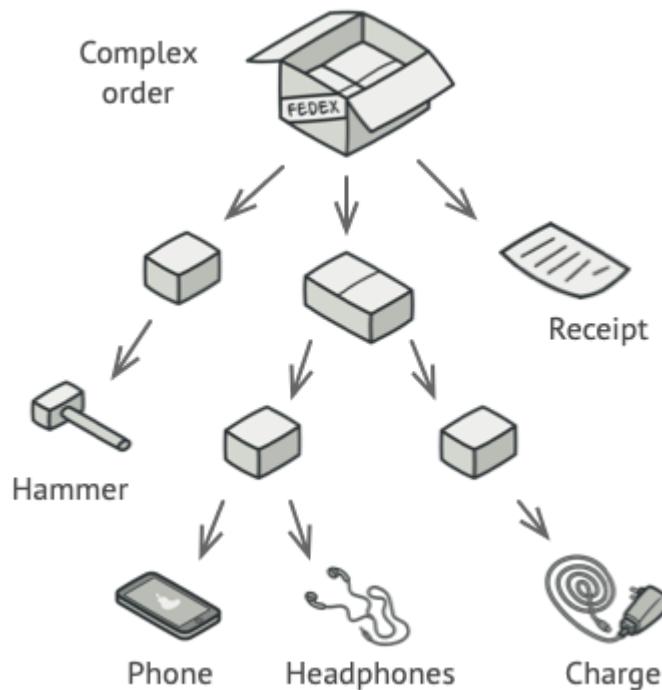


### طرح مسئله

استفاده از این الگوی طراحی زمانی معنا پیدا می‌کند که بتوان مدل اصلی یا Core برنامه‌тан را بصورت درختی نمایش دهید.

به طور مثال فرض کنید که دو نوع شیء دارید، Product و Boxes. یک جعبه می‌تواند شامل چندین محصول و یا حتی چندین جعبه‌ی کوچکتر باشد. و این جعبه‌های کوچکتر هم به همین روال می‌توانند هر کدام یک سری محصول یا جعبه‌ها کوچکتر از خود را در درون خود جای دهند.

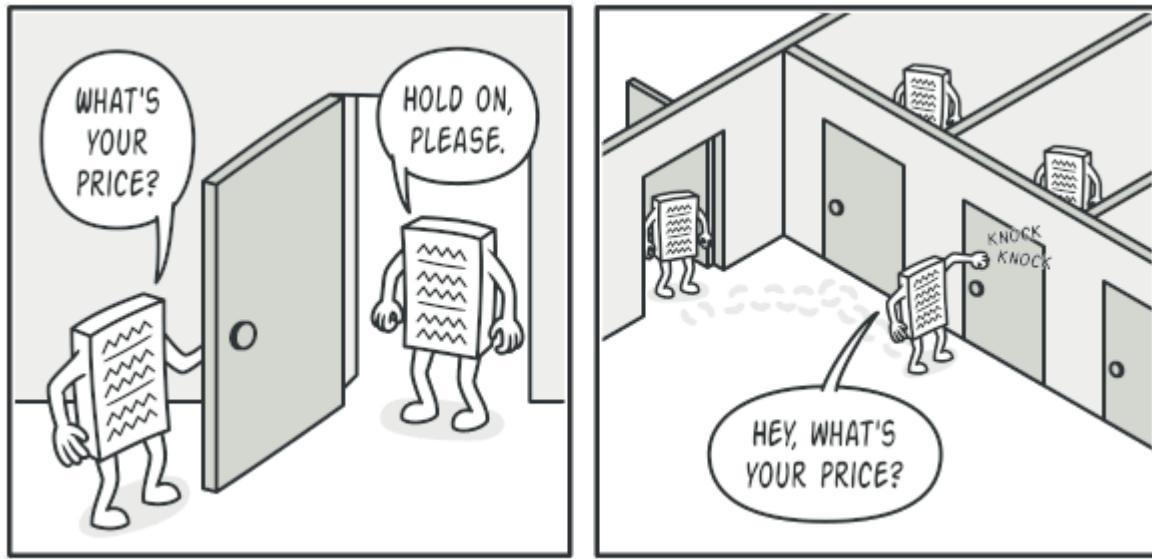
فرض کنید که می‌خواهیم سیستمی برای مدیریت سفارش‌ها بسازید که از این کلاس‌ها استفاده می‌کند. سفارش‌ها می‌توانند شامل محصولات بدون بسته‌بندی و یا جعبه‌های پر از محصولات و جعبه‌های دیگر باشند. چگونه قیمت کل همچین سفارشی را تعیین می‌کنید؟



می‌توانیم اولین روشی که به ذهنمان می‌رسد را انتخاب کنیم، یعنی تمام جعبه‌ها را باز کنیم و محصولات داخل آن‌ها را ببینیم و سپس قیمت کل را محاسبه کنیم که این کار در دنیای واقعی هم قابل اجرا می‌باشد. اما در یک برنامه این کار به سادگی اجرای یک حلقه نیست! شما باید از قبل اطلاعات محصولات و جعبه‌هایی که از آنها می‌گذرد، میزان تو در تو بودن آنها و ... را بدانید. این موارد باعث می‌شوند که راهکار اولیه کمی ناخواهایند باشد. دیگر چه راه حلی برای حل این موضوع می‌توان به کار برد؟

## راه حل

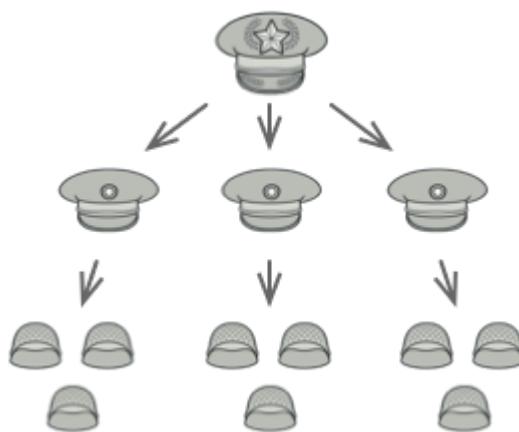
این الگو پیشنهاد می‌کند تا برای کار کردن با `Box` و `Product` از یک `interface` مشترک استفاده کنید و این `interface` یک متاد برای محاسبه قیمت در خودش داشته باشد. اما این متاد چگونه کار می‌کند؟ برای یک `Product` این امر ساده است و قیمت خود محصول را برمی‌گرداند ولی برای هر جعبه به سراغ آیتم‌ها موجود در آن می‌رود و قیمت هر کدام از آنها را بدست می‌آورد و جمع نهایی را برمی‌گرداند. اگر یکی از این آیتم‌ها خودش هم یک جعبه باشد این امر دوباره تکرار می‌شود و این امر تا زمانی ادامه پیدا می‌کند که قیمت تمامی محصولات بدست آمده باشد. یک جعبه حتی می‌تواند یک سری هزینه اضافی مانند هزینه بسته‌بندی را نیز به قیمت نهایی اضافه کند.



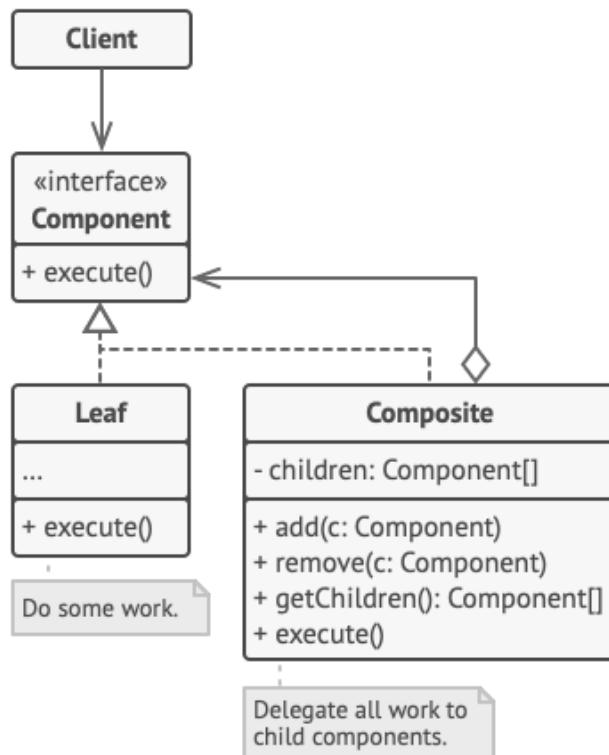
بزرگترین مزیت این رویکرد این است که دیگر لازم نیست به زیرکلاس‌های اشیائی که درخت را تشکیل می‌دهند اهمیت دهید. دیگر لازم نیست به اینکه شء یک محصول است یا یک جعبه‌ی پیچیده اهمیت دهید.

با همه‌ی این موارد می‌توانید با استفاده از یک interface عمومی برخورد یکسانی داشته باشید. وقتی یک متده را فرا می‌خوانید خود اشیاء درخواست را برای درخت مورد نظر ارسال می‌کنند.

### مقایسه با دنیای واقعی



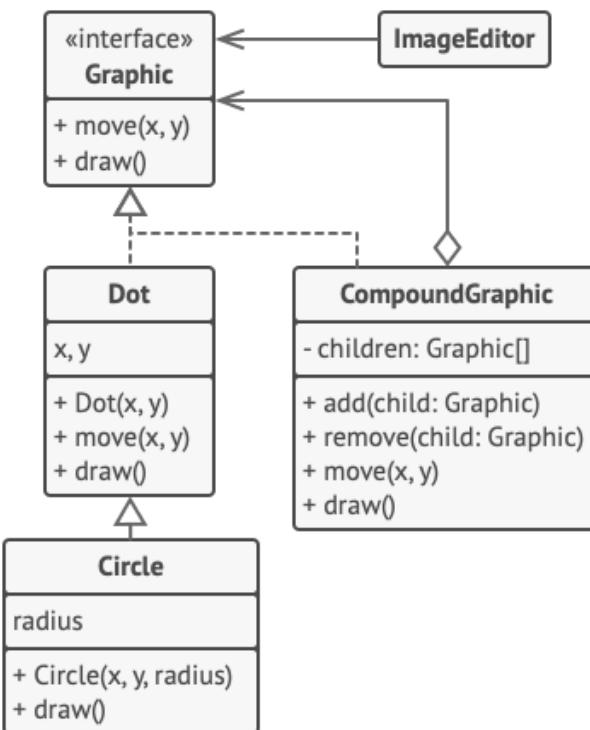
اکثر ارتش‌های دنیا ساختاری سلسله‌مراتبی دارند. یک ارتش از چندین بخش تشکیل شده است. یک ارتش شامل یک سری لشکر است که هر کدام از این لشکرها خود شامل یک سری تیپ می‌باشد و هر کدام از این تیپ‌ها خودشان هم می‌توانند شامل یک سری جوخه باشند. در نهایت یک جوخه کوچکترین گروه از یک سری سرباز واقعی است. دستورات در بالای این سلسله مرتب داده می‌شود و به هر سطح منتقل می‌شود و این امر تا زمانی ادامه می‌یابد که هر کدام از سرباز‌ها از این دستورات آگاه شوند.



- این واسط وظیفه‌ی تعریف عملیاتی را دارد که بین بخش‌های ساده و پیچیده‌ی درخت مشترک است.
- ساده‌ترین قسمت یک درخت است که هیچ زیرمجموعه‌ای ندارد. معمولاً Leaf‌ها (برگ‌ها) نقطه‌ی پایانی انجام کارها می‌باشند و کارهای اصلی را انجام می‌دهند زیرا دیگر در لایه‌ی زیرین خود بخشی را ندارند که کارها را به او واگذار کنند.
- عنصری است که زیرشاخه‌های دیگر را در خودش دارد، مواردی مانند برگ‌ها و یا Container‌های دیگر. یک Container خبری از اجزای داخلی‌اش ندارد و فقط با استفاده از یک واسط با آنها در ارتباط است.  
پس از دریافت یک دستور، Container کار را به عناصر داخلی خودش واگذار می‌کند، سپس نتایج هر بخش را بررسی می‌کند و نتیجه نهایی را برمی‌گرداند.
- کلاینت با تمام عناصر واسط Component کار می‌کند. در نتیجه Client می‌تواند با هر دو عنصر ساده و پیچیده در درخت کار کند.

## مثال

در این مثال، الگوی composite به شما امکان می دهد انباشته کردن اشکال هندسی را در یک ویرایشگر گرافیکی پیاده سازی کنید.



کلاسِ CompoundGraphic یک Container است که شامل یک سری شکل‌ها و همچنین یک سری اشکال ترکیبی است. یک شکل ترکیبی مانند یک شکل ساده دارد. با این وجود بجای اینکه کارها را خودش انجام دهد درخواست‌ها را به صورت بازگشتی به زیر-کلاس‌هایش می‌فرستد و نتایج نهایی را جمع‌آوری می‌کند. Client با همه‌ی شکل‌ها از طریق یک واسط عمومی ارتباط برقرار می‌کند. بنابراین Client اطلاعی از اینکه در حال حاضر با یک شکل ساده و یا یک شکل ترکیبی ارتباط برقرار می‌کند ندارد. Client می‌تواند با کدهای پیچیده بدون اینکه به آنها وابسته شود کار کند.

```

// The component interface declares common operations for both
// simple and complex objects of a composition.
interface Graphic is
    method move(x, y)
    method draw()

// The leaf class represents end objects of a composition. A
// leaf object can't have any sub-objects. Usually, it's leaf
// objects that do the actual work, while composite objects only
// delegate to their sub-components.
class Dot implements Graphic is
    field x, y

    constructor Dot(x, y) { ... }

    method move(x, y)
    method draw()

```

```

method move(x, y) is
    this.x += x, this.y += y

method draw() is
    // Draw a dot at X and Y.

// All component classes can extend other components.
class Circle extends Dot is
    field radius

constructor Circle(x, y, radius) { ... }

method draw() is
    // Draw a circle at X and Y with radius R.

// The composite class represents complex components that may
// have children. Composite objects usually delegate the actual
// work to their children and then "sum up" the result.
class CompoundGraphic implements Graphic is
    field children: array of Graphic

    // A composite object can add or remove other components
    // (both simple or complex) to or from its child list.
    method add(child: Graphic) is
        // Add a child to the array of children.

    method remove(child: Graphic) is
        // Remove a child from the array of children.

    method move(x, y) is
        foreach (child in children) do
            child.move(x, y)

    // A composite executes its primary logic in a particular
    // way. It traverses recursively through all its children,
    // collecting and summing up their results. Since the
    // composite's children pass these calls to their own
    // children and so forth, the whole object tree is traversed
    // as a result.
    method draw() is
        // 1. For each child component:
        //     - Draw the component.
        //     - Update the bounding rectangle.
        // 2. Draw a dashed rectangle using the bounding
        //    coordinates.

```

```
// The client code works with all the components via their base
// interface. This way the client code can support simple leaf
// components as well as complex composites.
class ImageEditor is
    field all: CompoundGraphic
    method load() is
        all = new CompoundGraphic()
        all.add(new Dot(1, 2))
        all.add(new Circle(5, 3, 10))
        // ...
    // Combine selected components into one complex composite
    // component.
    method groupSelected(components: array of Graphic) is
        group = new CompoundGraphic()
        foreach (component in components) do
            group.add(component)
            all.remove(component)
        all.add(group)
        // All components will be drawn.
        all.draw()
    
```

## چه زمانی باید از این الگو استفاده کنیم؟

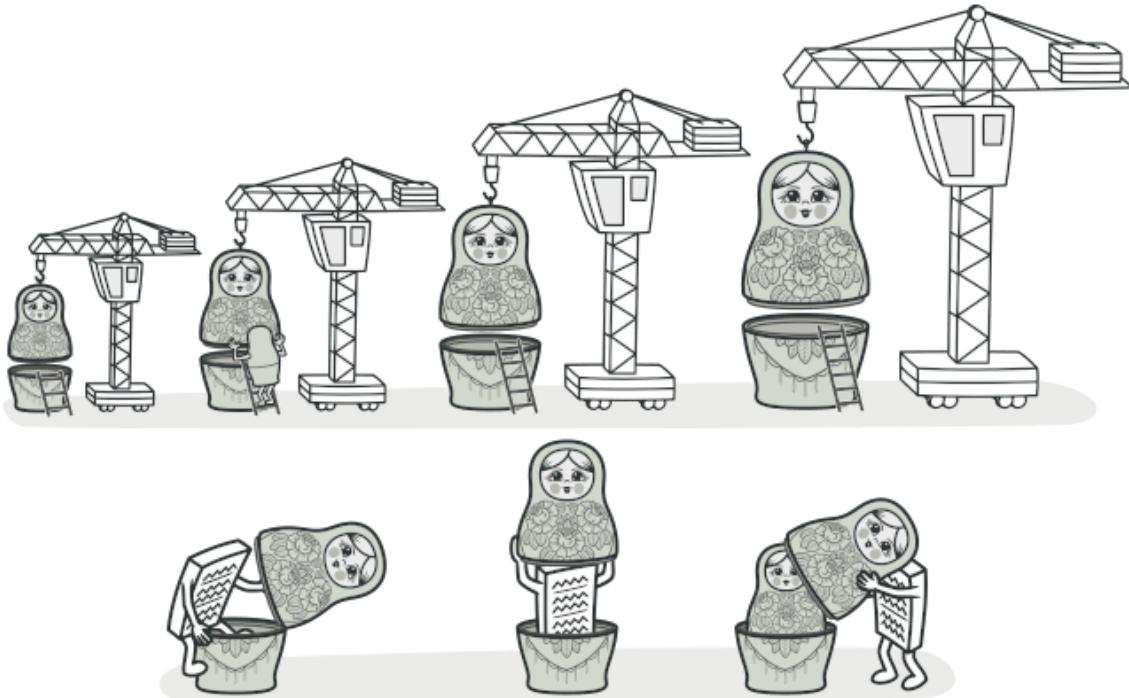
- زمانی از این الگو استفاده کنید که مجبورید اشیاء خود را در ساختاری درختی پیاده سازی کنید.  
این الگو این اجازه را به شما می‌دهد تا با استفاده از دو عنصر که از یک واسط مشترک استفاده می‌کنند ساختار برنامه خود را بسازید. برگهای ساده با container های پیچیده. این container ها خود می‌توانند شامل برگها و یا container های دیگر شوند. این کار به شما کمک می‌کند تا بتوانید یک ساختار تو در توی درختی از اشیاء خود بسازید.
- زمانی از این الگو استفاده کنید که می‌خواهید client تان با همه‌ی عناصر ساده و یا پیچیده برنامه‌تان به یک صورت رفتار کند.  
تمام عناصری که توسط این الگو ایجاد شده باشند، از یک واسط مشترک استفاده می‌کنند. بنابراین client نگرانی‌ای از نحوه‌ی پیاده سازی کلاس‌های داخلی ندارد.

## معایب و مزایا

- ❖ می‌توانید راحت‌تر با ساختارهای درختی پیچیده کار کنید و از مواردی مثل polymorphism و recursive به سود خود استفاده کنید.
- ❖ با استفاده از این الگو اصل Open/Closed SOLID رعایت می‌شود چرا که می‌توانید عناصر جدیدی به کدتان اضافه کنید بدون اینکه تغییری در کدهای قبلی ایجاد کنید.
- اینکه برای کلاس‌هایی که دارای functionality های بسیار متفاوتی هستند یک واسط مشترک بسازیم کاری دشوار است. در سناریوهای خاص باید رابطه‌ی بین مولفه‌های را خیلی تعمیم دهیم و این کار باعث سخت‌تر شدن درک کد می‌شود.

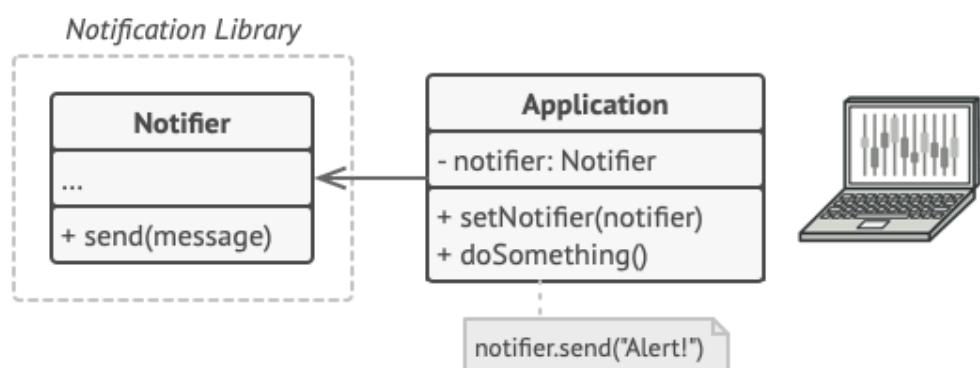
## Decorator

الگوی طراحی Decorator این قابلیت را در اختیارمان قرار می‌دهد تا با قرار دادن یک شیء در درون یک شیء دیگر که به عنوان wrapper عمل می‌کند رفتارهای جدیدی را به شیء خود اضافه کنید.

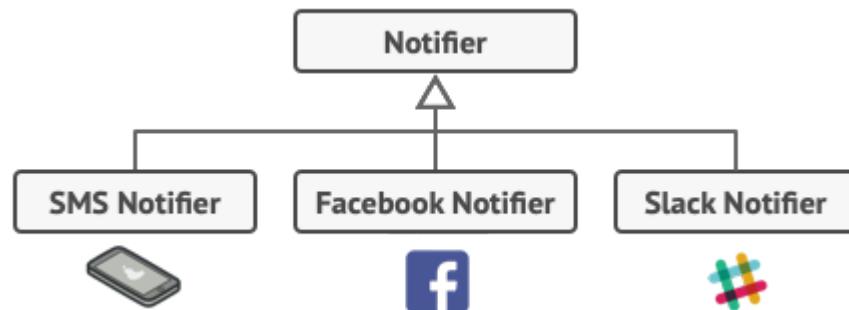


### طرح مسئله

فرض کنید بر روی یک کتابخانه Notification کار می‌کنید که کاربران را از رویدادهای مهم باخبر می‌کند. نسخه‌ی اولیه‌ی برنامه بر اساس کلاسی با نام Notifier بوده است که فقط شامل یک سازنده و یک متند با نام send است. این متند یک متن از سمت client دریافت می‌کند و آن را به ایمیل‌هایی که در سازنده‌اش فرستاده شده ارسال می‌کند. سپس یک 3d-party Client را بازی می‌کند، یک بار کلاس Notifier را می‌سازد و هر زمان که اتفاق مهمی افتاد از آن استفاده می‌کند.

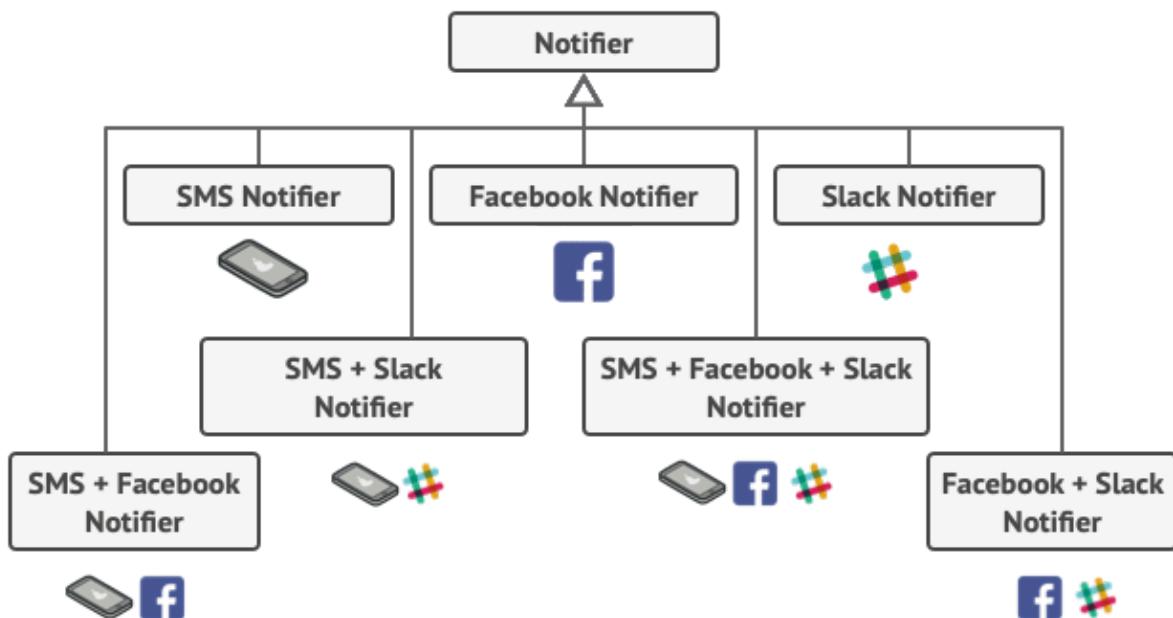


گاهی اوقات متوجه می‌شوید که کاربران انتظارات بیشتری نسبت به ارسال از طریق ایمیل دارند. تعدادی از آنها علاقه دارند که اعلان‌های خیلی مهم را به صورت SMS دریافت کنند. یا تعدادی دیگر علاقه دارند که اعلان‌هایشان را در slack و یا facebook دریافت کنند!



این کار چقدر می‌تواند مشکل باشد؟ اینکه کلاس Notifier را گسترش دهید و متدهای اضافه را در زیر-کلاس‌های جدید قرار دهید. در این حالت Client باید از همه‌ی این زیر-کلاس‌ها یک شیء بسازد و از آنها برای اعلان‌های بعدی اش استفاده کند.

اما اگر شخصی به طور منطقی به شما گفت که "چرا نمی‌توانید از چندین نوع اعلان به طور همزمان استفاده کنید؟ اگر خانه‌تان آتش گرفته است احتمالاً می‌خواهید از طریق چند کanal مطلع شوید" چه پاسخی دارید؟ برای حل این مشکل شما سعی می‌کنید با ایجاد یک سری زیر-کلاس‌های خاص که چندین متد برای اعلان را در خود ترکیب می‌کنند این مشکل را حل کنید. اما این کار علاوه بر مخدوش کردن کدهای Library، کدهای Client را نیز گنج می‌کند.



باید راه دیگری برای ساختار کلاس‌هایمان پیدا کنیم، چرا که اگر به همین روال ادامه دهیم، دریابی از کلاس‌ها را در برنامه خود داریم!!

## راه حل

زمانی که می‌خواهید رفتار یک شیء را تغییر دهید، ساختن زیر-کلاس از آن اولین راهکاری است که به ذهنتان می‌آید. با این حال ارثبری چندین مورد خطرناک دارد که باید از آنها آگاه باشید.

- ارثبری static یا ایستا است، به اینگونه که نمی‌توانید رفتاری شیء‌ای که از قبل ایجاد شده را در زمان اجرا عوض کنید. فقط می‌توانید تمام آن شیء را با یک زیر-کلاس دیگر جایگزین کنید.
- زیر-کلاس فقط می‌تواند یک والد یا Parent داشته باشد، در اکثر زبان‌های برنامه نویسی ارثبری به زیر-کلاس این اجازه را نمی‌دهد که چندین والد داشته باشد.

یکی از راههای غلبه بر این اختوارها، استفاده از Composition و Aggregation در ارثبری است. هر دو گزینه تقریباً به یک شکل عمل می‌کنند: یک شیء به شیء دیگری reference دارد و کاری را به آن واگذار می‌کند، در حالیکه در وراثت خود شیء قادر به انجام آن کار است و رفتار را از کلاس Parent خود به ارث می‌برد.

با این رویکرد، می‌توانید به آسانی از یک سری اشیاء کمکی استفاده کنید و آنها را با یکدیگر جایگزین کنید و رفتار container را در زمان اجرا تغییر دهید. یک شیء می‌تواند از رفتار کلاس‌های دیگر استفاده کند و همینطور به چندین کلاس دیگر reference داشته باشد و انواع کارها را به آنها واگذار کند. Aggregation و Composition نکته‌ی کلیدی پشت اکثر الگوهای طراحی‌اند که Decorator یکی از آنهاست.

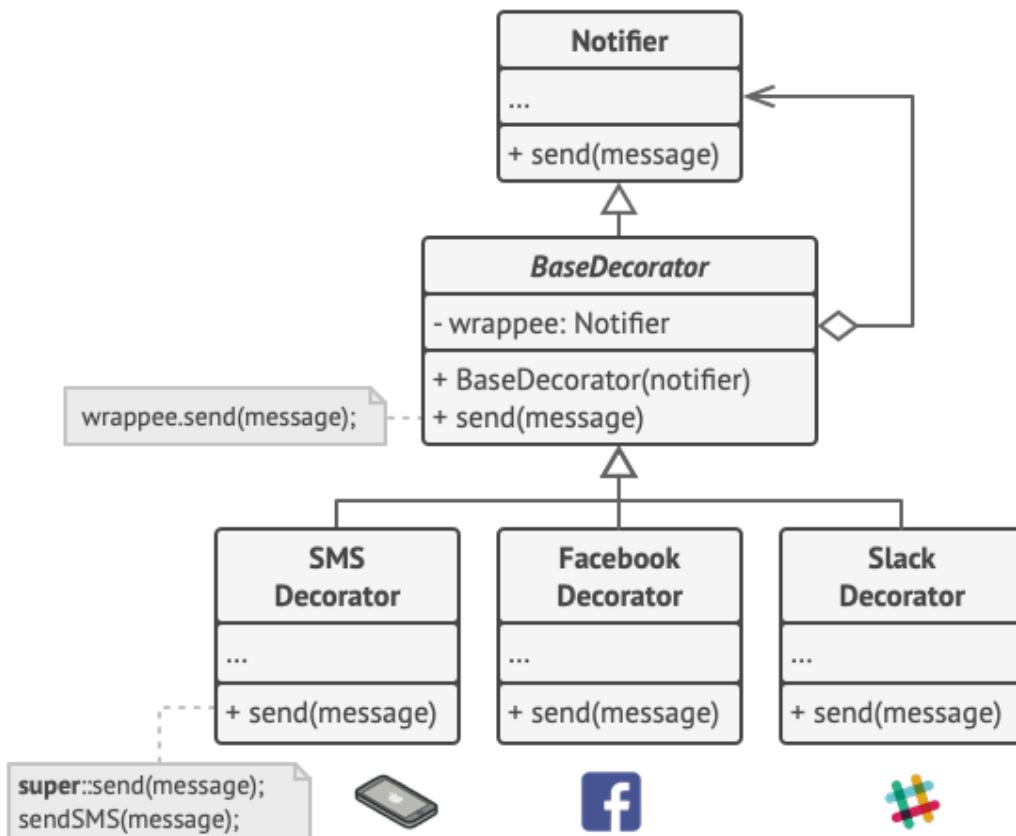


کلمه‌ی Wrapper بهترین واژه‌ای است که می‌تواند الگوی طراحی Decorator را توصیف کند. یک Wrapper شیء‌ای است که می‌تواند با اشیاء هدف ارتباط بگیرد. Wrapper شامل متدهای شیء هدف است و تمام درخواست‌هایی که دریافت می‌کند را با شیء هدف در جریان می‌گذارد! البته wrapper ممکن است نتیجه‌ای متفاوت برگرداند. چرا که می‌تواند کارهایی را قبل و یا بعد از در جریان گذاشتن با شیء هدف انجام دهد.

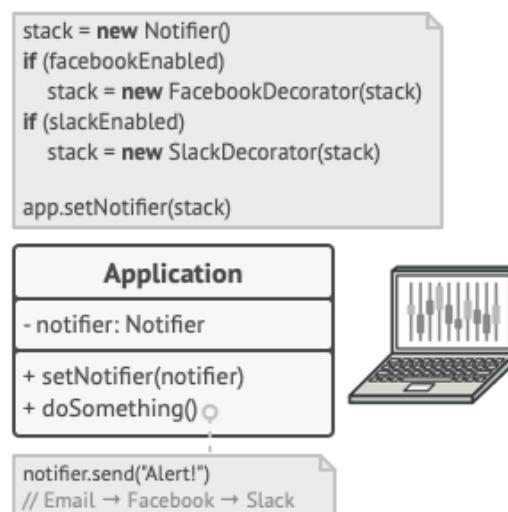
چه زمانی یک wrapper ساده تبدیل به یک decorator می‌شود؟ همانطور که قبلاً گفته شد، یک wrapper تمامی واسطه‌ها و رفتارها را همانند شیء‌ایی که wrap شده است پیاده‌سازی می‌کند. به همین دلیل است که Client هر دوی آنها را یکسان می‌داند. فیلدي که به wrapper ارجاع دارد باید به گونه‌ای باشد که هر نوع شیء از جنس شیء‌هدف را بپذیرد.

با استفاده از این روش شما می‌توانید یک شیء را با استفاده از چندین wrapper تحت پوشش قرار دهید و رفتار ترکیبی تمام این wrapper ها را به آن اضافه کنید.

باید در مثال Notifier رفتار ارسال یک ایمیل ساده را در کلاس Notifier پیاده‌سازی کنیم اما سایر روش‌های اعلان را به decorator تبدیل کنیم.



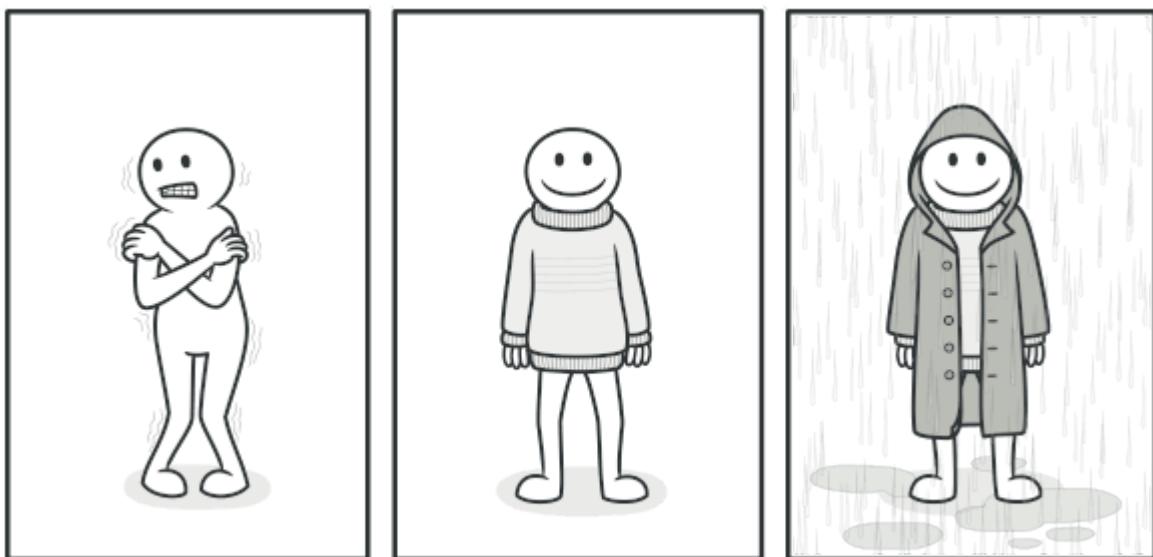
باشد یک سری از اشیاء پایه را که با خواسته‌هایش مطابقت دارند در مجموعه‌ای از Client ها قرار دهد. این اشیاء در پایان ساختاری مانند پشته (stack) خواهند داشت.



آخرین شیء در این پشته، همان شیء‌ای است که `client` با آن کار می‌کند. تا زمانی که تمام `decorator`‌ها واسط یکسانی را به عنوان `Notifier` پیاده‌سازی می‌کنند، `client` به اینکه آیا با خود شیء `Notifier` کار می‌کند یا با یک شیء `Wrapper` آن، اهمیتی نمی‌دهد.

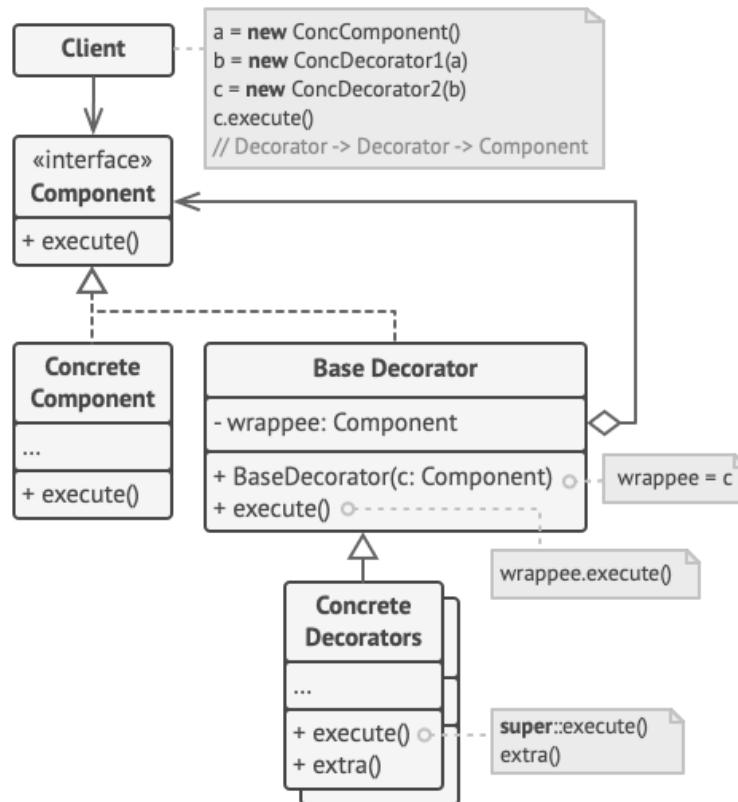
می‌توانیم از این رویکرد در پیاده‌سازی رفتارهای دیگر مانند قالب‌بندی پیام‌ها و ... نیز استفاده کنیم. `Client` می‌تواند با استفاده از هر `decoratory` شیء خود را بسازد، به شرطی که از واسطه که بقیه اشیاء پیاده‌سازی کرده‌اند پیروی کند.

## مقایسه با دنیای واقعی



پوشیدن لباس می‌تواند مثالی از الگوی `decorator` در دنیای واقعی باشد. زمانی که سرutan باشد، پُلیور می‌پوشید. اگر پُلیور پوشیدید و هنوز سرutan است، بر روی آن یک کاپشن نیز می‌پوشید. اگر هوا بارانی است، بر روی آن یک کت بارانی نیز می‌پوشید. تمامی این لباس‌ها یک رفتار از شما را گسترش می‌دهند ولی بخشی از شما نیستند بنابراین هر زمان که به هر کدام از آنها نیاز نداشته باشید می‌توانید آن را در بیاورید.

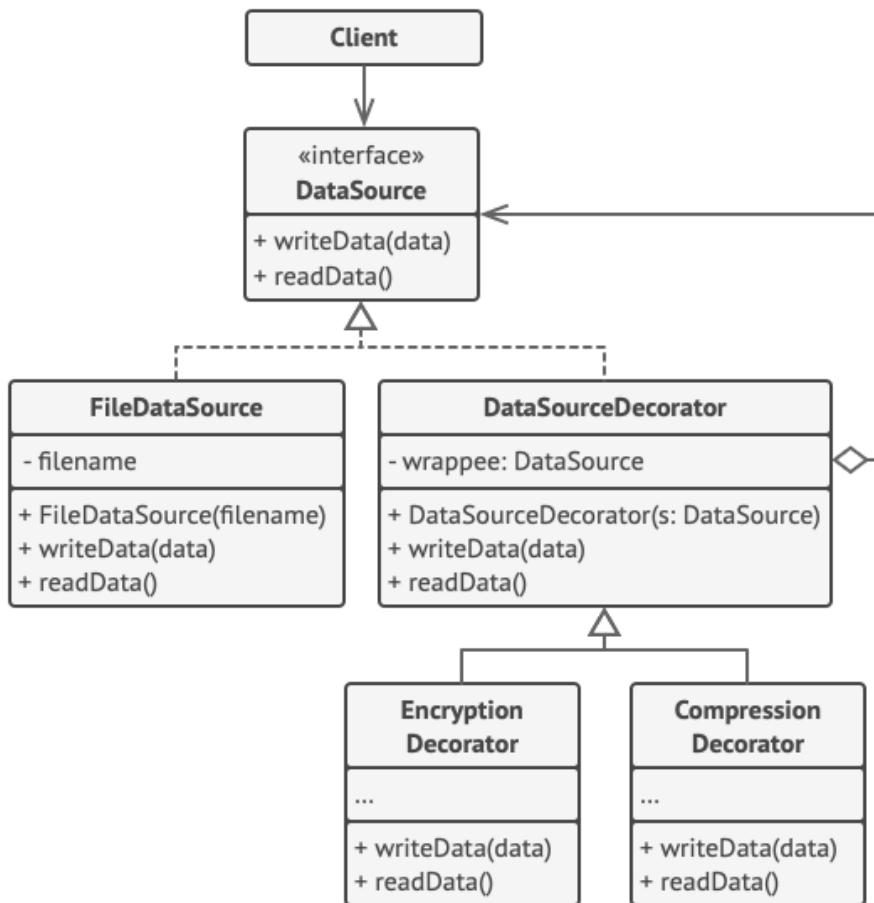
## ساختار



- **Component**: واسط مشترک بین کلاس‌ی **Wrapper** و شیء ای که قرار است wrap شود را تعریف می‌کند.
- **Concrete Component**: کلاسی است که اشیاء wrap شده را در خود نگه می‌دارد. این کلاس رفتارها را در خود نگه میدارد که هر کدام از این رفتارها می‌توانند توسط **decorator** ها تغییر کنند.
- **Base decorator**: شامل یک فیلد است که به شیء **Wrap** شده ارجاع دارد. نوع این فیلد می‌تواند از نوع واسطِ **component** باشد، بنابراین می‌تواند هم از نوع **concrete component** و یا از نوع **decorator** ها باشد. **base decorator** تمام کارها را به شیء **wrap** شده واگذار می‌کند.
- **Concrete decorators**: یک سری رفتارهای اضافی را تعریف می‌کنند که به صورت پویا می‌توان آن را به **component** اضافه کرد. **Concrete decorator** ها، متدهای **base decorator** را override می‌کنند و آنها را قبل یا بعد از اینکه توسط متدهای والدشان فراخوانی شوند، اجرا می‌کند.
- **Client**: این کلاس می‌تواند **component** ها را در چندین لایه از **decorator** ها همزمان با اشیائی که با واسطِ **component** کار می‌کنند **wrap** کند.

## مثال

در این مثال الگوی طراحی **decorator** این اجازه را به شما می‌دهد تا داده‌های حساس را مستقل از کدهایی که از این داده‌ها استفاده می‌کنند فشرده و رمزگذاری کنید.



این برنامه با استفاده از دو wrapper اشیاء را wrap می‌کند. هر دوی این ها خواندن و نوشتند در دیسک را تغییر می‌دهند :

- قبل از اینکه داده‌ها بر روی دیسک نوشته شوند، decorator ها آنها را رمزگذاری و فشرده می‌کنند.
- کلاس اصلی دیتاهای محافظت شده را در یک فایل بدون اینکه از تغییرات آن اطلاعاتی داشته باشد رمزگذاری می‌کنند.
- بلافاصله بعد از اینکه داده‌ها از دیسک خوانده شدند، از همان decorator هایی که آنها را رمزگذاری و فشرده کرده اند می‌گذرند و سپس از حالت فشرده خارج شده و رمزگشایی می‌شوند.
- و کلاس data یک واسط مشترک را پیاده‌سازی می‌کنند که این کار باعث تغییر پذیری آنها در کدهای Client می‌شود.

```

// The component interface defines operations that can be
// altered by decorators.
interface DataSource is
    method writeData(data)
    method readData():data

// Concrete components provide default implementations for the
// operations. There might be several variations of these
// classes in a program.
class FileDataSource implements DataSource is
    constructor FileDataSource(filename) { ... }
  
```

```

method writeData(data) is
    // Write data to file.

method readData():data is
    // Read data from file.

// The base decorator class follows the same interface as the
// other components. The primary purpose of this class is to
// define the wrapping interface for all concrete decorators.
// The default implementation of the wrapping code might include
// a field for storing a wrapped component and the means to
// initialize it.

class DataSourceDecorator implements DataSource is
    protected field wrappee: DataSource

    constructor DataSourceDecorator(source: DataSource) is
        wrappee = source

    // The base decorator simply delegates all work to the
    // wrapped component. Extra behaviors can be added in
    // concrete decorators.

    method writeData(data) is
        wrappee.writeData(data)

    // Concrete decorators may call the parent implementation of
    // the operation instead of calling the wrapped object
    // directly. This approach simplifies extension of decorator
    // classes.

    method readData():data is
        return wrappee.readData()

// Concrete decorators must call methods on the wrapped object,
// but may add something of their own to the result. Decorators
// can execute the added behavior either before or after the
// call to a wrapped object.

class EncryptionDecorator extends DataSourceDecorator is
    method writeData(data) is
        // 1. Encrypt passed data.
        // 2. Pass encrypted data to the wrappee's writeData
        // method.

    method readData():data is
        // 1. Get data from the wrappee's readData method.
        // 2. Try to decrypt it if it's encrypted.
        // 3. Return the result.

// You can wrap objects in several layers of decorators.

```

```

class CompressionDecorator extends DataSourceDecorator is
    method writeData(data) is
        // 1. Compress passed data.
        // 2. Pass compressed data to the wrappee's writeData
        // method.

    method readData():data is
        // 1. Get data from the wrappee's readData method.
        // 2. Try to decompress it if it's compressed.
        // 3. Return the result.

// Option 1. A simple example of a decorator assembly.
class Application is
    method dumbUsageExample() is
        source = new FileDataSource("somefile.dat")
        source.writeData(salaryRecords)
        // The target file has been written with plain data.

        source = new CompressionDecorator(source)
        source.writeData(salaryRecords)
        // The target file has been written with compressed
        // data.

        source = new EncryptionDecorator(source)
        // The source variable now contains this:
        // Encryption > Compression > FileDataSource
        source.writeData(salaryRecords)
        // The file has been written with compressed and
        // encrypted data.

// Option 2. Client code that uses an external data source.
// SalaryManager objects neither know nor care about data
// storage specifics. They work with a pre-configured data
// source received from the app configurator.
class SalaryManager is
    field source: DataSource

    constructor SalaryManager(source: DataSource) { ... }

    method load() is
        return source.readData()

    method save() is
        source.writeData(salaryRecords)
        // ...Other useful methods...

```

```
// The app can assemble different stacks of decorators at
// runtime, depending on the configuration or environment.
class ApplicationConfigurator is
    method configurationExample() is
        source = new FileDataSource("salary.dat")
        if (enabledEncryption)
            source = new EncryptionDecorator(source)
        if (enabledCompression)
            source = new CompressionDecorator(source)

        logger = new SalaryManager(source)
        salary = logger.load()
    // ...

```

چه زمانی باید از این الگو استفاده کنیم؟

- زمانی از این الگو استفاده کنید که میخواهید بدون اینکه object هایتان و کدهایی که از این object ها استفاده میکنند تغییری کنند، یک سری رفتارها را بر روی آنها اعمال کنید.
- الگوی decorator این اجازه را به شما میدهد تا منطق برنامه‌تان را در چندین لایه ساختاربندی کنید. برای هر لایه یک decorator بسازید و اشیائی با انواع ترکیب‌های منطق برنامه‌تان را در زمان اجرا بسازید. تا زمانی که این decorator ها از یک واسطه مشترک پیروی کنند، client به صورت یکسان با آنها برخورد خواهد کرد.
- زمانی از این الگو استفاده کنید که گسترش یک شیء با استفاده از ارث بری نامناسب یا غیرممکن است.

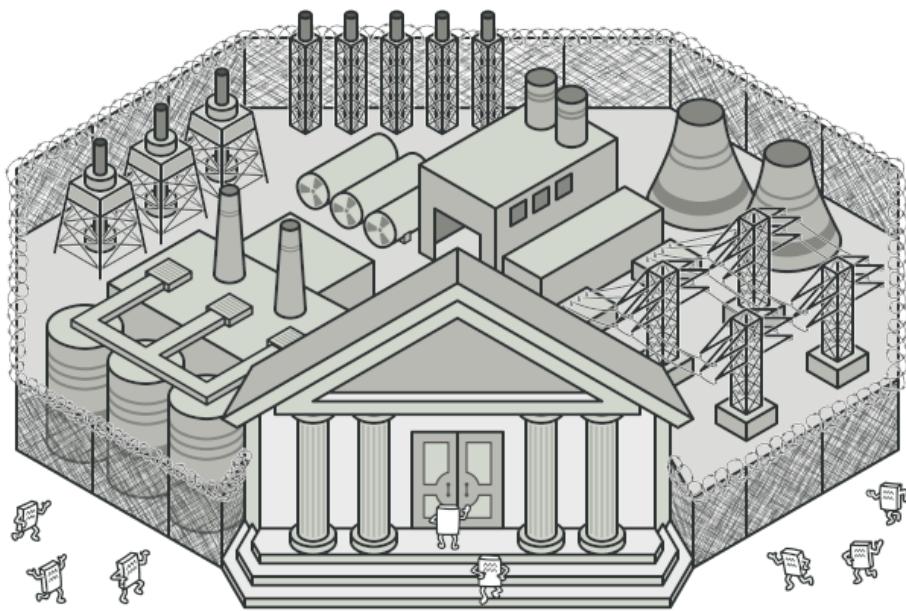
بسیاری از زبان‌های برنامه‌نویسی کلیدواژه‌ایی به نام final دارند. این کلیدواژه مانع از گسترش کلاس‌مان می‌شود. برای کلاس‌های فاینان تنها راهی که بتوان دوباره از رفتارهایش استفاده کرد، ایجاد یک کلاس wrapper با استفاده از الگوی طراحی decorator است.

## معایب و مزایا

- ❖ می‌توانید یک شیء را بدون ایجاد زیر-کلاس گسترش دهید.
- ❖ می‌توانید یک سری مسئولیت‌ها را در زمان اجرا به شیء خود اضافه کنید.
- ❖ می‌توانید با استفاده از wrap کردن اشیاء در decorator ها رفتارهایشان را باهم ترکیب کنید.
- ❖ با استفاده از این الگو اصل single responsibility رعایت می‌شود. شما می‌توانید یک کلاس یکپارچه که رفتارهای گوناگونی را پیاده سازی می‌کند به کلاس‌های کوچکتری تبدیل کنید.
- اینکه یک wrapper خاص را از پشتی‌های wrapper ها حذف کنید کمی سخت است.
- اینکه بخواهید یک decorator ای که با ترتیب decorator های موجود در پشتی همخوانی ندارد را پیاده سازی کنید کمی دشوار است.
- تنظیم اولیه لایه‌ها در کد خیلی زشت می‌شود!

## Facade

با استفاده از الگوی طراحی facade می‌توانید برای یک کتابخانه، فریمورک و یا هر کلاس پیچیده‌ای یک واسط ساده بسازید.



### طرح مسئله

فرض کنید کدتان باید با یک سری شیء که مربوط به یک کتابخانه و یا فریمورک پیچیده هستند کار کند. باید تمام این Object‌ها را مقداردهی اولیه کنید، dependency‌های مربوطه را اعمال کنید و متدها را به ترتیب third-party اجرا کنید. با این کار منطق اصلی برنامه‌تان بیش از حد به پیاده‌سازی کلاس‌هایی که برای هستند وابسته می‌شود. این کار نگهداری و درک کد را دشوار می‌کند.

### راه حل

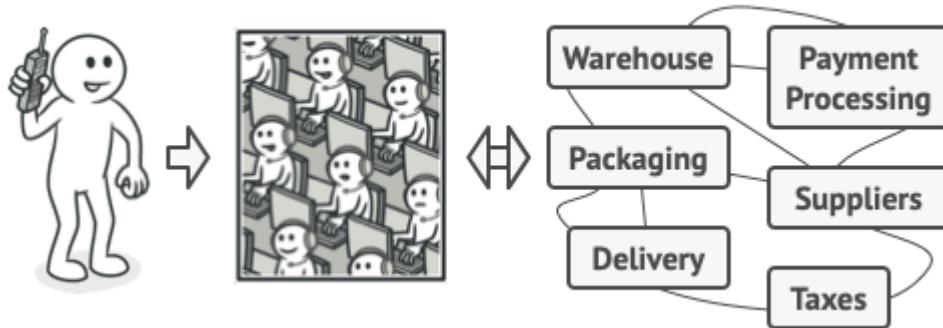
یک واسط ساده برای سیستمهای پیچیده که شامل قطعات متغیری هستند ایجاد می‌کند. یک facade ممکن است عملکرد های محدودی را برای ارتباط مستقیم به یک سیستم ارائه دهد. با این حال سعی بر این است تا feature‌هایی را ارائه دهد که برای client مهم است.

داشتن facade زمانی مهم است که نیاز دارید برنامه خود را با کتابخانه‌ی پیچیده‌ای که دارای دهها ویژگی است یکپارچه کنید و در عین حال به همه‌ی functionality‌های آن کتابخانه نیاز ندارید.

به طور مثال یک برنامه که یک سری ویدیوهای خنده دار از گربه‌ها را در فضای مجازی به اشتراک می‌گذارد می‌تواند از یک کتابخانه‌ی تبدیل ویدیوی حرفه‌ای استفاده کرده باشد!! در صورتی که تمام چیزی که این برنامه

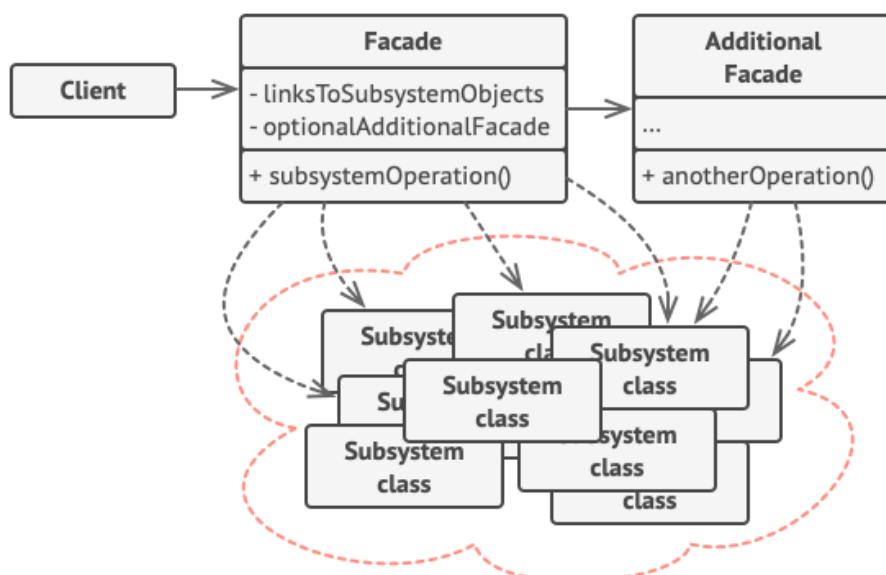
از آن کتابخانه نیاز دارد فقط متدهای با عنوان `encode(filename, format)` است. بعد از ایجاد چنین کلاسی و اتصال آن به کتابخانه مورد نظر، شما اولین `facade` خود را دارید.

### مقایسه با دنیای واقعی



زمانی که برای یک سفارش تلفنی با یک مغازه تماس می‌گیرید، اپراتور تلفن نمایانگر تمامی خدمات آن فروشگاه است. این اپراتور یک رابط صوتی ساده برای سیستم سفارش، درگاه پرداخت و خدمات مختلف تحويل را در اختیار شما قرار می‌دهد.

### ساختار

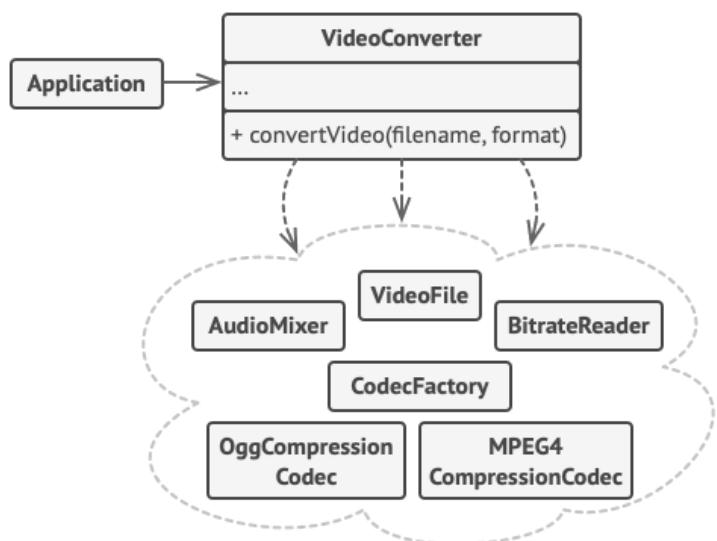


- **Facade**: دسترسی راحت به بخش خاصی از عملکرد یا functionality یک زیر سیستم را فراهم می‌کند. این کلاس می‌داند که چگونه درخواست‌های client را هدایت کند.

- **Additional facade**: از این کلاس می‌توان برای جلوگیری از آلوده شدن یک facade به ویژگی‌های نامرتبه که ممکن است باعث پیچیدگی آن شود استفاده کرد. این کلاس هم توسط Client و هم توسط دیگر facade‌ها می‌تواند مورد استفاده قرار گیرد.
  - **Complex subsystem**: از ده‌ها سیستم‌های زیرشاخه‌ی پیچیده تشکیل شده است. برای اینکه هر کدام از این سیستم‌های زیرشاخه کار مفیدی انجام دهند باید به جزئیات عمیق پیاده‌سازی آنها نگاه کرد مانند مقدار دهی اولیه‌ی Object‌ها و یا ارائه‌ی آنها به ترتیب و فرمت مناسب.
  - کلاس‌های سیستم‌های زیرشاخه از وجود facade‌ها آگاه نیستند. آنها در درون سیستم‌ها کار می‌کنند و مستقیماً با یکدیگر تعامل دارند.
  - **Client**: این بخش به جای ارتباط مستقیم با سیستم‌ها با facade‌ها ارتباط برقرار می‌کند.

## مثال

در این مثال الگوی طراحی facade ارتباط با فریمورک تیدیل ویدئو را برایمان ساده سازی می‌کند.



به جای اینکه کد شما به طور مستقیم با دهها کلاس از یک فریمورک کار کند، یک کلاس facade ایجاد کنید تا functionality های موردنظرتان را مدیریت کند و آن‌ها را از بقیه‌ی کدتان پنهان کند.

این کار به شما کمک می‌کند تا کمتر به ارتقای نسخه‌ی آن فریمورک و یا تغییر به یک فریمورک دیگر وابسته باشید. تنها چیزی که شما نیاز دارید، تغییر کلاس facade است.

```
// These are some of the classes of a complex 3rd-party video  
// conversion framework. We don't control that code, therefore  
// can't simplify it.
```

```
class VideoFile
```

```

class OggCompressionCodec
// ...

class MPEG4CompressionCodec
// ...

class CodecFactory
// ...

class BitrateReader
// ...

class AudioMixer
// ...

// We create a facade class to hide the framework's complexity
// behind a simple interface. It's a trade-off between
// functionality and simplicity.
class VideoConverter is
    method convert(filename, format) :File is
        file = new VideoFile(filename)
        sourceCodec = (new CodecFactory).extract(file)
        if (format == "mp4")
            destinationCodec = new MPEG4CompressionCodec()
        else
            destinationCodec = new OggCompressionCodec()
        buffer = BitrateReader.read(filename, sourceCodec)
        result = BitrateReader.convert(buffer, destinationCodec)
        result = (new AudioMixer()).fix(result)
        return new File(result)

// Application classes don't depend on a billion classes
// provided by the complex framework. Also, if you decide to
// switch frameworks, you only need to rewrite the facade class.
class Application is
    method main() is
        convertor = new VideoConverter()
        mp4 = convertor.convert("funny-cats-video.ogg", "mp4")
        mp4.save()

```

## چه زمانی باید از این الگو استفاده کنیم؟

- زمانی از این الگو استفاده کنید که نیاز به یک واسط محدود و ساده برای ارتباط با یک سیستم پیچیده دارید.

در بیشتر اوقات، با گذشت زمان سیستم‌ها یا همان سیستم‌های زیرشاخصه پیچیده تر می‌شوند. حتی استفاده از الگوهای طراحی هم باعث بیشتر شدن کلاس‌ها و این پیچیدگی می‌شوند. یک سیستم کوچک می‌تواند منعطف تر و قابل استفاده‌تر باشد اما این را در نظر بگیرید که با تعدد آنها باید config‌های بیشتری نیز انجام دهید و هر لحظه هم ممکن است تعدادشان بنا به درخواست client بیشتر شود. Facade این قابلیت را در اختیارمان قرار می‌دهد تا تنها با پیاده‌سازی feature‌های مورد نیاز و ایجاد shortcut از آنها، بیشتر نیازهای Client را برآورده کنیم.

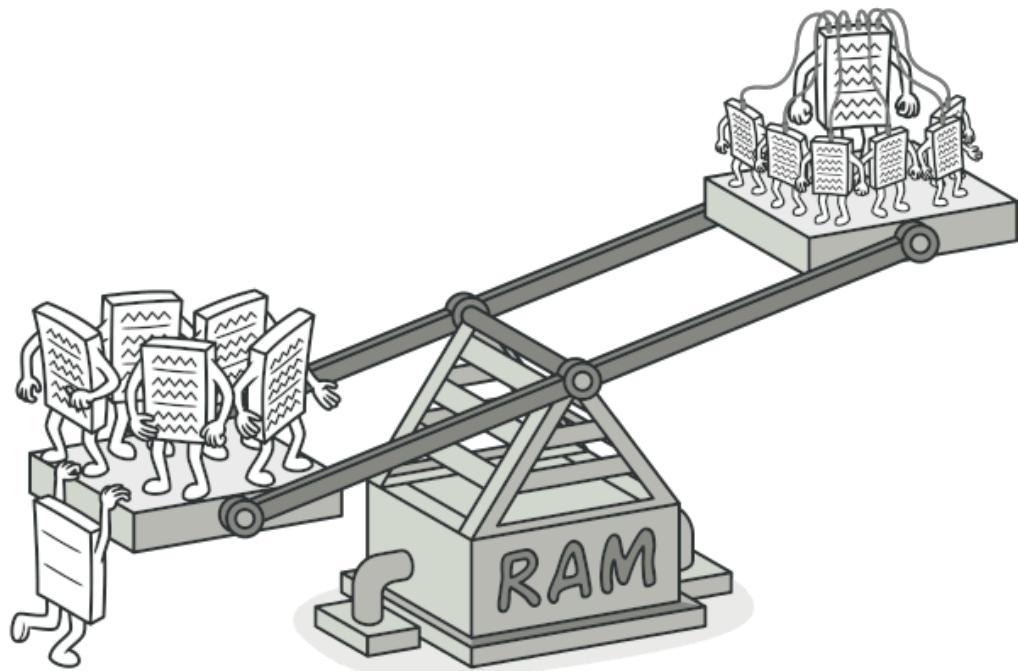
- زمانی از این الگو استفاده کنید که می‌خواهید سیستم‌های تودرتو بسازید.  
برای تعیین نقاط ورود به هر کدام از سیستم‌ها یک facade ایجاد کنید. شما می‌توانید از facade‌ها به عنوان تنها راه ارتباطی این سیستم‌ها استفاده کنید.  
برای درک بهتر ببایید به مثال فریم ورک تبدیل کننده‌ی ویدئو برگردیم. می‌توانیم این مورد را به دو لایه بشکنیم. کارهای مربوط به ویدئو و کارهای مربوط به صدا. می‌توانیم برای هرکدام این لایه‌ها یک facade بسازیم و هر کدام از این لایه‌ها از این facade‌ها برای ارتباط با هم استفاده کنند.

## معایب و مزایا

- ❖ پیچیدگی سیستم‌ها را از کدتان جدا می‌کنید.
- یک facade می‌تواند به یک شیء اصلی یا همان God Object تبدیل شود که با تمامی قسمت‌های برنامه در ارتباط است.

## Flyweight

با استفاده از این الگوی طراحی می‌توانید Object های بیشتری را در حافظه RAM خود نگه دارید. به جای اینکه تمامی داده‌ها را در یک شیء نگهداری کنید، با به اشتراک گذاری رفتارهای مشترک مدیریت حافظه را بهبود ببخشید.



### طرح مسئله

برای داشتن یک سرگرمی بعد از ساعتها کار، تصمیم می‌گیرید که یک videogame ساده بسازید. در این بازی، بازیکنان در نقشه حرکت می‌کنند و به سمت همدیگر شلیک می‌کنند.

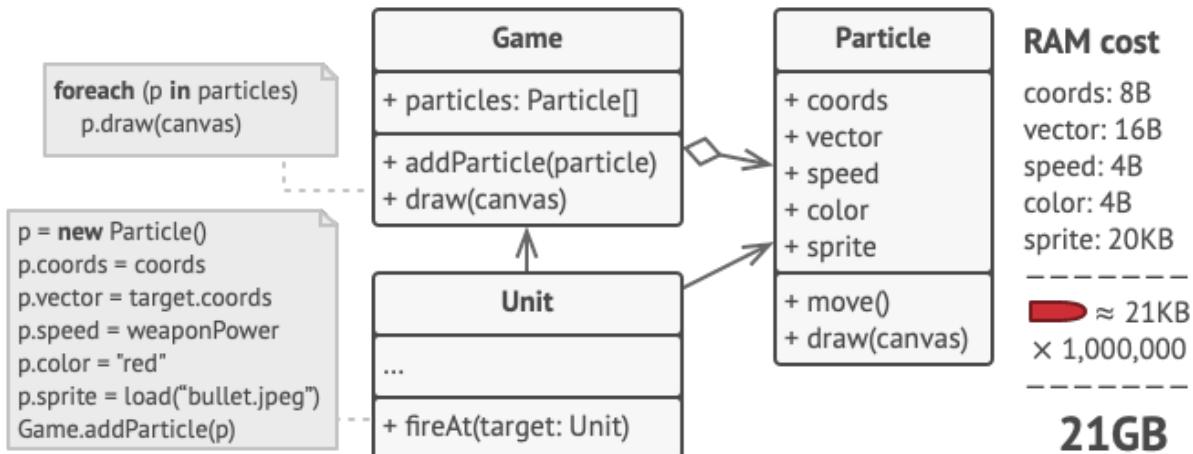
شما تصمیم گرفته‌اید تا یک سیستم برای مدیریت حرکت اشیاء در محیط را در بازی خود پیاده‌سازی کنید و آن را به یک ویژگی خاص از بازی خود تبدیل کنید. تعداد زیادی از گلوله‌ها، موشک‌ها و ترکش‌های انفجار باید در سرتاسر نقشه پخش شوند و یک فضای هیجان انگیز را برای مخاطب به ارمغان بیاورند.

آخرین نسخه از برنامه‌تان را نوشته‌اید. Commit می‌کنید و آن را برای دوست خود می‌فرستید تا تست‌های مربوط به drive را انجام دهد. در عین حال که بازی بسیار روان بر روی سیستم شما اجرا می‌شد، دوست‌تان نمی‌تواند برای مدت طولانی بازی را اجرا کند! مدتی بعد از شروع بازی، crash می‌کند و از بازی خارج می‌شود. بعد از ساعتها تلاش و دیباگ و خواندن لاغ‌ها متوجه می‌شوید که عملیات crash بخار ناکافی بودن میزان **RAM** است.

معلوم شد که مشخصات سیستم دوست‌تان از سیستم شما ضعیفتر است و به همین دلیل خیلی سریع با مشکل مواجه شد.

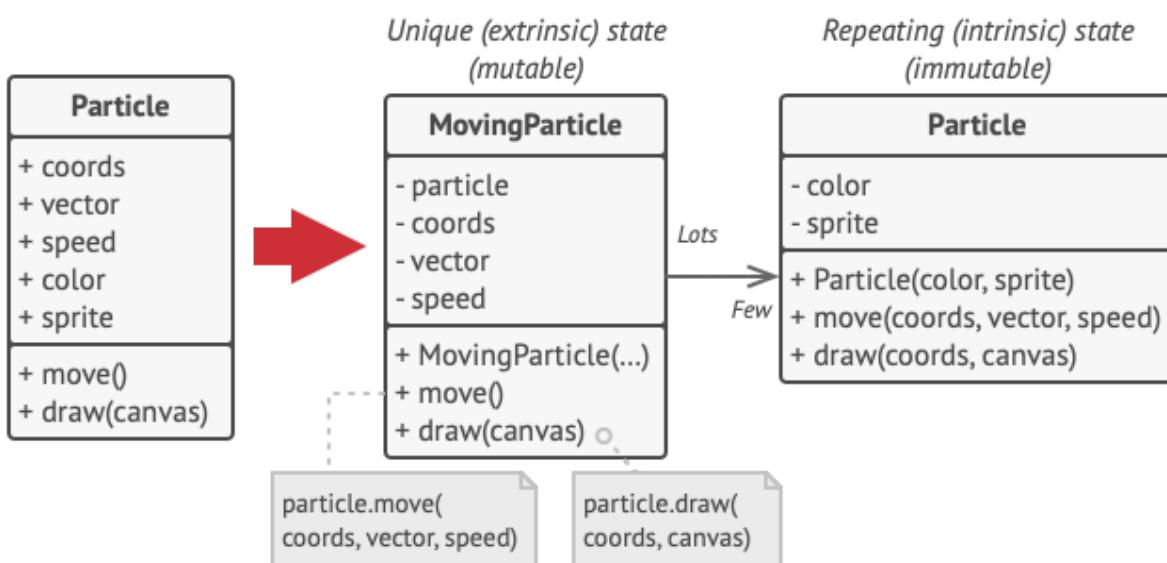
مشکل اصلی از سیستم مدیریت حرکاتی است که در بازی پیاده‌سازی کرده بودید. هر کدام از بخش‌های این سیستم مانند یک گلوله، یک موشک و یا یک ترکش یک Object هایی با داده‌های بسیار زیادی هستند.

زمانی که در صحنه‌ی بازی یک قتل عام رخ دهد(!) دیگر فضای کافی در RAM برای نگهداری این حجم از داده‌ها وجود ندارد. بنابراین برنامه از کار می‌افتد. ساختار زیر را برای بازی خود در نظر بگیرید. برای حل این مشکل چه راه حلی به ذهن شما می‌رسد؟



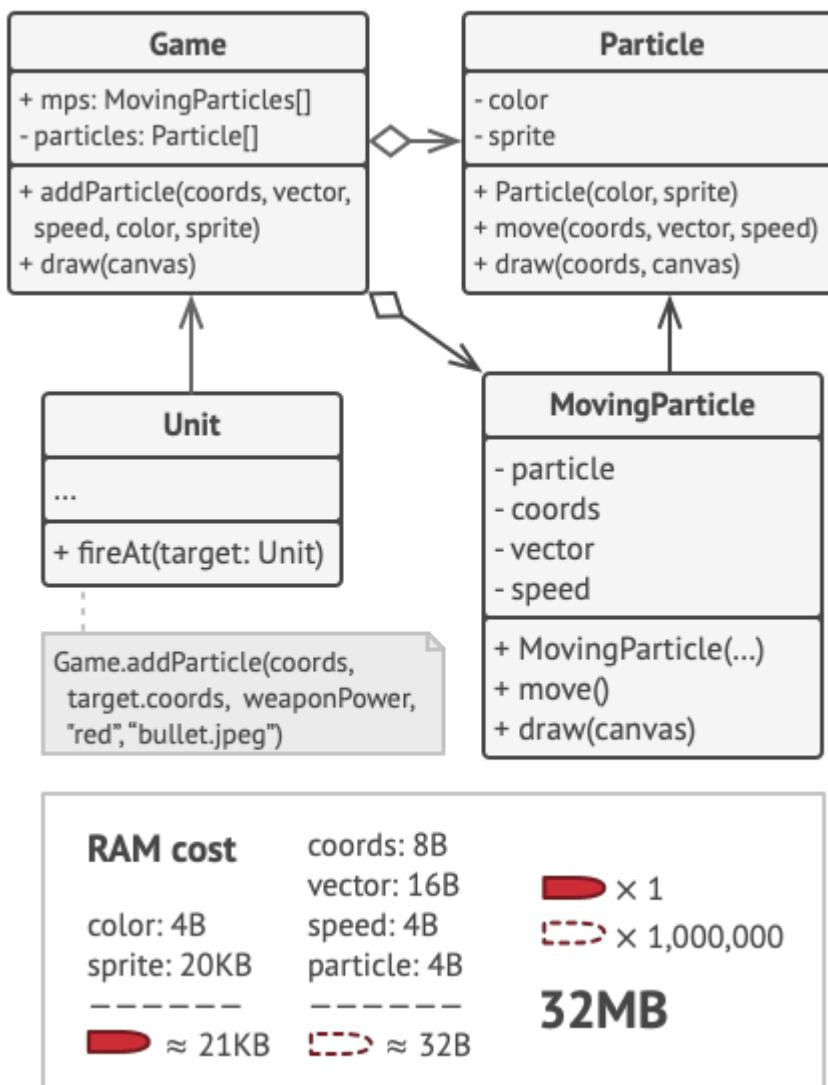
## راه حل

با بررسی بیشتر کلاس particle (ذره) متوجه می‌شویم که فیلدی‌های color و sprite (عکسی که نمایانگر آن ایمان است) مقدار حافظه‌ی بیشتری را مصرف می‌کنند. بدتر این است که این دو فیلد داده‌های تقریباً یکسانی را در همه ذرات ذخیره می‌کنند. به طور مثال یک گلوله رنگ و sprite یکسانی دارد.



در بخش دیگری از یک ایمان، مواردی مانند مختصات، بردار سرعت و حرکت برای همهی المان‌ها یکسان است. با این حال مقدار این فیلدها در طول زمان تغییر خواهد کرد. اما رنگ و sprite در طول بازی یکسان خواهد بود.

به این داده‌های یکسان یک شیء *intrinsic state* یا حالت ذاتی می‌گویند. این حالت از بدرو تولد یک شیء با آن زندگی می‌کند. اشیاء دیگر فقط می‌توانند از آنها استفاده کنند و قادر به تغییر آنها نخواهند بود. به آن قسمت از یک شیء که از بیرون قابل تغییر است حالت بیرونی یا *extrinsic state* می‌گویند. الگوی طراحی Flyweight پیشنهاد می‌کند تا از ذخیره‌ی این حالات بیرونی در درون شیء خودداری کنید و این کار را به متدهای خاصی بسپارید و تنها حالات ذاتی شیء را در آن نگه دارید. این کار این اجازه را به شما می‌دهد تا در زمینه‌های متفاوتی از شیء خود استفاده کنید. در نتیجه شما به تعداد کمتری از این اشیاء نیاز دارید زیرا آنها فقط در حالتهای ذاتی با هم متفاوتند و تنوع کمتری در حالتهای بیرونی دارند.

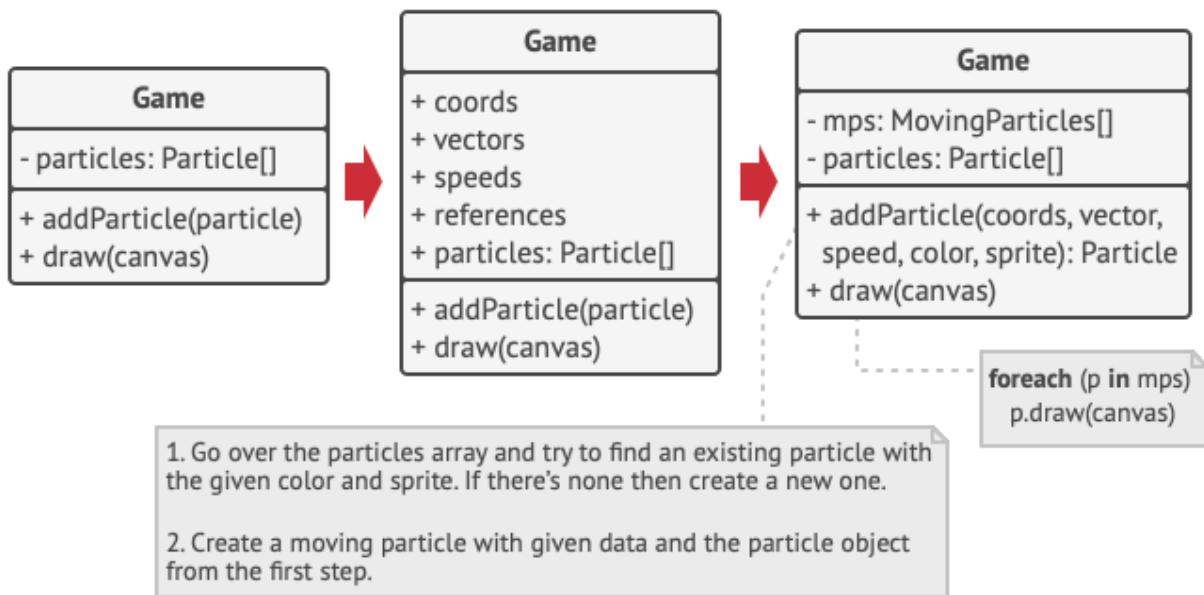


باید به بازی ای که ساختیم برگردیم. با فرض اینکه حالات بیرونی را از المان‌های خود خارج کرده باشیم، تنها سه جسم مختلف برای نمایش تمام المان‌ها در بازی کافی است. یک گلوله، یک موشک و یک قطعه ترکش. احتمالاً تا الان حدس زده باشید که آن Object ای که حالات ذاتی را در خود نگه می‌دارد یک flyweight است.

### ذخیره‌سازی حالات بیرونی

به نظرتان این حالات بیرونی کجا ذخیره می‌شوند. شاید بعضی از کلاس‌ها آنها را ذخیره می‌کنند، اینطور نیست؟ در بیشتر مواقع این حالات به Object هایی تحت عنوان Container Object منتقل می‌شوند که قبل از اینکه الگویی را پیاده‌سازی کنیم در آن ذخیره می‌شوند.

در مورد این بازی، شیء اصلی Game است که تمامی المان‌ها را در فیلد particles نگهداری می‌کند. برای اینکه حالات بیرونی را به این کلاس منتقل کنیم، باید چندین فیلد از نوع آرایه برای ذخیره‌سازی مختصات و بردارهای سرعت هر المان ایجاد کنیم. ولی این همه‌ی کاری که باید انجام دهیم نیست. شما به یک آرایه‌ی دیگر برای ذخیره‌سازی reference های به یک flyweight خاص که نشان دهنده‌ی یک المان است هم نیاز دارید. این آرایه‌ها باید با هم sync یا همگام باشند تا با استفاده از یک index به اطلاعات المان مورد نظرتان دسترسی داشته باشید.



یک راه حل زیباتر این است که کلاسی جدا بسازیم و حالات بیرونی به همراه reference هایشان را در آن نگه داریم. با این رویکرد فقط نیاز به یک آرایه در کلاس Container مان داریم. لحظه‌ای صبر کنید! آیا نیاز نیست به همان تعدادی که در ابتدا داشتیم از این Object ها داشته باشیم؟ از نظر فنی پاسخ "بله" است. ولی چیزی که مشخص است این است که این اشیاء خیلی کوچکترند. بیشتر فیلدهایی که حافظه‌مان را مصرف می‌کردند فقط به تعداد کمی از اشیاء flyweight منتقل شدند. این به این

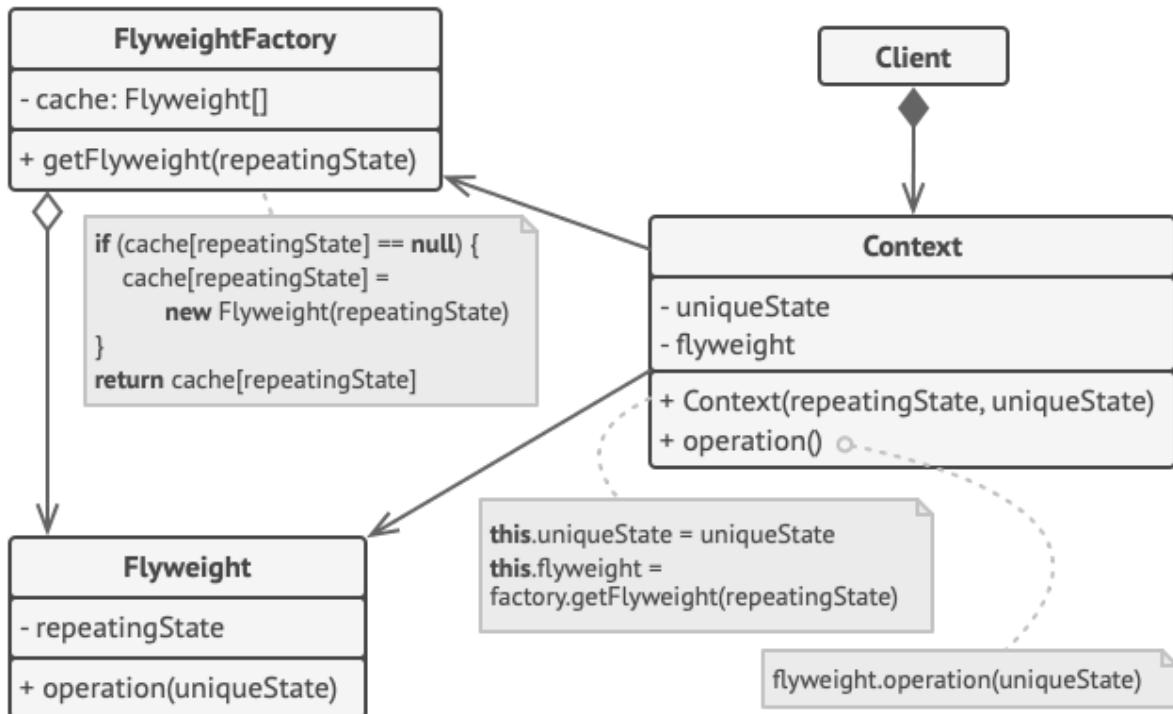
معنا است که هزاران اشیاء کوچک می‌توانند از یک شیء سنگین flyweight به جای ذخیره هزاران نسخه از داده‌ها استفاده کنند. به این ترتیب، مصرف حافظه بهبود یافته و عملکرد بهتری برای برنامه خواهیم داشت.

### **Flyweight و تغییر ناپذیری**

فرض کنید یک شیء Flyweight دارید که باید در موارد مختلفی مورد استفاده قرار گیرد. حالا، اگر بخواهیم این شیء را تغییر دهیم (به عبارتی وضعیتش را تغییر دهیم)، ممکن است اثرات غیرمنتظره‌ای در دیگر قسمت‌های برنامه ایجاد شود. در الگوی طراحی Flyweight، اصل بر این است که وقتی یک شیء را ایجاد می‌کنیم، مقدارش را فقط یک بار می‌توانیم تنظیم کنیم (مثلًا از طریق سازنده). بعد از آن دیگر نمی‌توانیم وضعیت داخلی‌اش را تغییر دهیم. به عبارت ساده‌تر، مثل یک «شیء فقط خواندنی» است. یک بار ایجاد می‌شود و می‌توانیم از آن در موارد مختلف استفاده کنیم، اما دیگر نمی‌توانیم مقدارش را تغییر دهیم. این کار از ایجاد مشکلات غیرمنتظره در برنامه جلوگیری می‌کند.

### **Flyweight کارخانه‌ی**

برای دسترسی راحت‌تر به Flyweight Object‌ها، می‌توانیم متدهایی بنویسیم که وظیفه‌ی ایجاد این اشیاء را برعهده بگیرند. این متدها حالات ذهنی یک Flyweight را دریافت می‌کنند و در صورتی که Flyweight ای با مشخصات درخواستی وجود داشته باشد آن را بر می‌گردانند در غیر اینصورت یک Flyweight جدید با حالات ذاتی وارد شده ایجاد می‌کنند و آن را به Flyweight pool اضافه می‌کنند.  
گزینه‌های متفاوتی برای اینکه این متدها را در کجا برنامه‌مان قرار دهیم وجود دارد. واضح ترین کاری که می‌توان انجام داد این است که کلاسی تحت عنوان Flyweight Container بسازیم یا یک factory class برای انجام این کار ایجاد کنیم.



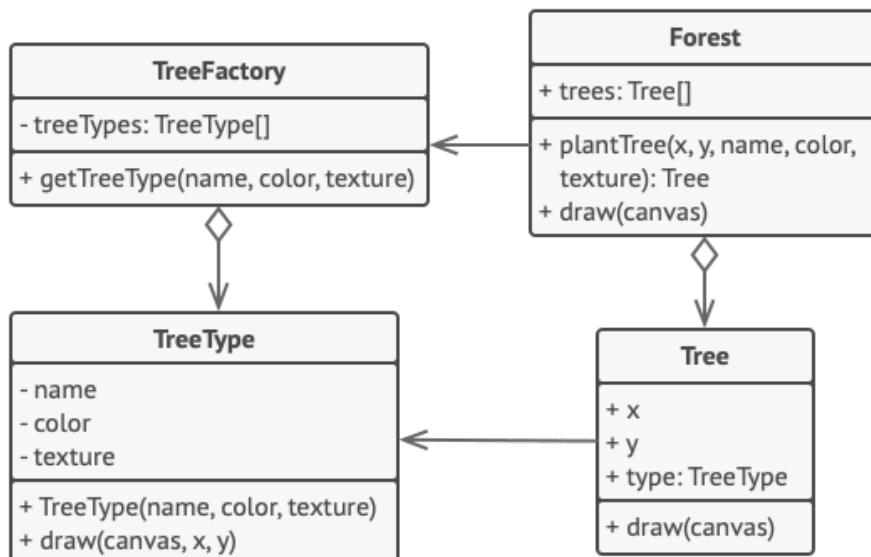
- **Flyweight Pattern:** این الگو یک راه بهینه سازی است. پس قبل از اینکه آن را پیاده‌سازی کنید، از این که برنامه‌تان به دلیل زیاد شدن تعداد Object ها مشکل مصرف RAM دارد مطمئن شوید. و اطمینان حاصل کنید که این مشکل را نمی‌توان با راه حل معنادار دیگری حل کرد.
- **Flyweight:** این کلاس شامل بخشی از حالات اصلی یک شیء است که می‌تواند بین چندین شیء دیگر به اشتراک گذاشته شود. از Flyweight های مشترک می‌توان در زمینه های مختلف استفاده کرد. به حالتی که در یک Flyweight ذخیره می‌شود حالت ذاتی یا درونی و به حالاتی که به این Flyweight ها پاس داده می‌شوند حالات بیرونی می‌گویند.
- **Context:** این کلاس شامل تمامی حالات بیرونی است که در تمام شیء های اصلی منحصر به فرد است. زمانی که یک context به یک flyweight وصل شود، تمامی حالات یک شیء پوشش داده می‌شود.
- تصور کنید یک شیء Flyweight داریم که برخی از رفتارهای خود را دارد. در بیشتر مواقع، این رفتارها در همان کلاس Flyweight باقی می‌مانند. حالا، وقتی کسی می‌خواهد یکی از این رفتارها را اجرا کند، باید همه جزئیات مورد نیاز از حالات بیرونی را به عنوان ورودی به متدهای فراخوانی شده بدهد. حالا یک انتخاب دیگر هم وجود دارد. می‌توانیم این رفتارها را به کلاس Context منتقل کنیم. در این صورت، کلاس Context به عنوان یک نوع جعبه اطلاعاتی از Flyweight استفاده می‌کند و رفتارها به عنوان

یک نوع داده مورد استفاده قرار می‌گیرند. این کار می‌تواند موجب ساده‌تر شدن کدها و کاهش پیچیدگی شود.

- در این لایه، وضعیت حالات بیرونی شناسایی و یا ذخیره می‌شوند. از دیدگاه Client، شیء Flyweight یک شیء الگو است که می‌تواند در زمان اجرا با ارسال برخی از داده‌های متغیر به پارامترهای متدهای آن Config شود.
- factory: این کلاس Flyweight های موجود را مدیریت می‌کند. با استفاده از کدهای client به طور مستقیم Flyweight ایجاد نمی‌کنند. در عوض فقط تعدادی از حالات ذاتی یک شیء را به Flyweight مورد نظر پاس می‌دهند. ابتدا به دنبال Flyweight درخواستی Flyweight می‌گردد. اگر Flyweight ای وجود داشته باشد آن را برمی‌گرداند در غیر این صورت یک Flyweight جدید می‌سازد.

## مثال

در این مثال الگوی طراحی Flyweight به ما کمک می‌کند تا میزان حافظه‌ی مورد استفاده را برای نمایش تعداد زیادی درخت بر روی یک بوم نقاشی کاهش دهیم.



در الگوی Flyweight اطلاعات مشترک (مثل نوع درخت) از کلاس اصلی درخت خارج شده و در کلاس جدیدی به نام TreeType قرار می‌گیرد. حالا به جای این که هر درخت این اطلاعات را خودش نگه دارد، این اطلاعات در چندین شیء Flyweight قرار می‌گیرد و به درخت‌های مختلف متصل می‌شوند. زمانی که می‌خواهیم یک درخت جدید رارسم کنیم، از Flyweight factory، از استفاده می‌کنیم. این کارخانه پیچیدگی را از ما پنهان می‌کند و به ما امکان می‌دهد درخت را ایجاد کنیم و یا اگر قبلًا ایجاد شده باشد، از آن استفاده

کنیم. این کار به ما کمک می‌کند تا از حافظه بهتر استفاده کنیم و میلیون‌ها درخت را با بهره‌وری بیشتری رسم کنیم.

```
// The flyweight class contains a portion of the state of a
// tree. These fields store values that are unique for each
// particular tree. For instance, you won't find here the tree
// coordinates. But the texture and colors shared between many
// trees are here. Since this data is usually BIG, you'd waste a
// lot of memory by keeping it in each tree object. Instead, we
// can extract texture, color and other repeating data into a
// separate object which lots of individual tree objects can
// reference.
class TreeType is
    field name
    field color
    field texture
    constructor TreeType(name, color, texture) { ... }
    method draw(canvas, x, y) is
        // 1. Create a bitmap of a given type, color & texture.
        // 2. Draw the bitmap on the canvas at X and Y coords.

// Flyweight factory decides whether to re-use existing
// flyweight or to create a new object.
class TreeFactory is
    static field treeTypes: collection of tree types
    static method getTreeType(name, color, texture) is
        type = treeTypes.find(name, color, texture)
        if (type == null)
            type = new TreeType(name, color, texture)
            treeTypes.add(type)
        return type

// The contextual object contains the extrinsic part of the tree
// state. An application can create billions of these since they
// are pretty small: just two integer coordinates and one
// reference field.
class Tree is
    field x,y
    field type: TreeType
    constructor Tree(x, y, type) { ... }
    method draw(canvas) is
        type.draw(canvas, this.x, this.y)

// The Tree and the Forest classes are the flyweight's clients.
// You can merge them if you don't plan to develop the Tree
// class any further.
class Forest is
    field trees: collection of Trees
```

```

method plantTree(x, y, name, color, texture) is
    type = TreeFactory.getTreeType(name, color, texture)
    tree = new Tree(x, y, type)
    trees.add(tree)

method draw(canvas) is
    foreach (tree in trees) do
        tree.draw(canvas)

```

## چه زمانی باید از این الگو استفاده کنیم؟

- زمانی از این الگو استفاده کنید که برنامه‌تان باید از تعداد زیادی از اشیاء پشتیبانی کند و

این تعداد زیاد از شیء به سختی در RAM قرار می‌گیرند.

بهره بردن از مزایای این الگو به شدت به اینکه کجا و چگونه از آن استفاده کنید وابسته است:

1. برنامه نیاز به ایجاد تعداد زیادی اشیاء مشابه دارد.
2. این اشیاء باعث پرشدن تمام حافظه موجود می‌شوند.
3. اشیاء دارای وضعیت‌های تکراری هستند که می‌توانند از بین چندین اشیاء استخراج و به اشتراک گذاشته شوند.

## معایب و مزایا

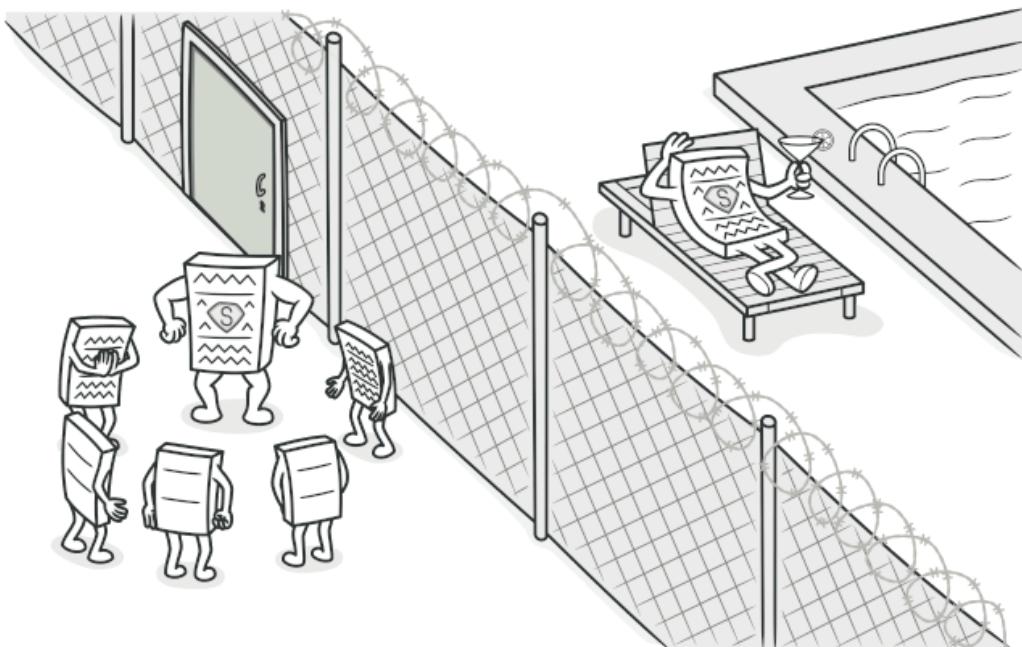
❖ زمانی که برنامه تان هزاران اشیاء تکراری دارد به راحتی می‌توانید درصد زیادی از RAM را ذخیره کنید.

→ زمانی که از الگوی Flyweight استفاده می‌کنید در واقع حافظه‌ی RAM دستگاه‌تان را با چرخه‌های پردازشی CPU معامله می‌کنید. این معامله زمانی رخ می‌دهد که بخشی از اطلاعات مرتبط با محیط باید هر بار که متند یک شی Flyweight فراخوانی می‌شود، دوباره محاسبه شوند.

→ کدهایمان به شدت پیچیده‌تر می‌شوند، اعضای جدید تیم همیشه سوال می‌کنند که چرا وضعیت‌های یک موجودیت به این گونه از هم جدا شده‌اند!

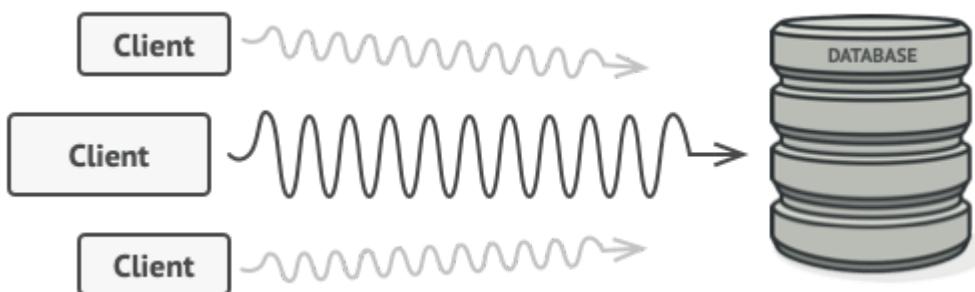
## Proxy

با استفاده از این الگو می‌توانید برای اشیاء خود یک جایگزین بسازید. یک proxy دسترسی‌ها به یک object و عملیاتی که باید قبل یا بعد از ارسال درخواست استفاده از آن object انجام شوند را کنترل می‌کند.



### طرح مسئله

سوالی که پیش می‌آید این است که چرا اصلاً باید دسترسی به یک Object را کنترل کنیم؟ بباید یک مثال بزنیم. فرض کنید شیء عظیمی دارید که مقدار زیادی از منابع سیستم را مصرف می‌کند. بنا به کاری که انجام می‌دهید هر از گاهی به این سیستم نیاز پیدا می‌کنید. نه همیشه!

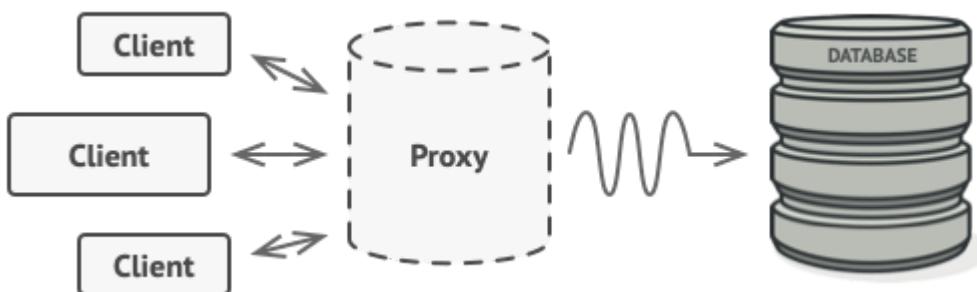


می‌توانیم از روش Lazy استفاده کنیم، یعنی هر زمان که واقعاً به آن شیء نیاز داشتیم آن را بسازیم. در این صورت تمام کدهایی که از این اشیاء استفاده می‌کنند باید مراحلی را انجام بدھند تا این اشیاء را با این روش بسازند. این کار باعث تکرار کد می‌شود و مشکل‌هایی را ایجاد خواهد کرد.

در حالت ایده آل، میخواهیم کدهای مربوط به ایجاد اشیاء به روش Lazy را در خود کلاس شیء قرار دهیم. اما این همیشه قابل اجرا نیست. چرا که ممکن است کلاسمان جزو یک کتابخانه‌ی خارجی از یک third-party باشد. راه حل شما برای ایجاد این اشیاء چیست؟؟

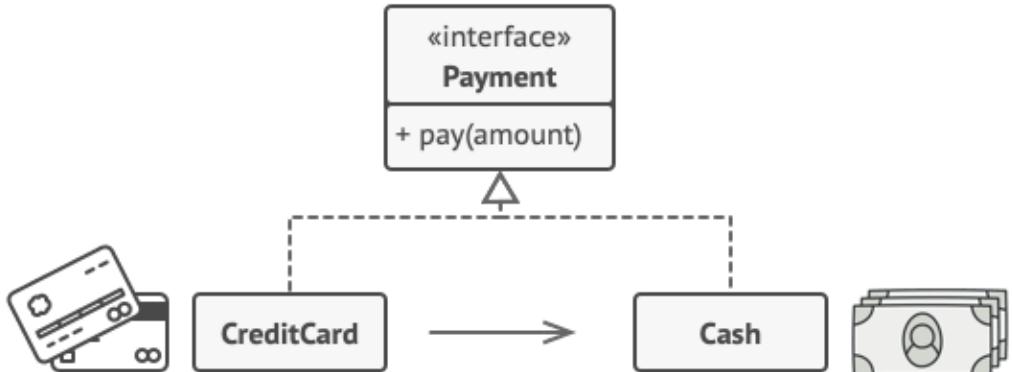
## راه حل

الگوی طراحی proxy برای حل این مورد پیشنهاد می‌کند تا کلاس جدیدی تحت عنوان Proxy با همان ویژگی‌ها و رفتارهای شیء اصلی برنامه‌مان که خدماتی را ارائه می‌کند ایجاد کنیم. سپس برنامه خود را به گونه‌ای تغییر دهیم تا به جای اینکه شیء اصلی را ارسال کنیم، شیء Proxy را به تمامی client‌ها منتقال دهیم. زمانی که یک client درخواستی را ارسال می‌کند، proxy یک شیء واقعی از شیء ای که سرویس ارائه می‌کند را ایجاد می‌کند و تمام کارها را به آن منتقل می‌کند. این کار به ما این امکان را می‌دهد تا به عنوان یک واسطه (proxy) کنترل بیشتری بر روی ارتباط client با شیء سرویس دهنده داشته باشیم.



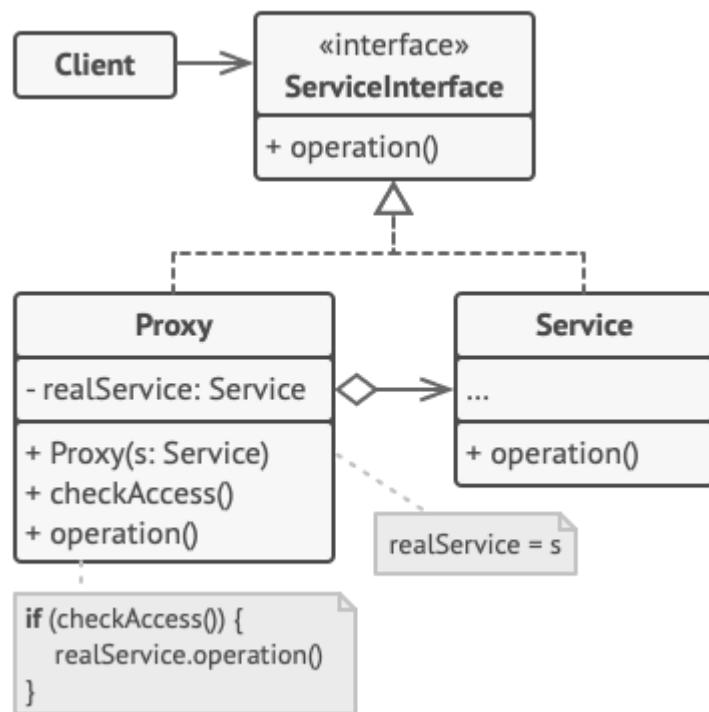
اما این کار چه فایده‌ای دارد؟ اگر نیاز به اجرای عملیاتی قبل و یا بعد از منطق اصلی کلاس دارید، proxy به شما این امکان را می‌دهد تا این کار را بدون تغییر در آن کلاس انجام دهید. از آنجایی که همان واسطه کلاس اصلی را پیاده‌سازی می‌کند، می‌تواند به هر client که انتظار یک شیء واقعی را دارد، منتقل شود.

## مقایسه با دنیای واقعی



در دنیای واقعی، کارت‌های اعتباری هم‌زمان هم نمایندهٔ حساب بانکی شما و هم نمایندهٔ پول‌های نقد شما هستند. هر دوی اینها یک رابط مشترک دارند؛ به عبارت دیگر، هر دو می‌توانند برای انجام پرداخت مورد استفاده قرار گیرند. کاربر از اینکه دیگر نیاز نیست مقدار زیادی پول را همراه خود حمل کند احساس رضایت می‌کند. صاحب مغازه نیز خوشحال است زیرا درآمد حاصل از تراکنش به صورت الکترونیکی به حساب بانکی مغازه اضافه می‌شود بدون اینکه خطر از دادن سپرده یا سرقت در راه بانک او را تهدید کند.

## ساختار

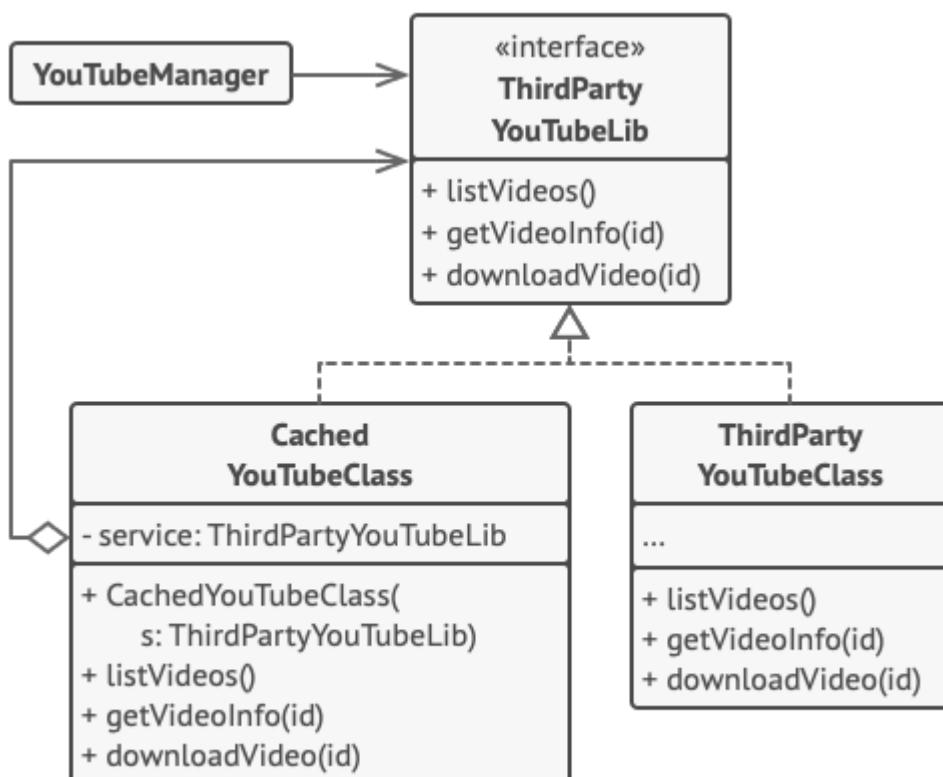


- واسطه‌های سرویس در این واسط تعريف می‌شوند.
- در این کلاس بعضی از business logic کاربردی تعريف می‌شود.

- Proxy: در این کلاس reference هایی به اشیاء Service نگهداری می‌شود. بعد از اینکه فرایندهای مورد نیاز(مانند مقداردهی های اولیه، لگ انداختن، کنترل دسترسی‌ها، و ...) را پردازش کرد، آن‌ها را به شیء سرویس پاس می‌دهد. به طور کلی، Proxy ها چرخه‌ی اشیاء Service ای که در آن‌ها وجود دارد را مدیریت می‌کنند.
- Client: کلاینت باید هم با proxy و هم با service با استفاده از یک واسطه صحبت کند. با این کار می‌توانید از proxy مورد نظر در هر جایی که از service استفاده می‌شود استفاده کنید.

## مثال

در این مثال خواهیم دید که چگونه الگوی طراحی Proxy به ما کمک می‌کند تا یک سری مقداردهی‌های اولیه و cache کردن‌ها را هنگام استفاده از یک کتابخانه‌ی 3rd-party یوتیوب، انجام دهیم.



فرض کنید یک library دارید که یک کلاس برای دانلود ویدئو فراهم می‌کند. اما این library به نحوی ناکارآمد است. اگر client چندین بار درخواست دانلود یک ویدئو را بدهد، همیشه آن را دوباره دانلود می‌کند، به جای اینکه اولین فایل دانلود شده را ذخیره و مجدد استفاده کند.

با استفاده از الگوی proxy، شما یک proxy برای این کلاس دانلود ایجاد می‌کنید. این proxy همان واسطه (متدها و ویژگی‌ها) کلاس دانلود اصلی را دارد و آن را پیاده‌سازی می‌کند. وظیفه‌ی پراکسی این است که قبل از فراخوانی متدهای دانلود اصلی، فرایندهایی را انجام دهد و سپس کار را به متدهای اصلی واگذار کند.

همزمان اطلاعات دریافتی را نگه می‌دارد و هر زمان که برنامه چندین بار درخواست تکراری برای دانلود همان ویدئو را دهد، به جای دانلود دوباره نتیجه ذخیره شده را برمی‌گرداند.

```
// The interface of a remote service.
interface ThirdPartyYouTubeLib is
    method listVideos()
    method getVideoInfo(id)
    method downloadVideo(id)

// The concrete implementation of a service connector. Methods
// of this class can request information from YouTube. The speed
// of the request depends on a user's internet connection as
// well as YouTube's. The application will slow down if a lot of
// requests are fired at the same time, even if they all request
// the same information.
class ThirdPartyYouTubeClass implements ThirdPartyYouTubeLib is
    method listVideos() is
        // Send an API request to YouTube.

    method getVideoInfo(id) is
        // Get metadata about some video.

    method downloadVideo(id) is
        // Download a video file from YouTube.

// To save some bandwidth, we can cache request results and keep
// them for some time. But it may be impossible to put such code
// directly into the service class. For example, it could have
// been provided as part of a third party library and/or defined
// as `final`. That's why we put the caching code into a new
// proxy class which implements the same interface as the
// service class. It delegates to the service object only when
// the real requests have to be sent.
class CachedYouTubeClass implements ThirdPartyYouTubeLib is
    private field service: ThirdPartyYouTubeLib
    private field listCache, videoCache
    field needReset

    constructor CachedYouTubeClass(service: ThirdPartyYouTubeLib)
is
    this.service = service

    method listVideos() is
        if (listCache == null || needReset)
            listCache = service.listVideos()
        return listCache
```

```

method getVideoInfo(id) is
    if (videoCache == null || needReset)
        videoCache = service.getVideoInfo(id)
    return videoCache

method downloadVideo(id) is
    if (!downloadExists(id) || needReset)
        service.downloadVideo(id)

// The GUI class, which used to work directly with a service
// object, stays unchanged as long as it works with the service
// object through an interface. We can safely pass a proxy
// object instead of a real service object since they both
// implement the same interface.
class YouTubeManager is
    protected field service: ThirdPartyYouTubeLib

constructor YouTubeManager(service: ThirdPartyYouTubeLib) is
    this.service = service

method renderVideoPage(id) is
    info = service.getVideoInfo(id)
    // Render the video page.

method renderListPanel() is
    list = service.listVideos()
    // Render the list of video thumbnails.

method reactOnUserInput() is
    renderVideoPage()
    renderListPanel()

// The application can configure proxies on the fly.
class Application is
    method init() is
        aYouTubeService = new ThirdPartyYouTubeClass()
        aYouTubeProxy = new CachedYouTubeClass(aYouTubeService)
        manager = new YouTubeManager(aYouTubeProxy)
        manager.reactOnUserInput()

```

## چه زمانی باید از این الگو استفاده کنیم؟

- مقدار دهی به صورت **lazy**. فرض کنید سرویس بزرگی دارید که همیشه در حال اجرا است و مقدار زیادی از منابع سیستم شما را مصرف می‌کند اما شما فقط گاهی اوقات به آن نیاز دارید نه همیشه!

بنابراین به جای اینکه هر زمان برنامه اجرا شد یک شی بسازید، هر زمان که به آن نیاز پیدا شد این کار را انجام دهید. به عبارت دیگر، با استفاده از الگوی proxy، شما یک proxy مجازی ایجاد می‌کنید که همان ویژگی‌ها و رفتار شیء اصلی را دارد. این proxy مجازی وظیفه نگهداری شیء اصلی را دارد و هنگامی که واقعاً نیاز به استفاده از شیء دارید، شیء اصلی را ایجاد و به آن اجازه اجرا می‌دهد. این کار باعث می‌شود تا منابع سیستمی برنامه صرفاً در زمان نیاز، مصرف شوند و بهینه‌سازی شود.

- برای مدیریت دسترسی(protection proxy): زمانی که می‌خواهید فقط یک سری کلاینت‌های خاص از اشیاء سرویستان استفاده کنند از این الگو استفاده کنید، به طور مثال زمانی که اشیاء شما بخش‌های حیاتی یک سیستم عامل و کلاینت‌های شما انواع مختلفی از برنامه‌ها هستند. با استفاده از این الگوی طراحی فقط زمانی که اطلاعات کلاینت مورد نظر مناسب بود می‌تواند به اشیاء سرویس دسترسی داشته باشد.

- از این الگو برای اتصال ریموت به یک سرویس استفاده کنید. (**Remote proxy**)  
در این مورد proxy درخواست‌های کلاینت را بر روی شبکه می‌فرستد و تمامی جزئیات شبکه را هندل می‌کند.

- از این الگو برای ذخیره‌سازی تاریخچه‌ی سرویس‌ها یا همان درخواست‌های لاغ استفاده کنید.  
از این می‌تواند جزئیات هر درخواستی را قبل از اینکه به سرور برسد لاغ بزند.

- از این الگو برای کش کردن استفاده کنید. می‌توانید برای کش کردن در خواست‌های کلاینت و مدیریت چرخه‌ی کش‌ها از این الگو استفاده کنید.  
از این الگو برای زمان‌هایی که درخواست‌های یکسان و زیادی دارید می‌توانید استفاده کنید.

- از این الگو می‌توانید برای مدیریت منابع سنگین استفاده کنید. **Proxy** می‌تواند با بررسی کلاینت‌ها در صورت فعل نبودن آنها، منابع را آزاد کند.  
proxy می‌تواند کلاینت‌هایی که اشیاء سرویس با آن ها کار می‌کرده‌اند را پیگیری کند و وضعیت فعل بودن یا نبودن آن ها را بررسی کند. اگر کلاینت‌ها غیرفعال باشند و لیست کلاینت‌ها خالی شود proxy می‌تواند منابع را آزاد کند.

## معایب و مزایا

- ❖ بدون اینکه اطلاعاتی از کلاینت‌ها داشته باشید می‌توانید سرویس‌های خود را مدیریت کنید.
- ❖ می‌توانید اشیاء service خود را در زمان‌هایی که کلاینت‌ها با آنها کار ندارند مدیریت کنید.
- ❖ proxy در زمانی که اشیاء service قادر به فعالیت نباشند و یا در دسترس نباشند هم کار می‌کند.
- ❖ با استفاده از این الگوی طراحی، اصل SOLID از اصول Open/Closed نیز رعایت می‌شود. چرا که می‌توانید هر زمان یک proxy جدید اضافه کنید بدون اینکه service و یا client را تغییر دهید.
- ممکن است کمی باعث پیچیدگی کد شود چرا که باید کلاس‌های زیادی ایجاد کنید.
- دریافت پاسخ از سرور ممکن است با تاخیر مواجه شود.

## Behavioral

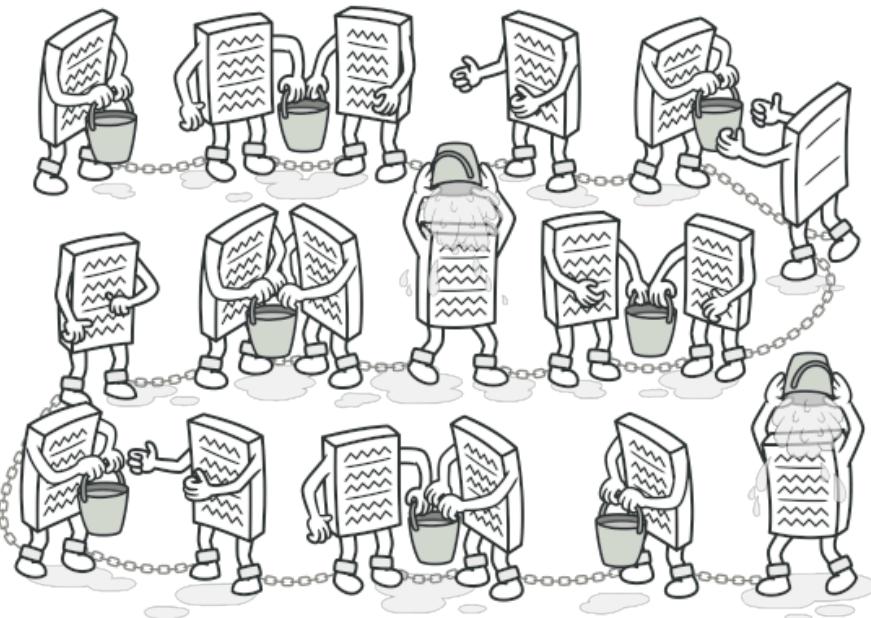
این الگوها بر رفتار و تعامل میان اشیاء در برنامه تمرکز دارند. آنها به حل مسائل مربوط به توزیع و اجرای وظایف و عملکردهای مختلف در سیستم کمک می‌کنند.

انواع الگوهای طراحی behavioral عبارتند از :

- Chain Of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template method
- Visitor

## Chain Of Responsibility

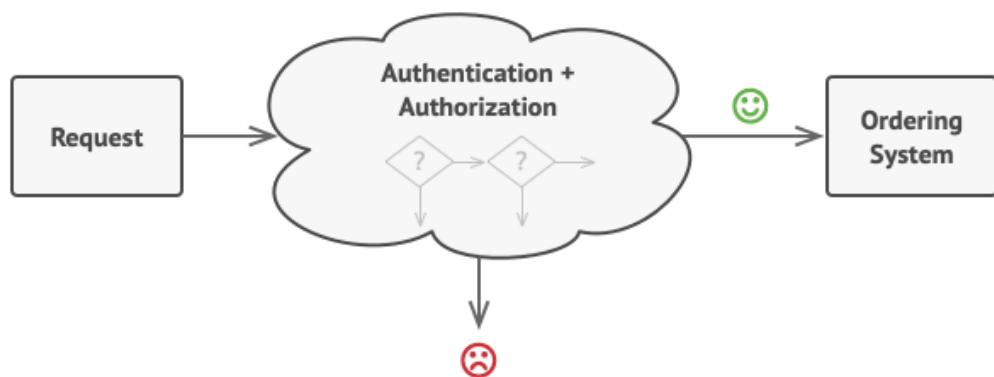
این الگوی طراحی به ما این امکان را می‌دهد تا هندل کردن request‌ها را به زنجیره‌ای از handler‌ها بسپاریم. بعد از دریافت هر درخواست، هر هندلر تصمیم می‌گیرد که خودش کارهای مربوط به این درخواست را انجام دهد یا اینکه آن را به یک هندلر دیگر واگذار کند.



### طرح مسئله

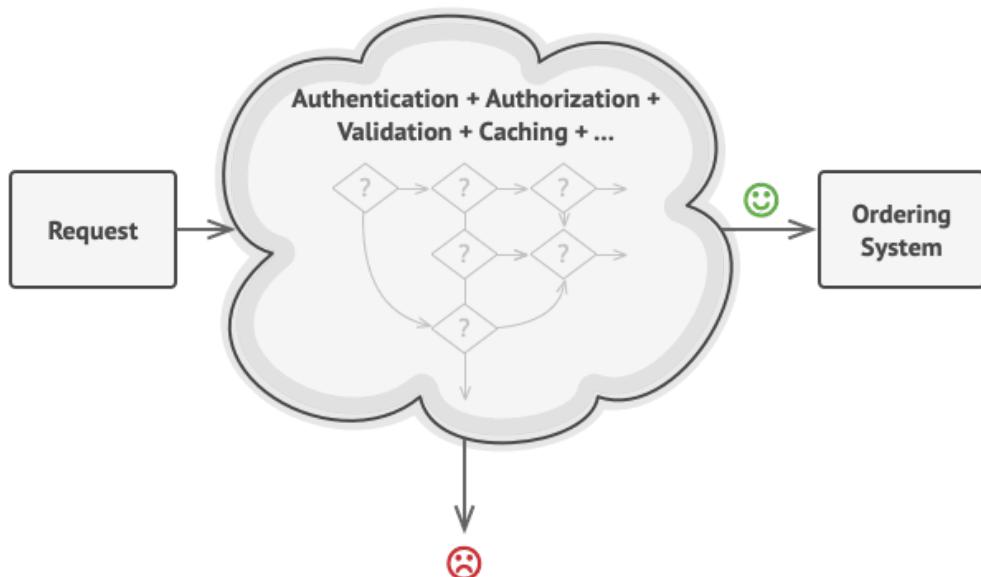
تصور کنید که شما در حال کار بر روی یک سیستم سفارش آنلاین هستید. می‌خواهید دسترسی به سیستم محدود شود، به طوری‌که تنها کاربران احراز هویت شده بتوانند سفارش ایجاد کنند. همچنین کاربرانی که دسترسی مدیریتی دارند، باید به تمام سفارشات دسترسی کامل داشته باشند.

بعد از کمی برنامه‌ریزی، متوجه می‌شوید که یک سری از بررسی‌ها باید به ترتیب انجام شوند. برنامه می‌تواند هر زمان که یک درخواست حاوی اطلاعات احراز هویت کاربر را دریافت می‌کند، تلاش کند که کاربر را در سیستم احراز هویت کند. در نتیجه، اگر این اطلاعات احراز هویت صحیح نباشند و احراز هویت ناموفق باشد، هیچ دلیلی برای ادامه بررسی‌های دیگر وجود ندارد.



بعد از گذشت چندین ماه، چندین سیستم برای بررسی این حالات ایجاد کرده‌اید. این سناریوها را در نظر بگیرید:

- یکی از همکاران شما پیشنهاد داد که ارسال داده‌های خام به سیستم سفارش می‌تواند خطرناک باشد. بنابراین شما یک مرحله‌ی اعتبارسنجی اضافی برای پاکسازی داده‌های یک درخواست اضافه می‌کنید.
- بعد از مدتی یک دیگر از همکارانتان به شما می‌گوید که سیستم شما در برابر حملات پیدا کردن رمزهای عبور به شدت آسیب پذیر است. فوراً یک سری بررسی‌های دیگر اضافه می‌کنید تا جلوی IP هایی که تلاش‌های ناموفق برای ورود به سامانه در چند مرحله‌ی تکراری داشتند را بگیرید.
- همکار دیگر شما این پیشنهاد را می‌دهد که می‌توانید برای افزایش سرعت سیستم از کش کردن دیتا برای درخواست‌های تکراری و یکسان استفاده کنید. پس یک مرحله برای بررسی دیتاها اضافه می‌کنید تا در صورتی که درخواست غیر تکراری‌ای دیدید آن را به سمت سرور ارسال کنید.



با هر حالتی که برای چک کردن یک سری از حالات متفاوت اضافه می‌کنید، کدهایتان کثیفتر و کثیفتر می‌شوند. این را هم باید در نظر بگیرید که تغییر یکی از این چک کردن‌ها ممکن است بر روی بررسی‌های دیگر تاثیر بگذارد. بدتر از همه، زمانی است که سعی می‌کنید برای حفاظت از بخش‌های دیگر سیستم از این بررسی‌ها استفاده کنید و باید برخی از قسکت‌های کدتان را کپی کنید. چرا که ممکن است آن بخش‌ها به برخی از این چک‌ها نیاز داشته باشند، نه به همه‌ی آنها. سیستمی که نوشته‌ید را به سختی می‌توان درک کرد و از آن نگهداری کرد. زمانی با کدتان کلنجر می‌روید و زمانی باید تصمیم بگیرید که آن را بازنگری کنید و به دنبال راه حلی برای بهبود این وضعیت باشید.

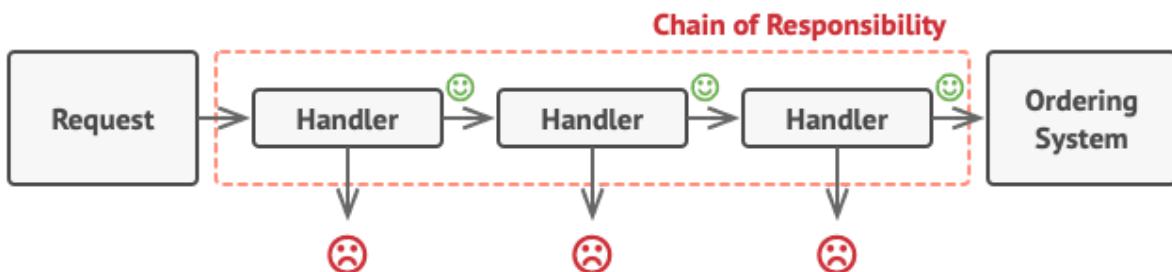
## راه حل

مانند بسیاری از الگوهای طراحی behavioral، الگوی Chain of Responsibility بر اساس تبدیل رفتارهای خاص به اشیاء مستقل، به handler تکیه دارد. در مثالی که زدیم هر کدام از چکها باید به کلاس‌هایی با یک متدهای مستقل تبدیل شوند و بررسی موارد درخواستی به عهده‌ی آن متدهای باشد و درخواست‌ها به همراه داده‌هایشان به عنوان آرگومان ورودی به این متدها داده شوند.

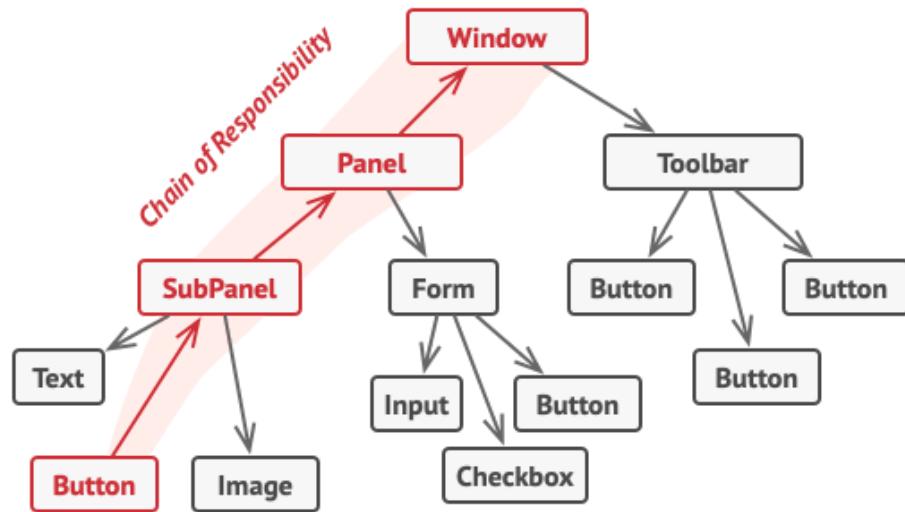
این الگو پیشنهاد می‌کند که handler‌ها مثل یک زنجیر به هم متصل باشند. هر کدام از این handler‌ها یک رفرنس از یک handler بعدی را در خود دارند تا بتوانند زنجیره‌های بعدی را پیدا کنند. در خصوص پردازش یک درخواست، handler‌ها درخواست را در کل زنجیره پاس می‌دهند. یک درخواست از بین تمامی این زنجیره‌ها می‌گذرد تا تمامی handler‌ها شанс پردازش آن را داشته باشند.

بهترین بخش کار این است که هر کدام از handler‌ها می‌توانند تصمیم بگیرند که یک درخواست را دیگر در زنجیره ارسال نکنند و هرجایی که مورد نیاز بود پردازش را قطع کنند.

در مثالی که از سیستم ثبت سفارش زدیم، handler تصمیم می‌گیرد که پردازش درخواست در زنجیره را ادامه دهد یا نه. در صورتی که دیتاهای یک درخواست صحیح باشند، تمام handler‌ها کار اصلی خود را انجام می‌دهند. این کار می‌تواند کش کردن اطلاعات و یا احراز هویت باشد.

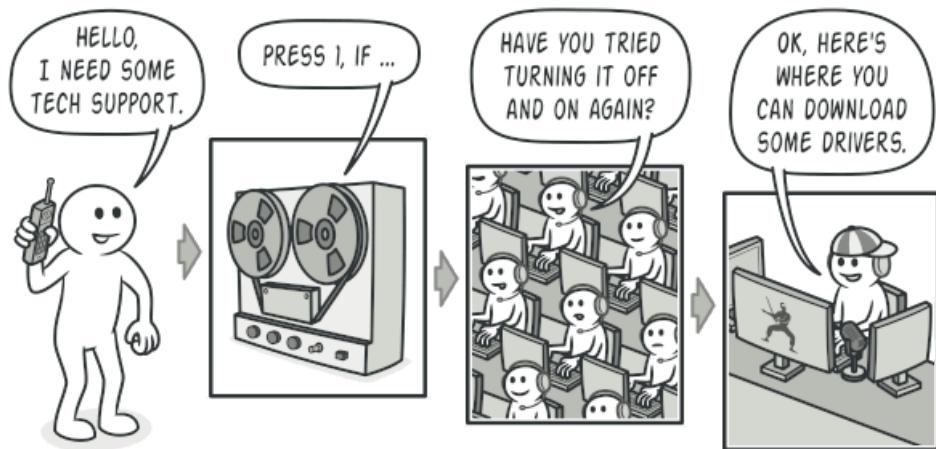


اما یک رویکرد دیگر هم وجود دارد و آن این است که یک درخواست یک پس از دریافت یک handler می‌کند که آیا می‌تواند آن را پردازش کند یا خیر. اگر بتواند درخواست را پردازش کند، دیگر درخواست را به زنجیره‌های بعدی ارسال نخواهد کرد. بنابراین یا فقط یک handler درخواست را پردازش می‌کند یا هیچکدام. این رویکرد بسیار در رابطه‌های کاربری برای تصمیم گیری درباره‌ی نوع برخورد با پشته‌ای از رخدادهای مربوط به هر ایمان متداول است. به طور مثال هر زمان که کاربر بر روی یک دکمه کلیک می‌کند، این رخداد بین تمام زنجیره‌های GUI که با عملیات دکمه آغاز می‌شوند منتشر می‌شود. پس از آن به سراغ پنل‌ها، فرم‌ها و .. می‌رود و در انتهای در پنجره‌ی اصلی اپلیکیشن به کاربر نمایش داده می‌شود. در ابتدا این رخداد توسط handler‌ای که توانایی پردازش آن را دارد پردازش می‌شود. این مثال بسیار قابل توجه است چرا که می‌توان چرخه‌ی این زنجیره‌ها را به صورت درخت نشان داد.



این موضوع بسیار حیاتی است که تمامی این *handler* ها باید یک واسط مشترک را پیاده‌سازی کنند. تمامی *handler* ها باید متدهای *execute* تحت عنوان *execute* را در خود داشته باشند. با این کار می‌توانید این زنجیره‌ها را در زمان اجرا با استفاده از انواع *handler* ها با یکدیگر ترکیب کنیم بدون اینکه نگران وابسته شدن کدهایمان به کلاس‌های *handler* شویم.

### مقایسه با دنیای واقعی



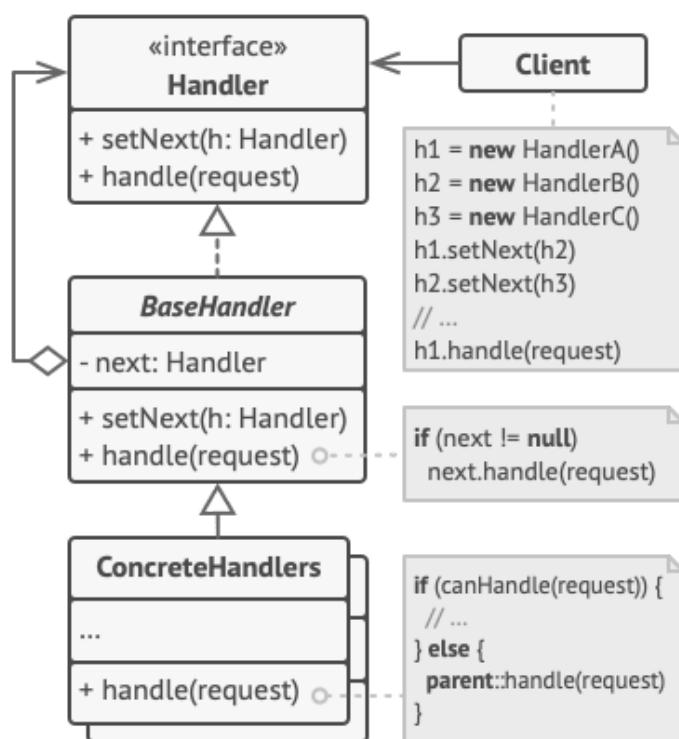
فرض کنید که یک قطعه‌ی سخت‌افزاری برای کامپیوتر خود خریده‌اید. شما یک geek هستید و بر روی سیستم‌تان چند سیستم عامل دارید. تصمیم می‌گیرید که همه‌ی این سیستم‌عامل‌ها را اجرا کنید تا ببینید سخت‌افزار‌تان آنها را پشتیبانی می‌کند یا خیر. سیستم عامل ویندوز شما به صورت اتوماتیک سخت‌افزار را شناسایی می‌کند. اما لینوکسی که بر روی سیستم‌تان نصب دارید با مشکل مواجه می‌شود. بعد از کمی ناامیدی تصمیم می‌گیرید تا با شماره‌ی پشتیبانی که بر روی جعبه نوشته شده است تماس بگیرید.

در مرحله‌ی اول صدای اپراتور را می‌شنوید که به شما ۹ گزینه پیشنهاد می‌دهد تا با توجه به مشکلتان یکی از آنها را انتخاب کنید. اما هیچ کدام از آنها مشابه مشکل شما نیست. بنابراین اپراتور شما را به یک تماس صوتی با یکی از کارشناسان بخش متصل می‌کند. بعد از کمی گفتگو کارشناس این بخش هم نمی‌تواند مشکل شما را حل کند. مدام یک سری دستورالعمل را از روی کتاب راهنمای خواندن و علاقه‌های به شنیدن صحبت‌های شما ندارد! بعد از شنیدن جمله‌ی "آیا سیستم خود را روشن و خاموش کرده‌اید؟" به تعداد ۱۰ بار، از او می‌خواهید که لطفاً شما را به یک مهندس متخصص متصل کند.

سرانجام، اپراتور تماس شما را به یکی از مهندسان منتقل می‌کند، که احتمالاً ساعت‌ها در آرزوی یک چت زنده انسانی در اتاق سرور تنها‌یش در زیرزمین تاریک یک ساختمان اداری نشسته بود.

مهندس به شما می‌گوید که درایورهای مناسب سخت افزار جدید خود را از کجا دانلود کنید و چگونه آنها را روی لینوکس نصب کنید. بالاخره راه حل را پیدا کرده‌اید و تماس را با شادی به پایان می‌رسانید.

## ساختار



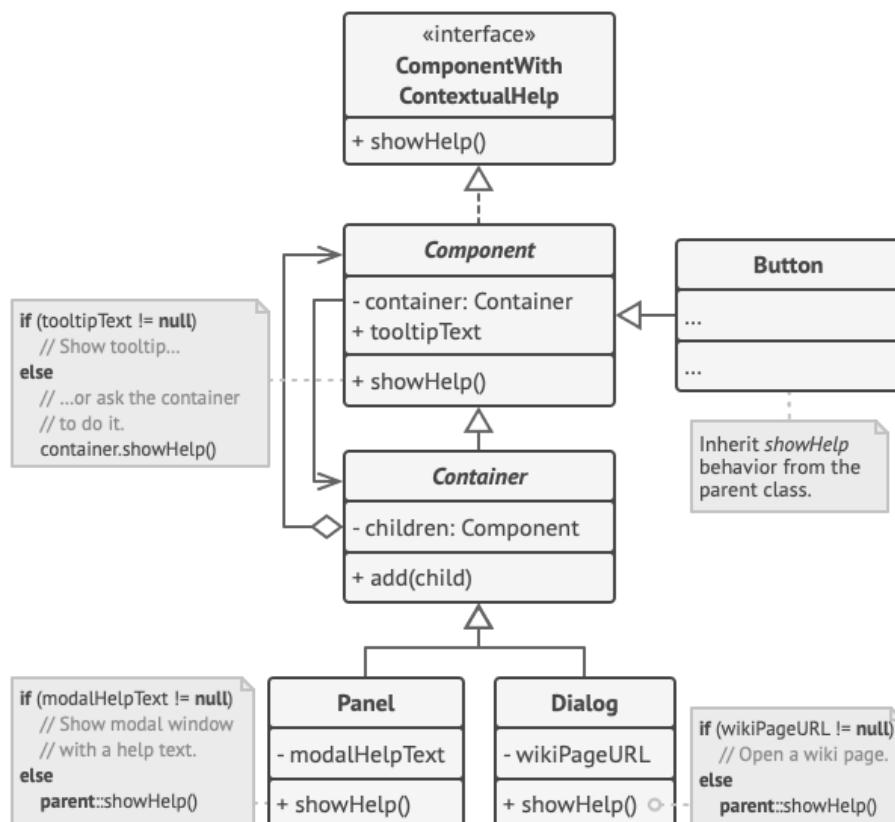
- **Handler**: یک واسط را تعریف می‌کند تا بقیه‌ی **Handler**‌های زیرشاخه نیز آن را پیاده‌سازی کنند. معمولاً دارای یک متده است که پاسخگوی نیاز یک `request` است اما گاهی اوقات دارای متدهای بیشتری برای تنظیم کردن **handler**‌های بعدی نیز هست.
- **Base Handler**: این کلاس اختیاری است و می‌توان مواردی که بین **Handler**‌ها مشترک است را در آن قرار داد. معمولاً این کلاس یک فیلد در خودش نگه می‌دارد که به بعدی **Handler** بعدی reference دارد. می‌تواند یک زنجیره را با پاس دادن **Handler**‌ها به سازنده‌ی این کلاس یا استفاده از **Client** ایجاد کند.

setter های این کلاس بسازد. این کلاس همچنین باید رفتارهای پایه‌ای Handler را نیز پیاده‌سازی کند، یعنی می‌تواند اجرای یک درخواست را بعد از بررسی وجود handler های بعدی به آنها واگذار کند.

- Concrete Handlers** • پردازش اصلی درخواست‌ها در این Handler های زیرشاخه اتفاق می‌افتد. بعد از دریافت هر درخواست Handler باید تصمیم بگیرد که خودش آن را پردازش کند و یا اینکه آن را به Handler دیگر پاس بدهد. Handler ها معمولاً مستقل و غیر قابل تغییر هستند و تمام داده‌های مورد نیاز را فقط یک بار از طریق سازنده دریافت می‌کنند.
- Client** • بسته به منطق برنامه client ممکن است زنجیره‌ها را فقط یک بار و یا به حالت پویا بسازد. توجه کنید که یک درخواست می‌تواند به هر handler ای در یک زنجیره ارسال شود و لازم نیست که حتماً به اولین handler فرستاده شود.

## مثال

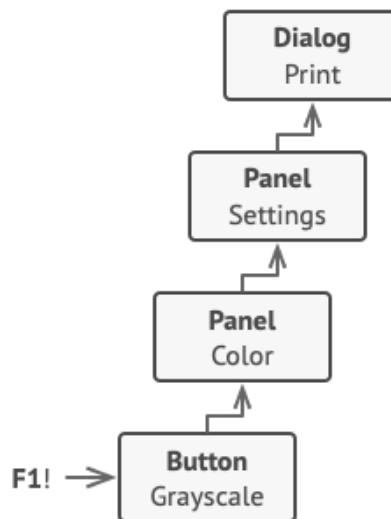
در این مثال، الگوی Chain of Responsibility مسئول نمایش اطلاعات راهنمایی متنی برای عناصر فعال رابط کاربری گرافیکی (GUI) است.



معمولاً Object هایی که در رابطهای کاربری برنامه‌ها می‌بینیم ساختاری درختی دارند. به طور مثال کلاس **Dialog** که صفحه‌ی اصلی برنامه را نشان می‌دهد ریشه‌ی درخت است. **Dialog** شامل یک سری **Panel** است

که هر کدام از این Panel‌ها خودشان می‌توانند شامل یک سری Panel‌ها دیگر و یا ایمان‌های سطح پایین تر مانند دکمه‌ها و TextField‌ها باشند.

هر کامپوننت می‌تواند توضیح مختصری از ساختار متنی را به همراه راهنمایی‌های مختلفی ارائه کند. اما یک سری از کامپوننت‌ها فقط متن‌های خاصی را نمایش می‌دهند و یک کار مختص به خود را دارند مانند دکمه‌ی "باز کردن در یک مرورگر جدید".



زمانی که کاربر ماوس خودش را بر روی ایمانی می‌برد و بر روی دکمه‌ی F1 کلیک می‌کند، برنامه کامپوننت مرتبط با آن دکمه را شناسایی می‌کند و درخواست help را ارسال می‌کند. این درخواست بین تمامی عناصر می‌چرخد تا اطلاعات مربوط به help نمایش داد شود.

```

// The handler interface declares a method for executing a
// request.
interface ComponentWithContextualHelp is
    method showHelp()

// The base class for simple components.
abstract class Component implements ComponentWithContextualHelp is
    field tooltipText: string

        // The component's container acts as the next link in the
        // chain of handlers.
        protected field container: Container

        // The component shows a tooltip if there's help text
        // assigned to it. Otherwise it forwards the call to the
        // container, if it exists.
        method showHelp() is
  
```

```

        if (tooltipText != null)
            // Show tooltip.
        else
            container.showHelp()

// Containers can contain both simple components and other
// containers as children. The chain relationships are
// established here. The class inherits showHelp behavior from
// its parent.
abstract class Container extends Component is
    protected field children: array of Component

    method add(child) is
        children.add(child)
        child.container = this

// Primitive components may be fine with default help
// implementation...
class Button extends Component is
    // ...

// But complex components may override the default
// implementation. If the help text can't be provided in a new
// way, the component can always call the base implementation
// (see Component class).
class Panel extends Container is
    field modalHelpText: string

    method showHelp() is
        if (modalHelpText != null)
            // Show a modal window with the help text.
        else
            super.showHelp()

// ...same as above...
class Dialog extends Container is
    field wikiPageURL: string

    method showHelp() is
        if (wikiPageURL != null)
            // Open the wiki help page.
        else
            super.showHelp()

// Client code.

```

```

class Application is
    // Every application configures the chain differently.
    method createUI() is
        dialog = new Dialog("Budget Reports")
        dialog.wikiPageURL = "http://..."
        panel = new Panel(0, 0, 400, 800)
        panel.modalHelpText = "This panel does..."
        ok = new Button(250, 760, 50, 20, "OK")
        ok.tooltipText = "This is an OK button that..."
        cancel = new Button(320, 760, 50, 20, "Cancel")
        // ...
        panel.add(ok)
        panel.add(cancel)
        dialog.add(panel)

    // Imagine what happens here.
    method onF1KeyPress() is
        component = this.getComponentAtMouseCoords()
        component.showHelp()

```

## چه زمانی باید از این الگو استفاده کنیم؟

- از الگوی **Chain of responsibility** زمانی استفاده کنید که انتظار می‌رود برنامه شما انواع مختلفی از درخواست‌ها را به روش‌های مختلف پردازش کند، اما نوع دقیق درخواست‌ها و تعداد آنها از قبل مشخص نیست.

این الگو این امکان را می‌دهد تا چندین Handler را در قالب یک زنجیره به یکدیگر وصل کنید و زمانی که با یک درخواست روبرو شدید، می‌توانید با استفاده از هر Handler ای که توانایی پردازش آن درخواست را داشته باشد آن را پردازش کنید. با این کار تمامی Handler‌ها شанс پردازش درخواست را دارند.

- از این الگو زمانی استفاده کنید که نیاز به اجرای چند Handler با یک ترتیب خاصی دارد.

زمانی که Handler‌ها در یک زنجیره قرار می‌دهید، درخواست‌ها به همان ترتیبی که در برنامه‌تان تنظیم کرده اید توسط Handler‌ها بررسی می‌شوند.

- از این الگو زمانی استفاده کنید که مجموعه‌ی Handler‌ها و ترتیب آنها در زمان اجرا تغییر می‌کند.

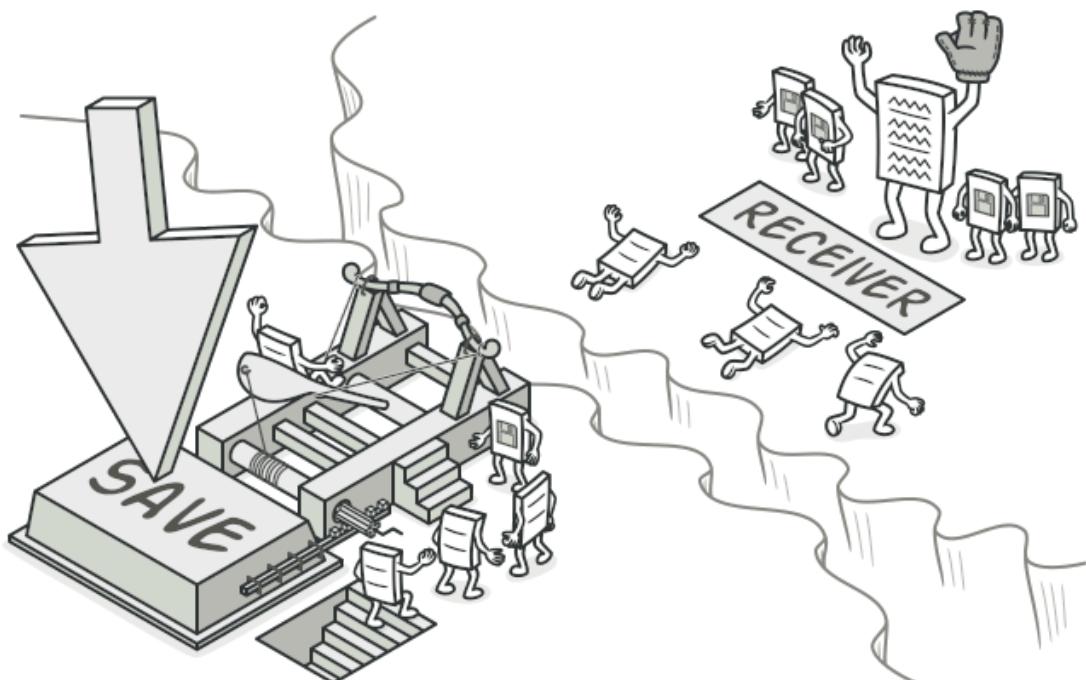
اگر از setter برای رفرنس دادن به بقیه‌ی Handler‌ها استفاده شود می‌توان آنها را حذف، اضافه و یا ترتیب آنها را تغییر داد.

## معایب و مزایا

- ❖ می‌توان ترتیب پردازش درخواست‌ها را کنترل کرد.
- ❖ با استفاده از این الگو اصل Single responsibility SOLID از اصول رعایت می‌شود چرا که می‌توانید کلاس‌هایی که وظیفه‌ی فرآخوانی عملیات را دارند از کلاس‌هایی که وظیفه‌ی انجام عملیات را دارند جدا کنید.
- ❖ با استفاده از این الگو اصل Open/Closed SOLID از اصول رعایت می‌شود چرا که در هر لحظه می‌توانید یک Handler جدید به برنامه اضافه کنید بدون اینکه کدهای Client را تغییر دهید.
- برخی از درخواست‌ها ممکن است در نهایت رسیدگی نشوند.

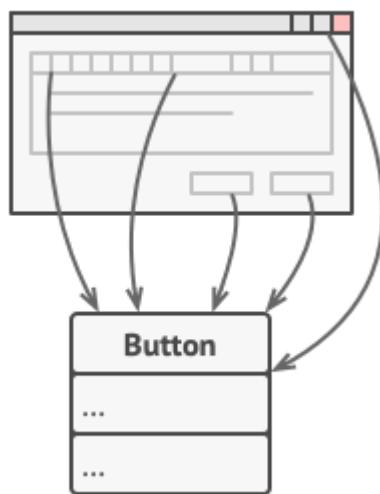
## Command

با استفاده از الگوی طراحی command می‌توانیم درخواست‌هایمان را به Object ای واگذار کنیم که شامل تمامی اطلاعات آن درخواست باشد. با این کار می‌توانیم درخواست‌های خودمان را به عنوان آرگومان ورودی متدها پاس دهیم، زمان پردازش درخواست‌ها را customize کنیم و از یک سری کارهایی که در آن لحظه قابل انجام نیستند پشتیبانی کنیم.

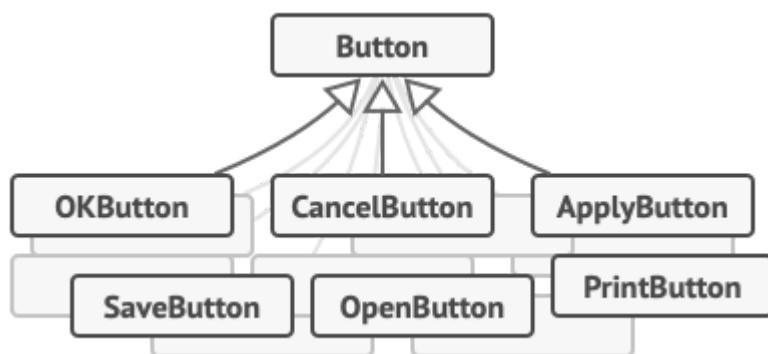


## طرح مسئله

فرض کنید بر روی یک برنامه‌ی text-editor کار می‌کنید. تسک شما این است که یک toolbar برای انواع عملیات‌ها در این editor ایجاد کنید. یک کلاس بسیار منظم و دقیق برای تمامی دکمه‌های toolbar و دکمه‌های عمومی در سطح برنامه ایجاد می‌کنید.



با اینکه این دکمه‌ها کاملاً شبیه به هم به نظر می‌رسند، اما کارهای مختلفی انجام می‌دهند. سوال من این است که کد مربوط به هندل کردن این دکمه‌ها را در کجا قرار می‌دهیم؟ ساده‌ترین راهی که وجود دارد این است که به ازای هر عملیات یک زیر-کلاس button ایجاد کنیم و آن کلاس شامل یک سری فرایندها برای کلیک روی آن دکمه باشد.



بعد از کمی فکر کردن متوجه می‌شویم که این کار کمی نقص دارد. در نگاه اول شما تعداد زیادی زیر-کلاس دارید که خطرهای زیادی ممکن است گریبانگیر شما شود. به طور مثال ممکن است در هر باری که کلاس

Button را تغییر می‌دهید مجبور باشید تمامی زیر-کلاس‌ها را نیز تغییر دهید. اگر بخواهم به طور ساده بگویم کد شما به شدت به منطق بیزینس شما وابسته شده است.

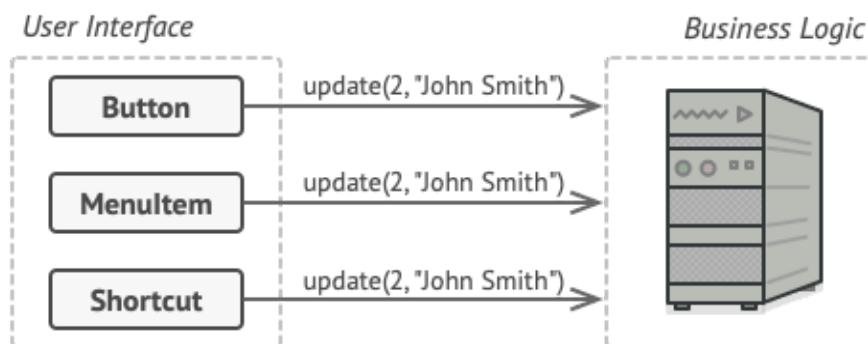


و این نقطه رشت ترین جای این کار است. برخی از عملیات‌ها مانند copy یا paste کردن ممکن است از چندین جا فراخوانی شوند. به طور مثال کاربر ممکن است بر روی دکمه‌ی copy داخل toolbar کلیک کند یا از طریق گزینه‌ی منو این کار را انجام دهد و یا اصلاً از کلیدهای `ctrl + c` استفاده کند. اگر برنامه‌ی شما فقط toolbar داشت، منطقی بود که تمامی عملیات‌ها را در درون زیر-کلاس‌ها پیاده‌سازی کنید و همچنین نگهداری کدی برای کپی کردن متن در درون زیر-کلاس CopyButton هم منطقی به نظر می‌آمد. اما زمانی که toolbar، منو و کلیدهای میانبر دارید باید یک قطعه کد را در همه‌ی این‌ها تکرار کنید یا اینکه تمامی این منوها را به دکمه‌ها وابسته کنید که این گزینه‌ی بدتری است! اما راه حل چیست؟

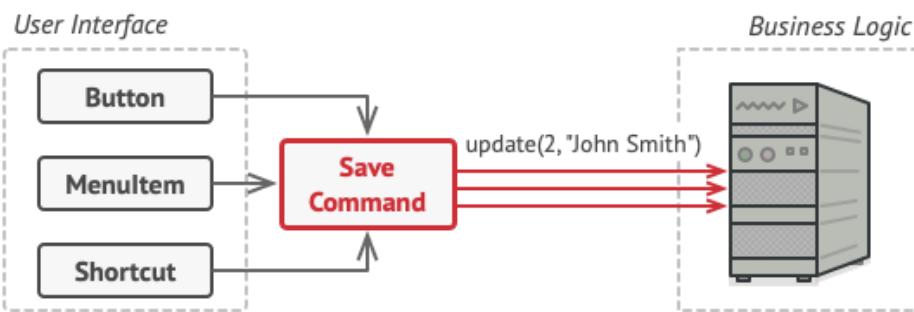
## راه حل

یک طراحی نرم‌افزار ایده‌آل، جداسازی انواع نگرانی‌های مسائل موجود در برنامه را اساس کار خود قرار می‌دهد و معمولاً برنامه را به چندین لایه‌ی کوچکتر می‌شکند. به طور مثال یک لایه برای رابط کاربری و یک لایه‌ی جدای دیگر برای بیزینس.

لایه‌ی رابط کاربری وظیفه‌ی خروجی گرفتن از عکس‌های زیبا در صفحات برنامه، دریافت ورودی‌های متفاوت و نشان دادن خروجی مناسب را بر عهده دارد و هر زمان که نوبت به کارهایی همچون محاسبه‌ی مسیر ماه(!) یا تهیه‌ی گزارش‌های سالانه و ... می‌رسد لایه‌ی گرافیکی آنها را به لایه‌ی بیزینس واگذار می‌کند. اگر بخواهیم از سمت کد به این موضوع نگاه کنیم به این شکل می‌شود که Object‌های رابط کاربری متدهای لایه‌ی بیزینس را فراخوانی می‌کنند و تعدادی آرگومان برای آنها می‌فرستند. این فرایند اینگونه توصیف می‌شود که یک شی به شی دیگر درخواست ارسال می‌کند.

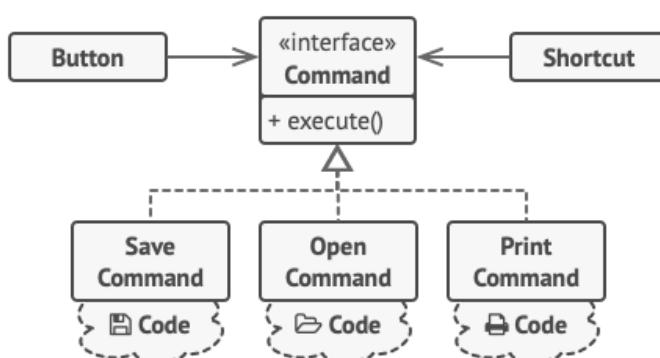


الگوی طراحی Command پیشنهاد می‌کند تا رابط کاربری به طور مستقیم این درخواست‌ها را ارسال نکند و به جای آن تمامی جزئیات درخواست مانند Object ای که قرار است فراخوانی شود، نام متده است، لیست آرگومان‌ها و ... را در کلاسی جداگانه تحت عنوان command که یک متده برای راه‌اندازی این درخواست دارد نگه دارد. در واقع کلاس‌های command رابط بین رابط کاربری و لایه‌ی بیزینس هستند. با این کار لایه‌ی GUI یا همان رابط کاربری دیگر نیازی به اینکه چه اتفاقی در لایه‌ی بیزینس می‌افتد ندارد. اینکه چگونه قرار است این درخواست دریافت و یا پردازش شود هیچ ارتباطی به لایه‌ی GUI ندارد. GUI فقط دستوری که باید یک سری موارد را کنترل کند فعال می‌کند.



گام بعدی این است که تمامی command ها یک interface را پیاده‌سازی کنند. این معمولاً دارای یک متده است که هیچ آرگومانی هم دریافت نمی‌کند. این کار به شما این اجازه را می‌دهد تا command های متفاوتی از یک ارسال کننده درخواست را بدون اینکه به یک command خاصی وابسته شوید هندل کنید. به عنوان یک امتیاز، الان شما می‌توانید به راحتی بین command هایی که از سمت sender می‌آید جابجا شوید و حتی به طور موثر رفتار sender یا همان ارسال کننده درخواست را تغییر دهید.

اکنون باید به این تیکه‌ی گم شده از پازل یعنی پارامترهای یک درخواست توجه کنیم. یک شیء GUI ممکن است یک Object را به لایه‌ی بیزینس پاس بدهد. اما جلوتر گفتم که متده interface که متن‌بندی آرگومانی را به عنوان ورودی دریافت نمی‌کند. پس چگونه باید این کار را انجام داد؟ چگونه جزئیات یک درخواست را به گیرنده ارسال کنیم؟ بنظر می‌رسد که command مورد نظر باید از قبل کانفیگ شده باشد و یا در درون خود این جزئیات را داشته باشد.



باید به مثال text-editor برگردیم. بعد از اینکه تصمیم گرفتیم از الگوی Command استفاده کنیم، دیگر نیازی به اینکه تمامی زیر-کلاس‌ها را برای انواع مختلفی از عملیات‌ها پیاده‌سازی کنیم نداریم. اینکه فقط یک فیلد در درون کلاس Button نگه داریم تا رفرانسی به یک command داشته باشد کافی است تا هر زمان که دکمه‌ی مورد نظر کلیک شد command متناظر با آن را اجرا کند.

در این سناریو شما تعداد کلاس command تعریف می‌کنید و هر کدام را براساس رفتاری که دارد به عملیات مربوط به خودش متصل می‌کنید.

تمامی موارد دیگر GUI مانند منوها، shortCut ها و ... نیز به همین صورت زمانی که کاربر با هر کدام از آنها ارتباطی داشته باشد به command مربوط به خودشان متصل می‌شوند. احتمالاً تا الان حدس زده‌اید که ایمان‌هایی که عملیات یکسانی را انجام می‌دهند به Command یکسانی هم متصل می‌شوند و اینگونه از تکرار شدن کد جلوگیری می‌شود.

در پایان command به عنوان یک واسطه بین GUI و لایه‌ی بیزینس عمل می‌کند و میزان وابستگی این دو لایه به همدیگر را کاهش می‌دهد. و این مقدار کوچکی از مزایایی است که این الگو در اختیارمان قرار می‌دهد.

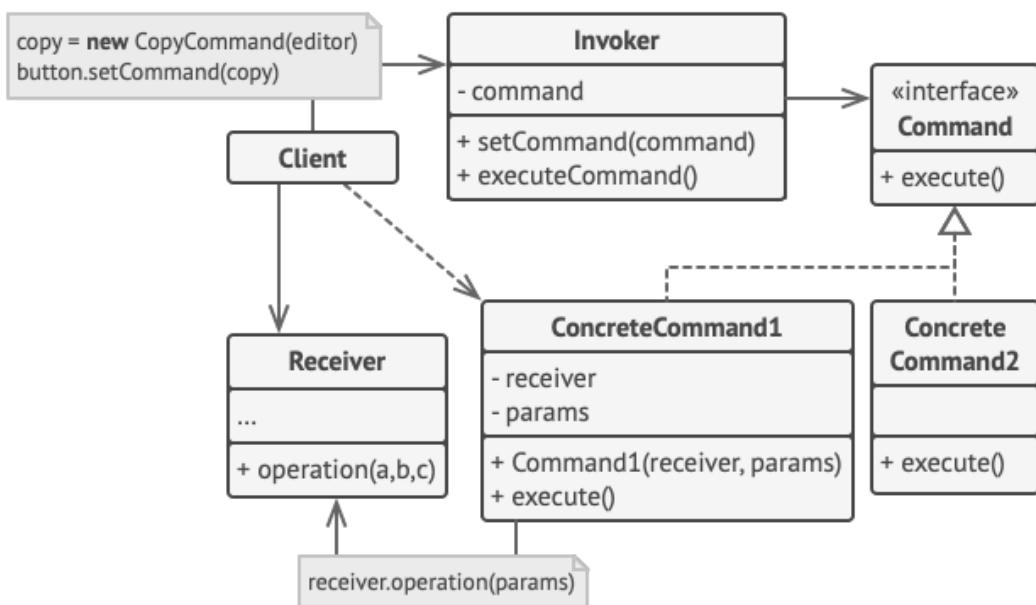
### مقابسه با دنیای واقعی



بعد از ساعتها قدم زدن در شهر، یک رستوران خوب پیدا می‌کنید و وارد آن می‌شوید و به روی میزی در کنار صندلی می‌نشینید. یک گارسون خوش بخورد به سرعت پیش شما می‌آید و سفارش شما را می‌گیرد و آن را بر روی یک کاغذ یادداشت می‌کند. گارسون به آشپزخانه می‌رود و یادداشت را به دیوار سفارشات می‌چسباند. بعد از مدتی سرآشپز سفارش را می‌بیند و شروع به آشپزی می‌کند. آشپز غذا را به همراه آن یادداشت در درون سینی می‌گذارد. گارسون سینی را می‌گیرد و یادداشت را چک می‌کند تا مطمئن شود همه چی درست است و سپس غذا را برایتان می‌آورد.

در اینجا آن تکه کاغذ نقش command را بازی می‌کند. این کاغذ تا زمانی که آشپز به سراغ آن باید در صفحه می‌ماند. در آن کاغذ تمامی موارد مورد نیاز برای اینکه کار طبخ این درخواست را شروع کند نوشته شده است. این فرایند به آشپز این اجازه را می‌دهد تا به جای اینکه به این ورود و آن ورود تا جزئیات سفارش را بگیرد، صرفاً به آشپزی مشغول باشد.

## ساختار

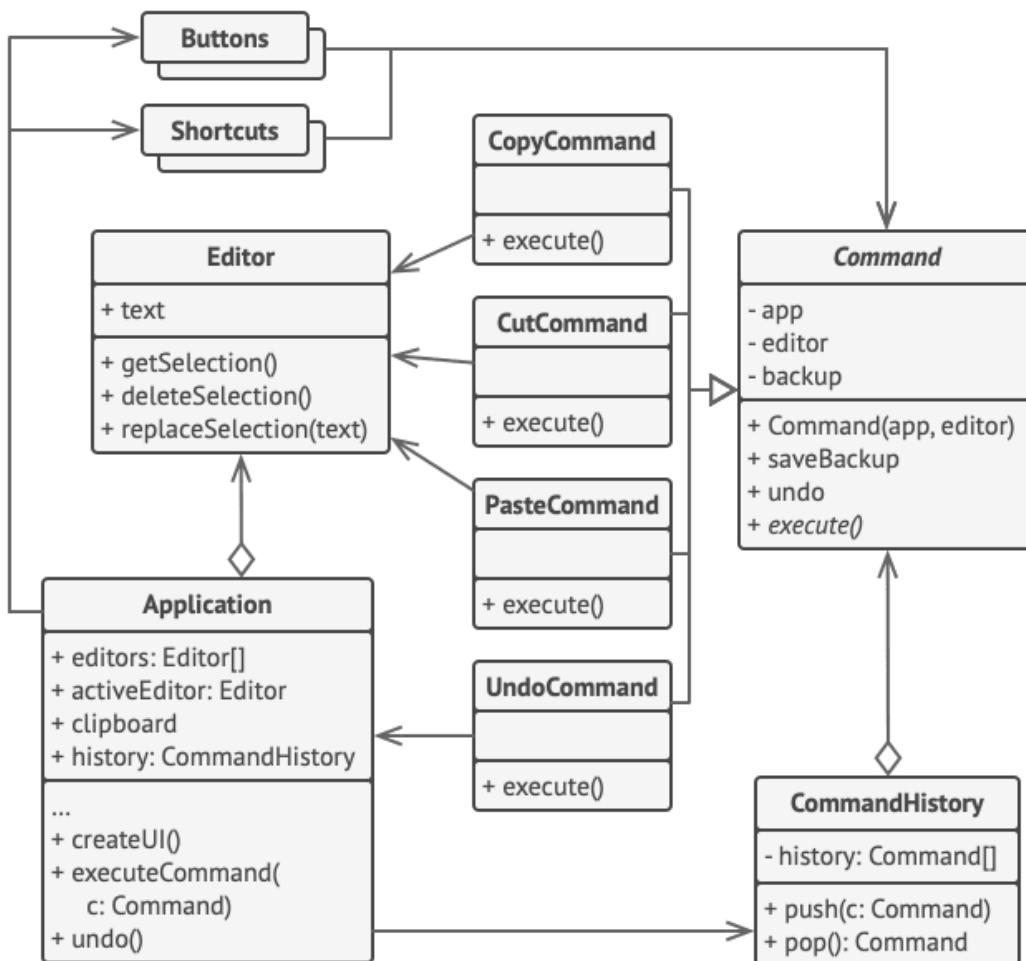


- **Sender:** این کلاس وظیفه‌ی تعریف یک درخواست را بر عهده دارد. این کلاس باید یک فیلد برای ذخیره‌ی رفرنس به یک command را در خود نگه داشته باشد. Sender به جای اینکه مستقیماً خودش درخواست را بفرستد از command برای این کار استفاده می‌کند تا درخواست به دست گیرنده برسد. به این نکته باید توجه کرد که sender وظیفه‌ی ایجاد command را بر عهده ندارد و به طور معمول یک command از پیش ساخته شده را دریافت می‌کند.
- **Command:** این واسطه معمولاً فقط یک متده دارد تا command مورد نظر را استخراج کند.
- **Concrete command:** این کلاس‌ها انواع مختلفی از دستورات را پیاده‌سازی می‌کنند. این را باید در نظر بگیریم که یک concrete command کارهای مربوط به یک درخواست را خودش انجام نمی‌دهد و آن درخواست را به لایه‌ی بیزینس منتقل می‌کند. البته که می‌توان برای ساده‌تر شدن کد این فرایند را با هم ادغام کرد. پارامترهای ورودی متدهایی که فرایندهای یک درخواست در آنها انجام می‌شوند می‌توانند به عنوان فیلد در concrete command ها تعریف شوند. می‌توانید این command ها را تغییر ناپذیر کنید به این صورت که تنها فقط با استفاده از سازنده بتوان آنها را ایجاد کرد.

- **Receiver**: این کلاس شامل منطق بیزینس برنامه‌مان می‌شود. تقریباً هر Object می‌تواند به عنوان یک receiver عمل کند. اکثر command‌ها جزئیات اینکه یک درخواست چگونه به یک receiver می‌رسد را هندل می‌کنند و از طرفی receiver خودش تمام کارها را انجام می‌دهد.
- **Client**: این کلاس Object‌های concrete command را ایجاد می‌کند. کلاینت باید تمام پارامترهای یک درخواست که شامل نمونه‌ای از receiver می‌باشد را به سازنده‌ی command ارسال کند. بعد از این کار یک command ممکن است به یک یا چند گیرنده متصل شود.

## مثال

در این مثال الگوی طراحی command به ما کمک می‌کند تا تاریخچه‌ی اجرای یک سری فرایند را پیگیری کنیم و در صورت نیاز هر کدام از این فرایندها را معکوس کنیم.



قبل از اجرای فرایندهایی مانند مانند برش، کپی و ... که باعث تغییر وضعیت editor مان می‌شوند، یک نسخه‌ی پشتیبان از حالت فعلی editor براساس command‌های موجود باید تهیه شود. بعد از اینکه یک command اجرا شد، command مورد نظر به همراه وضعیت editor و نسخه‌ی پشتیبان به تاریخچه command‌ها اضافه می‌شوند. در مراحل بعدی اگر کاربر نیاز به برگرداندن عملیاتی داشته باشد، برنامه

می‌تواند جدیدترین دستور را از تاریخچه دریافت کند، نسخه‌ی مربوط به وضعیت editor را بخواند و آن را بازیابی کند.

بدین صورت رابط کاربری‌مان دیگر به concrete command ها وابسته نیست چرا که فقط با یک interface از جنس command در ارتباط است. این کار این اجازه را می‌دهد تا هر زمان که نیاز بود بدون تغییر دادن کد هایمان هر command جدیدی که خواستیم به برنامه اضافه کنیم.

```
// The base command class defines the common interface for all
// concrete commands.

abstract class Command is
    protected field app: Application
    protected field editor: Editor
    protected field backup: text

    constructor Command(app: Application, editor: Editor) is
        this.app = app
        this.editor = editor

    // Make a backup of the editor's state.
    method saveBackup() is
        backup = editor.text

    // Restore the editor's state.
    method undo() is
        editor.text = backup

    // The execution method is declared abstract to force all
    // concrete commands to provide their own implementations.
    // The method must return true or false depending on whether
    // the command changes the editor's state.
    abstract method execute()

// The concrete commands go here.

class CopyCommand extends Command is
    // The copy command isn't saved to the history since it
    // doesn't change the editor's state.
    method execute() is
        app.clipboard = editor.getSelection()
        return false

class CutCommand extends Command is
    // The cut command does change the editor's state, therefore
    // it must be saved to the history. And it'll be saved as
    // long as the method returns true.
    method execute() is
        saveBackup()
```

```

        app.clipboard = editor.getSelection()
        editor.deleteSelection()
        return true

class PasteCommand extends Command is
    method execute() is
        saveBackup()
        editor.replaceSelection(app.clipboard)
        return true

// The undo operation is also a command.
class UndoCommand extends Command is
    method execute() is
        app.undo()
        return false

// The global command history is just a stack.
class CommandHistory is
    private field history: array of Command

    // Last in...
    method push(c: Command) is
        // Push the command to the end of the history array.

    // ...first out
    method pop():Command is
        // Get the most recent command from the history.

// The editor class has actual text editing operations. It plays
// the role of a receiver: all commands end up delegating
// execution to the editor's methods.
class Editor is
    field text: string

    method getSelection() is
        // Return selected text.

    method deleteSelection() is
        // Delete selected text.

    method replaceSelection(text) is
        // Insert the clipboard's contents at the current
        // position.

// The application class sets up object relations. It acts as a

```

```

// sender: when something needs to be done, it creates a command
// object and executes it.
class Application is
    field clipboard: string
    field editors: array of Editors
    field activeEditor: Editor
    field history: CommandHistory

    // The code which assigns commands to UI objects may look
    // like this.

    method createUI() is
        // ...
        copy = function() { executeCommand(
            new CopyCommand(this, activeEditor)) }
        copyButton.setCommand(copy)
        shortcuts.onKeyPress("Ctrl+C", copy)

        cut = function() { executeCommand(
            new CutCommand(this, activeEditor)) }
        cutButton.setCommand(cut)
        shortcuts.onKeyPress("Ctrl+X", cut)

        paste = function() { executeCommand(
            new PasteCommand(this, activeEditor)) }
        pasteButton.setCommand(paste)
        shortcuts.onKeyPress("Ctrl+V", paste)

        undo = function() { executeCommand(
            new UndoCommand(this, activeEditor)) }
        undoButton.setCommand(undo)
        shortcuts.onKeyPress("Ctrl+Z", undo)

    // Execute a command and check whether it has to be added to
    // the history.

    method executeCommand(command) is
        if (command.execute())
            history.push(command)

    // Take the most recent command from the history and run its
    // undo method. Note that we don't know the class of that
    // command. But we don't have to, since the command knows
    // how to undo its own action.

    method undo() is
        command = history.pop()
        if (command != null)
            command.undo()

```

## چه زمانی باید از این الگو استفاده کنیم؟

- زمانی از این الگو استفاده کنید که میخواهید Object هایتان را با استفاده از یک سری عملیات به صورت پارامتری در بیاورید.

الگوی طراحی command میتواند فرایند فراخوانی یک متده است که میتواند یک شیء مستقل تبدیل کند. این تغییر قابلیت‌های جالب زیادی را پیش روی‌مان قرار می‌دهد. به طور مثال میتوانیم یک command را به عنوان پارامتر ورودی یک متده است بدهیم و یا اینکه آنها را در Object های دیگری ذخیره کنیم و حتی بین command ها در زمان اجرا جابجا شویم.

برای مثال شما در حال توسعه یک رابط کاربری گرافیکی هستید و میخواهید یک منو برای کاربران ایجاد کنید. میخواهید این قابلیت در منو وجود داشته باشد که وقتی کاربر بر روی یک گزینه کلیک میکند عملیاتی شروع شود.

- از این الگو زمانی استفاده کنید که میخواهید یک سری عملیات با ترتیب خاص، زمان خاص و یا بصورت ریموت اجرا شوند.

مثل هر Object دیگری یک command هم میتواند serialize شود. بنابراین میتوان آن را در یک فایل یا دیتابیس به صورت رشته‌ای از کاراکترها نوشت. این رشته میتواند مانند Object اولیه بازیابی شود. بنابراین میتوانید هر زمانی که خواستید آنها را اجرا کنید.

اما چیزهایی فراتر از این موارد نیز وجود دارد، به طور مشابه میتوانید هر کدام از این Command ها را در queue بگذارید و یا حتی آنها را بر بستر شبکه پخش کنید.

- از این الگو زمانی استفاده کنید که میخواهید عملیات برگشت به حالت قبلی را پیاده‌سازی کنید. قطعا راههای زیادی برای پیاده‌سازی undo/redo وجود دارد اما الگوی command یکی از محبوب ترین راه هاست.

برای اینکه بتوانید عملی که انجام شده را باز پس بگیرید، باید تاریخچه انجام عملیات را ذخیره کنید. تاریخچه ای عملیات، یک پشته از فرایندهای اجرا شده به همراه نسخه و وضعیت برنامه میباشد.

این روش دو اشکال دارد. اولاً، ذخیره وضعیت یک برنامه چندان آسان نیست زیرا برخی از آن ها میتوانند خصوصی باشند. این مشکل را میتوان با الگوی Memento کاهش داد.

دوم، پشتیبان‌گیری لحظه به لحظه از وضعیت برنامه ممکن است مقدار زیادی RAM مصرف کند. بنابراین، گاهی اوقات میتوانید به یک پیاده‌سازی جایگزین متول شوید. مثلاً به جای بازیابی حالت گذشته، دستور عملیات معکوس را انجام دهید.

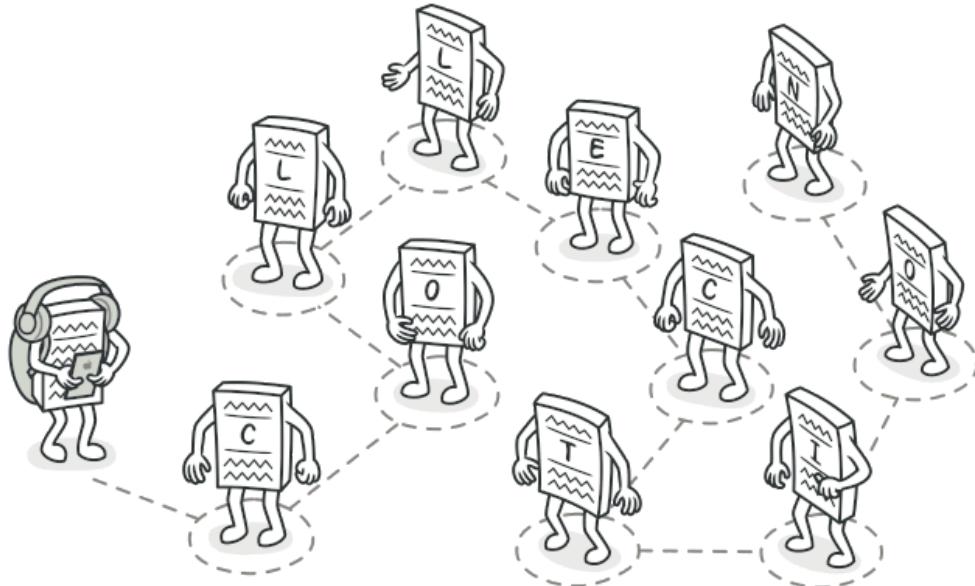
البته که عملیات معکوس نیز هزینه دارد: ممکن است اجرای آن سخت یا حتی غیر ممکن باشد.

## معایب و مزایا

- ❖ با استفاده از این الگوی طراحی اصل Single responsibility از SOLID رعایت می‌شود. چرا که می‌توانید کلاس‌هایی را که فقط عملیات را فراخوانی می‌کنند از کلاس‌هایی که عملیات را اجرا می‌کنند جدا کنید.
- ❖ با استفاده از این الگو اصل SOLID از اصول Open/Closed رعایت می‌شود چرا که بدون اینکه کد را تغییری دهید می‌توانید یک command جدید اضافه کنید.
- ❖ با استفاده از این الگو می‌توانید سیستم undo/redo را پیاده‌سازی کنید.
- ❖ با استفاده از این الگو می‌توانید اجرای بعضی از عملیات‌ها را به تعویق بیندازید.
- ❖ می‌توانید چند command ساده را در یک command پیچیده جمع‌آوری کنید.
- ➔ ممکن است که کد کمی پیچیده‌تر شود چرا که یک لایه کاملاً جدید بین فرستنده و گیرنده ایجاد می‌کنید.

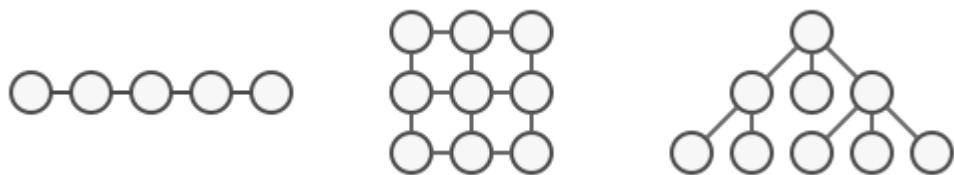
## Iterator

الگوی طراحی Iterator به ما کمک می‌کند تا بتوانیم یک مجموعه (Collection) را بدون در نظر گرفتن پیاده‌سازی هایش پیمایش کنیم.



### طرح مسئله

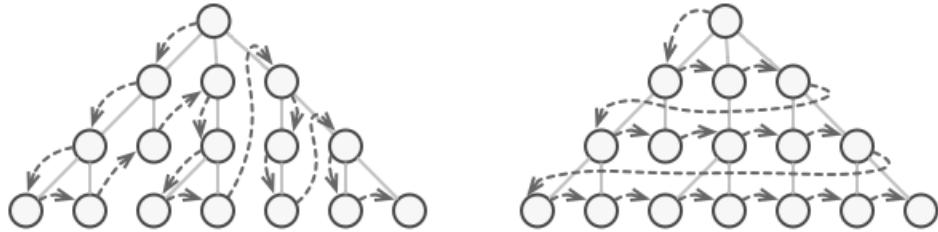
کالکشن‌ها یکی از پرکاربردترین data type ها در دنیای برنامه‌نویسی می‌باشند. با این حال یک کالکشن فقط ظرفی برای نگهداری یک سری Object است.



بیشتر کالکشن‌ها ایمان‌های خود را در یک لیست ساده نگه می‌دارند. اگر چه بعضی از آنها ساختار stack و انواع دیگری از ساختمان داده‌های پیچیده را دارند.

اما اینکه ساختار یک کالکشن چگونه است اهمیتی ندارد. چرا که کالکشن باید راهی برای دستیابی به ایمان‌های خود ارائه کند تا بقیه‌ی قسمت‌های برنامه بتوانند از آنها استفاده کنند. این راه باید به گونه‌ای باشد که بتوان به همه‌ی ایمان‌های داخل کالکشن بدون اینکه یک ایمان را دو بار پیمایش کنیم دسترسی داشت. اگر از لیست در برنامه‌هایتان استفاده کرده باشید این امر ساده به نظر می‌رسد. فقط کافیست تا یک حلقه بر روی لیستمان بزنیم. اما چگونه می‌توان عنصر یک ساختمان داده‌ی پیچیده، مانند درخت را به‌طور متوالی طی کرد؟

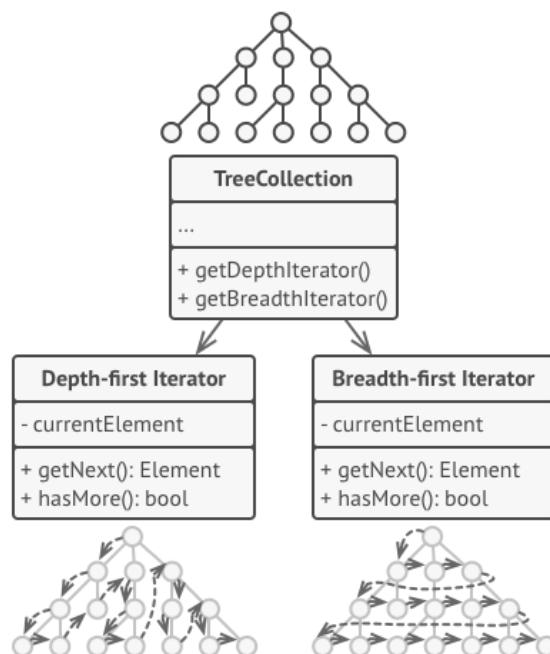
به طور مثال یک روز ممکن است که شما به پیمایش depth-first یا عمقی نیاز داشته باشید و روز بعد به پیمایش breadth-first یا اول سطح نیاز داشته باشید. به همین شکل ممکن است بعدها نیاز به دسترسی تصادفی به یک المان داشته باشد.



افزودن هر چه بیشتر الگوریتم برای جستجوی یک المان، کالکشن را از هدف اصلی اش که ذخیره‌سازی داده‌ها است دورتر می‌کند. علاوه بر این بعضی از الگوریتم‌ها ممکن است که فقط برای یک برنامه‌ی خاص طراحی شده باشند و استفاده از آنها بر روی یک کالکشن عمومی یک کار بیهوده و اضافی باشد. از سوی دیگر هم کلاینت به اینکه کالکشن چگونه داده‌ها را ذخیره می‌کند اهمیتی نمی‌دهد. با این حال از آنجایی که کالکشن‌ها خودشان یک سری راه‌ها را برای اینکه به المان‌هایشان دسترسی داشته باشیم ارائه می‌دهند، راهی جز اینکه کدام را به آن کالکشن وابسته کنیم نداریم. اما چه باید بگوییم تا از این بحث‌ها خلاص شویم؟

## راه حل

ایده‌ی اصلی الگوی Iterator این است که رفتاری که برای پیمایش یک کالکشن در نظر داریم را در درون یک Object جدا به نام Iterator قرار دهیم.

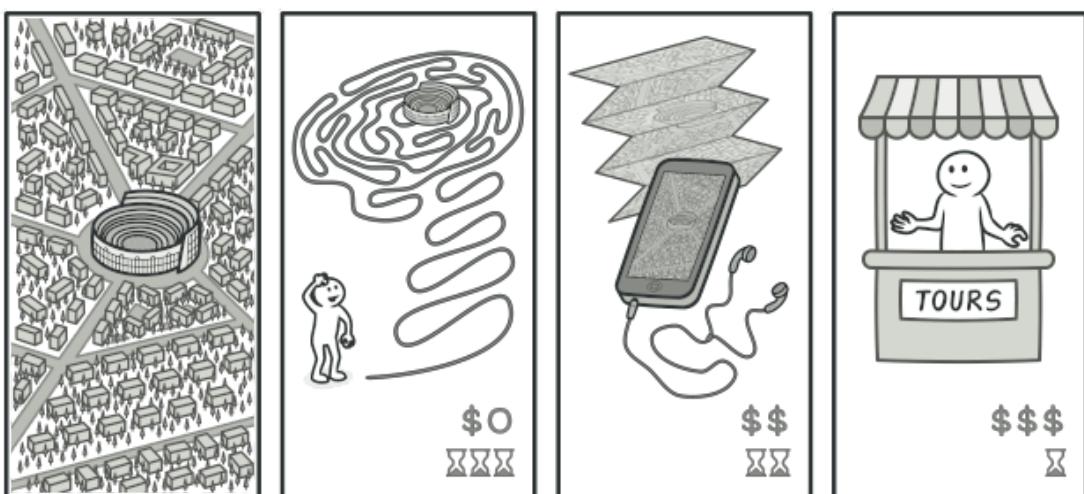


علاوه بر پیاده‌سازی خود الگوریتم، یک شیء Iterator تمام جزئیات پیمایش مانند موقعیت فعلی و تعداد عناصر باقی مانده تا پایان را نیز در خود نگه می‌دارد. به همین دلیل چندین Iterator می‌توانند هم با یکدیگر و هم به صورت مستقل در یک کالکشن باشند.

معمولاً Iterator ها یک متدهای واحد برای گرفتن یک المان از کالکشن در خود دارند. کلاینت می‌تواند این متدها را اجرا کند و تا زمانی که المان بر می‌گرداند از آن استفاده کند، چرا که در غیر این صورت یعنی به پایان کالکشن رسیده‌ایم.

نکته‌ای که باید به آن توجه کرد این است که تمامی Iterator ها باید یک interface یکسان را پیاده‌سازی کنند. این کار باعث می‌شود تا کلاینت بتواند از هر نوع کالکشن و یا الگوریتم پیمایشی که به آن نیاز دارد استفاده کند. زمانی که به یک الگوریتم جدید نیاز دارید فقط کافی است که یک Iterator جدید ایجاد کنید بدون اینکه به کدهای کالکشن‌تان دست بزنید.

### مقایسه با دنیای واقعی

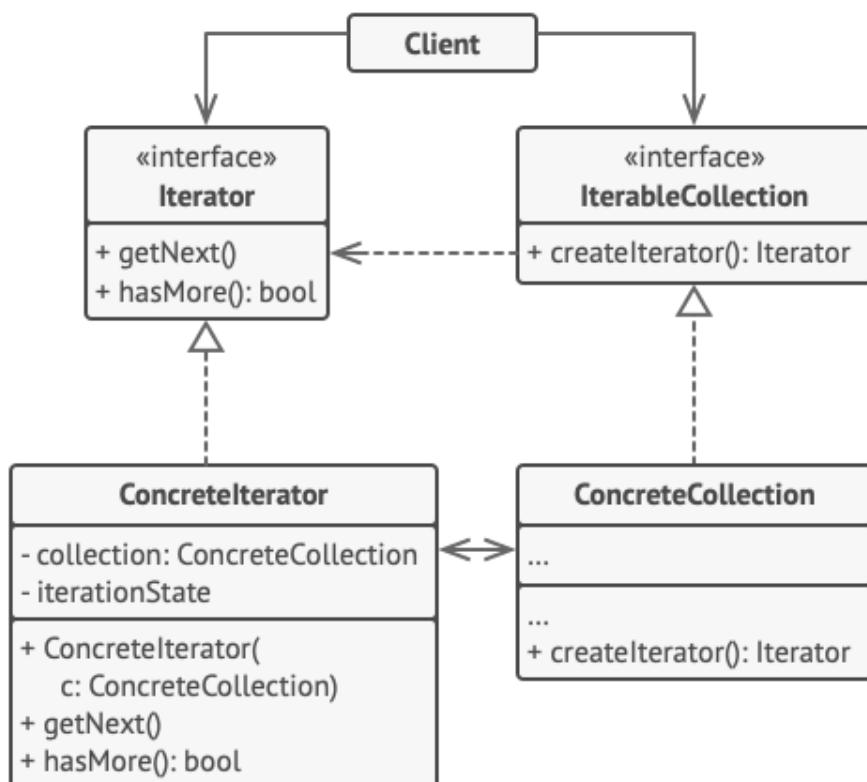


شما تصمیم گرفته‌اید که از شهر زیبای رم دیدن کنید و به مکان‌های جذاب و دیدنی‌اش سر بزنید. اما وقتی به رم می‌رسید زمان‌های زیادی را صرف گشتن می‌کنید و هی دور خود می‌چرخید به طوری که حتی Colosseum که یکی از معروف‌ترین مکان‌های دیدنی این شهر است را هم نمی‌توانید پیدا کنید. از طرفی هم می‌توانید که یک اپلیکیشن راهنمای مجازی برای گوشی تلفن همراه‌تان خریداری کنید و از آن برای مسیریابی کمک بگیرید. این برنامه هوشمند و ارزان است و تا هر زمانی که بخواهید می‌توانید در هر مکانی بمانید. راه حل دیگر هم این است که بخشی از بودجه‌ی سفرتان را برای استخدام یک فرد که رم را مثل کف دستش می‌شناسد در نظر بگیرید و از او راهنمایی بخواهید.

راهنما می‌تواند تور را مطابق میل شما تنظیم کند، هر جاذبه‌ای را به شما نشان دهد و داستان‌های هیجان‌انگیز زیادی را تعریف کند. این حتی سرگرم کننده‌تر خواهد بود؛ اما افسوس، گران‌تر است. خیلی هم گران‌تر است!

هر کدام از این گزینه‌ها یعنی دور زدن در شهر تا پیدا کردن یک جاذبه‌ی گردشگری، استفاده از اپلیکیشن راهنمای مجازی و استفاده کردن از یک فردی که کامل به رم آشناست، به عنوان یک Iterator عمل می‌کنند. که قرار است مکان‌های دیدنی رم را پیمایش کنند.

## ساختار

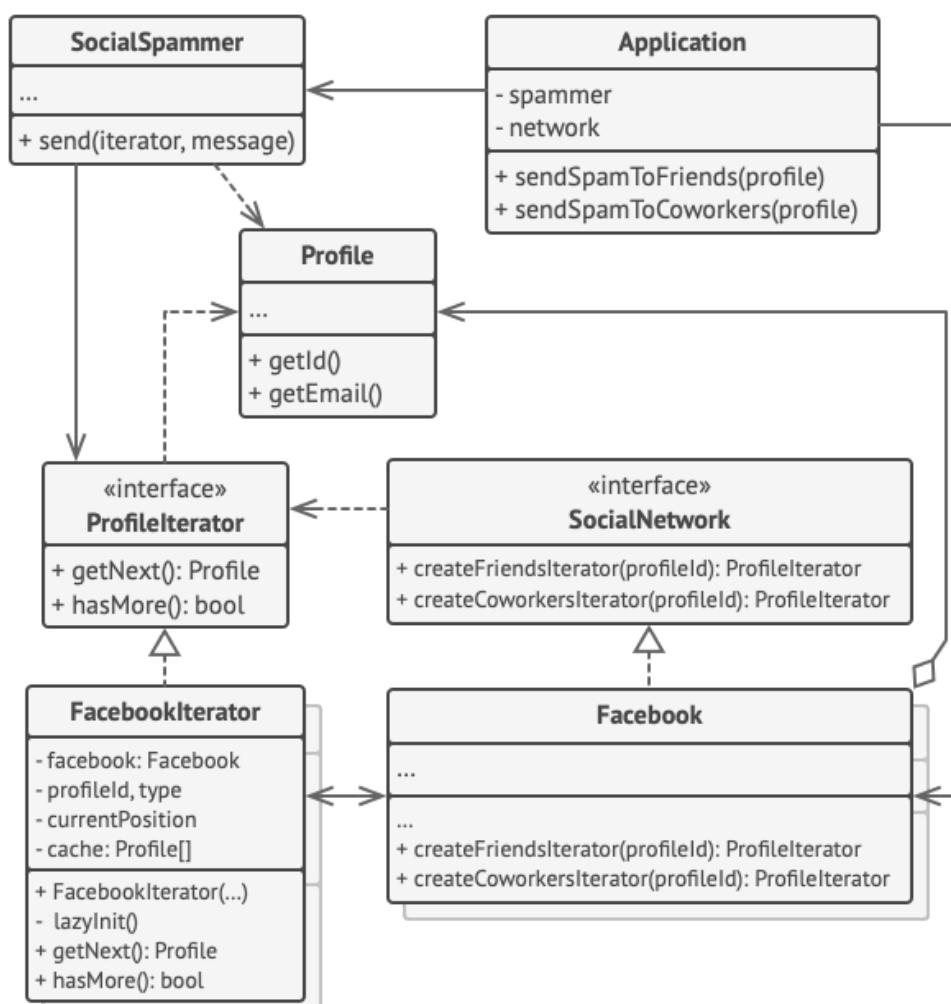


- **Iterator:** این Interface راههایی که باید برای پیمایش یک کالکشن طی کنیم را در خود تعریف می‌کند. کارهایی مانند گرفتن یک المان، بازگرداندن یک المان و ... .
- **Concrete Iterators:** این کلاس الگوریتم‌های خاص پیمایش را پیاده‌سازی می‌کند. شیء Iterator باید مسیر پیمایش یک کالکشن را دنبال کند. این کار باعث می‌شود تا چندین Iterator بتوانند مستقل از هم با یک کالکشن کار کنند.
- **Collection:** این Interface متدهایی تعریف می‌کند تا لیست Iterator هایی که با آن سازگار هستند را دریافت کند. به این نکته باید توجه شود که مقدار بازگشتی این متدها باید از جنس واسطه Iterator باشد و کالکشن‌هایی که از واسطه Collection ارث می‌برند می‌توانند انواع مختلفی از Iterator را به عنوان مقدار بازگشتی این متدها برگردانند.

- این کلاس‌ها در هر باری که کلاینت آنها را فراخوانی کند، یک نمونه‌ی جدید از Iterator بر می‌گردانند. باید تا الان تعجب کرده باشید و به این فکر کرده باشید که پس کد اصلی پیاده‌سازی کالکشن‌ها در کجا قرار می‌گیرد؟ نگران نباشید، آنها هم در همین کلاس‌ها پیاده‌سازی می‌شوند اما چون بحث اصلی این الگو چیز دیگری است راجع به آنها صحبت نمی‌کنیم.
- Client: کلاینت هم با کالکشن و هم با Iteratorها با استفاده از واسطه‌ایشان صحبت می‌کند. این کار باعث می‌شود تا کلاینت به کلاس‌های زیرین وابستگی‌ای نداشته باشد و بتواند با انواع کالکشن‌ها و Iteratorها کار کند.
- به طور کلی کلاینت‌ها خودشان Iterator ایجاد نمی‌کنند و آنها را از کالکشن‌ها دریافت می‌کنند. در حالت‌های خاصی کلاینت خودش یک Iterator ایجاد می‌کند و آن هم زمانی است که می‌خواهد یک Iterator مختص خودش ایجاد کند.

## مثال

در این مثال از الگوی طراحی Iterator استفاده شده است تا بتوانیم گراف‌های مربوط به شبکه‌ی اجتماعی فیسبوک را پیمایش کنیم. کالکشن چندین Iterator را در اختیارمان قرار می‌دهد تا با شیوه‌های گوناگونی پروفایل‌ها را پیمایش کنیم.



پیمایشگر 'friends' می‌تواند برای پیمایش دوستان یک کاربر خاص مورد استفاده قرار گیرد. پیمایشگر 'colleagues' نیز همین کار را می‌کند با این تفاوت که دوستانی که در همان شرکت کار می‌کنند را در نظر نمی‌گیرد و آنها را حذف می‌کند. هر دوی این Iterator ها یک Interface را پیاده‌سازی می‌کنند و به کلاینت این اجازه را می‌دهند تا بدون اینکه درگیر جزئیات پیاده‌سازی شود پروفایل‌ها را دریافت کند. کدهای کلاینت دیگر به کلاس‌های لایه‌ها داخلی وابسته نیست چرا که از با استفاده از واسطه‌ها با کالکشن‌ها و Iterator ها در ارتباط هستند. اگر هم زمانی تصمیم به اضافه کردن یک شبکه‌ی اجتماعی جدید گرفتیم، فقط کافی است که یک کالکشن و Iterator جدید اضافه کنید و نیازی به تغییر کدهای موجود نیست.

```
// The collection interface must declare a factory method for
// producing iterators. You can declare several methods if there
// are different kinds of iteration available in your program.
interface SocialNetwork {
    method createFriendsIterator(profileId) : ProfileIterator
    method createCoworkersIterator(profileId) : ProfileIterator

    // Each concrete collection is coupled to a set of concrete
    // iterator classes it returns. But the client isn't, since the
    // signature of these methods returns iterator interfaces.
class Facebook implements SocialNetwork {
    // ... The bulk of the collection's code should go here ...

    // Iterator creation code.
    method createFriendsIterator(profileId) is
        return new FacebookIterator(this, profileId, "friends")
    method createCoworkersIterator(profileId) is
        return new FacebookIterator(this, profileId, "coworkers")

    // The common interface for all iterators.
    interface ProfileIterator {
        method getNext() : Profile
        method hasMore() : bool

    // The concrete iterator class.
    class FacebookIterator implements ProfileIterator {
        // The iterator needs a reference to the collection that it
        // traverses.
        private field facebook: Facebook
        private field profileId, type: string

        // An iterator object traverses the collection independently
        // from other iterators. Therefore it has to store the
```

```

// iteration state.
private field currentPosition
private field cache: array of Profile

constructor FacebookIterator(facebook, profileId, type) is
    this.facebook = facebook
    this.profileId = profileId
    this.type = type

private method lazyInit() is
    if (cache == null)
        cache = facebook.socialGraphRequest(profileId, type)

    // Each concrete iterator class has its own implementation
    // of the common iterator interface.

method getNext() is
    if (hasMore())
        result = cache[currentPosition]
        currentPosition++
    return result

method hasMore() is
    lazyInit()
    return currentPosition < cache.length

// Here is another useful trick: you can pass an iterator to a
// client class instead of giving it access to a whole
// collection. This way, you don't expose the collection to the
// client.
//
// And there's another benefit: you can change the way the
// client works with the collection at runtime by passing it a
// different iterator. This is possible because the client code
// isn't coupled to concrete iterator classes.

class SocialSpammer is
    method send(iterator: ProfileIterator, message: string) is
        while (iterator.hasMore())
            profile = iterator.getNext()
            System.sendEmail(profile.getEmail(), message)

// The application class configures collections and iterators
// and then passes them to the client code.

class Application is
    field network: SocialNetwork
    field spammer: SocialSpammer

```

```

method config() is
    if working with Facebook
        this.network = new Facebook()
    if working with LinkedIn
        this.network = new LinkedIn()
    this.spammer = new SocialSpammer()

method sendSpamToFriends(profile) is
    iterator = network.createFriendsIterator(profile.getId())
    spammer.send(iterator, "Very important message")

method sendSpamToCoworkers(profile) is
    iterator = network.createCoworkersIterator(profile.getId())
    spammer.send(iterator, "Very important message")

```

## چه زمانی باید از این الگو استفاده کنیم؟

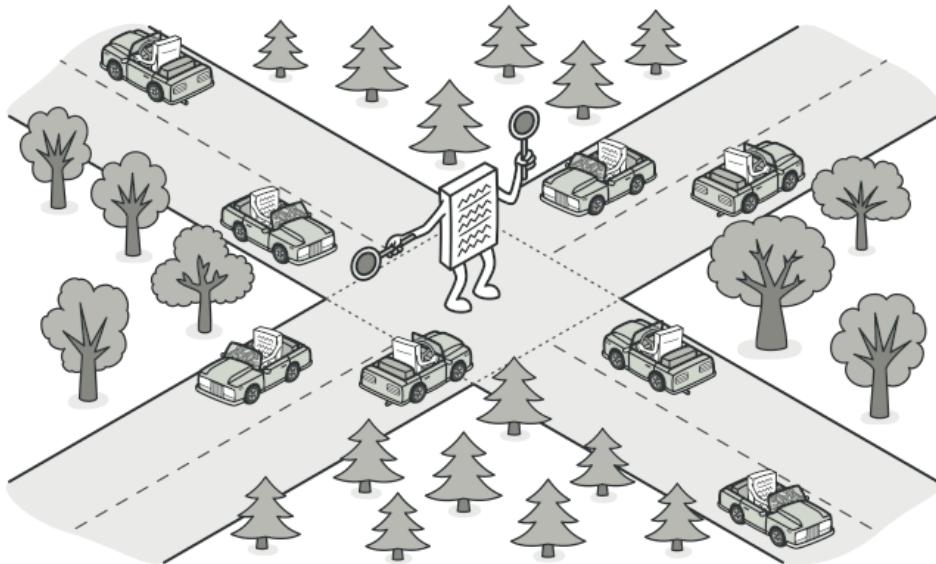
- زمانی از این الگو استفاده کنید که کالکشن‌تان از ساختمان داده‌ی پیچیده‌ای در پشت صحنه استفاده می‌کند و می‌خواهید این پیچیدگی را از چشم کلاینت پنهان کنید.(به دلایل امنیتی و یا قراردادی بین خودتان)
- از این الگو استفاده کنید تا از تکرار کد برای پیمایش موارد مختلف در برنامه‌تان جلوگیری کنید.
- اگر در دفعات زیاد الگوریتم‌های پیمایشتان را تکرار کنید، کدی کثیف خواهید داشت. وقتی این کثیفی در بیزینس اصلی برنامه‌تان وارد شود، کدهایتان غیر قابل نگهداری می‌شود و رفته رفته کد اصلی‌تان محو می‌شود. استفاده از Iterator ها باعث می‌شود تا کدهایی ظرفی و تمیز داشته باشید.
- زمانی از این الگو استفاده کنید که می‌خواهید بین انواع ساختمان داده‌ها جابجا شوید و یا در ابتداء اینکه نوع ساختمان داده چه چیزی باشد هنوز مشخص نیست.
- این الگو یک جفت Interface کلی برای کالکشن و Iterator ایجاد می‌کند. پس تا زمانی که کالکشن و Iterator ای که این واسطه‌ها را پیاده‌سازی کند به کدتان اضافه کنید کدتان کار می‌کند.

## معایب و مزایا

- ❖ با استفاده از این الگوی طراحی اصل Single responsibility از اصول SOLID رعایت می‌شود. چرا که می‌توانید کدهای کلاینت و کالکشن‌ها را با استفاده از الگوریتم‌های پیمایش از هم جدا کنید و کدی تمیز تر داشته باشید.
  - ❖ با استفاده از این الگو اصل Open/Closed از اصول SOLID رعایت می‌شود چرا که می‌توانید در هر زمان یک کالکشن و یا Iterator جدید بدون اینکه کد هایتان را تغییر دهید اضافه کنید.
  - ❖ می‌توانید به صورت موازی چند کالکشن را پیمایش کنید چرا که هر Iterator وضعیت مستقل خودش را دارد.
  - ❖ هر زمان که نیاز داشته باشید می‌توانید یک پیمایش را به تعویق بیندازید و دوباره آن را ادامه دهید.
- زمانی که برنامه‌تان فقط از کالکشن‌های ساده استفاده کند، استفاده از این الگو نابود کننده است!
- استفاده از Iterator ممکن است کارایی کمتری نسبت به اینکه عناصر یک سری از کالکشن‌های به خصوص را بررسی کنیم داشته باشد.

## Mediator

الگوی طراحی Mediator به ما این امکان را می‌دهد تا وابستگی‌های آشفته‌ی بین Object‌ها را کاهش دهیم. این الگو Object‌ها را به نحوی محدود می‌کند تا تنها از طریق راه ارتباطی‌ای که تعیین شده است با هم ارتباط داشته باشند.



### طرح مسئله

فرض کنید در حال توسعه رابط کاربری پروفایل برای مشتری هستید و می‌خواهید یک دیالوگ (Dialog) به آن نمایش دهید که دارای اجزایی شامل فیلد‌های متñی، چک باکس‌ها، دکمه‌ها و ... می‌باشد. این اجزا ممکن است به یکدیگر وابسته باشند. برای مثال، اگر چک باکس "من سگ دارم" انتخاب شود، یک فیلد مخفی برای وارد کردن نام سگ نشان داده می‌شود یا با زدن دکمه ارسال قبل از ذخیره داده‌ها، مقادیر تمام فیلد‌ها باید اعتبارسنجی شوند.

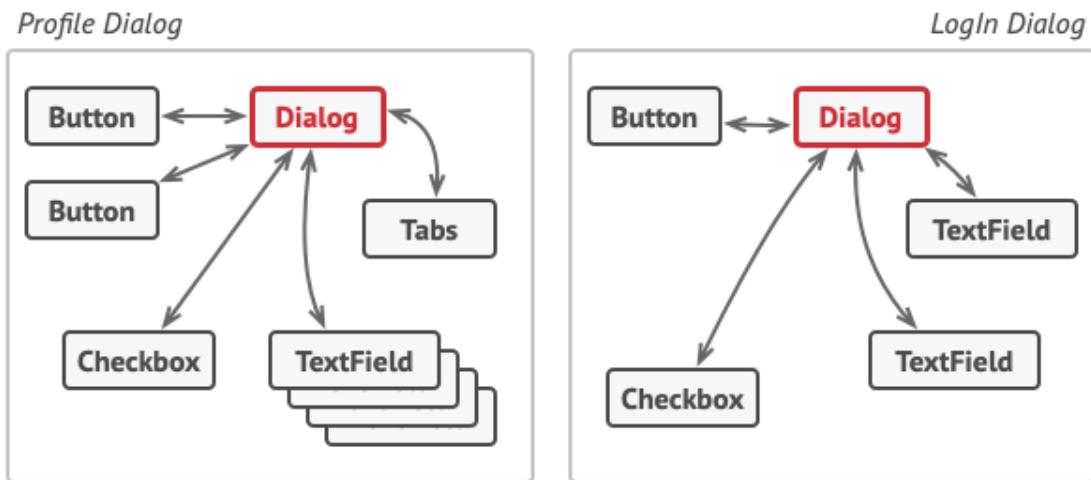


اگر این کلاس‌ها این گونه با هم ارتباط داشته باشند، دیگر نمی‌شود از آنها در جاهای دیگر برنامه به راحتی استفاده کرد. به طور مثال از چک‌باکسی که تعریف شده است دیگر نمی‌شود در کلاس دیگری استفاده کرد چرا که این چک باکس فقط برای اسم حیوان سگ کاربرد دارد.

## راه حل

الگوی طراحی Mediator پیشنهاد می‌کند که ارتباط بین اجزایی که می‌خواهید مستقل باشند را با یکدیگر قطع کنید و به جای آن با استفاده از یک Object میانی بین آنها ارتباط برقرار کنید و این Object میانی فراخوانی‌ها را به کامپوننت‌های متناظر با آن متصل می‌کند. با این کار کامپوننت‌ها به جای اینکه به حجم وسیعی از Object‌ها وابسته باشند فقط با شیء Mediator در ارتباطند.

در مثالی که زده شد، کلاس Dialog می‌تواند نقش یک Mediator را بازی کند. این کلاس همچنان از إلمان‌های زیرشاخه‌ی خود جداست و لازم نیست که وابستگی جدیدی در این کلاس اضافه کنید.

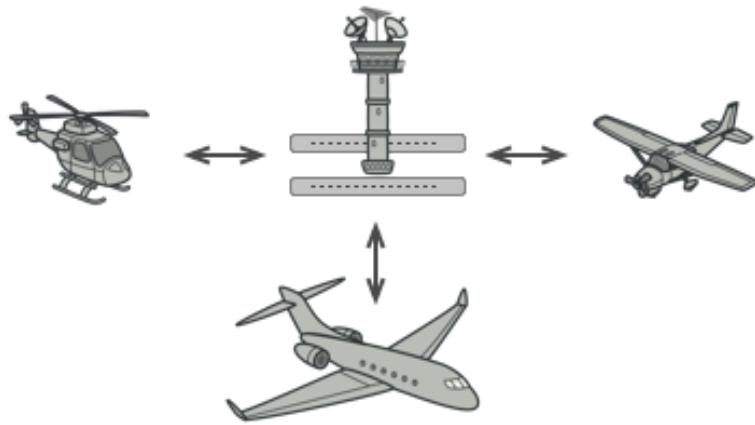


اتفاق اصلی در داخل المان‌ها می‌افتد. بباید دکمه‌ی submit را در نظر بگیریم. قبل‌اً، هر بار که کاربر روی دکمه کلیک می‌کرد، باید مقادیر تمام عناصر فرم را تأیید می‌کرد. اما حالا فقط این موضوع را به Dialog اطلاع می‌دهد و Dialog بعد از دریافت همچین خبری خودش اعتبارسنجی‌ها را انجام می‌دهد و یا این وظیفه را به عناصر جداگانه‌ی مستقلی محوی کند. به جای اینکه دکمه‌ی submit به چندین المان و عنصر از فرم اعتبارسنجی گره بخورد، فقط به کلاس Dialog وابسته است.

هر چه جلوتر می‌روید با ایجاد Interface‌های مشترک بین Dialog‌ها می‌توانید وابستگی‌ها را کاهش دهید. این واسطه‌ها باید دارای متدهای برای اطلاع‌رسانی از سمت همه‌ی المان‌ها باشند تا المان‌ها بتوانند هر رخدادی که ایجاد می‌شود را به گوش Dialog برسانند. در این صورت دکمه‌ی submit می‌تواند با هر Dialog‌ای که Interface عمومی را پیاده‌سازی کرده باشد ارتباط برقرار کند.

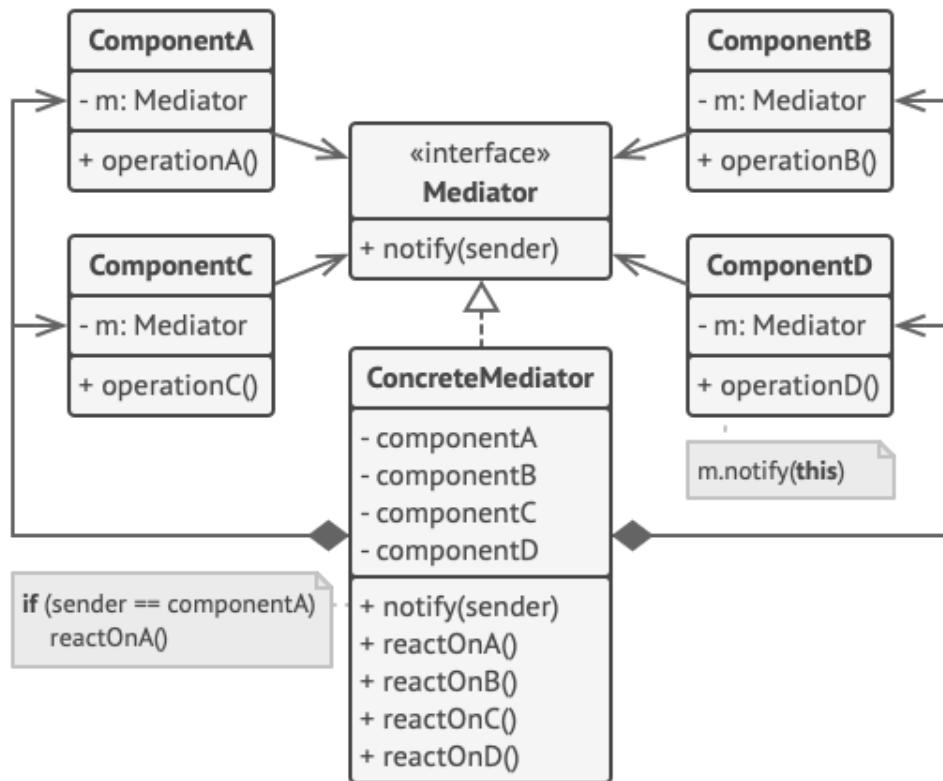
تا به اینجای کار، الگوی Mediator این موقعیت را فراهم کرده است تا رابطه‌های پیچیده بین المان‌ها یک صفحه‌ی وب را ساده‌سازی کنیم و وظیفه‌ی این ارتباط‌ها را به دوش یکی شیء Mediator بیندازیم.

## مقایسه با دنیای واقعی



دقت کنید که خلبان‌هایی که به برج مراقبت نزدیک می‌شوند و یا در حال خارج شدن از باند هستند هیچ وقت به صورت مستقیم با یکدیگر ارتباط برقرار نمی‌کنند. آنها با شخصی در برج مراقبت در ارتباطند که وظیفه‌ی آن کنترل این کار است. اگر این فرد نباشد، تمام خلبانان باید خودشان بدانند که چه هواپیماهی الان در نزدیکی باند است باید این را هم بدانند که اولویت پرواز و یا فرود در حال حاضر با کدام یک از این هواپیماها است و باید همزمان با دهها خلبان صحبت کنند. این کار احتمالاً آمار سقوط‌های هواپیمایی را افزایش می‌دهد!

کنترل کننده نیازی به اینکه تمامی پروازها را کنترل کند ندارد و فقط پروازهایی که در لحظه در نزدیکی باند هستند را کنترل می‌کند. چرا که کنترل این تعداد از هواپیماها برای یک خلبان کار دشواری است.

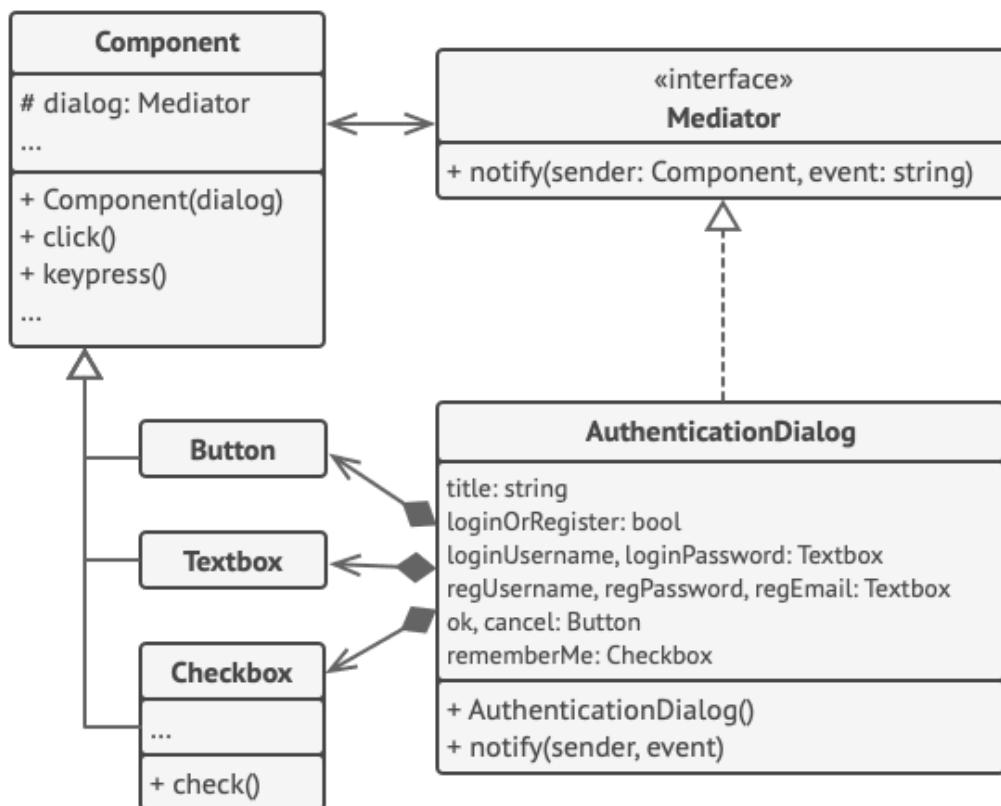


- **Components:** کامپوننت‌ها کلاس‌های متفاوتی هستند که انواع منطق بیزینس‌مان در آن وجود دارد. هر کامپوننت رفرنسی به یک Interface Mediator که از نوع Interface معمومی است دارد. کامپوننت اطلاعی از کلاس اصلی Mediator ندارد، بنابراین می‌توانید از این کامپوننت در جاهای دیگر برنامه تنها با تغییر Mediator آن استفاده کنید.
- **Mediator Interface:** این متدهایی که برای ارتباط بین کامپوننت‌ها نیاز است را تعریف می‌کند. معمولاً این واسطه دارای یک متدهای اطلاع‌رسانی رخدادها می‌باشد. کامپوننت‌ها هر فیلڈی را ممکن است به عنوان ورودی به این متدهای دهد. به طور مثال می‌توانند Object هایی از جنس خودشان نیز ارسال کنند اما باید توجه داشت که نباید بین کامپوننت ارسال کننده و گیرنده وابستگی‌ای به وجود بیايد.
- **Concrete Mediator:** این کلاس‌ها در واقع پایه‌ی روابط بین کامپوننت‌ها هستند. این کلاس‌ها معمولاً رفرنس تمامی کامپوننت‌هایی که با آنها در ارتباط هستند را در خود نگه می‌دارند و گاهی وقت هم چرخه‌ی حیات آنها را کنترل می‌کنند.
- نکته‌ای که باید به آن توجه کرد این است که کامپوننت‌ها نباید از وجود هم اطلاعی داشته باشند. اگر اتفاق مهمی قرار است که از کامپوننتی به کامپوننتی دیگر منتقل شود، باید به mediator اطلاع داده شود. بعد از اینکه mediator خبری دریافت کرد، به راحتی ارسال کننده‌ی پیام را تشخیص

می‌دهد و همین اطلاعات کافی است تا گیرنده‌ی مناسب را پیدا کند و خبر را به دست آن برساند. از دید یک کامپونت همه چیز مانند یک جعبه‌ی سیاه است. فرستنده نمی‌داند که چه کسی در نهایت درخواستی که دارد را انجام می‌دهد و همچنین گیرنده نیز نمی‌داند که در مرحله‌ی اول چه کسی خبر را فرستاده است.

## مثال

در این مثال، الگوی Mediator به شما کمک می‌کند تا وابستگی‌های متقابل بین کلاس‌های مختلف ای را حذف کنید: دکمه‌ها، چک باکس‌ها و برچسب‌های متن.



المانی که توسط کاربر فعل می‌شود به طور مستقیم با دیگر المان‌ها ارتباط ندارد، هر چند که شاید از بیرون اینگونه به نظر برسد اما در داخل اینگونه نیست چرا که المان فقط باید واسط خودش را پیدا کند و از وجود یک رویداد جدید آن را آگاه کند و هرگونه اطلاعات متنی را برای آن ارسال کند.

در این مثال قسمت dialog احراز هویت همانند یک mediator عمل می‌کند. این قسمت می‌داند که المان‌ها چگونه باید با یکدیگر ارتباط برقرار کنند و باعث ساده‌سازی ارتباط بین آنها می‌شود. هر زمان که از وجود یک رخداد مطلع شود، dialog تصمیم می‌گیرد که این رخداد از سمت چه المانی باید کنترل شود.

```

// The mediator interface declares a method used by components
// to notify the mediator about various events. The mediator may
// react to these events and pass the execution to other
// components.
interface Mediator is
    method notify(sender: Component, event: string)

// The concrete mediator class. The intertwined web of
// connections between individual components has been untangled
// and moved into the mediator.
class AuthenticationDialog implements Mediator is
    private field title: string
    private field loginOrRegisterChkBx: Checkbox
    private field loginUsername, loginPassword: Textbox
    private field registrationUsername, registrationPassword,
        registrationEmail: Textbox
    private field okBtn, cancelBtn: Button

    constructor AuthenticationDialog() is
        // Create all component objects by passing the current
        // mediator into their constructors to establish links.

    // When something happens with a component, it notifies the
    // mediator. Upon receiving a notification, the mediator may
    // do something on its own or pass the request to another
    // component.
    method notify(sender, event) is
        if (sender == loginOrRegisterChkBx and event == "check")
            if (loginOrRegisterChkBx.checked)
                title = "Log in"
                // 1. Show login form components.
                // 2. Hide registration form components.
        else
            title = "Register"
            // 1. Show registration form components.
            // 2. Hide login form components

        if (sender == okBtn && event == "click")
            if (loginOrRegister.checked)
                // Try to find a user using login credentials.
                if (!found)
                    // Show an error message above the login
                    // field.
            else
                // 1. Create a user account using data from the
                // registration fields.

```

```

        // 2. Log that user in.
        // ...

// Components communicate with a mediator using the mediator
// interface. Thanks to that, you can use the same components in
// other contexts by linking them with different mediator
// objects.
class Component is
    field dialog: Mediator

    constructor Component(dialog) is
        this.dialog = dialog

    method click() is
        dialog.notify(this, "click")

    method keypress() is
        dialog.notify(this, "keypress")

// Concrete components don't talk to each other. They have only
// one communication channel, which is sending notifications to
// the mediator.
class Button extends Component is
    // ...

class Textbox extends Component is
    // ...

class Checkbox extends Component is
    method check() is
        dialog.notify(this, "check")
    // ...

```

چه زمانی باید از این الگو استفاده کنیم؟

- زمانی از این الگو استفاده کنید که کلاسی به شدت به تعداد زیادی از کلاس‌ها وابسته است و تغییر دادن آن کار دشواری است.
- این الگو اجازه می‌دهد تا وابستگی بین کلاس‌ها را به یک کلاس جدایانه بسپاریم و اینگونه به راحتی می‌توان مستقل از دیگر کامپوننت‌ها تغییرات مورد نظر را اعمال کرد.

- زمانی از این الگو استفاده کنید که نمی‌توانید از یک کامپوننت در یک برنامه‌ی متفاوت دیگر استفاده کنید. چرا که آن کامپوننت به شدت به دیگر کامپوننت‌ها وابسته است.  
بعد از اینکه از Mediator استفاده کنید دیگر کامپوننت‌ها از وجود یکدیگر خبر ندارند. آنها همچنان می‌توانند به وسیله‌ی mediator با یکدیگر ارتباط داشته باشند.
- هر زمان که نیاز به استفاده از کامپوننتی در یک برنامه‌ی متفاوت دارید، فقط کافی است که یک Mediator جدید برای آن ایجاد کنید.
- هنگامی که احساس کردید تعداد زیادی زیر-کلاس از کامپوننت ایجاد می‌کنید تا از برخی رفتارهای اساسی در زمینه‌های مختلف دوباره استفاده کنید، از Mediator استفاده کنید.  
از آنجایی که از mediator برای ارتباط بین کامپوننت‌ها استفاده می‌شود، ایجاد راههای جدید برای ارتباط بین کامپوننت‌ها بدون اینکه در آنها تغییراتی ایجاد کنیم بسیار آسان است.

## معایب و مزایا

- ❖ با استفاده از این الگوی طراحی اصل SOLID از اصول Single responsibility می‌شود چرا که می‌توانید روابط موجود در برنامه را به یک مکان مستقل انتقال دهید و این کار باعث ساده‌تر شدن مدیریت کد و درک آن می‌شود.
- ❖ با استفاده از این الگوی طراحی اصل Open/Closed از اصول SOLID می‌شود چرا که بدون اینکه تغییراتی در کامپوننت‌هایمان ایجاد کنیم می‌توانیم Mediator ها جدید اضافه کنیم.
- ❖ با استفاده از این الگو می‌توان وابستگی بین کامپوننت‌ها را کاهش داد.
- ❖ می‌توانید هر تعدادی که می‌خواهیم از کامپوننت‌هایمان استفاده کنیم.

→ باید به این توجه داشت که بعد از مدتی ممکن است که یک God Object به یک Mediator تبدیل شود.

## Memento

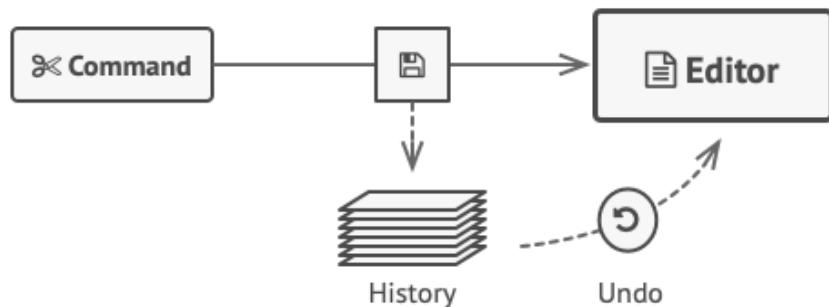
الگوی طراحی Memento این امکان را به ما می‌دهد تا بتوانیم وضعیت‌های قبلی یک Object را بدون آشکار شدن جزئیاتش ذخیره و بازیابی کنیم.



### طرح مسئله

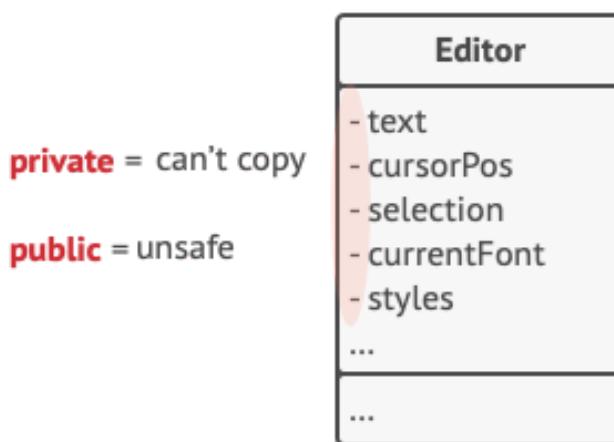
فرض کنید می‌خواهید برنامه‌ی ویرایشگر متن بنویسید. علاوه بر عملیات ساده‌ی ویرایش متن، برنامه‌ی شما قابلیت‌های دیگری مانند تصحیح متن، اضافه کردن عکس در بین متن و ... را نیز دارد.

تصمیم گرفتید تا قابلیت undo را به برنامه‌تان اضافه کنید. این قابلیت ساله‌است که هر برنامه‌ی ویرایشگر متنی ای آن را دارد و کاربران نیز انتظار دارند تا برنامه‌ی شما هم این قابلیت را داشته باشد. برای پیاده‌سازی این قابلیت یک رویکرد کاملاً مستقیم را انتخاب می‌کنید. یعنی قبل از انجام هر عملیاتی تمام وضعیت‌های برنامه را در مقداری از حافظه ذخیره می‌کنید. در واقع انگار از هر وضعیت برنامه یک عکس فوری (snapshot) می‌گیرید و آن را در حافظه ذخیره می‌کنید. پس اطلاعات را در کلاسی با عنوان History ذخیره می‌کنید و آن را در حافظه می‌ریزیم. بعد از مدتی که کاربر تصمیم به برگرداندن عملیات گرفت، برنامه آخرین تاریخچه که همان snapshot است را از حافظه می‌خواند و بر اساس آن وضعیت همه‌ی اشیاء موجود در برنامه را بازیابی می‌کند.



بیایید به این فکر کنیم که برنامه چگونه این snapshot را از حافظه می‌خواند. در حالت کلی احتمالاً نیاز دارد تا تمام فیلدهای یک Object را ببینید و مقدارشان را در حافظه ذخیره کنید. این در صورتی کار می‌کند که بتوان به راحتی به فیلدهای هر Object دسترسی داشت تا بتوان محتویات آن را بررسی کرد. اما در واقعیت اشیاء به همین راحتی این اجازه را نمی‌دهند و برای دسترسی به محتویات‌شان محدودیت‌هایی اعمال می‌کنند.

فعلاً این مورد را نادیده می‌گیریم و فکر می‌کنیم که اشیاء مان همچین محدودیت‌هایی را در نظر نگرفته‌اند. با اینکه این کار مشکلمان را خیلی سریع حل می‌کند اما هنوز مشکلات جدی تری نیز وجود دارند. در آینده شما تصمیم می‌گیرید که بخشی از کلاس editor تان را ویرایش کنید یا تعدادی از فیلدها را حذف و یا اضافه کنید. اما این کار بر روی کلاس‌هایی که مسئول کپی کردن وضعیت‌های کلاس‌ها هستند هم تاثیر می‌گذارد و باید همه‌ی آن‌ها را نیز تغییر دهید.



بیایید آن snapshot که از وضعیت یک Object می‌گرفتیم را بررسی کنیم. به نظرتان شامل چه داده‌هایی است؟ این snapshot باید حداقل شامل متن واقعی، مختصات مکان‌نما(cursor)، موقعیت اسکرول فعلی و... باشد. با این کار یک snapshot داریم که تعداد فیلدهای زیادی برای ذخیره‌سازی اطلاعات هر Object دارد. سپس برای اینکه یک تاریخچه یا همان snapshot را از وضعیت فعلی Object-مان تهیه کنیم باید این مقادیر را جمع آوری کنیم و در چیزی مانند یک ظرف(container) که حاوی فیلد های Object است بریزیم. در واقع شما تعداد زیادی از این ظرف‌ها را دارید که با کنار هم گذاشتن این‌ها به تاریخچه‌ی Object ها دسترسی پیدا می‌کنید. انگار این کانتینرها یا همان ظرف‌ها خودشان Object ای از یک کلاس هستند. این کلاس می‌تواند هیچ متدهای نداشته باشد اما فیلدهایی دارد که این فیلدها نمایانگر وضعیت Object در هر لحظه هستند.

برای اینکه بتوانیم اطلاعات یک Object را بخوانیم احتمالاً باید به فیلدهایش دسترسی public بدهیم تا بتوانیم این اطلاعات را از آن snapshot ای که می‌گیریم استخراج کنیم. اما این کار باعث می‌شود اطلاعات همه‌ی این فیلدها در همه جا آشکار شوند و دیگر فیلد خصوصی ای نداشته باشیم. در ضمن به ازای هر

تغییر کوچکی در کلاس snapshot، بقیه ای کلاس‌ها به آن وابسته می‌شوند. به عبارت دیگر، اگر کلاس snapshot تغییر کند، کلاس‌های دیگری که از آن استفاده می‌کنند نیز ممکن است تحت تأثیر قرار گیرند. به این مورد باید توجه کرد که در صورتی که فیلدهای کلاس snapshot به صورت private تعریف شده باشند تغییرات این کلاس تاثیری بر روی دیگر کلاس‌ها نخواهد داشت، چرا که اطلاعات آن فیلد فقط مختص همان کلاس است. اما اگر این فیلدها به صورت public باشند و بقیه ای کلاس‌هایی که از snapshot استفاده می‌کنند به آنها دسترسی داشته باشند، هر تغییری که در این فیلدها ایجاد شود بر روی همه کلاس‌ها تاثیر می‌گذارد.

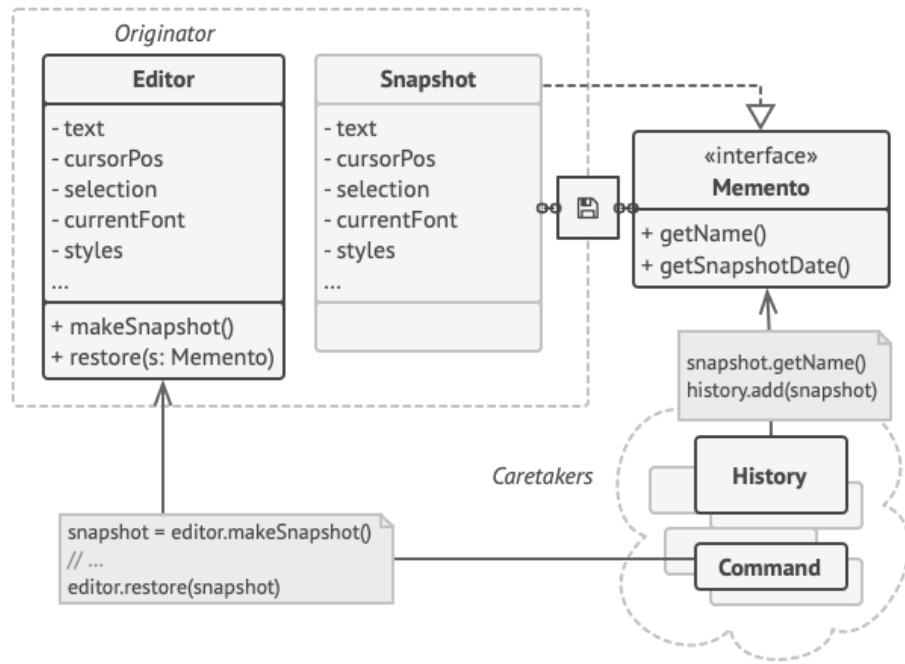
فکر کنم به یک بن بست رسیده ایم. شما یا تمام جزئیات کلاس را public می‌کنید و آن را در منظر عموم قرار می‌دهید. یا اینکه آنها را private می‌کنید که با این کار دیگر نمی‌توانید از کلاس snapshot به آن دسترسی داشته باشید و تاریخچه را ذخیره کنید. آیا راه حل دیگری برای ایجاد قابلیت Undo به ذهن شما می‌رسد؟؟

## راه حل

تمام مشکلاتی که تا به حال با آنها روبرو شده بودیم بخاطر حفظ حریم شخصی هر فیلد است! بعضی از Object‌ها سعی می‌کنند بیشتر از آنچه که باید کار کنند. برای اینکه یک سری داده‌های را برای انجام کارهایی جمع آوری کنند، به حریم خصوصی Object‌های دیگر حمله می‌کنند. در صورتی که باید آن کار را به همان Object‌ها محول کنند.

الگوی طراحی memento ایجاد snapshot از وضعیت هر Object را به خودش واگذار می‌کند. با این کار به جای اینکه کلاس‌های دیگر مانند snapshot در کار مداخله کنند خود Object وارد عمل می‌شود و ذخیره‌سازی تاریخچه‌اش را انجام می‌دهد. از این رو دیگر به مشکل دسترسی به فیلدها برنمی‌خوریم چرا که خود Object به همه فیلدها و وضعیت خودش دسترسی دارد.

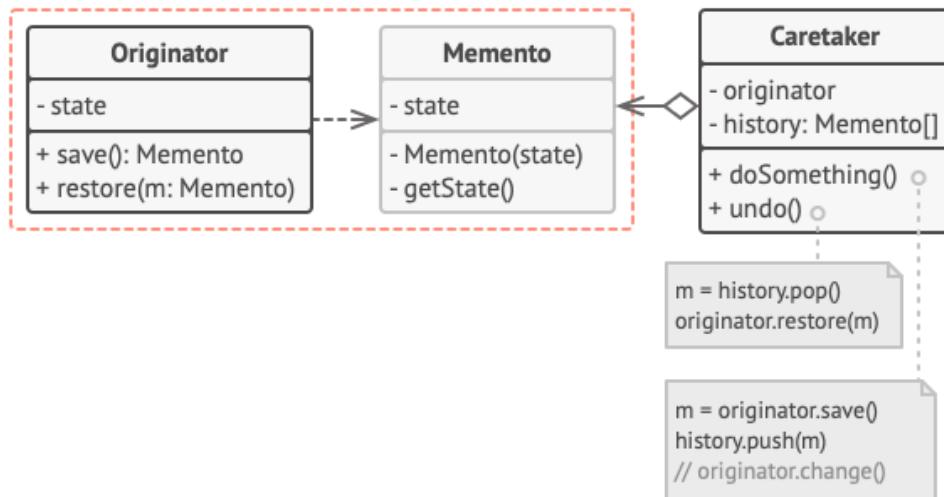
این الگو پیشنهاد می‌کند تا یک کپی از وضعیت فعلی را در کلاسی تحت عنوان memento ذخیره کنیم. هیچ Object ای به غیر از خود Object ای که این memento را ایجاد کرده است به محتوای درون آن دسترسی ندارد. کلاس‌های دیگر فقط از طریق یک واسط که آن هم محدودیت‌هایی دارد می‌توانند با این memento ارتباط برقرار کنند و داده‌های مختصی مانند تاریخ ایجاد memento، نام عملیات و ... را دریافت کنند و نمی‌توانند به وضعیت Object اصلی که در حال ذخیره سازی تاریخچه‌اش هستیم دسترسی ای داشته باشند.



چنین سیاستگذاری هایی این اجازه را به شما می دهد تا memento ها را در داخل اشیاء دیگر که معمولاً به آنها می گوییم نگهداری کنید. از آنجایی هم که caretaker از طریق واسط محدود با ارتباط برقرار می کند نمی تواند وضعیت ذخیره شده در memento را دستکاری کند. در عین حال کسی که memento را ایجاد کرده است به تمام فیلدهای داخل آن دسترسی دارد و به راحتی می تواند وضعیت های قبلی خود را بازیابی کند.

در مثال ویرایشگر متمنی که قبل تر بحث کردیم، می توانیم یک کلاس تاریخچه جدا بسازیم تا مانند یک caretaker عمل کنیم. پشته ای که در memento وجود دارد در داخل caretaker ذخیره می شود تا در هر باری که یک عملیات صورت گرفت به این پشته اضافه شود. شما می توانید خروجی این پشته را در رابط کاربری خودتان نشان دهید تا کاربر تاریخچه کارهایی که انجام داده است را مشاهده کند. هنگامی که کاربر Undo را انتخاب کند تاریخچه برنامه جدید ترین memento را از پشته می گیرد و آن را به editor می فرستد. از آنجایی هم که editor به memento خودش دسترسی کامل دارد، وضعیت جدید خودش را جایگزین می کند.

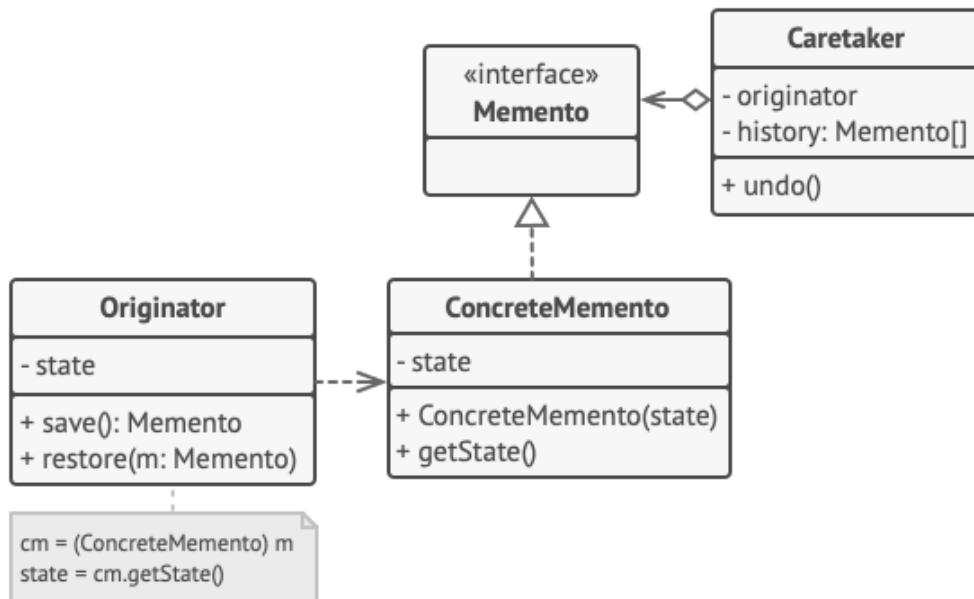
❖ پیاده‌سازی برای اساس کلاس‌های داخلی



- **Originator:** این کلاس می‌تواند یک snapshot از وضعیت فعلی خودش ایجاد کند و یا هر زمان که نیاز بود snapshot مورد نیاز را بازیابی کند.
- **Memento:** شیء ای است که همانند یک snapshot عمل می‌کند و وضعیت‌های Originator را ذخیره می‌کند. خوبی خوب است که memento را تغییر ناپذیر یا Immutable کنیم و تنها یک بار با استفاده از سازنده آن را مقداردهی کنیم.
- **Caretaker:** این کلاس علاوه بر اینکه می‌داند چه زمانی و چرا باید وضعیت یک originator را دریافت کند، از این که چه زمانی باید آن را بازیابی کند هم باخبر است. Caretaker می‌تواند تاریخچه‌ی یک originator را با استفاده از پشته‌ای از memento های آن دنبال کند. زمانی که originator باید یک گام به عقب برگردد، caretaker آخرین خانه از پشته‌ی memento ها را برمی‌دارد و آن را به متدهای originator پاس می‌دهد.
- در این پیاده‌سازی کلاس memento در داخل originator قرار دارد. این کار باعث می‌شود تا originator به تمامی فیلدهای داخل memento حتی اگر private تعریف شده باشند هم دسترسی کامل داشته باشد. از طرف دیگر، caretaker دسترسی محدودی به فیلدها و متدهای داخل memento دارد. این کار باعث می‌شود تا بتواند آنها را در داخل پشته ذخیره کند اما نتواند وضعیت آنها را مشاهده کند.

### ❖ پیاده سازی بر اساس واسط میانی

این پیاده سازی برای زبان های برنامه نویسی ای می باشد که از کلاس های داخلی(nested classes) استفاده می کنند.

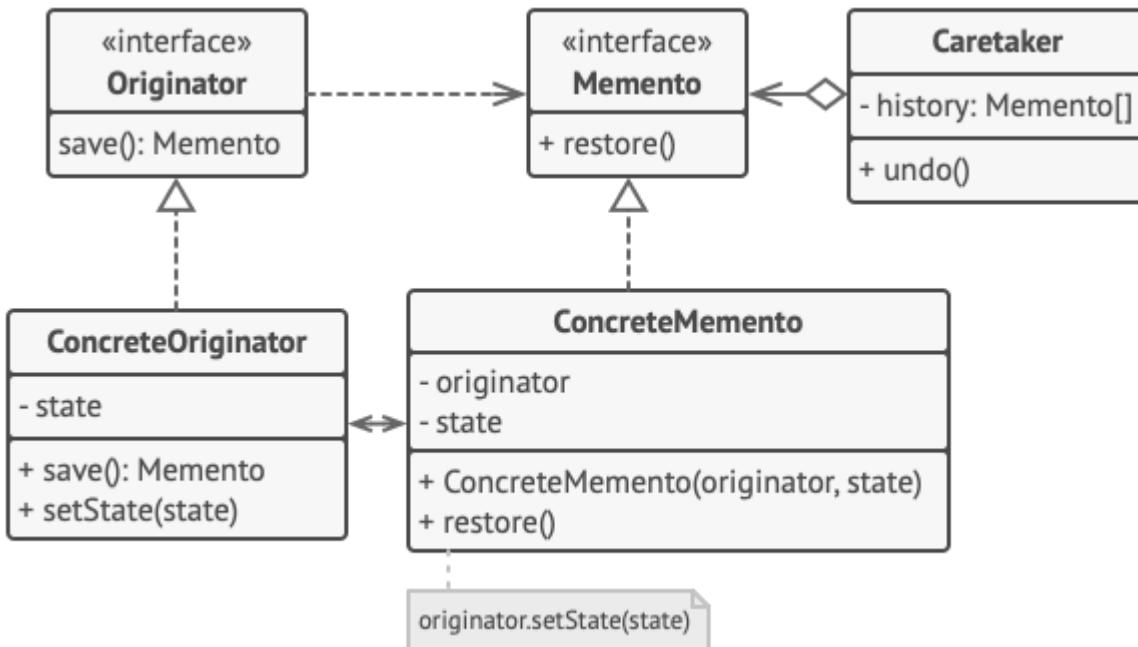


- در این مورد که نمی توان از کلاس داخلی استفاده کرد، می توان با ایجاد قراردادی برای کلاس caretaker یک سری محدودیت برای دسترسی به memento ایجاد کرد. به گونه ای که caretaker فقط با استفاده از یک واسط میانی بتواند با memento ارتباط برقرار کند. این واسط میانی فقط شامل متدهایی برای دریافت یک سری از اطلاعات خاص memento می باشد.

- از طرفی دیگر، originator می تواند به صورت مستقیم با memento در ارتباط باشد و به متدها و فیلدهای آن دسترسی داشته باشد. پشت پرده ای این پیاده سازی به این گونه است که باید تمام فیلدهای داخل memento را به صورت public تعریف کنید. با این کار مشکلی پیش نخواهد آمد چرا که از آن طرف دسترسی caretaker را محدود کرده اید و فقط از طریق یک واسط میانی می تواند با memento ارتباط برقرار کند.

❖ پیاده‌سازی با کپسوله سازی ساختگیرانه تر

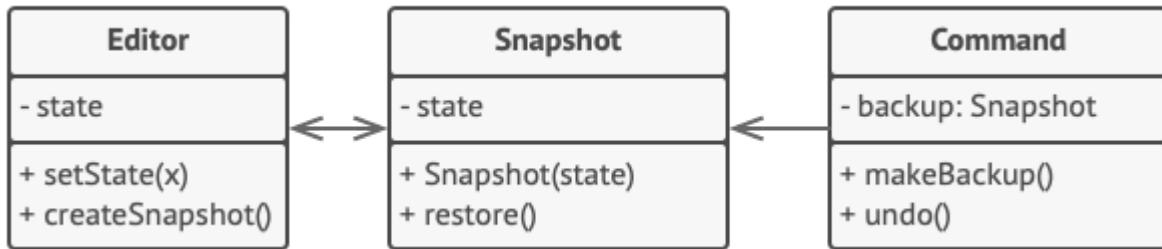
پیاده‌سازی دیگری وجود دارد که زمانی مفید است که نمی‌خواهید حتی کوچکترین شانسی را برای دسترسی کلاس‌های دیگر به وضعیت memento از طریق Originator باقی بگذارد.



- این پیاده‌سازی این اجازه را می‌دهد تا بتوانیم انواع مختلفی از originator و memento داشته باشیم. هر originator با کلاس memento مربوط به خودش کار می‌کند و هیچ‌کدام از آنها وضعیت خودشان را در دید دیگران قرار نمی‌دهند.
- Caretaker ها هم به وضوح دچار محدودیت‌های بیشتری برای ذخیره‌سازی وضعیت memento ها شده‌اند. علاوه بر این caretaker کاملاً از originator مستقل می‌شود چرا که متدهای بازیابی هم اکنون در memento پیاده‌سازی می‌شوند.
- هر کجا از memento ها فقط با originator ای که آنها را ایجاد کرده است در ارتباطند. Originator خودش را به همراه مقادیر فیلدهایش و وضعیتی که اکنون دارد به سازنده‌ی memento پاس می‌دهد. به دلیل اینکه این دو باهم خیلی ارتباط نزدیکی دارند یک memento می‌تواند وضعیت ای که آن را ایجاد کرده است را بازیابی کند.

## مثال

در این مثال از الگوی طراحی memento به همراه الگوی طراحی استفاده می‌کنیم تا snapshot های مختلفی از وضعیت یک editor ثبت کنیم و هر زمان که نیاز بود در سریع ترین زمان ممکن این snapshot ها را بازیابی کنیم.



در اینجا اشیاء command همانند caretaker عمل می‌کنند. قبل از اینکه عملیات مربوط به هر دستوری اجرا شود، های memento را واکشی می‌کنند. زمانی که کاربر تصمیم به برگرداندن عملیات بگیرد، می‌تواند از command هایی که در memento مربوطه ذخیره شده‌اند استفاده کند تا به وضعیت قبلی خودش بازگردد.

کلاس memento هیچ فیلد public و getter , setter در خود تعریف نمی‌کند. بنابراین هیچ Object ای نمی‌تواند محتوای آن را تغییر دهد. Memento به اشیاء editor ای که آن را ایجاد کرده است متصل است. این اتصال باعث می‌شود تا memento بتواند وضعیت editor ای که آن را ایجاد کرده است را از طریق هایش (editor های setter) ذخیره کند.

از آنجایی که هر memento به یک editor خاصی متصل است می‌توانید کاری کنید تا برنامه شما از چندین پنجره‌ی editor که کاملاً مستقل از همیگر هستند با یک پشتیبانی بازگردانی تاریخچه‌ی متمرکز پشتیبانی کند.

```

// The originator holds some important data that may change over
// time. It also defines a method for saving its state inside a
// memento and another method for restoring the state from it.
class Editor is
    private field text, curX, curY, selectionWidth

    method setText(text) is
        this.text = text

    method setCursor(x, y) is
        this.curX = x
        this.curY = y

    method setSelectionWidth(width) is
        this.selectionWidth = width
  
```

```

// Saves the current state inside a memento.
method createSnapshot():Snapshot is
    // Memento is an immutable object; that's why the
    // originator passes its state to the memento's
    // constructor parameters.
    return new Snapshot(this, text, curX, curY, selectionWidth)

// The memento class stores the past state of the editor.
class Snapshot is
    private field editor: Editor
    private field text, curX, curY, selectionWidth

    constructor Snapshot(editor, text, curX, curY, selectionWidth)
is
    this.editor = editor
    this.text = text
    this.curX = x
    this.curY = y
    this.selectionWidth = selectionWidth

    // At some point, a previous state of the editor can be
    // restored using a memento object.
    method restore() is
        editor.setText(text)
        editor.setCursor(curX, curY)
        editor.setSelectionWidth(selectionWidth)

// A command object can act as a caretaker. In that case, the
// command gets a memento just before it changes the
// originator's state. When undo is requested, it restores the
// originator's state from a memento.
class Command is
    private field backup: Snapshot

    method makeBackup() is
        backup = editor.createSnapshot()

    method undo() is
        if (backup != null)
            backup.restore()
        // ...

```

## چه زمانی باید از این الگو استفاده کنیم؟

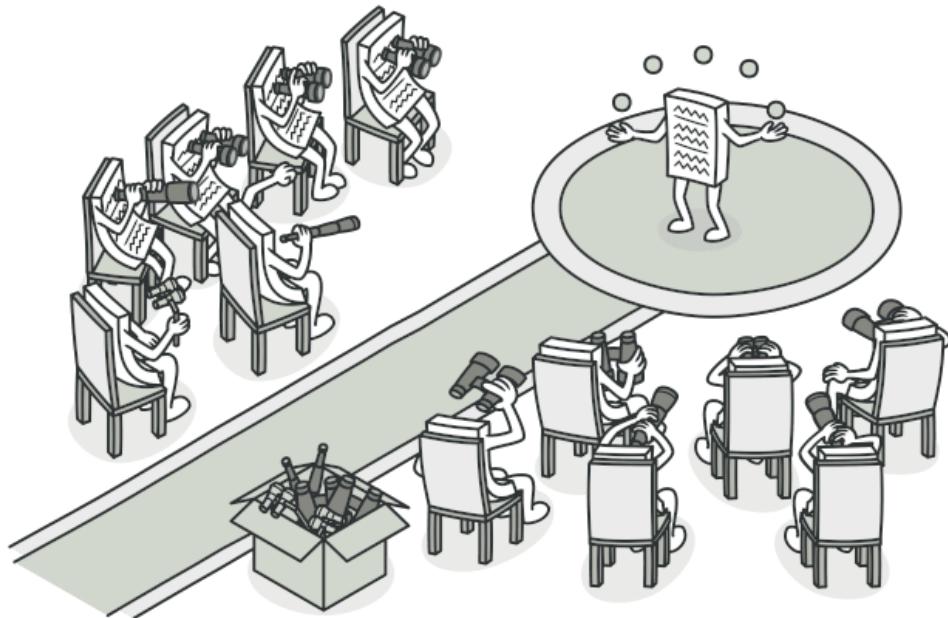
- از این الگو زمانی استفاده کنید که می‌خواهید از هر لحظه از وضعیت Object تان یک تاریخچه داشته باشید و آن را به وضعیت قبلی خودش برگردانید.
- الگوی طراحی memento این اجازه را می‌دهد تا بتوانیم یک کپی کامل از وضعیت Object مان حتی با فیلدهای private شان بگیریم و آن‌ها را مستقل از خود Object اصلی ذخیره کنیم. در حالی که بیشتر برنامه‌نویس‌ها این الگو رو با قابلیت "Undo" به یاد می‌آورند، این الگو در هنگامی که با یک سری تراکنش سر و کار داریم هم خیلی به کارمان می‌آید (به طور مثال هر زمان که با خطایی مواجه شدیم عملیات به عقب برگردانده شود).
- زمانی از این الگو استفاده کنید که دسترسی مستقیم به فیلدها، **getter** و **setter** های یک Object باعث نقض حریم خصوصی آنها می‌شود.
- الگوی memento وظیفه‌ی ذخیره‌سازی وضعیت یک Object را فقط به خودش محول می‌کند. هیچ دیگری اجازه‌ی گرفتن snapshot را ندارد و این باعث می‌شود تا وضعیت داده‌های یک Object در نقطه‌ی امنی قرار بگیرد.

## معایب و مزایا

- ❖ بدون اینکه حریم خصوصی یک Object را نقض کنید می‌توانید از آن snapshot تهیه کنید.
- ❖ می‌توانید کدهای Originator را ساده‌تر کنید چرا که وظیفه‌ی مدیریت تاریخچه‌ی آن را برعهده دارد.
- اگر کلاینت همیشه یک memento ایجاد کند ممکن است باعث بیش از حد مصرف شدن RAM شود.
- CareTaker ممکن است که چرخه‌ی یک Originator را دنبال کند و یک memento را از بین ببرد.
- بیشتر زبان‌های برنامه‌نویسی پویا، مثل PHP, Python و JavaScript نمی‌توانند تضمین کنند که وضعیتی که در یک memento نگهداری می‌شود همیشه دست نخورده باقی بماند.

## Observer

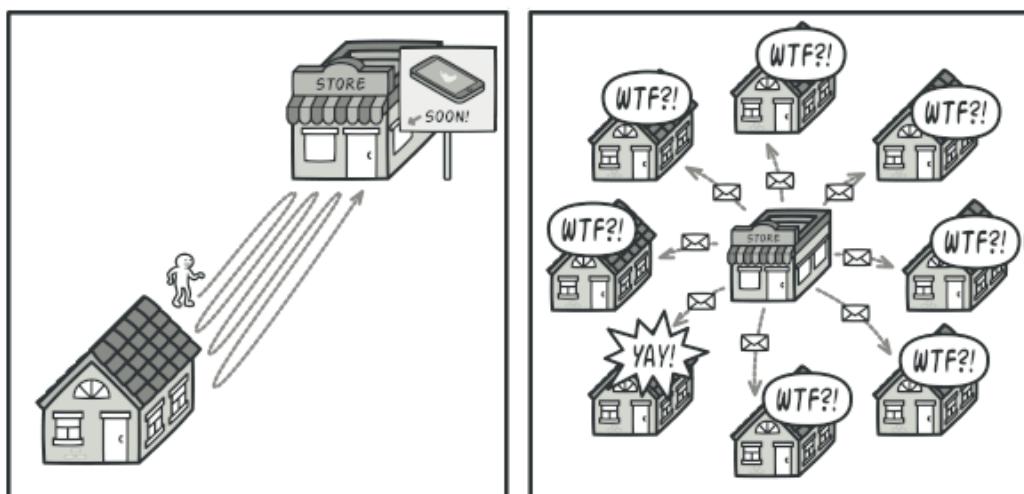
با استفاده از الگوی طراحی Observer می‌توانیم یک محیط اشتراکی برای Object هایمان ایجاد کنیم تا هر ای که در این محیط وجود دارد از اطلاعات و رویدادهای جدیدی که رخ می‌دهد با خبر شود.



### طرح مسئله

فرض کنید دو نوع Object یکی از جنس Customer و دیگری هم از نوع Store دارید. مشتری علاقه‌ی شدیدی به یک برندهای خاصی از یک محصول دارد که به زودی قرار است در فروشگاه موجود شود. به طور مثال نسخه‌ی جدید iPhone.

مشتری می‌تواند هر روز فروشگاه را چک کند و ببیند که این کالا موجود شده است یا نه. اما زمانی که این محصول هنوز در راه است و موجود نشده تمام این چک کردن‌ها کاری بیهوده بوده است.



از طرفی دیگر هم فروشگاه می‌تواند تعداد زیادی ایمیل برای همه مشتریان بفرستد (ممکن است به عنوان spam در نظر گرفته شود) که کالای مورد نظرشان موجود شده است. این از مراجعه‌ی بیهوده مشتریان به فروشگاه جلوگیری می‌کند. اما از طرفی هم باید آن مشتریانی که علاقه‌ای به آن محصول ندارند را هم در نظر بگیریم.

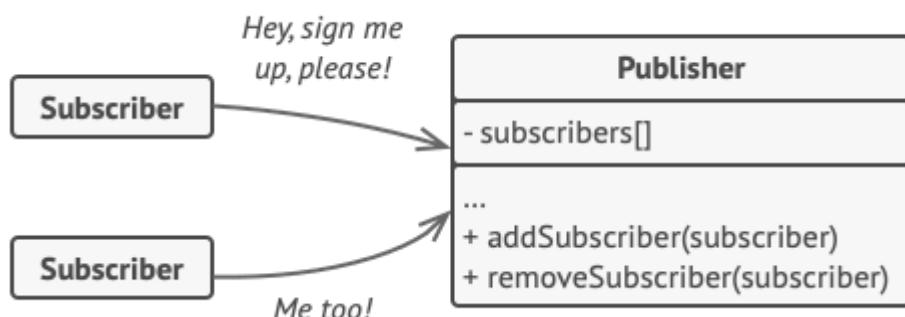
اینجور که پیداست دچار تداخل شده ایم. اینکه مشتری زمانش را با مراجعه‌های بیهوده هدر دهد یا اینکه فروشگاه منابع خودش را الکی صرف اطلاع‌رسانی به همه مشتری‌ها کند؟

## راه حل

به طور کلی زمانی که علاقه داریم از تغییر وضعیت یک شیء با خبر شویم، به آن شیء Subject می‌گوییم. اما در این مثال چون شیءمان می‌خواهد تغییر وضعیتش را با بقیه به اشتراک بگذارد و به آنها اطلاع دهد به آن subscribers publisher می‌گوییم و به بقیه اشیائی که می‌خواهند تغییرات Publisher را دنبال کنند می‌گوییم.

الگوی طراحی Observer پیشنهاد می‌کند تا یک مکانیسم تهیه اشتراک برای Object هایمان ایجاد کنیم تا بقیه اشیائی که علاقه به دنبال کردن شیء مورد نظر دارند با ایجاد یک اشتراک بتوانند این کار را به راحتی انجام دهند و به صورت لحظه‌ای از تغییراتی که در publisher رخ می‌دهد آگاه شوند و به همین روال هم می‌توانند اشتراک خود را حذف کنند تا دیگر این اطلاعات را دریافت نکنند.

نترسید! همه چیز اینقدر که پیچیده به نظر می‌آید نیست. این مکانیسم فقط شامل یک آرایه از رفرنس‌های اشیاء دیگر و یک سری متدهای مدیریت این آرایه می‌باشد. متدهایی مانند حذف و اضافه کردن رفرنس‌ها.

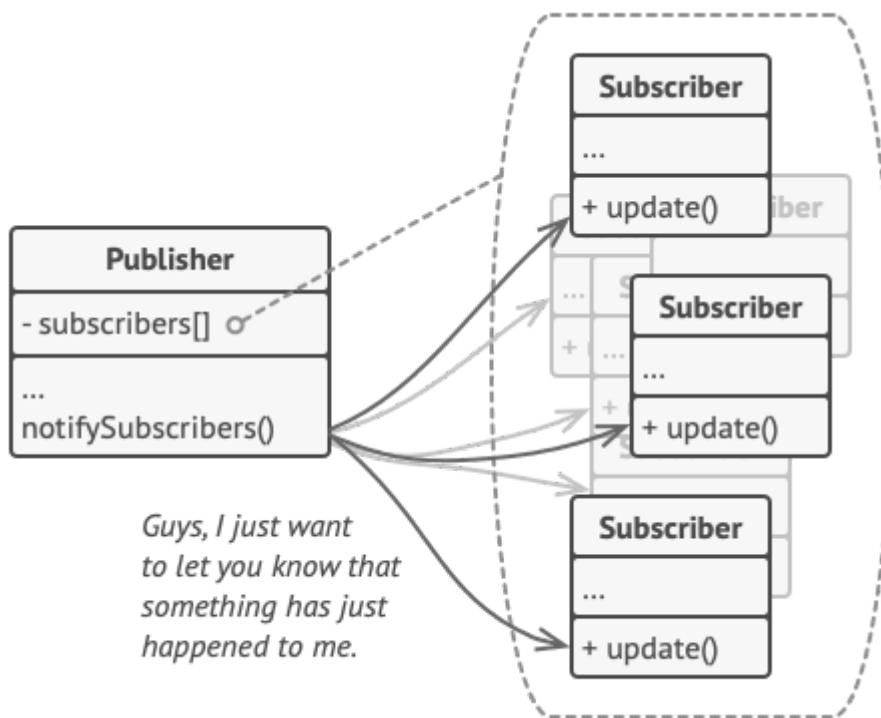


با این کار هر زمان که اتفاق مهمی برای publisher بیفتند تمامی اشیائی که به آنها رفرنس ایجاد شده است هم آگاه می‌شوند.

برنامه‌های واقعی ممکن است که دهها کلاس برای ایجاد اشتراک داشته باشند تا هر کدام از این‌ها یک رویداد از کلاس publisher را ردیابی کنند. شما نمی‌خواهید که کلاس Publisher به همه این کلاس‌ها

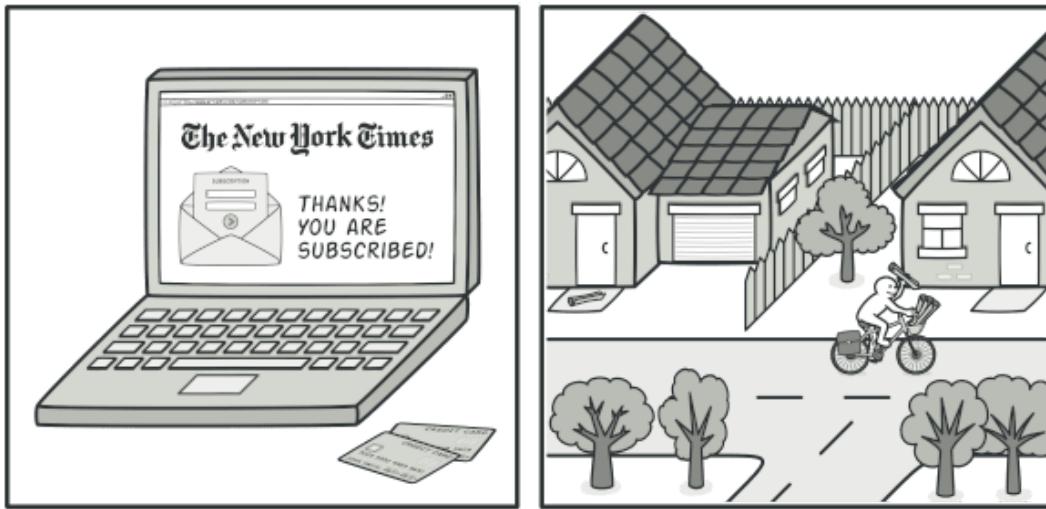
متصل شود چرا که این کار باعث ارتباط زیاد (coupling) بین کلاس‌ها می‌شود و حتی ممکن است از برخی از این کلاس‌ها هنگام نوشتمن کد Publisher خود اطلاعی نداشته باشید، به خصوص اگر کلاس Publisher برای استفاده‌ی دیگران طراحی شده باشد.

به همین خاطر این موضوع مهم است که تمامی subscriber‌ها یک Interface مشترک را پیاده‌سازی کنند و کلاس publisher با استفاده از این Interface با آنها در ارتباط باشد. این Interface باید یک متده‌ی update() باشد که این متده‌ی یک سری پارامترها را به عنوان ورودی خود دریافت می‌کند. اطلاع رسانی در خود داشته باشد که این متده‌ی update() را به عنوان ورودی خود را ارسال می‌کند. Publisher با استفاده از این متده‌ی update() از داده‌های متنی خود را ارسال می‌کند.



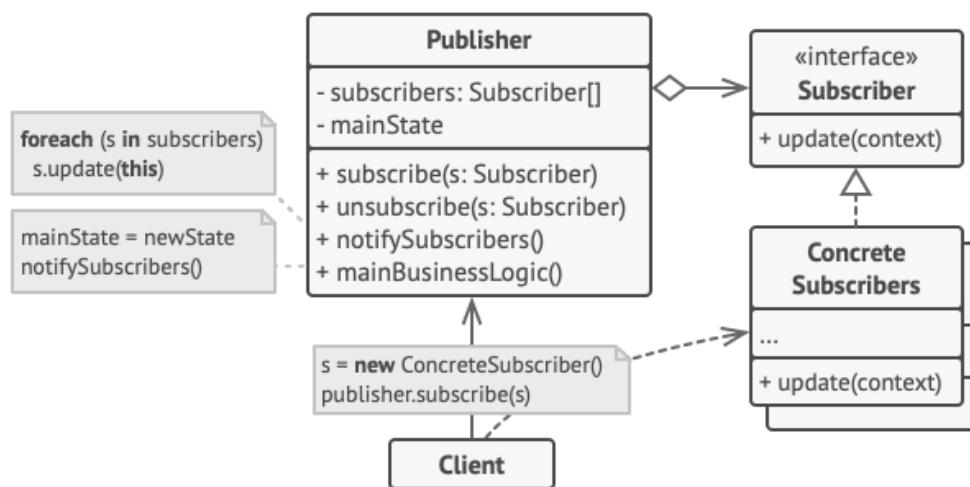
اگر برنامه‌تان تعداد زیادی publisher و می‌خواهید تمام subscriber‌ها را با آنها همگام کنید، می‌توانید کاری کنید تا تمام publisher‌هایتان یک interface کلی را پیاده‌سازی کنند. این فقط کافی است تعداد کمی از متدهای ایجاد اشتراک را در خودش تعریف کند. این Interface به subscriber‌ها این اجازه را می‌دهد تا بتوانند بدون اینکه به publisher وابسته شوند وضعیت آن را ردیابی کنند.

## مقایسه با دنیای واقعی



اگر شما اشتراک یک روزنامه یا مجله را بخرید، دیگر لازم نیست به یک فروشگاه بروید تا از جدید ترین خبرها آگاه شوید. به جای آن ناشر پس از انتشار خبر خودش آنها را به شما ایمیل می‌کند. ناشر لیستی از مشترکین را در پیش خودش دارد و می‌داند که هر کدام از مشترکین به چه چیزهایی علاقه دارند. مشترکین نیز هر زمان که علاقه‌ای به دنبال کردن آن موضوع نداشته باشند می‌توانند اشتراکشان را لغو کنند تا دیگر خبری در رابطه با آن موضوع برایشان ارسال نشود.

## ساختار



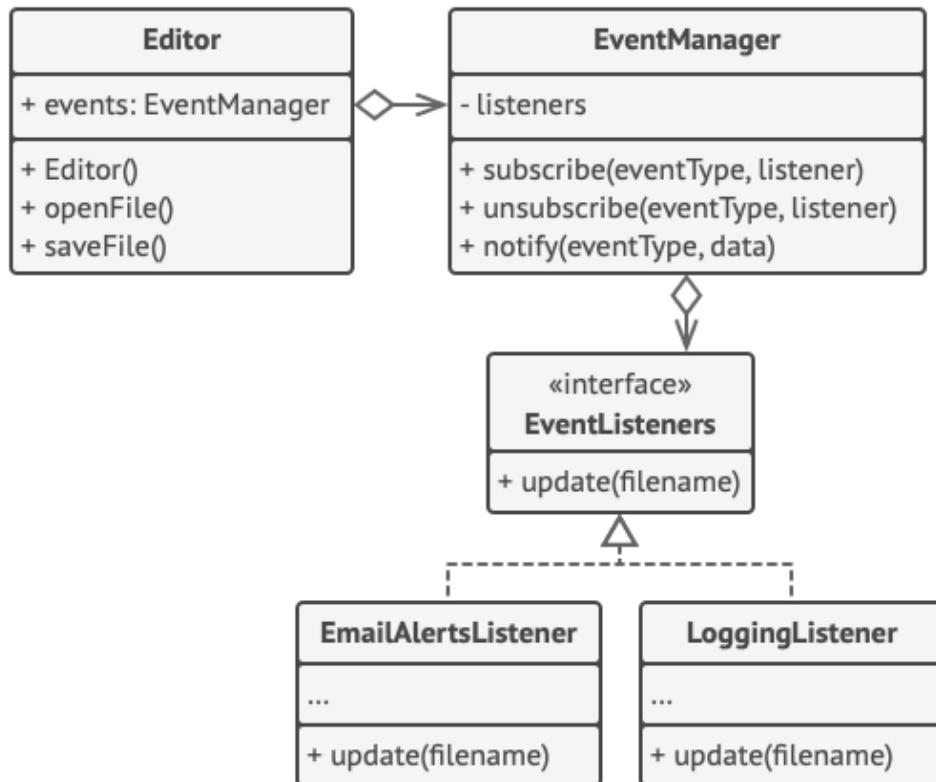
• **Publisher**: این کلاس رویدادهای مورد علاقه‌ی بقیه‌ی اشیاء را منتشر می‌کند. این رویدادها زمانی رخ می‌دهند که publisher وضعیت خودش را تغییر دهد و یا برخی از رفتارها را انجام دهد.

Publisher ها زیرساختهایی برای اضافه کردن یک مشترک جدید و یا حذف مشترکین قبلی در خود دارند.

- زمانی که یک اتفاق جدید رخ می‌دهد، publisher در بین لیست مشترکین می‌گردد و متند ای که در واسط subscriber تعریف کرده است را فراخوانی می‌کند.
- **Subscriber**: این واسط، واسط، notification را در خودش تعریف می‌کند. در بیشتر مواقع این واسط فقط یک متند تحت عنوان update را در خودش دارد. این متند می‌تواند چندین پارامتر داشته باشد تا publisher بتواند جزئیات فرایندهای خودش را از این طریق به مشترکین ارسال کند.
- **Concrete Subscribers**: کارهایی که باید در مقابله با notification ها انجام شوند در این کلاس‌ها پیاده‌سازی می‌شوند. تمام این کلاس‌ها باید یک Interface کلی را پیاده‌سازی کنند تا فقط به یک subscriber publisher وابسته نباشد.
- معمولاً subscriber ها نیاز به یک سری اطلاعات کامل‌تری دارند تا بتوانند متند update را بهتر پیاده‌سازی کنند. به همین دلیل publisher ها یک سری داده‌های متنی را هم به عنوان پارامتر می‌فرستند. در ضمن publisher می‌تواند خودش را هم به عنوان آرگومان متند بفرستد که در این صورت subscriber ها به همه‌ی اطلاعات به صورت مستقیم دسترسی دارند.
- **Client**: کلاینت Object های publisher و subscriber را به صورت مستقل می‌سازد و سپس subscriber ها را آماده‌ی هر نوع بروزرسانی از سمت publisher می‌کند.

## مثال

در این مثال الگوی طراحی observer این امکان را فراهم می‌کند تا object ویرایشگر متغیر بتواند وضعیت فعلی خود را به دیگر سرویس‌ها خبر دهد.



لیست مشترکین به صورت پویا جمع‌آوری می‌شود. بسته به رفتار مورد نظری که برنامه دارد، Object ها می‌توانند از زمان آغاز برنامه شروع به گوش دادن به اعلان‌ها کنند و یا گوش دادن به آنها را متوقف کنند. در این پیاده‌سازی کلاس `editor` خودش وظیفه‌ی مدیریت لیست `subscriber` ها را برعهده ندارد. این کار را به یک شیء کمک‌کننده‌ی ویژه که تنها به این کار اختصاص دارد واگذار کرده است. می‌توانید این شیء را به عنوان یک میانجی مرکزی برای ارسال رویدادها ارتقاء دهید تا هر شیء دیگری هم به عنوان یک ناشر (publisher) عمل کنند.

تا زمانی که که تمامی `subscriber` ها یک `Interface` را پیاده‌سازی می‌کنند اضافه که کردن هر `subscriber` به برنامه کار بسیار راحتی است.

```

// The base publisher class includes subscription management
// code and notification methods.

class EventManager is
    private field listeners: hash map of event types and listeners

    method subscribe(eventType, listener) is
  
```

```

        listeners.add(eventType, listener)

    method unsubscribe(eventType, listener) is
        listeners.remove(eventType, listener)

    method notify(eventType, data) is
        foreach (listener in listeners.of(eventType)) do
            listener.update(data)

// The concrete publisher contains real business logic that's
// interesting for some subscribers. We could derive this class
// from the base publisher, but that isn't always possible in
// real life because the concrete publisher might already be a
// subclass. In this case, you can patch the subscription logic
// in with composition, as we did here.

class Editor is
    public field events: EventManager
    private field file: File

    constructor Editor() is
        events = new EventManager()

    // Methods of business logic can notify subscribers about
    // changes.
    method openFile(path) is
        this.file = new File(path)
        events.notify("open", file.name)

    method saveFile() is
        file.write()
        events.notify("save", file.name)

    // ...

// Here's the subscriber interface. If your programming language
// supports functional types, you can replace the whole
// subscriber hierarchy with a set of functions.

interface EventListener is
    method update(filename)

// Concrete subscribers react to updates issued by the publisher
// they are attached to.
class LoggingListener implements EventListener is
    private field log: File
    private field message: string

    constructor LoggingListener(log_filename, message) is

```

```

    this.log = new File(log_filename)
    this.message = message

    method update(filename) is
        log.write(replace('%s', filename, message))

class EmailAlertsListener implements EventListener is
    private field email: string
    private field message: string

    constructor EmailAlertsListener(email, message) is
        this.email = email
        this.message = message

    method update(filename) is
        system.email(email, replace('%s', filename, message))

// An application can configure publishers and subscribers at
// runtime.
class Application is
    method config() is
        editor = new Editor()

        logger = new LoggingListener(
            "/path/to/log.txt",
            "Someone has opened the file: %s")
        editor.events.subscribe("open", logger)

        emailAlerts = new EmailAlertsListener(
            "admin@example.com",
            "Someone has changed the file: %s")
        editor.events.subscribe("save", emailAlerts)

```

## چه زمانی باید از این الگو استفاده کنیم؟

- زمانی از این الگو استفاده کنید که تغییر وضعیت یک Object باعث تغییر دیگری که

نمی‌شناسید و نمی‌دانید کی ایجاد می‌شود هم بشود.

عموماً با این مشکل بیشتر در زمانی که با کلاس‌های یک رابط کاربری کار می‌کنید مواجه می‌شوید. به طور مثال یک سری کلاس‌های سفارشی شده از دکمه ایجاد کرده اید و می‌خواهید بعد از فشردن این دکمه‌ها یک سری کارهای سفارشی شده‌ی دیگر هم انجام شود.

الگوی طراحی Observer این اجازه را می‌دهد تا هر Object ای که بواسطه subscriber را پیاده‌سازی کرده باشد بتواند به عنوان یک subscriber منتظر شنیدن اعلان‌های یک publisher باشد.

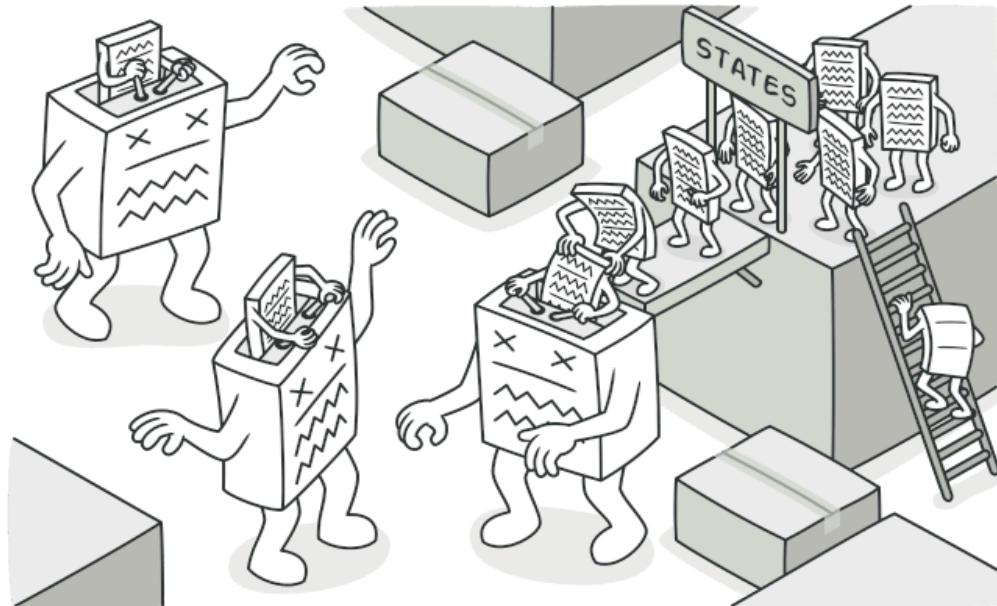
- از این الگو زمانی استفاده کنید که بعضی از اشیاء موجود در برنامه‌تان باید به یک سری از Object‌ها نگاه کنند و وضعیت آنها را به مدت محدودی یا در موارد خاصی ردیابی کنند.
- لیست subscriber‌ها پویاست و هر زمان که نیاز باشد می‌توان یک subscriber با آن اضافه و یا از آن کم کرد.

### معایب و مزایا

- ❖ با استفاده از این الگو اصل SOLID از اصول Open/Closed رعایت می‌شود چرا که می‌توانید در هر لحظه یک Subscriber به لیست تان اضافه کنید بدون اینکه کدهای publisher قسمت را تغییری دهید(و همینطور بالعکس).
  - ❖ می‌توانید یک سری از روابط را در زمان اجرا برقرار کنید.
- اطلاع رسانی به subscriber‌ها به ترتیب نیست.

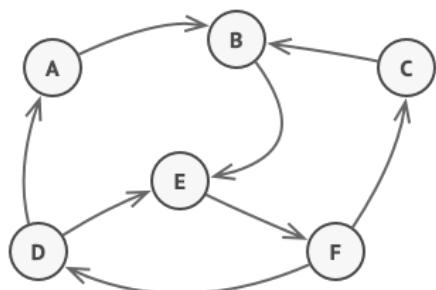
## State

با استفاده از این الگو Object ها می‌توانند در صورتی که وضعیت درونی‌شان تغییر کند رفتارشان را هم تغییر دهند. این کار مانند این است که یک شیء انگار کلاس خود را عوض کرده است.



### طرح مسئله

این الگو ارتباط بسیار نزدیکی با مفهوم Finite-State Machine دارد. اگر در رابطه با این مفهوم اطلاعاتی ندارید به طور خلاصه FSM به ما کمک می‌کند تا رفتار یک سیستم را در مقابل ورودی‌ها مدل کنیم. به عبارت ساده‌تر، ماشین حالت محدود ما مشخص می‌کند که در چه وضعیت‌هایی می‌توانیم با چه ورودی‌هایی رفتار کنیم و چه تغییراتی در وضعیت داریم.



ایده‌ی اصلی این است که در هر لحظه تعداد محدودی از state ها هستند که یک برنامه می‌تواند در آن باشد. در هر state یکتایی برنامه متفاوت عمل می‌کند و می‌تواند در جا بین این وضعیت‌ها جابجا شود و یا با توجه به وضعیت فعلی‌ای که برنامه در آن قرار دارد می‌تواند جابجا نشود. به این قوانین جابجایی transitions می‌گوییم که محدود و از پیش تعیین شده هستند.

شما همچنین می‌توانید این رویکرد را برای Object‌ها نیز در نظر بگیرید. تصور کنید که کلاسی به نام Document داریم. یک Document می‌تواند در هر کدام از وضعیت‌های Draft (پیش‌نویس)، Moderation (در حال نوشتمن) و یا انتشار یافته باشد. متدهای publish در هر کدام از این وضعیت‌ها کمی متفاوت است.

- در حالت Draft، متدهای publish بعد از فراخوانی document را به حالت moderation می‌برد.
- در حالت moderation متدهای publish بعد از فراخوانی document را در صورتی که کاربر نقش داشته باشد به حالت public در می‌آورد.
- در حالت moderation این متدهای هیچ کار انجام نمی‌دهد.



هموچنین معمولاً با عبارات شرطی زیادی مانند if و switch پیاده‌سازی می‌شوند تا بسته به وضعیت فعلی برنامه رفتار مناسب را انتخاب کنند. معمولاً هم این state‌ها مجموعه‌ای از مقادیر فیلدهای Object‌ها هستند. حتی اگر قبل از اینجا به FSM‌ها چیزی نشنیده باشید، احتمالاً برای یک بار هم که شده را پیاده‌سازی کرده اید. آیا ساختار کد زیر برای شما آشنا نیست؟؟

```

class Document {
  field state: string
  // ...
  method publish() {
    switch (state) {
      case "draft":
        state = "moderation"
        break
    }
  }
}
  
```

```

    "moderation":
      if (currentUser.role == "admin")
        state = "published"
      break
    "published":
      // Do nothing.
      break
    ...
  
```

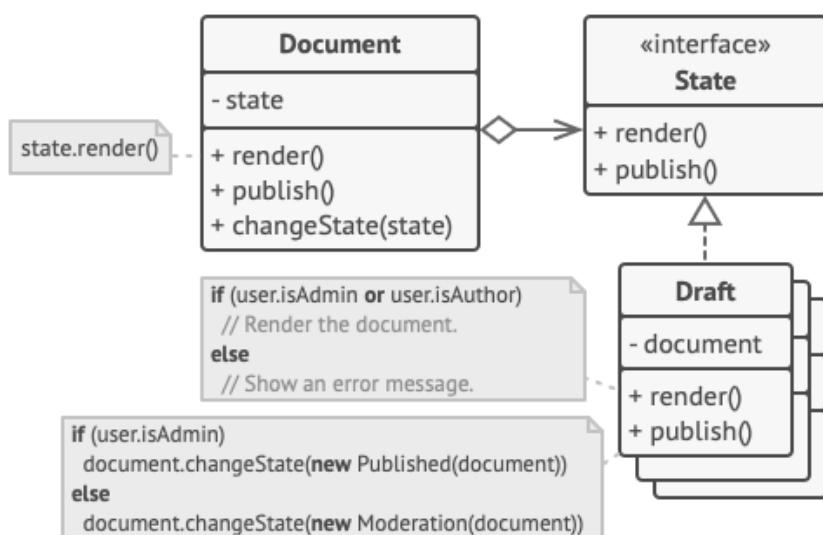
بزرگترین ضعف یک ماشین حالت زمانی خودش را نشان می‌دهد که شروع به اضافه کردن حالت‌های بیشتر که به کلاس Document مربوط هستند کنیم.

بیشتر متدها شامل شرط‌های غول پیکر می‌شوند تا رفتار متدها بر اساس وضعیت فعلی را تشکیل دهند. نگهداری از چنین کدی بسیار سخت و دشوار است چرا که هر گونه تغییری در منطق transition نیازمند تغییر شرط‌های هر state در متدها می‌باشد.

این مشکل حتی در ابعاد بزرگتری در پروژه هم می‌تواند بسط پیدا کند. پیش‌بینی همه حالت‌ها و transition‌های ممکن در مرحله طراحی بسیار دشوار است. از این رو، یک ماشین حالت خوب که با مجموعه‌ای محدود از شرط‌ها ساخته شده است، می‌تواند در طول زمان به یک آشفتگی بزرگ تبدیل شود. اما راه حل چیست؟

## راه حل

الگوی طراحی state پیشنهاد می‌کند که برای هر کدام از state‌های موجود در برنامه یک کلاس جدا ایجاد کنید و تمامی فرایندهای مربوط به این state‌ها را در این کلاس‌ها کنترل کنید. به جای اینکه تمامی رفتارها را در کلاس Document پیاده‌سازی کنید، هر کدام از حالات خودشان در یک کلاس جدا هندل شوند.

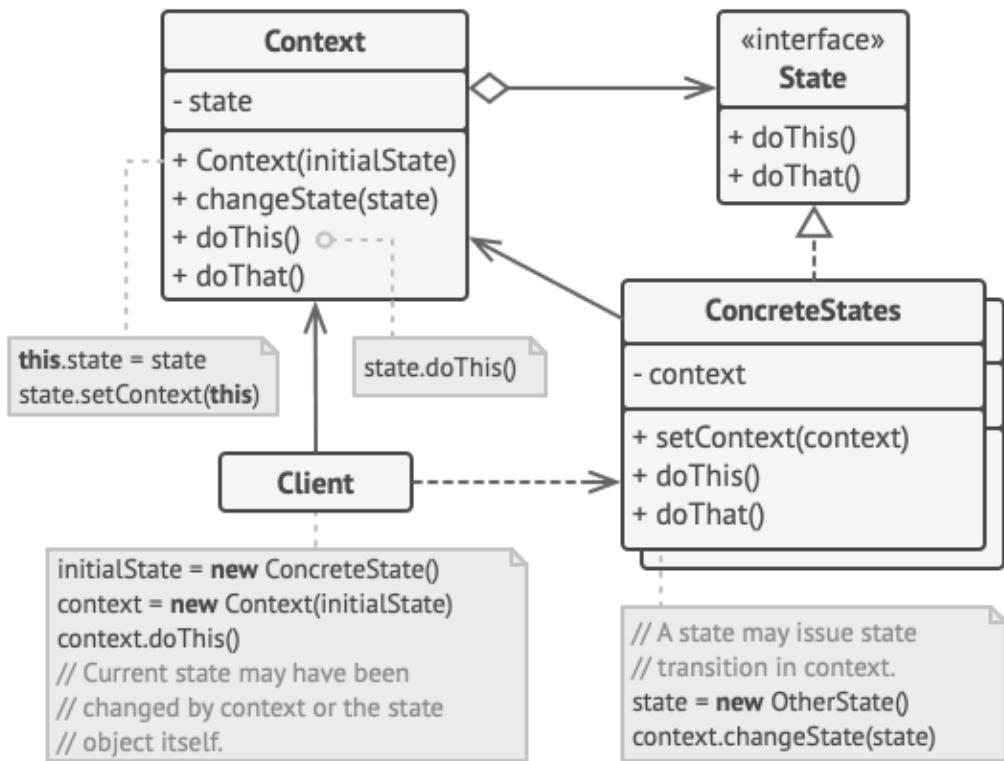


برای اینکه از وضعیتی به وضعیت دیگر بروید کافی است شیء state فعلی را با شیء state ای که میخواهیم با آن جایجا شویم تعویض کنیم. این در صورتی امکان پذیر است که تمامی state ها از یک Interface تبعیت کنند و کلاس اصلیمان با استفاده از این واسط با آنها ارتباط برقرار کند. این فرایند شاید خیلی شبیه به الگوی طراحی strategy باشد که در آینده راجع به آن صحبت خواهیم کرد. اما یک تفاوت اصلی وجود دارد اینکه در الگوی state ممکن است یک سری از state های خاص از وجود هم آگاه باشند و از حالتی به حالت دیگر منتقل شوند اما در الگوی strategy این فرایند وجود ندارد و ها هرگز از وجود هم آگاه نیستند.

### مقایسه با دنیای واقعی

دکمه‌ها و سوئیچ‌های روی تلفن همراه شما هر کدام بسته به حالت موبایل‌تان رفتارهای متفاوتی از خود نشان می‌دهند :

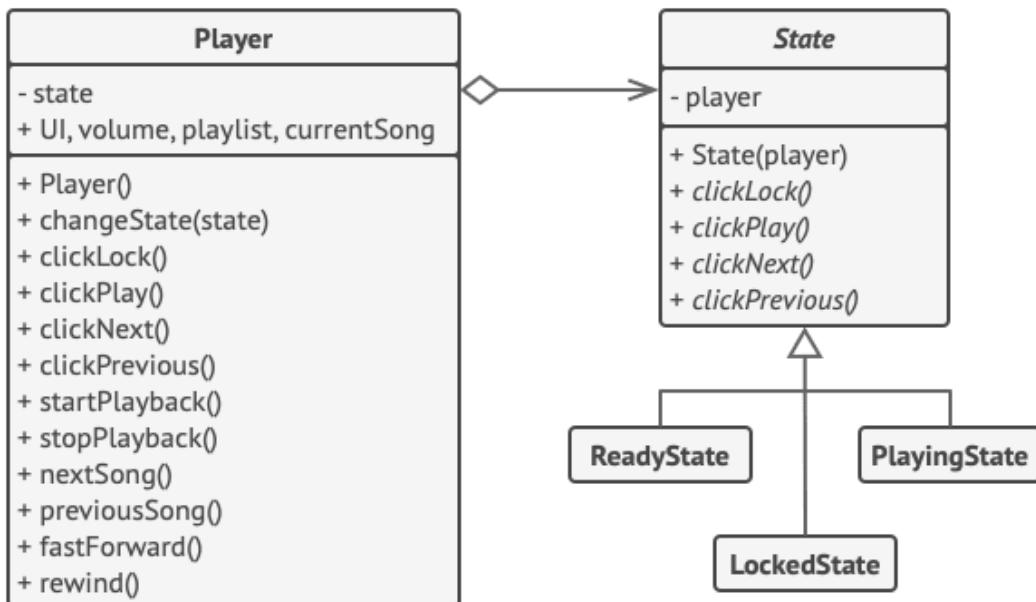
- زمانی که موبایل قفل نیست، فشار دادن دکمه‌ها فانکشنالیتی‌های متفاوتی دارند تا یک فرایندی را اجرا کنند.
- زمانی که موبایل قفل است، فشار دادن هر دکمه‌ای به منظور باز کردن قفل گوشی می‌باشد.
- زمانی که موبایل شارژ ندارد و خاموش است، بعد از اینکه شارژر را متصل می‌کنید فشار داد هر دکمه‌ای باعث نمایش میزان شارژ دستگاه می‌شود.



- **Context**: یک رفرنس به Object های state که در زیر شاخه قرار دارند نگه می‌دارد . به نوعی نماینده‌ی این state ها می‌باشد. Context با استفاده از یک interface کلی با state ها در ارتباط است. یک setter در خود دارد تا بتواند state را در آن set کند. این
- **State**: این واسط متدهای عمومی که هر state باید داشته باشد را در خودش تعریف می‌کند. این متدها باید برای همه‌ی state ها قابل فهم باشند چرا که نمی‌خواهیم state هایمان یک سری متدهایی داشته باشند که هرگز از آنها استفاده نمی‌کنند.
- **Concrete State**: متدهای مختص هر state را در خودشان پیاده‌سازی می‌کنند. اگر بخواهیم کد تکراری را کاهش دهیم و قسمتی از رفتارها را در چندین state به اشتراک بگذاریم، از یک کلاس انتزاعی میانی یا همان "intermediate abstract class" استفاده می‌کنیم. این کلاس انتزاعی میانی می‌تواند بخشی از رفتارهای مشترک بین چندین state را در خود نگه دارد.
- **Context**, state های Object ممکن است رفرنسی به شیء context را در خود نگه دارند. با این کار یک state می‌تواند هر نوع اطلاعاتی که از یک context نیاز دارد را بدست بیاورد.
- هر دوی context و concrete state می‌توانند state بعدی context را سیت کنند و با جایگزین کردن یک state مرتبط با context یک transition انجام دهند.

## مثال

در این مثال الگوی طراحی state این اجازه را می‌دهد تا کنترل کننده‌های پخش media بسته به وضعیت فعلی پخش کننده، رفتارهای متفاوتی داشته باشند.



اصلی Object مان همیشه به یک state از Object وصل است که این وضعیت بیشتر فعالیت‌های player را نشان می‌دهد. بعضی از فعالیت‌ها با state فعلی برنامه عوض می‌شوند تا player بتواند در مقابله با فعالیت‌های کاربر عملکرد مناسبی داشته باشد.

```

// The AudioPlayer class acts as a context. It also maintains a
// reference to an instance of one of the state classes that
// represents the current state of the audio player.
class AudioPlayer is
    field state: State
    field UI, volume, playlist, currentSong

    constructor AudioPlayer() is
        this.state = new ReadyState(this)

        // Context delegates handling user input to a state
        // object. Naturally, the outcome depends on what state
        // is currently active, since each state can handle the
        // input differently.
        UI = new UserInterface()
        UI.lockButton.onClick(this.clickLock)
        UI.playButton.onClick(this.clickPlay)
        UI.nextButton.onClick(this.clickNext)
  
```

```

    UI.prevButton.onClick(this.clickPrevious)

    // Other objects must be able to switch the audio player's
    // active state.
    method changeState(state: State) is
        this.state = state

    // UI methods delegate execution to the active state.
    method clickLock() is
        state.clickLock()
    method clickPlay() is
        state.clickPlay()
    method clickNext() is
        state.clickNext()
    method clickPrevious() is
        state.clickPrevious()

    // A state may call some service methods on the context.
    method startPlayback() is
        // ...
    method stopPlayback() is
        // ...
    method nextSong() is
        // ...
    method previousSong() is
        // ...
    method fastForward(time) is
        // ...
    method rewind(time) is
        // ...

// The base state class declares methods that all concrete
// states should implement and also provides a backreference to
// the context object associated with the state. States can use
// the backreference to transition the context to another state.
abstract class State is
    protected field player: AudioPlayer

    // Context passes itself through the state constructor. This
    // may help a state fetch some useful context data if it's
    // needed.
    constructor State(player) is
        this.player = player

    abstract method clickLock()
    abstract method clickPlay()
    abstract method clickNext()

```

```

abstract method clickPrevious()

// Concrete states implement various behaviors associated with a
// state of the context.

class LockedState extends State is

    // When you unlock a locked player, it may assume one of two
    // states.

    method clickLock() is
        if (player.playing)
            player.changeState(new PlayingState(player))
        else
            player.changeState(new ReadyState(player))

    method clickPlay() is
        // Locked, so do nothing.

    method clickNext() is
        // Locked, so do nothing.

    method clickPrevious() is
        // Locked, so do nothing.

// They can also trigger state transitions in the context.

class ReadyState extends State is

    method clickLock() is
        player.changeState(new LockedState(player))

    method clickPlay() is
        player.startPlayback()
        player.changeState(new PlayingState(player))

    method clickNext() is
        player.nextSong()

    method clickPrevious() is
        player.previousSong()

class PlayingState extends State is

    method clickLock() is
        player.changeState(new LockedState(player))

    method clickPlay() is
        player.stopPlayback()
        player.changeState(new ReadyState(player))

```

```

method clickNext() is
    if (event.doubleclick)
        player.nextSong()
    else
        player.fastForward(5)

method clickPrevious() is
    if (event.doubleclick)
        player.previous()
    else
        player.rewind(5)

```

### چه زمانی باید از این الگو استفاده کنیم؟

- زمانی از این الگو استفاده کنید که Object هایتان بسته به وضعیت‌های مختلف برنامه باید عملکردهای متفاوتی از خودشان نشان دهند یا تعداد این وضعیت‌ها بسیار زیاد است و مجبور به تغییر مکرر کدها می‌شوید.

این الگو پیشنهاد می‌کند تا تمام وضعیت‌های به خصوص برنامه را در یک سری کلاس جدا از هم قرار دهید. با این کار هر زمان که به یک state جدید در برنامه‌تان نیاز پیدا کنید می‌توانید به راحتی یک کلاس از جنس آن state به برنامه‌تان اضافه کنید.

- زمانی از این الگو استفاده کنید که کلاس‌هایتان به دلیل استفاده‌ی بیش از حد از شرط‌ها کثیف شده‌اند و باعث تغییر رفتار کلاستان در وضعیت‌های مختلف شده‌اند.

این الگو پیشنهاد می‌کند تا این شرط‌ها را به متدهایی که نمایانگر وضعیت کلاس در هر لحظه هستند تبدیل کنید. با این کار می‌توانید هر زمان که به این state ها نیازی نداشتید آن را از کلاستان پاک کنید.

- زمانی از این الگو استفاده کنید که مقدار زیادی کد تکراری برای یک وضعیت دارید و به یک ماشین شرط‌گذاری تبدیل شده‌اید.

این الگو این اجازه را می‌دهد تا بتوانید یک سلسله مراتب از state های مورد نیاز بسازید و در عین حال هر کدام از این state ها به صورت مستقل عمل کنند. می‌توانید هر جا که نیاز بود از کلاس‌های پایه‌ای و abstract ای که در راس این سلسله مراتب تعریف کرده‌اید استفاده کنید و اینگونه از تکرار کد جلوگیری کنید.

### معایب و مزايا

- ❖ با استفاده از این الگو اصل SOLID از اصول Single responsibility رعایت می‌شود چرا که برای هر کدام از state ها یک کلاس جدا ایجاد می‌کند.

- ❖ با استفاده از این الگو اصل SOLID از اصول Open/Closed رعایت می‌شود چرا که می‌توانید در هر جا که نیاز بود یک state به برنامه‌تان اضافه کنید بدون اینکه در کدهای قبلی تغییری ایجاد کنید.
- ❖ با استفاده از این الگو کدهای ساده‌تر با شرط‌های کمتری خواهیم داشت.

→ زمانی که برنامه از شرط‌های کمی برخوردار باشد این الگو می‌تواند آسیب زننده باشد.

## Strategy

با استفاده از این الگو می‌توانید خانواده‌ای از الگوریتم‌ها را ایجاد کنید و هر کدام از آنها را در درون کلاس‌های جداگانه‌ای قرار دهید تا هر زمان که نیاز بود Object‌های آنها را با یکدیگر تعویض کنید.



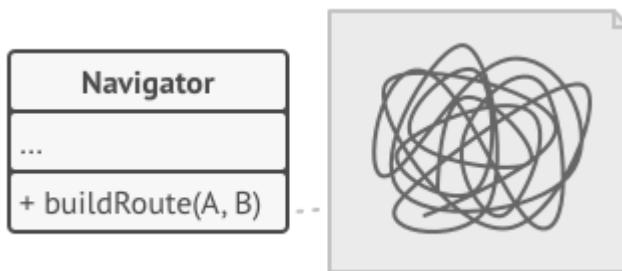
## طرح مسئله

فرض کنید یک روز تصمیم می‌گیرید تا یک برنامه‌ی مسیریابی برای مسافران بسازید. برنامه از یک نقشه‌ی زیبا برخوردار است که به کاربران این اجازه را می‌دهد تا سریع در داخل شهر جابجا شوند. یک از feature‌هایی که متقاضی زیادی دارد این است که مسیریابی اتوماتیک هم به برنامه‌تان اضافه شود. یک کاربر بتواند مقصد خود را تعیین کند و برنامه به صورت اتومات کوتاه‌ترین و سریع‌ترین مسیر تا به آنجا را بر روی نقشه نشان دهد.

نسخه‌ی اولیه‌ی برنامه فقط قادر است تا مسیرها را بر روی جاده‌های شهری داخل نقشه طراحی کند و آن را به کاربر نمایش دهد.

با این کار افرادی که با ماشین سفر می‌کردند خیلی خوشحال شدند. اما ظاهرا همه‌ی کاربران از این کار خوششان نیامد. چرا که بعضی‌ها هم دوست دارند تا پیاده در شهر بچرخند. بنابراین تصمیم می‌گیرید تا در نسخه‌ی بعدی برنامه‌ی خود گزینه‌ای برای ساخت مسیر‌های پیاده روی نیز اضافه کنید.

بلافاصله بعد از آن گزینه‌ی دیگری را اضافه کردید تا به مردم اجازه دهد از وسائل حمل و نقل عمومی در مسیرهای خود استفاده کنند. با اینکه این همه قابلیت اضافه کردید اما این تازه اول کار است. ممکن است بعداً بخواهید مسیرهایی را فقط برای رفت و آمد با دوچرخه ایجاد کنید و یا حتی بخواهید مسیرهایی برای دیدن جاذبه‌های گردشگری هر شهر ایجاد کنید.



در حالی که از نظر بیزینسی این برنامه بسیار موفقیت آمیز است، اما بخش فنی آن سردردهای زیادی را برای شما به وجود آورد. هر بار که یک الگوریتم جدید برای مسیریابی اضافه می‌کنید اندازه‌ی کلاس Navigator - تان دوبرابر می‌شود و نگهداری از آن دیگر کار هر کسی نیست!

برای هر تغییر کوچکی مانند برطرف کردن یک باگ یا تغییر مستقیم یک feature باید کل برنامه تغییر کند و ممکن است که با error‌های زیادی مواجه شوید.

علاوه بر این کار کردن بر روی یک برنامه به صورت تیمی خیلی دشوار خواهد شد. هم تیمی هایتان که بعد از انتشار نسخه‌های موفق از برنامه‌تان به تیم شما پیوسته اند، همیشه شکایت دارند که باید موارد زیادی را حین merge کردن برطرف کنند.

پیاده سازی یک feature جدید نیازمند ایجاد تغییرات بسیار زیادی در کلاس‌ها می‌باشد و این کار باعث ایجاد تداخل با دیگر برنامه‌نویسان در تیم‌تان می‌شود.

## راه حل

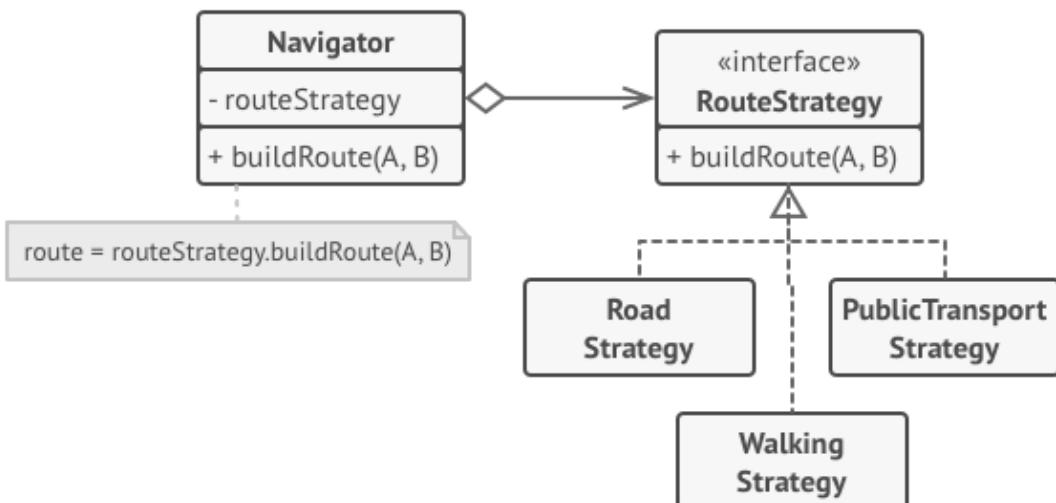
الگوی طراحی strategy پیشنهاد می‌کند یک کلاس را انتخاب کنید تا کارهای مختلف را با متدهای متفاوتی انجام دهد و در ادامه همه‌ی الگوریتم‌های مورد نیازتان را در کلاس‌های جداگانه به نام strategies قرار دهید.

کلاس اصلی که آن را context می‌نامیم، یک field برای داشتن ترتیب رفرنس‌ها به این استراتژی‌ها را در خود نگه می‌دارد.

در ادامه کلاس context به جای اینکه خودش کارها را انجام دهد کار را به الگوریتم مناسب واگذار می‌کند و خودش هیچ مسئولیتی در قبال انتخاب الگوریتم مناسب ندارد.

کلایینت نیز استراتژی درخواستی خود را به context می‌گوید اما این را باید دانست که در واقع context هیچ اطلاعی از این strategy‌ها ندارد. او با همه‌ی strategy‌ها فقط و فقط از طریق یک interface ارتباط برقرار می‌کند که این واسط معمولاً فقط یک متده‌ی تعویض الگوریتم که در strategy‌ها پیاده سازی شده است را در خود دارد.

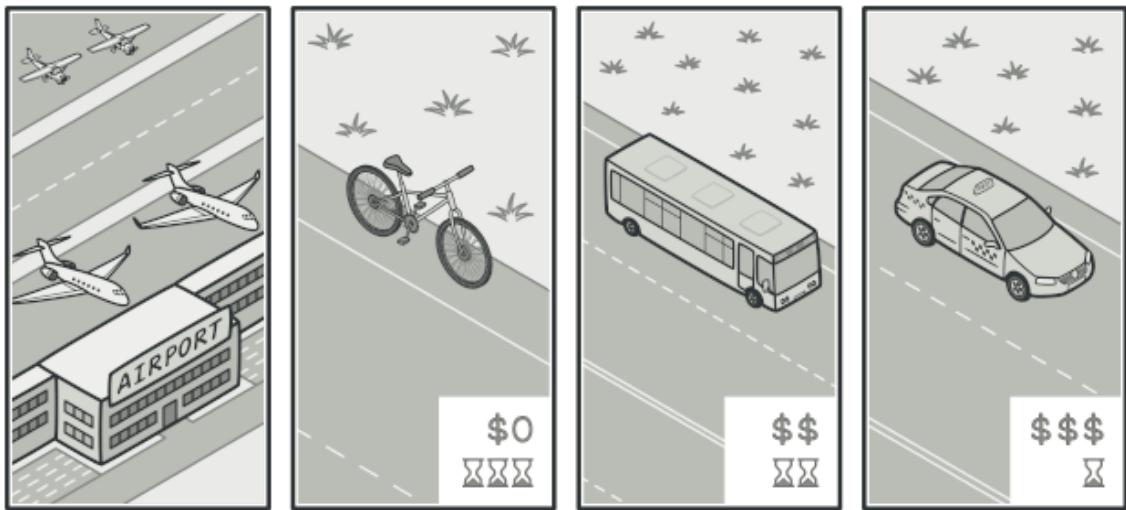
با این کار کلاس context کاملاً از strategy‌ها جدا می‌شود و هر زمان که نیاز داشته باشید می‌توانید یک الگوریتم و یا strategy جدید بدون اینکه کدهای اصلی context تان را تغییر دهید به برنامه‌تان اضافه کنید.



در برنامه‌ی مسیریابی که نوشتمیم، هر کدام از الگوریتم‌ها می‌توانند در کلاس خودشان تنها با یک متده به نام buildRoute خلاصه شوند. این متده تنها مبدأ و مقصد را به عنوان ورودی دریافت می‌کند و لیستی از مسیرها را به عنوان خروجی باز می‌گرداند. با این حال هر کدام از این کلاس‌های برای یک مبدأ و مقصد یکسان می‌توانند مسیرهای متفاوتی را پیشنهاد دهند و این موضوع هیچ اهمیتی برای مسیریاب ندارد چرا که تنها وظیفه‌ی او نشان دادن این مسیرها بر روی نقشه است.

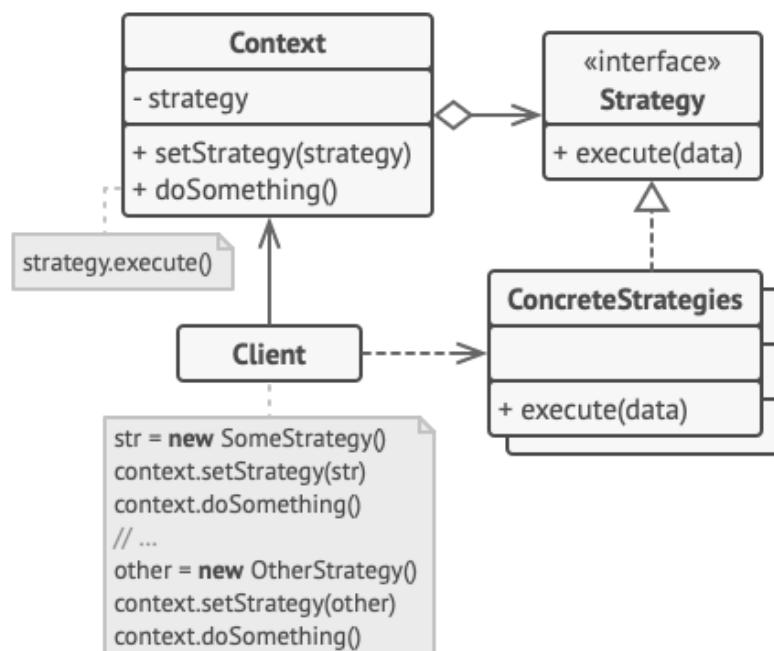
کلاسمان فقط شامل یک متده برای سوئیچ کردن بین strategy‌های فعال است بنابراین کلایینت می‌تواند تنها با فشردن دکمه‌ای که در رابط کاربری قرار داده شده است، هر کدام از مسیرهایی که خواست را انتخاب کند.

## مقایسه با دنیای واقعی



فرض کنید که باید به فرودگاه بروید. می‌توانید اتوبوس سوار شوید، تاکسی بگیرید و یا از دوچرخه استفاده کنید. این‌ها همه استراتژی‌های شما برای جابجایی است. شما می‌توانید هر کدام از این استراتژی‌ها را بر اساس بودجه یا محدودیت زمانی خود انتخاب کنید.

## ساختار



این کلاس یک رفرنس به یک استراتژی را در خود نگه می‌دارد و تنها با استفاده از واسطه **strategy** ها با آنها ارتباط برقرار می‌کند.

- **Strategy**: این واسط بین تمامی strategy ها مشترک است. شامل یک متدهای می‌شود تا از context استفاده کند.
- **Concrete strategies**: این کلاس‌ها انواع الگوریتم‌ها را پیاده سازی می‌کنند تا context بتواند از آن استفاده کند.
- کلاس context هر زمان که نیاز به اجرای یک الگوریتم داشته باشد، متدهایی که در strategy ها پیاده سازی شده‌اند را فراخوانی می‌کند. Context از این که با چه strategy ای کار می‌کند و یا چه الگوریتمی را اجرا می‌کند هیچ آگاهی‌ای ندارد.
- **Client**: کلاینت یک strategy می‌سازد و آن را به context پاس می‌دهد. یک setter برای Context کلاینت ایجاد می‌کند تا کلاینت بتواند در زمان اجرا strategy مورد نظر خود را تغییر دهد.

## مثال

در این مثال context از یک سری استراتژی برای اجرای برخی از عملگرهای محاسباتی استفاده می‌کند.

```
// The strategy interface declares operations common to all
// supported versions of some algorithm. The context uses this
// interface to call the algorithm defined by the concrete
// strategies.

interface Strategy {
    method execute(a, b)

    // Concrete strategies implement the algorithm while following
    // the base strategy interface. The interface makes them
    // interchangeable in the context.
}

class ConcreteStrategyAdd implements Strategy {
    method execute(a, b) {
        return a + b
    }
}

class ConcreteStrategySubtract implements Strategy {
    method execute(a, b) {
        return a - b
    }
}

class ConcreteStrategyMultiply implements Strategy {
    method execute(a, b) {
        return a * b
    }
}

// The context defines the interface of interest to clients.

class Context {
    // The context maintains a reference to one of the strategy
    // objects. The context doesn't know the concrete class of a
    // strategy. It should work with all strategies via the
    // strategy interface.
    private strategy: Strategy
}
```

```

// Usually the context accepts a strategy through the
// constructor, and also provides a setter so that the
// strategy can be switched at runtime.
method setStrategy(Strategy strategy) is
    this.strategy = strategy

// The context delegates some work to the strategy object
// instead of implementing multiple versions of the
// algorithm on its own.
method executeStrategy(int a, int b) is
    return strategy.execute(a, b)

// The client code picks a concrete strategy and passes it to
// the context. The client should be aware of the differences
// between strategies in order to make the right choice.
class ExampleApplication is
    method main() is
        Create context object.

        Read first number.
        Read last number.
        Read the desired action from user input.

        if (action == addition) then
            context.setStrategy(new ConcreteStrategyAdd())

        if (action == subtraction) then
            context.setStrategy(new ConcreteStrategySubtract())

        if (action == multiplication) then
            context.setStrategy(new ConcreteStrategyMultiply())

            result = context.executeStrategy(First number, Second
number)

        Print result.

```

چه زمانی باید از این الگو استفاده کنیم؟

- زمانی از این الگو استفاده کنید که می‌خواهید از انواع الگوریتم‌ها در یک استفاده کنید و در زمان اجرا بتوانید بین این الگوریتم‌ها جابجا شوید.
- الگوی طراحی strategy این امکان را به شما می‌دهد تا بتوانید رفتار یک object را در زمان اجرا با

ارتباط دادن آنها با یک سری Object فرعی دیگر تغییر دهید. هر کدام از این Object های فرعی می‌توانند کار خاصی را انجام دهند.

- از این الگو زمانی استفاده کنید که تعداد زیادی کلاس مشابه دارید که تنها تفاوتشان در اجرای یک رفتار است.

این الگو به شما این اجازه را می‌دهد تا انواع رفتارهای متفاوت را در یک سلسله مراتبی از کلاس‌های جدا از هم پیاده سازی کنید و کلاس‌های اصلی خودتان را هم در یک کلاس ترکیب کنید. با این کار از تکرار شدن کدهایتان جلوگیری می‌شود.

- زمانی از این الگو استفاده کنید که می‌خواهید منطق برنامه‌تان را از جزئیات پیاده سازی آنها جدا کنید.

با استفاده از این الگو می‌توانید کد، داده‌های داخلی و وابستگی‌های الگوریتم‌های مختلف را از باقی کدهایتان جدا کنید. این الگو یک رابط کاربری ساده را در اختیار کلاینت قرار می‌دهد تا در هر لحظه از زمان اجرا بتواند الگوریتم مناسب را انتخاب کند.

- زمانی از این الگو استفاده کنید که کلاستان یک سری شرط‌های عظیم در خودش دارد که بین الگوریتم‌های مختلف یکسان است.

الگوی طراحی strategy این اجازه را به شما می‌دهد تا هر کدام از الگوریتم‌های خود را در کلاس‌های جداگانه‌ای قرار دهید. هر کدام از این کلاس‌ها باید یک interface مشترک را پیاده‌سازی کنند. Object اصلی برنامه‌تان به جای اینکه تک تک این الگوریتم‌ها را پیاده‌سازی کند وظیفه‌ی اجرای آنها را به همان کلاس واگذار می‌کند.

## معایب و مزايا

- ❖ می‌توانید در زمان اجرا بین الگوریتم‌ها جابجا شوید.
- ❖ می‌توانید کدهای پیاده‌سازی هر الگوریتم را از کدهایی که باید از این الگوریتم‌ها استفاده کنند جدا کنید.
- ❖ می‌توانید ارثبری را با composition جایگزین کنید.
- ❖ با استفاده از این الگو، اصل Open/Closed از اصول SOLID رعایت می‌شود چرا که می‌توانید هر دفعه که خواستید یک strategy به برنامه‌تان اضافه کنید بدون اینکه context-тан را تغییر دهید.

→ اگر فقط چند الگوریتم دارید که به ندرت تغییر می‌کنند، دلیلی برای پیچیده‌تر کردن برنامه با کلاس‌ها و interface های جدیدی که این الگو می‌گوید وجود ندارد.

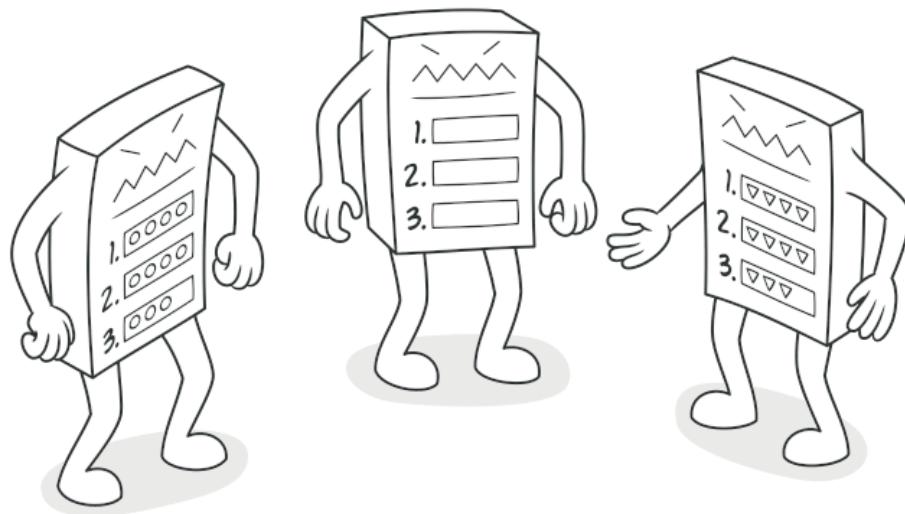
→ کلاینت باید تفاوت هر کدام از این strategy ها را با هم بداند تا بتواند یک strategy درست را انتخاب کند.

→ بسیاری از زبان‌های برنامه‌نویسی مدرن هم اکنون از anonymous function ها پشتیبانی می‌کنند. این توابع به شما این امکان را می‌دهند که نسخه‌های مختلف یک الگوریتم را به صورت توابع

جداگانه پیاده‌سازی کنید. بعداً می‌توانید از این توابع برای انجام الگوریتم‌ها استفاده کنید، بدون اینکه نیاز به ایجاد کلاس‌ها و رابطه‌ای اضافی باشد.

## Template Method

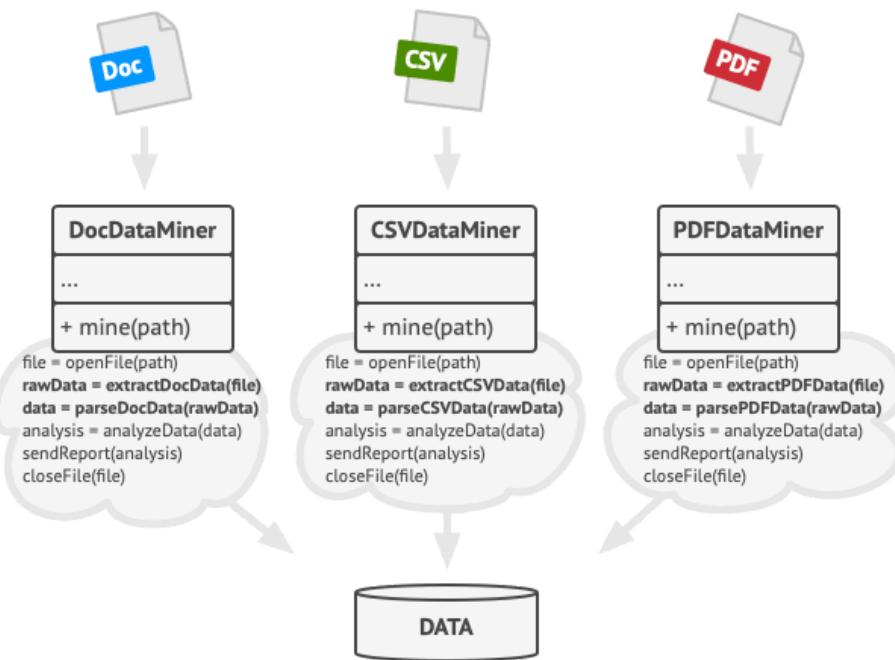
با استفاده از این الگو می‌توانید اسکلت اصلی الگوریتم‌هایتان را در کلاس‌های Parent ایجاد کنید و کلاس‌های فرزند بسته به نیاز خودشان آنها را شخصی سازی کنند بدون اینکه به ساختار آن آسیبی وارد شود.



### طرح مسئله

فرض کنید که برنامه‌ای برای داده‌کاوی مستندات یک شرکت بزرگ می‌نویسید. کاربران انواع فایل‌های مختلف مانند PDF, CSV, DOC, ... را در این برنامه دارند و برنامه‌ی شما باید سعی کند تا داده‌های معنا داری را از این اسناد در قالبهای یکسانی خارج کند.

نسخه‌ی اولیه برنامه‌ی شما فقط از فایل‌های با فرمت DOC پشتیبانی می‌کند. بعدها به این فکر می‌افتید که فرمتهای دیگری مانند CSV و PDF را نیز به برنامه‌تان اضافه کنید.



بعد از مدتی متوجه می‌شوید که هر سه‌ی این کلاس‌ها مقدار زیادی کد مشابه دارند. در حالی که کد مربوط به فرمتهای مختلف در همه‌ی کلاس‌ها کاملاً با هم فرق دارند کدهای مربوط به پردازش و تجزیه و تحلیل داده‌ها تقریباً با هم یکسان است. نظرتان در این رابطه که بدون اینکه به الگوریتم اصلی دست بزنیم کدهای تکراری را حذف کنیم چیست؟ به نظر ایده‌ی خوبی است!

مشکل دیگری هم وجود دارد و آن مربوط به کدهای کلاینتمان است که از این کلاس‌ها استفاده می‌کنند. این کلاس‌ها دارای شرط‌های زیادی هستند که با توجه به نوع کلاسی که اشیاء موجود را پردازش می‌کند، مسیر صحیح عملیات را انتخاب می‌کنند. به عبارت دیگر، اگر نوع کلاس پردازش مشخص باشد، کد کلاینت باید با استفاده از شرط‌ها تصمیم بگیرید که چه عملیاتی باید انجام شود.

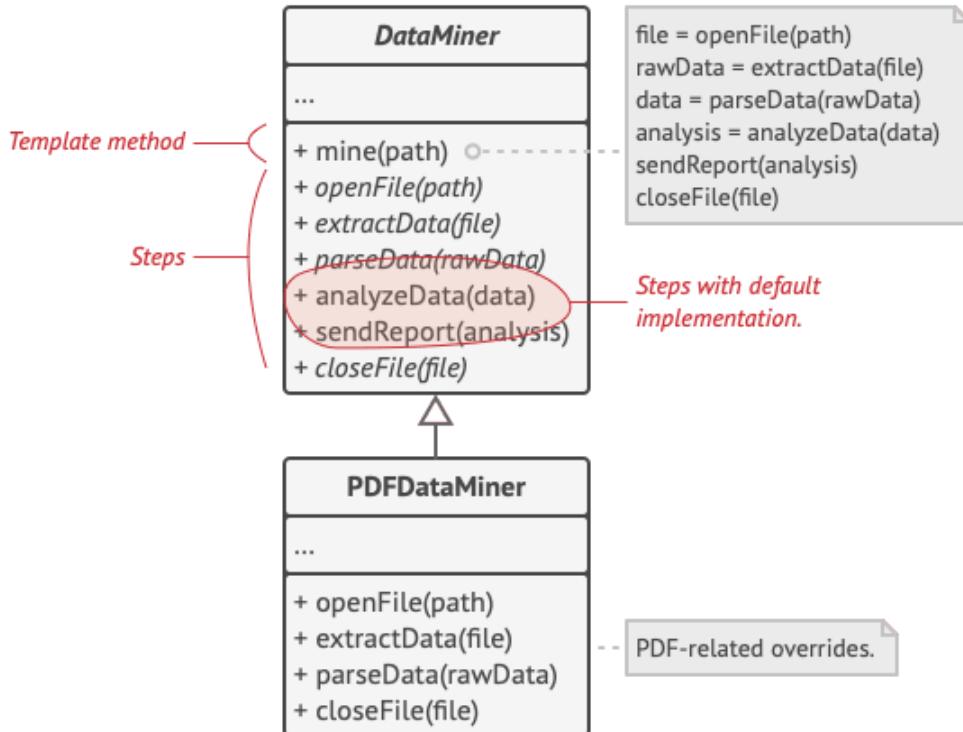
اما اگر هر سه تا کلاسی که برای پردازش فرمتهای مختلف داریم یک interface را پیاده‌سازی کنند به کلاینت این اجازه را می‌دهند تا بدون اینکه از شرط‌های مختلف برای شناسایی یک process استفاده کند، به راحتی با استفاده از متدهایی که بین این process‌ها مشترک است، عملیات مناسب را انجام دهد.

## راه حل

الگوی طراحی template method پیشنهاد می‌کند تا یک الگوریتم را به چندین مرحله‌ی کوچکتر تقسیم کنید و آن‌ها را به متدهای کنید. سپس به ترتیب هر کدام از این متدها را در یک Template method قرار دهید. پیاده‌سازی کنید.

هر کدام از این مراحل می‌توانند abstract باشند و یا به صورت پیشفرض پیاده‌سازی ای نیز داشته باشند. برای اینکه کلاینت از این الگوریتم استفاده کند باید تمام این مراحل را به صورت اختصاصی پیاده‌سازی کند.

بیینیم که چگونه می‌توانیم این رویکرد را در برنامه‌ی داده‌کاوی خود پیاده کنیم. می‌توانیم یک کلاس پایه برای هر سه الگوریتم خود ایجاد کنیم. این کلاس یک template method است که شامل فراخوانی یک سری از فرایندهای پردازش یک سند است.



در ابتدا می‌توانیم تمامی مراحل را abstract در نظر بگیریم و زیر-کلاس‌ها را مجبور کنیم تا به شکلی که خودشان نیاز دارند این مراحل را پیاده سازی کنند. در مثال ما، زیر-کلاس‌ها از قبل همه‌ی پیاده سازی‌ها را دارند، بنابراین فقط باید به signature متدۀای کلاس پدر که می‌خواهیم آن‌ها را پیاده سازی کنیم دقت کنیم.

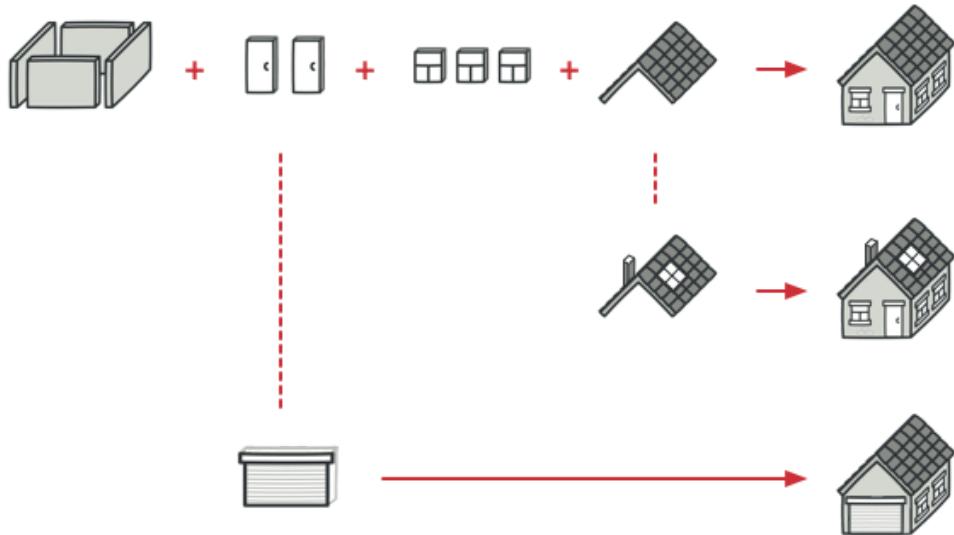
حالا بباید بیینیم برای خلاص شدن از دست کدهای تکراری چه کاری باید انجام دهیم. به نظر می‌رسد که کدهای مربوط به باز و بسته کردن فایل‌ها و کارهایی از این دست برای هر فرمات متفاوت است. در نتیجه نمی‌توان به این متدۀا دست زد و آنها را تغییر داد. اما کارهایی مانند آنالیز کردن دیتاباها و یا گزارشگیری‌ها در این سه پردازش شبیه به هم است. پس می‌توان این موارد را در کلاس پایه قرار داد تا زیر کلاس‌ها از آنها استفاده کنند.

همانطور که دیدید، ما دو مرحله داریم:

- مراحل abstract که باید توسط زیر-کلاس‌ها پیاده سازی شوند.
- مراحل optional که از قبل پیاده سازی دارند و می‌توان بسته به نیاز از آنها استفاده کرد.

یک نوع دیگر از این مراحل هم وجود دارد که به آن قلاب یا hook می‌گویند. Hook‌ها در واقع به همان بخش optional اشاره دارند ولی پیاده سازی ای ندارند. کاربرد hook‌ها در واقع قبل و یا بعد از انجام یک الگوریتم اصلی است. به نوعی که در صورت نیاز می‌توان قبل یا بعد از اجرای الگوریتم مواردی را اضافه کرد.

## مقایسه با دنیای واقعی

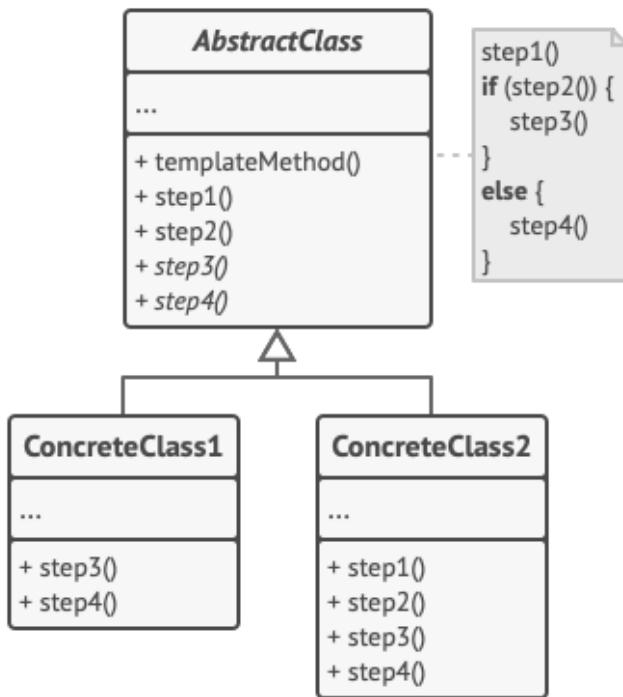


تصور کنید شما یک معمار هستید و یک پلان معمولی برای ساخت خانه‌های استاندارد دارید. این پلان شامل مراحل مختلفی است که برای ساخت هر خانه استفاده می‌شود، مثل ساخت پی، نصب اسکلت، ساخت دیوارها، نصب لوله‌کشی و سیم‌کشی برق و غیره.

حالا، شما می‌خواهید افراد مختلف بتوانند این پلان را به شکل دلخواه تغییر دهند بدون اینکه به ساختار کلی پلان دست بزنند. در اینجا الگوی template method به کمک می‌آید.

- **پایه‌گذاری (Foundation):** این یک مرحله مهم است که همه خانه‌ها نیاز به آن دارند. اما شما یک hook (قبل از مرحله) اضافه می‌کنید. این hook به مالکان این امکان را می‌دهد که نوع مواد مصرفی در پایه‌گذاری (مثل سنگ، بتن یا آجر) را انتخاب کنند.
  - **اسکلت (Framing):** در این مرحله نیز یک hook (بعد از مرحله) اضافه می‌شود. مثلاً مالکان می‌توانند تعداد و اندازه اتاق‌ها را تغییر دهند.
  - **سیستم‌های آب و برق (Plumbing and Wiring):** اینجا نیز hook هایی اضافه می‌شوند. مالکان می‌توانند جزئیاتی از لوله‌کشی و سیم‌کشی برق را تغییر دهند، مثل افزودن پکیج‌های هوشمند یا انتخاب لوازم بهداشتی مورد نظر.
- به این ترتیب، با استفاده از الگوی template method، شما اجازه می‌دهید تا جزئیات خاص هر خانه را تغییر داده و در عین حال از ساختار کلی و اجزای اصلی پلان برای همگان استفاده کنید.

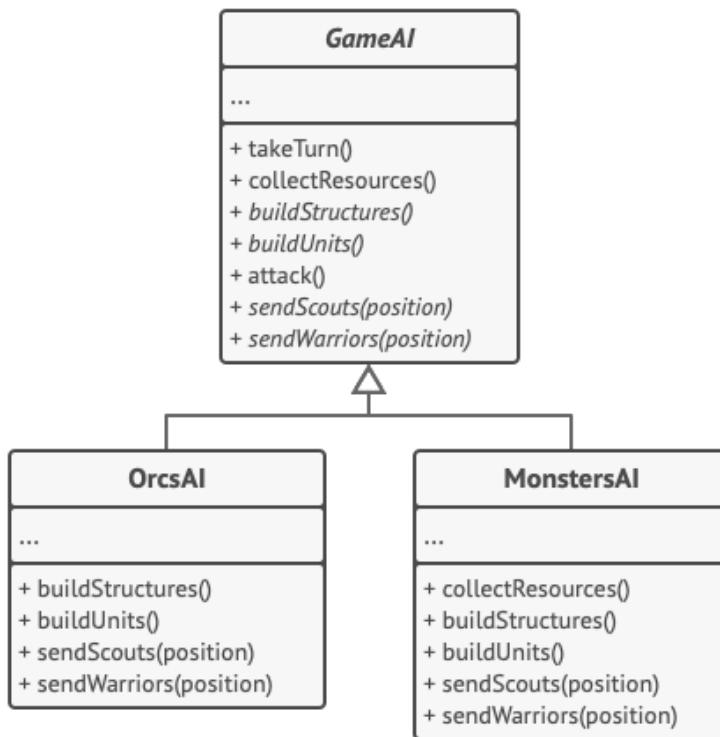
## ساختار



- این کلاس متدهایی که مراحل یک الگوریتم را تعریف می‌کنند را در خود دارد و abstract template method باشند یا اینکه به صورت پیشفرض پیاده‌سازی ای داشته باشند.
- این کلاس‌ها می‌توانند تمامی مراحل را override کنند. اما نمی‌توانند این کار را با template method ها نیز انجام دهند.

## مثال

در این مثال الگوی طراحی template method یک اسکلت برای انواع شاخه‌های هوش مصنوعی در یک بازی ویدیویی استراتژیک ایجاد می‌کند.



اکثر مسابقه‌های داخل بازی واحدها و سازه‌های یکسانی دارند. بنابراین می‌توانید از یک ساختار AI یکسان برای انواع مسابقه‌ها استفاده کنید و در عین حال بعضی از جزئیات را نیز Override کنید.

```

// The abstract class defines a template method that contains a
// skeleton of some algorithm composed of calls, usually to
// abstract primitive operations. Concrete subclasses implement
// these operations, but leave the template method itself
// intact.
class GameAI is
    // The template method defines the skeleton of an algorithm.
    method turn() is
        collectResources()
        buildStructures()
        buildUnits()
        attack()

    // Some of the steps may be implemented right in a base
    // class.
    method collectResources() is
        foreach (s in this.builtStructures) do
            s.collect()

```

```

// And some of them may be defined as abstract.
abstract method buildStructures()
abstract method buildUnits()

// A class can have several template methods.
method attack() is
    enemy = closestEnemy()
    if (enemy == null)
        sendScouts(map.center)
    else
        sendWarriors(enemy.position)

abstract method sendScouts(position)
abstract method sendWarriors(position)

// Concrete classes have to implement all abstract operations of
// the base class but they must not override the template method
// itself.
class OrcsAI extends GameAI is
    method buildStructures() is
        if (there are some resources) then
            // Build farms, then barracks, then stronghold.

    method buildUnits() is
        if (there are plenty of resources) then
            if (there are no scouts)
                // Build peon, add it to scouts group.
            else
                // Build grunt, add it to warriors group.

    // ...

    method sendScouts(position) is
        if (scouts.length > 0) then
            // Send scouts to position.

    method sendWarriors(position) is
        if (warriors.length > 5) then
            // Send warriors to position.

// Subclasses can also override some operations with a default
// implementation.
class MonstersAI extends GameAI is
    method collectResources() is
        // Monsters don't collect resources.

    method buildStructures() is
        // Monsters don't build structures.

```

```
method buildUnits() is
    // Monsters don't build units.
```

## چه زمانی باید از این الگو استفاده کنیم؟

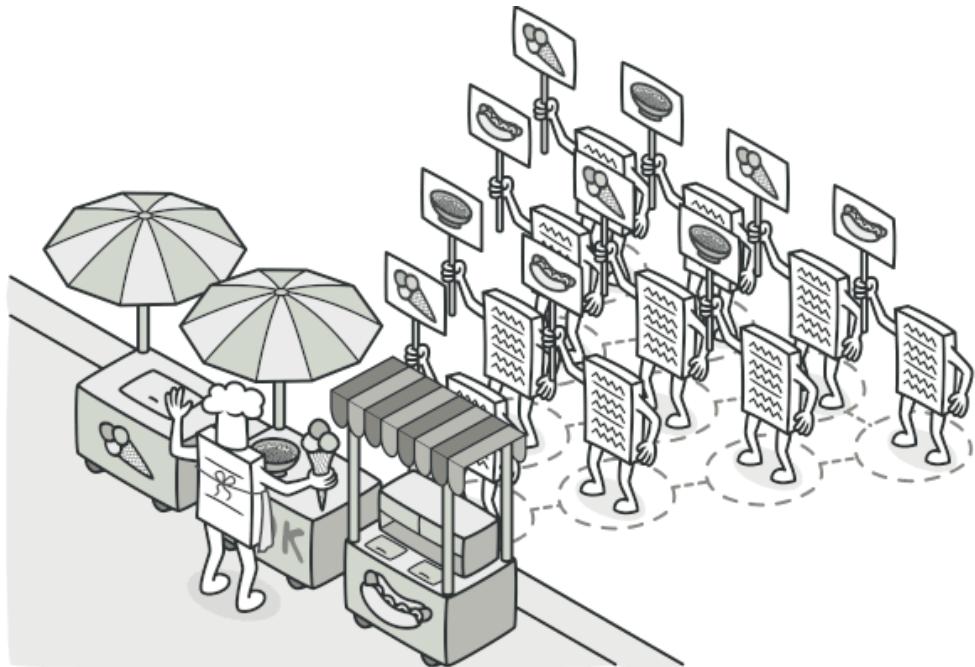
- زمانی از این الگو استفاده کنید که می‌خواهید کلاینت‌هایتان فقط مراحل خاصی از یک الگوریتم را گسترش دهند نه کل ساختار الگوریتم را.
  - الگوی طراحی template methods به شما امکان می‌دهد تا یک الگوریتم یکپارچه را به مجموعه ای از مراحل جداگانه تبدیل کنید که می‌توانند به راحتی توسط زیر کلاس‌ها گسترش یابند و در عین حال ساختار تعریف شده در کلاس پدر دست نخورده باقی بماند.
  - زمانی از این الگو استفاده کنید که الگوریتم‌های تقریباً یکسان اما با تفاوت‌های جزئی دارید. در نتیجه وقتی بخواهید تغییر کوچکی در الگوریتم ایجاد کنید مجبور می‌شوید که کل کلاس‌ها را تغییر دهید.
- زمانی که یک الگوریتم را به یک template method تبدیل می‌کنید، می‌توانید مراحلی که پیاده‌سازی های مشترکی دارند را در کلاس پدر قرار دهید و با این کار از تکرار کد جلوگیری کنید.

## معایب و مزایا

- ❖ به کلاینت‌هایتان این امکان را می‌دهید تا هر مرحله‌ای از الگوریتم را که می‌خواهند پیاده سازی کنند و با اینکار کمتر تحت تاثیر تغییر کد قسمت‌های دیگر برنامه قرار بگیرند.
- ❖ می‌توانید کدهای تکراری را در کلاس پدر قرار دهید.
- ➔ بعضی از کلاینت‌ها ممکن است که بر اساس اسکلتی که برای الگوریتم مان تعریف کرده ایم محدود شوند.
- ➔ ممکن است که با جایگزینی زیر-کلاس‌ها به جای یک الگوریتم اصلی اصل Liskov substitution از اصول SOLID را نقض کنید.
- ➔ هر چه تعداد template method ها بیشتر شوند نگهداری از آنها هم سخت تر خواهد بود.

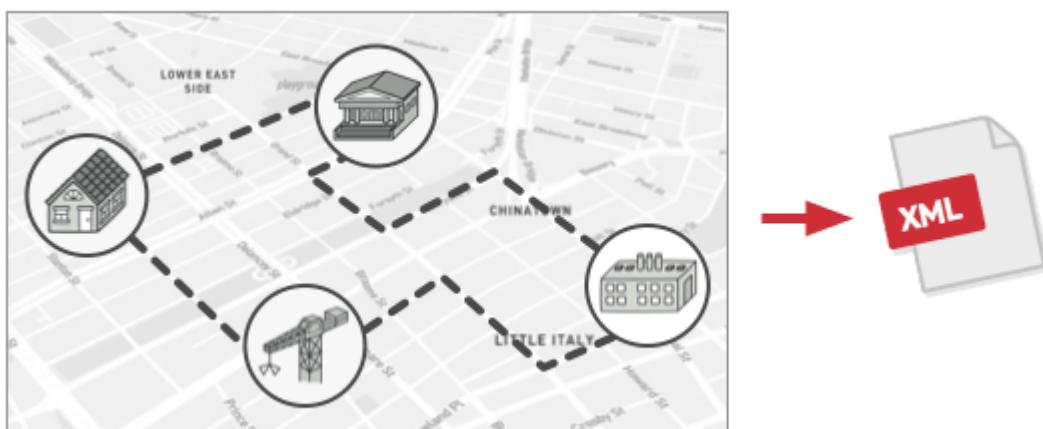
## Visitor

با استفاده از الگوی طراحی visitor می‌توانید الگوریتم‌ها را از Object هایی که بر روی آن‌ها کار می‌کنند جدا کنید.



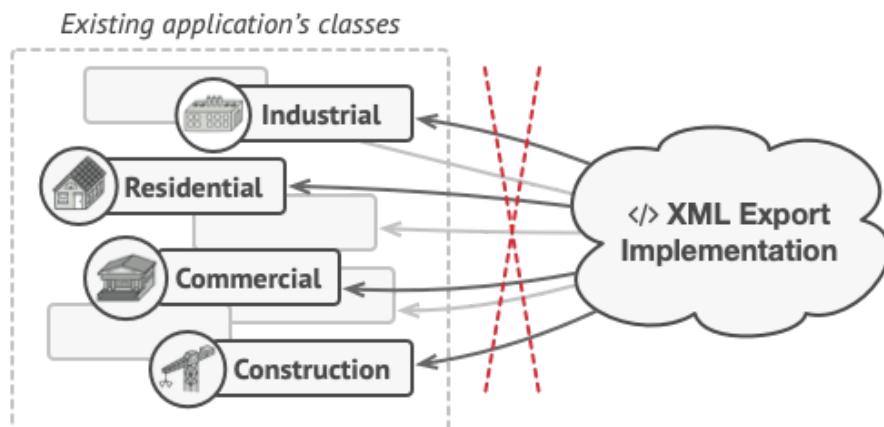
## طرح مسئله

فرض کنید تیم شما در حال توسعه برنامه‌ای است که با یک سری اطلاعات جغرافیایی و گراف‌های عظیم ساختار یافته‌ای سر و کار دارد. هر کدام از node های داخل گراف می‌توانند نشان دهنده‌ی یک موجودیت پیچیده مانند شهر و یا مناظر دیدنی و ... باشند. تمامی node ها در صورتی که یک جاده بین Object های آنها وجود داشته باشد به هم‌دیگر وصل خواهند شد. در پشت صحنه در واقع هر node توسط کلاسیش شناخته می‌شود و هر کدام از این node ها خودشان یک object هستند.



اکنون یک تسك به شما داده شده است که از گراف تشکیل شده خروجی XML بگیرید. در ابتدا کار بسیار ساده به نظر می‌رسد. تصمیم می‌گیرید تا یک متده‌ای با نام `export` به کلاس `node` اضافه کنید و به صورت بازگشتی از یک متده‌ای با نام `node` دیگر بروید و این متده را اجرا کنید. راه حل بسیار ساده و ظرفی است.

متاسفانه معمار سیستم به شما اجازه‌ای تغییر کلاس‌ها را نمی‌دهد. او به شما می‌گوید که کد در حال اجرا است و زیر بار است و نمی‌تواند این ریسک را پیذیرد که با تغییرات شما برنامه با باگ مواجه شود.



از طرفی هم از شما سوال می‌کند که آیا نگه داشتن کدهای مربوط به خروجی گرفتن XML در کلاس‌های node کاری منطقی است یا خیر؟ چرا که کار اصلی این کلاس‌ها کار کردن با داده‌های جغرافیایی است و خروجی گرفتن XML در این مکان جایز نیست.

و اما دلیل دیگری هم برای رد کردن درخواست شما وجود دارد. چرا که ممکن است در آینده از طرف تیم فروش و یا بازاریابی درخواست‌هایی مانند خروجی گرفتن در فرمتهای دیگر نیز داده شود. و این درخواست‌ها موجب می‌شود تا شما دوباره آن کلاس‌های مهم و ظرفی را تغییر دهید. پس چه باید کرد؟

## راه حل

الگوی طراحی visitor به شما پیشنهاد می‌دهد تا هر گونه رفتار جدیدی که می‌خواهید به برنامه‌تان اضافه کنید را به جای اینکه با کدهای قبلی ادغام کنید آنها را در کلاس‌های جداگانه‌ای قرار دهید. با این کار object اصلی ای که باید آن رفتار را پیاده‌سازی کند، اکنون به عنوان آرگومان ورودی به یکی از متدهای visitor پاس داده می‌شود و این متده دسترسی به تمام موارد اساسی آن object را فراهم می‌کند. اگر بتوانیم آن رفتار را بر روی object ها و کلاس‌های مختلفی اجرا کنیم چه می‌شود؟ در مثالی که زدیم قطعاً پیاده‌سازی فرایند خروجی گرفتن XML با انواع کلاس‌های node کمی متفاوت است. بنابراین کلاس visitor نباید فقط یک متده بلکه باید انواع متدها با انواع آرگومان‌های ورودی را در خود داشته باشد تا بتواند موارد مختلف را تحت پوشش قرار دهد. برای اینکه بهتر متوجه شوید به این کد توجه کنید:

```
class ExportVisitor implements Visitor {
    method doForCity(City c) { ... }
    method doForIndustry(Industry f) { ... }
    method doForSightSeeing(SightSeeing ss) { ... }
    // ...
}
```

اما چگونه باید با این متدها در تعامل باشیم؟ علی‌الخصوص زمانی که با کل گرافمان ارتباط برقرار می‌کنیم. این متدها signature‌های متفاوتی دارند پس نمی‌توان از polymorphism استفاده کرد. برای انتخاب یک متده visitor مناسب که بتواند یک Object به خصوص را پردازش کند، اول باید کلاس آن متده را بررسی کنیم. این کار شبیه به یک کابوس است!

```
foreach (Node node in graph)
    if (node instanceof City)
        exportVisitor.doForCity((City) node)
    if (node instanceof Industry)
        exportVisitor.doForIndustry((Industry) node)
    // ...
}
```

شاید سوال کنید که خب چرا از overloading استفاده نمی‌کنیم؟ چرا که overloading زمانی است که متدها نام یکسانی دارند اما signature هایشان متفاوت است. با فرض اینکه زبان برنامه‌نویسی ای که با آن کار می‌کنید از این قابلیت پشتیبانی کند هم این مورد کمکی به ما نمی‌کند. تا زمانی که کلاس دقیق یک node ناشناس باشد با overload کردن نمی‌توان تشخیص داد که کدام متده باید اлан اجرا شود و به صورت پیشفرض متده کلاس پایه‌ای node را به عنوان آرگومان دریافت کند اجرا می‌شود. با این حال الگوی طراحی visitor این مشکل را حل می‌کند. این الگو از تکنیک Double Dispatch استفاده می‌کند که به اجرای یک متده مناسب بر روی یک Object بدون ایجاد شرط‌های اضافی و دست و پا گیر کمک می‌کند.

چطور است به جای اینکه کلاینت تصمیم بگیرد چه متده را فراخوانی کند، این کار را به Object هایی که به عنوان آرگومان ورودی متدها پاس می‌دهیم واگذار کنیم؟ از آنجایی که Object ها کلاس‌های خودشان را می‌شناسند، می‌توانند بهترین متده را برای visitor انتخاب کنند. آن‌ها یک متده accept می‌کنند و به آن می‌گوینند که چه متدهایی باید اجرا شوند :

```
// Client code
foreach (Node node in graph)
    node.accept(exportVisitor)

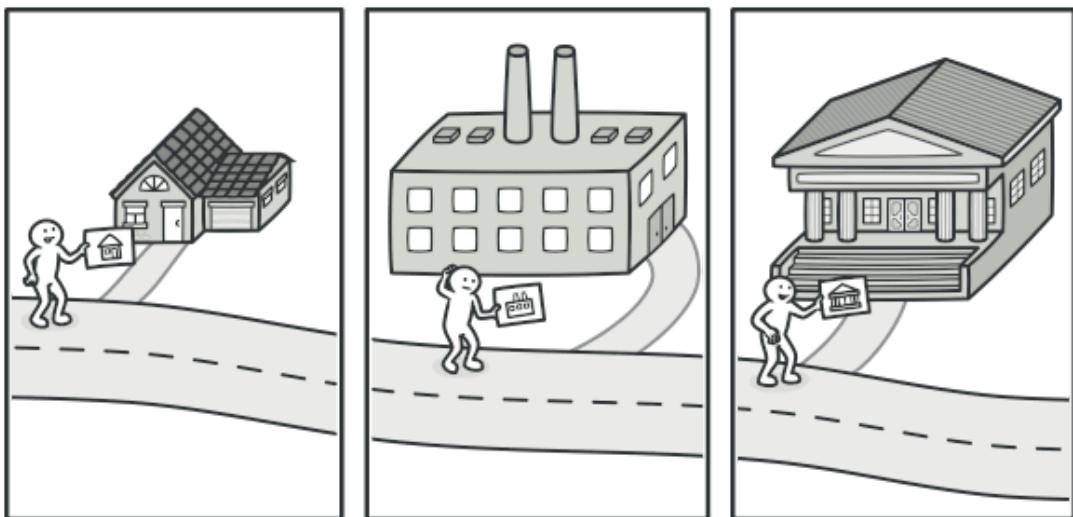
// City
class City {
    method accept(Visitor v) {
        v.doForCity(this)
    }
}

// Industry
class Industry {
    method accept(Visitor v) {
        v.doForIndustry(this)
    }
}
```

جا دارد که اعتراض کنم که مجبور شدیم تا در نهایت کدهای node را تغییر دهیم . اما این تغییر اهمیت چندانی ندارد و می‌توانیم بدون اینکه کدام را تغییر دهیم رفتارهای بیشتر اضافه کنیم.

حال اگر تمام visitor ها یک interface را پیاده‌سازی کنند دیگر تمام node ها می‌توانند با هر visitor جدیدی که به برنامه‌تان اضافه می‌کنند کار کنند. هر زمان احساس کردید که باید رفتار جدیدی به node تان اضافه شود، تنها کاری که باید انجام دهید این است که یک visitor جدید تعریف کنید.

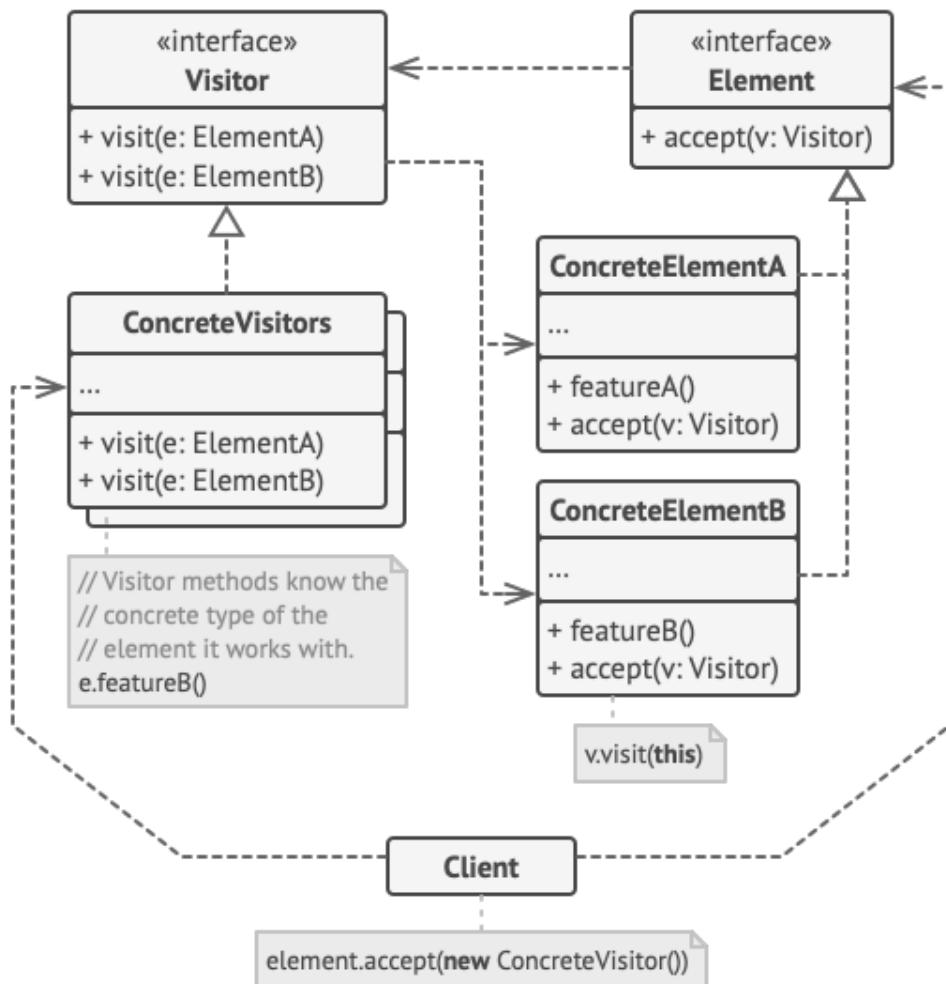
### مقایسه با دنیای واقعی



یک مشاور بیمه‌ی فصلی را در نظر بگیرید که می‌خواهد مشتری های خود را افزایش دهد. می‌تواند در یک مجله از یک خانه به خانه‌ی دیگر برود و به هر کدام از آنها بیمه بفروشد. بسته به نوع ساختمان‌ها و سازه‌ی آنها می‌تواند به هر کدام از آنها بیمه‌ی مناسب با آن ها را پیشنهاد دهد :

- اگر ساختمان مسکونی باشد، بیمه‌ی درمانی می‌فروشد.
- اگر بانک باشد، بیمه‌ی سرقت می‌فروشد.
- اگر کافی شاپ باشد بیمه‌ی آتش‌سوزی و سیل می‌فروشد.

## ساختار

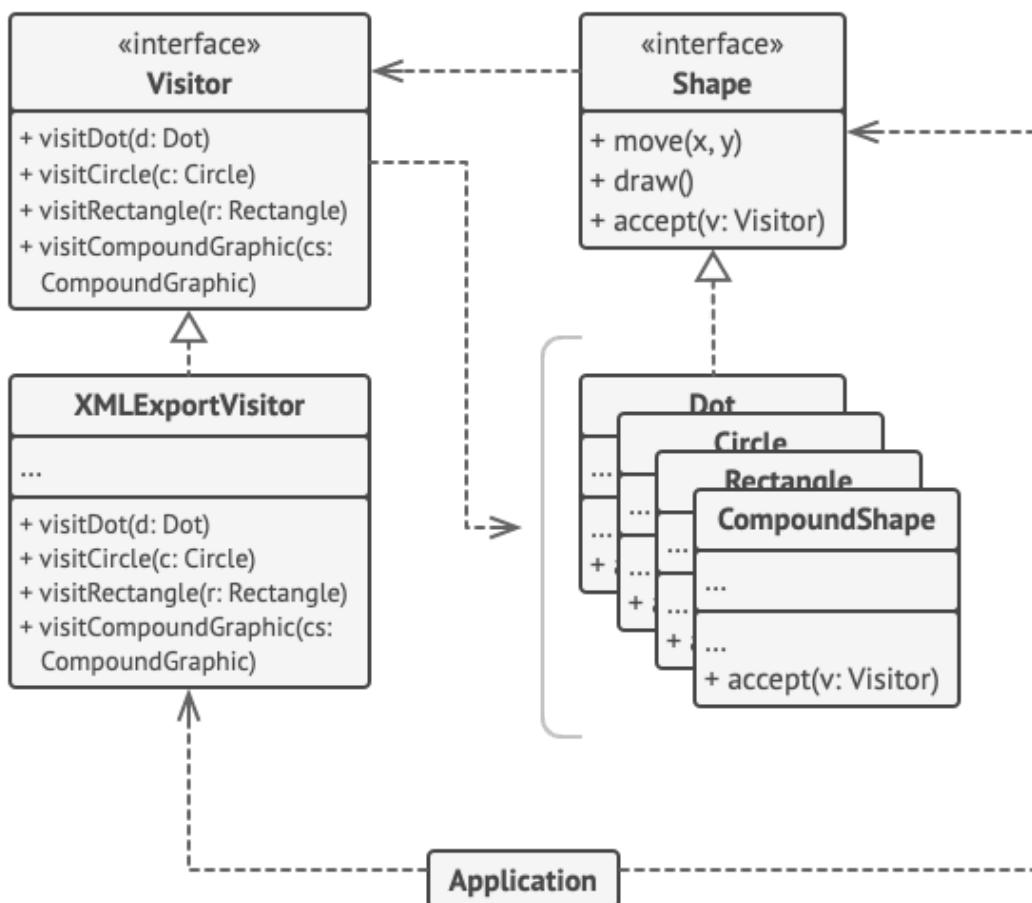


- **Visitor**: این واسط تعدادی متد در خودش تعریف می‌کند که هر کدام از این متدها عناصر تشکیل دهنده‌ی یک object را به عنوان ورودی دریافت می‌کنند. اگر زبان برنامه‌نویسی ای که با آن کار می‌کنید از overloading پشتیبانی کند، ممکن است اسامی این متدها با هم یکسان باشند اما آرگومان‌های ورودی متفاوتی داشته باشند.
- **Concrete visitor**: هر کدام از این کلاس‌ها چندین نسخه از رفتارهای یکسان که برای ایمان‌های مختلف هستند را پیاده‌سازی می‌کنند.
- **Element**: این کلاس یک متد برای `accept` کردن visitor‌ها در خودش دارد. بهتر است که این متد یک ورودی از نوع **Visitor** را دریافت کند.
- **Concrete Element**: هر کدام از این کلاس‌ها باید متدهای `accept` را پیاده‌سازی کنند. هدف اصلی این متد این است که متد مناسب برای element فعلی را از متدهای visitor فراخوانی کند. این را در نظر داشته باشید که حتی اگر کلاس پایه‌ی element این متد را پیاده‌سازی کرده باشند هم تمامی concrete element‌ها باید این متد را override کنند و متد مناسب visitor را بر روی object اعمال کنند.

- کلاینتها به object ها دسترسی دارند و می‌توانند به آن ها بگویند که از visitor مناسب برای پردازش ها استفاده کنند.

## مثال

در این مثال با استفاده از الگوی طراحی visitor، قابلیتی برای دریافت خروجی XML از اشکال هندسی را در برنامه ایجاد می‌کنیم.



```

// The element interface declares an `accept` method that takes
// the base visitor interface as an argument.
interface Shape is
    method move(x, y)
    method draw()
    method accept(v: Visitor)

// Each concrete element class must implement the `accept`
// method in such a way that it calls the visitor's method that
// corresponds to the element's class.
class Dot implements Shape is
    // ...
  
```

```

// Note that we're calling `visitDot`, which matches the
// current class name. This way we let the visitor know the
// class of the element it works with.
method accept(v: Visitor) is
    v.visitDot(this)

class Circle implements Shape is
    // ...
    method accept(v: Visitor) is
        v.visitCircle(this)

class Rectangle implements Shape is
    // ...
    method accept(v: Visitor) is
        v.visitRectangle(this)

class CompoundShape implements Shape is
    // ...
    method accept(v: Visitor) is
        v.visitCompoundShape(this)

```

```

// The Visitor interface declares a set of visiting methods that
// correspond to element classes. The signature of a visiting
// method lets the visitor identify the exact class of the
// element that it's dealing with.
interface Visitor is
    method visitDot(d: Dot)
    method visitCircle(c: Circle)
    method visitRectangle(r: Rectangle)
    method visitCompoundShape(cs: CompoundShape)

```

```

// Concrete visitors implement several versions of the same
// algorithm, which can work with all concrete element classes.
// 
// You can experience the biggest benefit of the Visitor pattern
// when using it with a complex object structure such as a
// Composite tree. In this case, it might be helpful to store
// some intermediate state of the algorithm while executing the
// visitor's methods over various objects of the structure.
class XMLExportVisitor implements Visitor is
    method visitDot(d: Dot) is
        // Export the dot's ID and center coordinates.

    method visitCircle(c: Circle) is
        // Export the circle's ID, center coordinates and
        // radius.

```

```

method visitRectangle(r: Rectangle) is
    // Export the rectangle's ID, left-top coordinates,
    // width and height.

method visitCompoundShape(cs: CompoundShape) is
    // Export the shape's ID as well as the list of its
    // children's IDs.

// The client code can run visitor operations over any set of
// elements without figuring out their concrete classes. The
// accept operation directs a call to the appropriate operation
// in the visitor object.
class Application is
    field allShapes: array of Shapes

    method export() is
        exportVisitor = new XMLExportVisitor()

        foreach (shape in allShapes) do
            shape.accept(exportVisitor)

```

چه زمانی باید از این الگو استفاده کنیم؟

- زمانی از این الگو استفاده کنید که نیاز دارید تغییری بر روی ساختار یک object پیچیده اعمال کنید.
- الگوی Visitor به شما امکان می دهد یک عملیات بر روی مجموعه ای از اشیاء با کلاس های مختلف را با استفاده از یک visitor که یک سری از فرایندها را پیاده سازی کرده است اجرا کنید.
- زمانی از این الگو استفاده کنید که می خواهید برای پاک کردن یک سری از منطق ها در برنامه تان از یک سری از رفتارهای کمکی استفاده کنید.
- این الگو به شما این امکان را می دهد تا کلاس های اصلی برنامه تان با استفاده از متدهایی که در کلاس visitor ایجاد می شوند، بیشتر بر روی وظیفه ای اصلی شان تمرکز کنند.
- زمانی از این الگو استفاده کنید که یک رفتار فقط در سلسله مراتب یک کلاس به چشم می آید نه در دیگر کلاس ها.
- شما می توانید این رفتار را به صورت جداگانه در یک کلاس visitor پیاده سازی کنید و فقط متدهای visitor را اجرا کنید.

## معایب و مزایا

- ❖ با استفاده از این الگو اصل SOLID از اصول Reuse می‌شود چرا که می‌توانید بدون اینکه تغییری در کدهایتان ایجاد کنید یک رفتار جدید اضافه کنید که بتواند با object هایی از کلاس‌های مختلف کار کند.
- ❖ با استفاده از این الگو اصل Single responsibility از اصول SOLID رعایت می‌شود چرا که می‌توانید آن دسته از رفتارهایی که مشابه یکدیگرند را در یک کلاس قرار دهید.
- ❖ یک شیء visitor می‌تواند داده‌های مفیدی را حین کار کردن با object های مختلف جمعآوری کند. این مورد زمانی اهمیت پیدا می‌کند که می‌خواهید یک شیء با ساختار پیچیده را مورد بررسی قرار دهید.
- هر زمان که کلاسی به سلسله مراتب element ها اضافه یا حذف شد باید تمام visitor ها را تغییر دهید.
- Visitor ها ممکن است دسترسی مورد نیاز به متدها و فیلدهای private آن object ای که با آن کار می‌کنند را نداشته باشند.