



موسسه آموزش عالی راهبرد شمال
پایان نامه کارشناسی - مهندسی کامپیووتر

الگوهای طراحی در شی گرائی

دانشجو

مهندی نیک ضمیر

استاد مربوطه

دکتر مرضیه فریدی

۱۴۰۰ بهمن

فهرست

۱.	چکیده	فهرست ب
۲.	فصل اول : مقدمه	
۳.	کلمات کلیدی:	
۴.	فصل دوم: پیشینه پژوهش	
۵.	فصل سوم: روش تحقیق	
۶.	فصل چهارم: تجزیه و تحلیل یافته ها	
۷.	الگوهای تکوینی	
۸.	الگوی Factory Method	
۱۵.	الگوی Abstract Factory	
۱۹.	الگوی Prototype	
۲۳.	الگوی Singleton	
۲۷.	الگوی Builder	
۳۱.	الگوهای ساختاری	
۳۲.	الگوی Adapter	
۳۷.	الگوی Bridge	
۴۵.	الگوی Composite	
۵۰.	الگوی Decorator	
۵۴.	الگوی Facade	
۵۸.	الگوی Proxy	
۶۳.	الگوی Flyweight	
۶۹.	الگوهای رفتاری	

٧٠	الگوی Chain Of Responsibility
٧٤	الگوی Command
٨٠	الگوی Observer
٨٧	الگوی Iterator
٩٠	الگوی State
٩٦	الگوی Strategy
١٠٠	الگوی Template Method
١٠٣	فصل پنجم : منابع

چکیده

کلمه الگوهای طراحی اولین بار توسط اریک گاما، ریچارد هلم و راف جانسون در سال ۱۹۹۴ استفاده شد. الگوهای طراحی راه حل هایی برای مشکلات رایج طراحی نرم افزار هستند. در کتاب "الگوهای طراحی: عناصر برنامه های شیء گرایی قابل استفاده مجدد" الگوهای طراحی به سه دسته توکینی، ساختاری و رفتاری دسته بندی شده اند. الگوهای توکینی برای مشکلات در حوزه نمونه سازی، الگوهای ساختاری در حوزه ترکیب اشیاء یا کلاس ها و الگوهای رفتاری در حوزه ارتباط اشیاء یا کلاس ها، راه حل هایی را ارائه می کنند. در کتاب مذکور ۲۳ الگو بیان شده است. با توجه به گذر زمان و پیشرفت و تغییر زبان های برنامه نویسی، برخی الگوها کاربرد خود را از دست داده اند. در این مقاله ۱۹ الگوی مهم و اساسی تعریف، بررسی و در زبان پایتون پیاده سازی شده است که شامل الگوهای Singleton، Prototype، Abstract Factory، Factory Method، Chain Of Responsibility، Flyweight، Proxy، Facade، Decorator، Composite، Brdige، Adapter، Builder، Template Method، Strategy، State، Iterator، Observer، Command مفاهیم تا حد امکان ساده سازی شده و مثال ها کاربردی و بروز باشند.

فصل اول : مقدمه

یکی از مهم ترین مباحث مهندسی نرم افزار شناخت اصول و استاندارد های برنامه نویسی است که در میان آنها، الگوهای طراحی از اهمیت ویژه ای برخوردار است. یکی از معضلات برنامه نویسانی که به تازگی برای کار روی پروژه ای استخدام شده اند، پی بردن به چگونگی عملکرد سیستم فعلی و شناخت اجزای مختلف پروژه است. این فرآیند بسیار زمان بر است و در بسیاری از موقع برنامه نویس جدید به درک درستی از سیستم مورد بحث نمی رساند. برای حل این مشکل چندین راه حل پیشنهاد می شود. یکی از این راه ها استاندارد سازی زبان های برنامه نویسی است. راه دیگر استفاده از کتابخانه های استاندارد است. اما مهم ترین و بهترین راه، شناخت و به کارگیری الگوهای طراحی است.

الگوهای طراحی از دو بعد دارای اهمیت بسیاری هستند:

۱- ارتباط میان توسعه دهنده‌گان مختلف را تسهیل می کنند.

اگر دو توسعه دهنده، هر دو الگوی طراحی "observer" را بدانند، دیگر نیازی به ساعت‌ها بحث و واکاوی کد نیست و هر دو می‌توانند به راحتی طرز کار سیستم یا بخشی از آن سیستم را درک کنند.

۲- راهکارهای آماده و آسانی برای حل مشکلات متعدد هستند.

اگر شما به الگوهای برنامه نویسی مسلط باشید، هنگام مواجه با مشکلات طراحی و برنامه نویسی می‌دانید که باید چه راهکاری را پیش بگیرید. به طور مثال می‌خواهید سیستمی طراحی کنید که در آن اطلاعات چندین شئ روندی یا چندین شئ مشتری گرفته شود اما نمی‌خواهید کوپلینگ (Coupling) را میان کلاس‌ها افزایش دهید. اگر به الگوهای طراحی مسلط باشید، می‌دانید که باید از الگوی Observer استفاده کنید.

هدف نویسنده از این مقاله، ساده سازی مفاهیم الگوهای طراحی و پیاده سازی آن با زبان Python است تا برنامه نویسان بتوانند از آن به عنوان منبعی کامل و خلاصه شده بهره بگیرند.

كلمات كليدي:

الگوهای طراحی – شئ گرایی – اجزاء طراحی – برنامه نویسی – الگوهای تکوینی – الگوهای ساختاری –
الگوهای رفتاری

فصل دوم: پیشینه پژوهش

در سال ۱۹۹۵ میلادی اریک گاما^۱، ریچارد هلم^۲، راف جانسن^۳ و جان ولیسایدز^۴ کتابی را تحت عنوان "الگوهای طراحی: عناصر برنامه‌های شی گرایی قابل استفاده مجدد"^۵ را منتشر کردند. این کتاب به عنوان پدر الگوهای طراحی شناخته می‌شود و جریان ساز این ایده بوده است. ۲۳ الگوی طراحی در این کتاب ذکر شده اند. نویسندهای کتاب، این ۲۳ الگو را بر حسب تجربه‌ی خود و به عنوان راهکارهایی برای مشکلات مرسوم زمان خود ارائه کردند. بعد از گذشت ۲۶ سال از انتشار این کتاب، الگوهای مختلفی ابداع شده است اما این ۲۳ الگو همچنان مهمترین الگوهای برنامه نویسی به حساب می‌آیند.

یکی دیگر از کتاب‌های خوب در این زمینه، "با کله برو تو الگوهای طراحی"^۶ به نوشتار اریک فریمن^۷، الیزابت رابسون^۸، برت بیتس^۹ و کتی سیرا^{۱۰} است که در سال ۲۰۰۴ منتشر گردید. ویرایش دوم این کتاب در سال ۲۰۲۰ نیز منتشر شده است. در این کتاب ۱۳ الگوی مهم بررسی شده است.

^۱ Eric Gamma

^۲ Richard Helm

^۳ Ralph Johnson

^۴ John Vlissides

^۵ Design Patterns: Elements of Reusable Object-Oriented Software

^۶ Head first Design Patterns

^۷ Eric Freeman

^۸ Elisabeth Robson

^۹ Bert Bates

^{۱۰} Kathy Sierra

فصل سوم: روش تحقیق

این مقاله یک مقاله مروری روایتی^{۱۱} است که در آن به ۲۳ الگوی طراحی در برنامه نویسی شئ گرایی می‌پردازد و سعی دارد تا تعاریف این مبحث را تا حد ممکن ساده سازی کرده و به فهم راحت تر مسئله کمک کند. در این مقاله هر الگو ابتدا تعریف و تشریح شده و سپس در مثالی آن الگو تشریح داده شده و سپس به زبان پایتون پیاده سازی می‌شود.

^{۱۱} Narrative Review Article

فصل چهارم: تجزیه و تحلیل یافته ها

الگو طراحی چیست؟

کرسیتوفر الکساندر، معمار مشهور اتریشی، الگوهای طراحی را این چنین توصیف می‌کند:
"هر الگو یک مشکل رایج و سپس ایده اصلی حل آن را ارائه می‌کند؛ به طوری که شما می‌توانید میلیون‌ها
بار از این راه حل استفاده کنید و هر بار به شکل جدیدی آن را انجام دهید"
درست است که الکساندر این حرف را در مورد معماری زده است اما حرف او در مورد الگوهای شئ گرایی نیز
صادق است. تنها فرق آن این است که ما در الگوهای طراحی از واژه‌های شئ و واسطه به جای در و دیوار
استفاده می‌کنیم.

به طور کلی هر الگو دارای چهار عنصر است:

- ۱ - نام الگو: عنوانی است که برای توصیف مشکل، راه حل و پیامدهای آن الگو در یک یا دو کلمه به کار گرفته می‌شود.
- ۲ - مشکل: مشکلی که الگو آن را حل می‌کند. بعضی مواقع شامل لیستی از شرایط است که در صورت وجود آن‌ها باید از الگوی مورد نظر استفاده کرد.
- ۳ - راه حل: راه حل عناصر تشکیل دهنده الگو، روابط، مسئولیت‌ها و همکاری‌ها را توصیف می‌کند. این راه حل قطعی و کاملاً مشخصی نیست و می‌توان آن را به شکل‌های مختلفی پیاده سازی کرد.
- ۴ - پیامد: پیامدها نتیجه‌ها و عوارض جانبی استفاده از الگو هستند. دانستن پیامد هر الگو می‌تواند به تصمیم گیری در انتخاب بهترین الگو به ما کمک کند.

به طور کلی سه دسته الگو طراحی تعریف می‌شود:

- ۱ - الگوهای تکوینی
- ۲ - الگوهای ساختاری
- ۳ - الگوهای رفتاری

الگوهای تکوینی^{۱۲}

الگوهای تکوینی فرآیند نمونه سازی را خلاصه می‌کنند. به کمک آن‌ها می‌توان یک سیستم را مستقل از چگونگی ساخت و ترکیب شیء‌ها و نمایش داده‌ها طراحی نمود. یک الگوی تکوینی از نوع کلاس، از ارث بری استفاده می‌کند تا کلاسی که از آن نمونه سازی شده است را تغییر دهد. در حالی یک الگوی تکوینی از نوع شیء، فرآیند نمونه سازی را به شیء دیگری واگذار می‌کند.

اهمیت الگوهای تکوینی هنگامی بیشتر می‌شود که سیستم بیشتر به ترکیب‌هایی از اشیاء وابسته است تا ارث بری‌های کلاس. به همین دلیل، تمرکز این الگوها به جای کدنویسی مجموعه‌ای از قوانین، روی تعریف مجموعه‌هایی از رفتارهای بنیادی است که خود می‌توانند تشکیل دهنده‌ی رفتارهای پیچیده باشند. بنابراین، ساخت اشیائی با رفتارهای خاص به چیزی بیشتر از نمونه سازی صرف از یک کلاس دارد.

این الگوها دو اصل دارند؛ اولاً همه‌ی آن‌ها اطلاعات کلاس‌هایی را که سیستم استفاده می‌کند، کپسوله سازی می‌کنند. دوماً، چگونگی ایجاد و کنار هم قرار گرفتن نمونه‌های این کلاس‌ها را نیز پنهان می‌کنند. الگوهای تکوینی به ما انعطاف پذیری بسیاری در اینکه چه چیزی ساخته شود، چه کسی آن را بسازد، چگونه و چه موقعی بسازد، می‌دهد. این الگوها به ما اجازه می‌دهند تا یک سیستم را با شیء‌های آماده ای که در ساختار و رفتار بسیار متفاوت هستند، پیکربندی کنیم. این پیکربندی می‌تواند ایستا^{۱۳} (در زمان کامپایل) و یا پویا^{۱۴} (در زمان اجرا) باشد.

^{۱۲} Creational Patterns

^{۱۳} Static

^{۱۴} Dynamic

الگوی Factory Method

تعریف

الگوی است که برای ساخت شئ از یک واسطه استفاده می‌کند و به کلاس زیرین اجازه می‌دهد تا خود تصمیم بگیرد چه شئ ای را تولید کند.

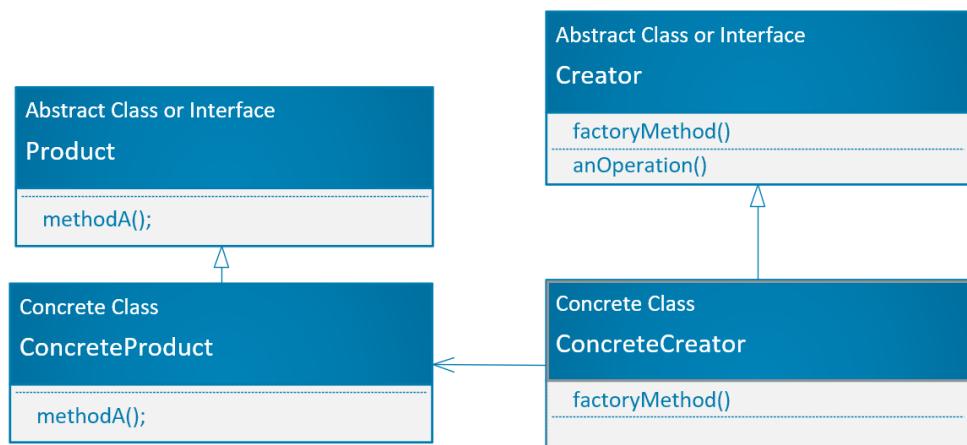
هدف

می‌خواهیم یک کلاس انتزاعی و یا یک واسطه تعریف کنیم که کلاسی که از آن ارث بری می‌کند قادر باشد فرآیند نمونه‌سازی از کلاس‌های دیگر را انجام دهد و با توجه به ورودی که به آن می‌دهیم، شئ ای از کلاس عینی مورد نظرمان را به ما برگرداند. داشتن یک کلاس انتزاعی به ما این اجازه را می‌دهد که فرآیند های مختلفی برای نمونه سازی تعریف کنیم. به متدهای که در آن کلاس وظیفه نمونه سازی را انجام می‌دهد، متدهای کارخانه گفته می‌شود.

موارد استفاده

- کلاس از نوع شئ ای که باید بسازد آگاه نیست.
- کلاس می‌خواهد زیرکلاس، خود نوع شئ ای را که می‌سازد تعیین کند.
- کلاس مسئولیت خود را به یکی از چند زیرکلاس کمکی واگذار کرده است و شما می‌خواهید اطلاعات اینکه این کار به کدام زیرکلاس واگذار شده است را محلی کنید.

ساختار



Class diagram in Factory Method Design Pattern - شکل ۱

:Creator کلاس انتزاعی یا واسطه

یک کلاس انتزاعی یا یک واسطه است که متده است که آبجکت مناسب را تولید می کند در آنجا اعلام شده است.
همچنین این کلاس می تواند سایر متدهای لازم را نیز اعلام کند.

:ConcreteCreator کلاس

کلاس عینی است که از کلاس Creator ارث بری می کند و متدهایی را که در آن اعلام شده اند، تعریف و پیاده سازی می کند. متده تولید کننده شئ مناسب در این کلاس پیاده سازی می شود و یک نمونه از کلاس ConcreteProduct را، که از نوع کلاس Product است، بر می گرداند. پس هنگامی که نیاز به تشخیص و تولید نوع مناسب شئ داریم، باید یک نمونه از این کلاس بسازیم.

:Product کلاس

یک کلاس انتزاعی یا یک واسطه است که متدها و متغیرهای لازم و مشترک هر نوع محصول در آن اعلام شده اند. به طور مثال کلاس Product می تواند کلاس پستانداران و کلاس ConcreteProduct یک نوع پستاندار مانند گاو باشد.

:ConcreteProduct کلاس

کلاسی است که از کلاس Product ارث بری می کند و تمام متدها و متغیرهای لازم را تعریف و پیاده سازی می کند.

فرض کنید یک شرکت خودروسازی به نام Medi در کانادا وجود دارد. این شرکت در حال حاضر سه خودرو M^۱, M^۲ و M^{۲X} به مشتریان خود عرضه می‌کند. هر کدام از این خودرو‌ها دارای مشخصات متفاوتی هستند. خودرو M^۱ یک کراس اوور با موتور ۲ لیتری است که ۱۹۰ اسب بخار توان و ۲۵۲ نیوتون متر گشتاور دارد. خودرو M^۲ یک سدان است با موتور ۱.۵ لیتری که ۱۴۰ اسب بخار توان و ۱۵۰ نیوتون متر گشتاور دارد. خودرو M^{۲X} نیز نسخه هاچ بک M^۲ است. این خودرو‌ها به ترتیب به قیمت ۲۶ هزار دلار، ۲۰ هزار دلار و ۲۲ هزار دلار به فروش می‌رسند. برنامه فروش این شرکت به این شکل است که ابتدا مشتری، خودروی مورد نظر خود را سفارش بدهد و سپس خودرو برای او ساخته شود. همچنین این شرکت تصمیم دارد شعبه‌ای در ایران باز کرده و مشخصات محصولات خود را با توجه به نیاز جامعه تغییر دهد.

حال می‌خواهیم سیستمی برای این شرکت طراحی کنیم. ابتدا باید به ویژگی‌های کلیدی این شرکت دقت کنیم:

۱. هر خودرو ویژگی‌های خاص خود را دارد ولی همه‌ی آنها خودرو هستند.
۲. شرکت از قبل نمی‌داند که مشتری کدام خودرو را سفارش خواهد داد و مشتری در لحظه خودرو خود را انتخاب می‌کند.
۳. هر خودروی این شرکت در ایران با تغییر در مشخصات فنی اما به همان روش فروخته می‌شود.

ابتدا کلاس Medi را می‌سازیم که کلاس اصلی شرکت است:

```
class Medi:
    registrationNumber = 598632
    name = "Medi Motors .Co"

    def order(carName):
        pass
```

شکل ۲ – کلاس کمپانی

سپس کلاس انتزاعی MediCars را طراحی می‌کنیم که کلاس‌های خودرو‌ها باید از آن ارث بری و متدها و ویژگی‌های آن را پیاده سازی کنند. متدهای سازنده این کلاس به ترتیب حجم موتور، گشتاور، اسب بخار و قیمت را دریافت و ذخیره می‌کند. همچنین این کلاس متدهای specs را پیاده سازی می‌کند که مشخصات خودرو را چاپ می‌کند. دو متدهای build و deliver هم وجود دارد که به ترتیب وظیفه ساختن خودرو با مشخصات مورد نظر و تحویل آن را بر عهده دارند.

```

class MediCars(ABC):
    type: str
    engineSize: float
    torque: int
    horsepower: int
    price: int

    def __init__(self, engineSize: float, torque: int, horsepower: int, price: int):
        self.engineSize = engineSize
        self.torque = torque
        self.horsepower = horsepower
        self.price = price

    def specs(self):
        print("Specs: \n engineSize:{0} \n horsepower:{1} \n torque:{2} \n price:{3}".format(
            self.engineSize, self.horsepower, self.torque, self.price))

    @abstractmethod
    def build(self):
        return

    @abstractmethod
    def deliver(self):
        return

```

شکل ۳ - کلاس انتزاعی خودرو *MediCars*

```

class M1(MediCars):
    type: "Crossover"

    def build(self):
        print("Your M1 is being built")

    def deliver(self):
        print("Here's your brand new M1 ^^")
class M2(MediCars):
    type: "Sedan"

    def build(self):
        print("M2 is being built")

    def deliver(self):
        print("Here's your brand new M2 ^^")

class M2X(MediCars):
    type: "Hatchback"

    def build(self):
        print("M2X is being built")

    def deliver(self):
        print("Here's your brand new M2X ^^")

```

شکل ۴ - پیاده سازی کلاس های خودرو های *M1,M2,M2X*

حال باید متده *order* را در کلاس *Medi* طوری پیاده سازی کنیم که مشتری نام خودرو را وارد کرده و شئ ای از آن خودرو ساخته و برگردانده شود (شرکت آن خودرو را برای او بسازد). برای اینکار باید یک عبارت

شرطی بنویسیم و چک کنیم که به طور مثال اگر نام خودرو وارد شده توسط مشتری M¹ بود، شی M¹ ساخته شود و برگردانده شود.

```
class Medi:
    registrationNumber = 598632
    name = "Medi Motors Co"

    def order(self, carName):
        if(carName == "M1"):
            self.car = M1(2, 252, 180, 26000)
        elif(carName == "M2"):
            self.car = M2(1.5, 150, 140, 20000)
        else:
            self.car = M2X(1.5, 150, 140, 22000)

    self.car.build()
    self.car.deliver()
```

شکل ۵ – پیاده سازی متدهای سفارش خودرو در کلاس Medi

```
medi = Medi()
car1 = medi.order("M2")
```

شکل ۶ – سفارش خودرو

```
M2 is being built
Here's your brand new M2 ^^
```

شکل ۷ – خروجی کد بالا

اما این کار با ایراداتی همراه است:

۱. اگر به تعداد خودرو ها افزوده شود، مجبور به ویرایش مکرر متدهای order هستیم.
۲. پیاده سازی شعبه ایران این شرکت، پردردسر خواهد بود. زیرا به طور مثال کاربر باید نام خودرو را به شکل ”M¹-iran“ وارد کند و ما در عبارت شرطی در متدهای order تاک تک این رشته ها را چک کنیم. این کار دو ایراد بزرگ دارد: ۱- نگهداری کد را به شدت سخت می کند. ۲- به دلیل چک کردن های مکرر و زیاد، تعداد اجرای عبارات شرطی را افزایش می دهیم.

پس کد ما باز برای گسترش و بسته برای ویرایش^{۱۵} نیست.

راه حل، استفاده از الگوی Factory Method است. این الگو فرآیند نمونه سازی را در یک یا چند کلاس کپسوله سازی می کند. در مثال ما، فرآیند نمونه سازی خودرو ها در چند کلاس کپسوله می شود زیرا دو کارخانه وجود دارد. یک کارخانه در کانادا و یک کارخانه در ایران که مشخصات فنی خودرو های تولید شده

^{۱۵}open for extension, closed for modification

در آن متفاوت است. به طور مثال هر کدام از آن کارخانه‌ها خودروی M^۱ را با مشخصات خاص مناسب آن کشور تولید می‌کنند. در کنار تمایز آن‌ها، وجه شباهتی نیز وجود دارد؛ هر دو کارخانه خودرو تولید می‌کنند. بنابراین باید یک کلاس انتزاعی کارخانه داشته باشیم که هر دو کارخانه‌ی کانادا و ایران از آن ارثبری می‌کنند.

```
class MediCarFactory(ABC):
    @abstractmethod
    def createCar(self, carName):
        return

class CanadaFactory(MediCarFactory):
    def createCar(self, carName):
        if(carName == "M1"):
            return M1(2, 252, 180, 26000)
        elif(carName == "M2"):
            return M2(1.5, 150, 140, 20000)
        else:
            return M2X(1.5, 150, 140, 22000)

class IranFactory(MediCarFactory):
    def createCar(self, carName):
        if(carName == "M1"):
            return M1(1.5, 180, 140, 56000000)
        elif(carName == "M2"):
            return M2(1.5, 150, 100, 46000000)
        else:
            return M2X(1.5, 135, 150, 50000000)
```

شکل ۸ - کلاس کارخانه‌های ایران و کانادا

اکنون دو کلاس createCar و CanadaFactory از MediCarFactory که متدهای آن‌ها شیء مناسب را ساخته و برمی‌گرداند. حال باید متدهای order در کلاس Medi را بازنویسی کنیم:

```
class Medi:
    registrationNumber = 598632
    name = "Medi Motors .Co"

    def __init__(self, factory: MediCarFactory):
        self.factory = factory

    def order(self, carName):
        self.car = self.factory.createCar(carName)
        self.car.build()
        self.car.deliver()
        self.car.specs()
```

شکل ۹ - بازنویسی متدهای order

در متدهای سازنده کلاس Medi ابتدا یک شیء از جنس MediCarFactory داده می‌شود که با توجه به سناریو موجود، شیء ای از کلاس IranFactory یا CanadaFactory خواهد بود و در متغیر factory ذخیره می‌شود. در متدهای order نیز با بهره گیری از چندین متد createCar از کلاس ذخیره شده در متغیر factory صدا

زده می‌شود که شئ مورد نظر را برمی‌گرداند و در متغیر car ذخیره می‌شود. سپس سه متد deliver, build و specs از شئ ذخیره شده در car صدا زده می‌شود.

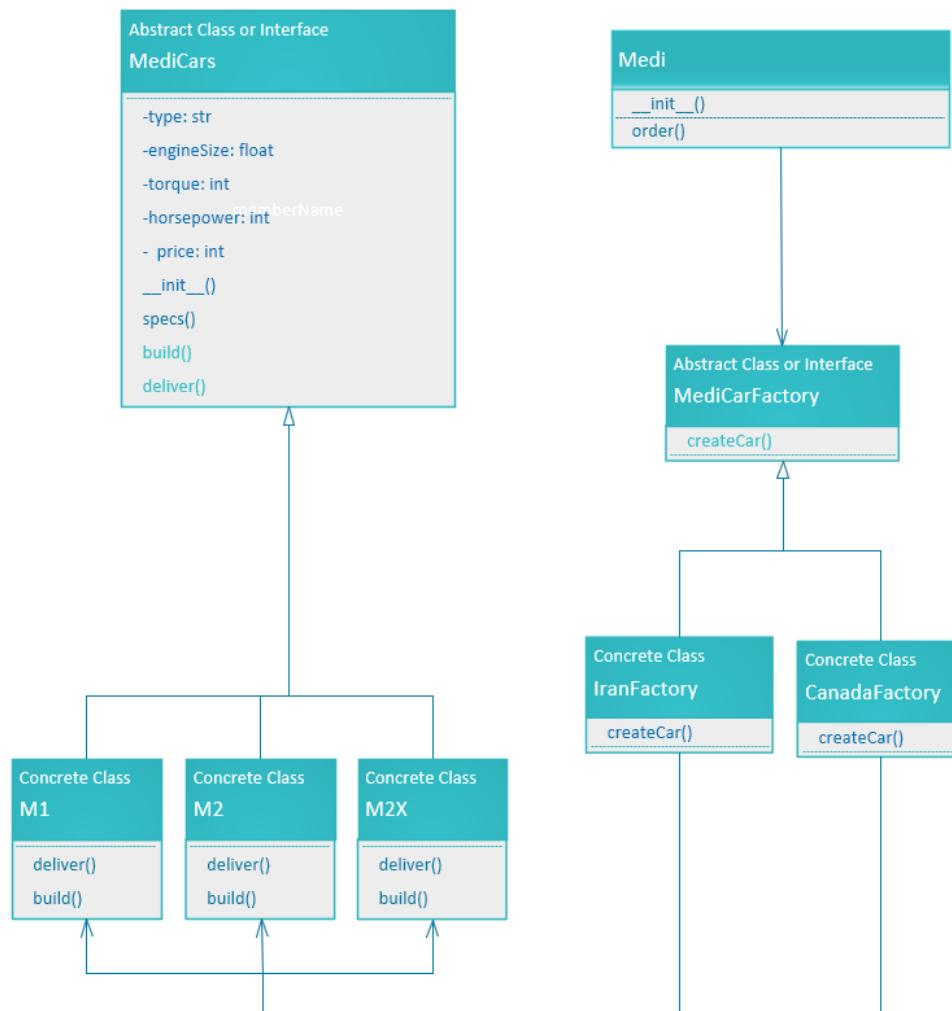
```
IranFactory = Medi(IranFactory())
IranFactory.order("M1")
```

شکل ۱۰ - سفارش خودرو M1 از کارخانه می‌ایران

```
Your M1 is being built
Here's your brand new M1 ^^
Specs:
engineSize:1.5
horsepower:140
torque:1.5
price:56000000
```

شکل ۱۱ - خروجی کد بالا

ساختار جدید سیستم این شرکت به صورت زیر خواهد بود:



شکل ۱۲ - طراحی جدید سیستم class diagram

الگوی Abstract Factory

تعریف

الگوی است که یک واسطه برای ساخت خانواده هایی از اشیاء مرتبط به هم ارائه می دهد بدون آنکه کلاس عینی آن شئ مشخص باشد.

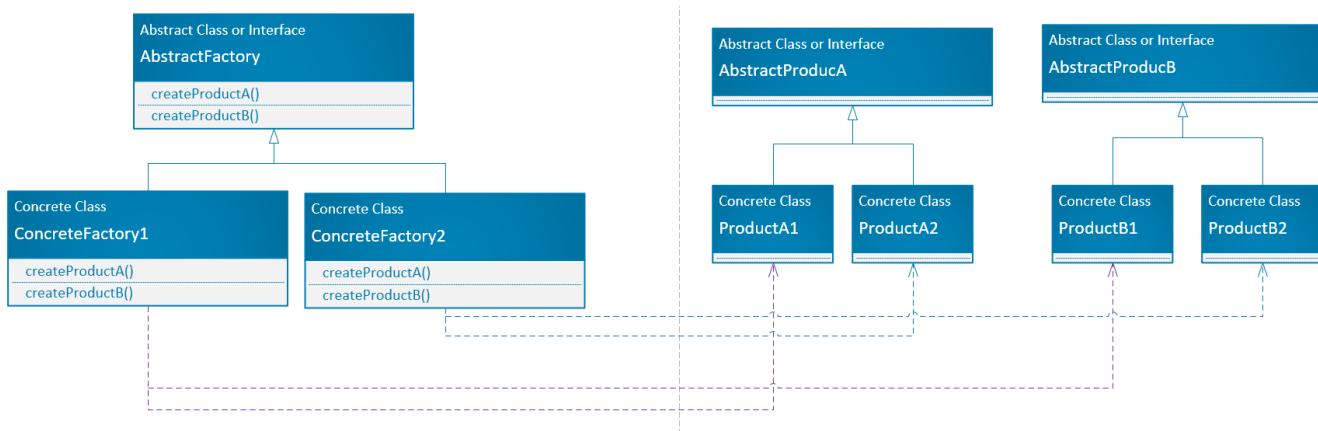
هدف

می خواهیم یک کلاس انتزاعی و یا یک واسطه تعریف کنیم که کلاسی که از آن ارث بری می کند قادر باشد فرآیند نمونه سازی از کلاس های دیگر را انجام دهد. فرق این الگو با الگوی Factory Method این است که اینبار گروهی از اشیاء ساخته می شوند و این اشیاء بهم مرتبط هستند و با هم یک محصول را تشکیل می دهند. به طور مثال شئ های فرمان، چرخ، موتور و غیره می توانند محصول خود را تشکیل دهند. در واقع در الگوی Abstract Factory چندین متده کارخانه وجود دارد.

موارد استفاده

- سیستم نباید به چگونگی تولید، ترکیب و نمایش محصولات خود وابسته باشد.
- سیستم باید با یکی از چندین خانواده محصولات پیکربندی شود.
- خانواده ای از اشیاء باید با هم استفاده شوند و شما باید این محدودیت را وضع کنید.
- می خواهید اطلاعات ساخت یک محصول و ترکیب اجزاء آن کپسوله سازی شود و شما از آن بی اطلاع باشید.

ساختار



شکل ۱۳ – ساختار الگوی Abstract Factory

کلاس انتزاعی یا واسطه :Abstract Factory

یک کلاس انتزاعی یا یک واسطه است که برای عملیاتی که اشیاء AbstractProduct تولید می‌کنند، تعریف شده است.

کلاس ConcreteFactory

کلاس عینی است که عملیاتی که اشیاء AbstractProduct را می‌سازد، را پیاده سازی می‌کند.

کلاس AbstractProduct

یک کلاس انتزاعی یا یک واسطه است که برای یک نوع از شئ product تعریف شده است.

کلاس ProductA^۱

کلاسی است که از کلاس AbstractProduct ارث بری می‌کند.

مثال

می‌خواهیم یک انتخاب‌کننده تاریخ^{۱۶} برای دو پلتفرم وب و اندروید بسازیم که برنامه نویسان بتوانند به راحتی از هر کدام آنها استفاده کنند. هر انتخاب‌کننده تاریخ از دو ماژول باکس و اسکرولر تشکیل شده است. هر یک از این ماژول‌ها یکبار به زبان java و یکبار به زبان javascript نوشته شده‌اند. ما باید سیستمی طراحی کنیم که توسعه دهنده‌گان از انتخاب‌کننده تاریخ پلتفرم مورد نظر خود استفاده کنند.

ابتدا باید به ویژگی‌های کلیدی این سیستم دقت کنیم:

۱. یک انتخاب‌کننده از دارای دو ماژول باکس و اسکرولر تشکیل شده است.
۲. ماژول باکس و اسکرولر برای اندروید به زبان java و برای وب به زبان javascript نوشته شده‌اند.
۳. ماژول باکس و اسکرولر به تنها‌یی استفاده نمی‌شوند.

با توجه به توضیحات بالا، الگوی انتخابی ما Abstract Method است.

^{۱۶} Date Picker

پس ابتدا یک کلاس انتزاعی به نام DatePickerFactory که دارای دو متده است. createDateBox() و createDateScroller()

```
class DatePickerFactory(ABC):
    @abstractmethod
    def createDateBox(self):
        return

    @abstractmethod
    def createDateScroller(self):
        return
```

شکل ۱۴ - پیاده سازی کلاس DatePickerCreator

سپس دو کلاس انتزاعی DateBox و DateScrollerer را می سازیم. این دو کلاس شمای کلی ماثول های اسکرولر و باکس را مشخص می کنند.

```
class DateBox(ABC):
    @abstractmethod
    def dateBox(self):
        return

class DateScroller(ABC):
    @abstractmethod
    def dateScroller(self):
        return
```

شکل ۱۵ - پیاده سازی کلاس های DateBox و DateScroller

حال نوبت کلاس های عینی JSDateBox ، JavaDateBox و JavaDateScroller است که دو کلاس اول از کلاس DateBox و دو کلاس دوم از کلاس DateScroller ارث بری می کنند.

```
class JSDateBox(DateBox):
    def dateBox(self):
        print("a datebox is created in javascript")

class JavaDateBox(DateBox):
    def dateBox(self):
        print("a datebox is created in java")

class JSDateScroller(DateScroller):
    def dateScroller(self):
        print("a date Scroller is created in javascript")

class JavaDateScroller(DateScroller):
    def dateScroller(self):
        print("a date Scroller is created in java")
```

شکل ۱۶ - پیاده سازی کلاس های عینی JavaDateScroller ، JSDateScroller ، JavaDateBox و JSDateBox

سپس دو کلاس عینی JavaDatePickerFactory و JSDatePickerController را پیاده سازی می کنیم. این دو کلاس از کلاس DatePickerFactory ارث بری می کنند. در این کلاس هاست که شئ هایی از جنس DateBox و DateScroller ساخته می شود.

```

class JSDatePickerFactory(DatePickerFactory):
    def createDateBox(self):
        self.DateBox = JSDateBox()

    def createDateScroller(self):
        self.DateBox = JSDateScroller()

class JavaDatePickerFactory(DatePickerFactory):
    def createDateBox(self):
        self.DateBox = JavaDateBox()
        self.DateBox.dateBox()

    def createDateScroller(self):
        self.DateBox = JavaDateScroller()
        self.DateBox.dateScroller()

```

شکل ۱۷ - پیاده سازی کلاس های DatePickerFactory و JSDatePickerFactory

و در انتها انتخاب کننده تاریخ جاوا را می سازیم:

```

javadatepicker = JavaDatePickerFactory()
javadatepicker.createDateBox()
javadatepicker.createDateScroller()

```

شکل ۱۸ - ساخت انتخاب کننده تاریخ جاوا - کد نهایی

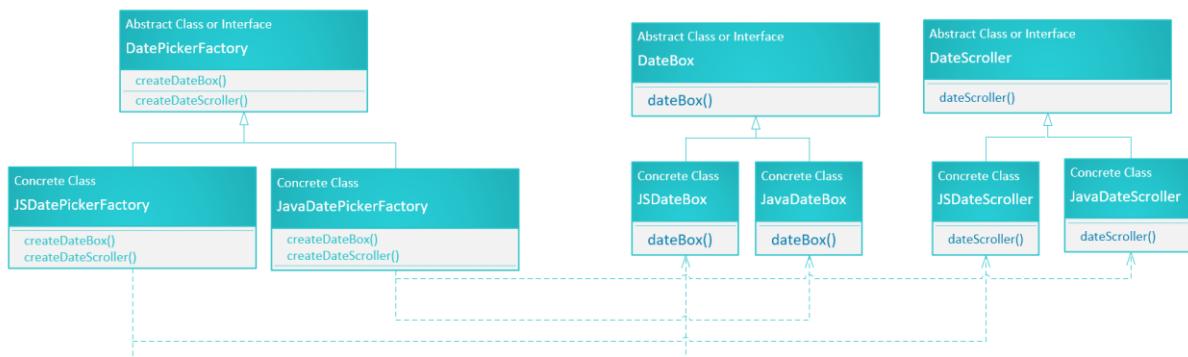
```

a datebox is created in java
a date Scroller is created in java

```

شکل ۱۹ - خروجی کد بالا

پس ساختار این سیستم به صورت زیر خواهد بود:



شکل ۲۰ - طراحی نهایی class class

الگوی Prototype

تعریف

الگوی Prototype انواع اشیائی که باید ساخته شوند را با استفاده از یک نمونه اولیه مشخص می‌کند و اشیاء جدید را با کپی کردن^{۱۷} این نمونه اولیه می‌سازد.

هدف

فرض کنید می خواهیم یک شئ را به طور کامل کپی کنیم. ابتدا باید یک شئ جدید از همان کلاس بسازیم. سپس تمامی ویژگی های شئ مورد نظر را در شئ جدید کپی کنیم. اما اینکار همیشه ممکن نیست زیرا ممکن است آن ویژگی های آن شئ شخصی باشند و ما نتوانیم به آن دسترسی داشته باشیم. همچنین باید کلاس شئ مورد نظر را بشناسیم و این کد ما را به آن کلاس وابسته می کند. حتی شناختن واسط آن کلاس به ما کمکی نمی کند زیرا باید از پیاده سازی های متدها نیز آگاه باشیم.

ایده اصلی الگوی Prototype این است که فرآیند کپی کردن یک شئ را، بدون آنکه ما را به کلاس آن شئ وابسته کند، ممکن می سازد. این الگو وظیفه کپی کردن را به شئ ای که می خواهیم از آن کپی کنیم محول می کند و این کار با استفاده از ارائه ای یک واسط مشترک برای تمامی اشیائی که قابلیت کپی شدن را دارند، انجام می دهد. این واسط به شما اجازه می دهد یک شئ را بدون وابسته شدن به کلاس آن شی انجام دهید. معمولاً تنها یک متده کپی در این واسط وجود دارد.

پیاده سازی های متده کپی در کلاس های مختلف بسیار مشابه هستند. بدین صورت که یک شی از کلاس فعلی ایجاد می کنند و تمام مقادیر شی قدیمی را به شی جدید منتقل می کند. حتی می توانید ویژگی های خصوصی را کپی کنید؛ زیرا اکثر زبان های برنامه نویسی به اشیاء اجازه می دهند به فیلدهای خصوصی دیگر اشیاء متعلق به همان کلاس دسترسی پیدا کنند..

شئ ای که از کپی شدن پشتیبانی می کند، نمونه اولیه نامیده می شود. هنگامی که اشیاء شما دارای دهها فیلد و صدها پیکربندی ممکن است، کپی کردن آنها می تواند به عنوان جایگزینی برای کلاس بندی فرعی باشد.

^{۱۷} Clone

موارد استفاده

هنگامی از الگوی Prototype استفاده کنید که یک سیستم مستقل از محصولات خود ساخته، ترکیب، و نمایش داده می‌شود. همچنین در موارد زیر نیز می‌توانید از این الگو بهره‌مند شوید:

- هنگامی که کلاس‌هایی که باید نمونه‌سازی شوند در زمان اجرا مشخص می‌شوند. به طور مثال بارگذاری پویا

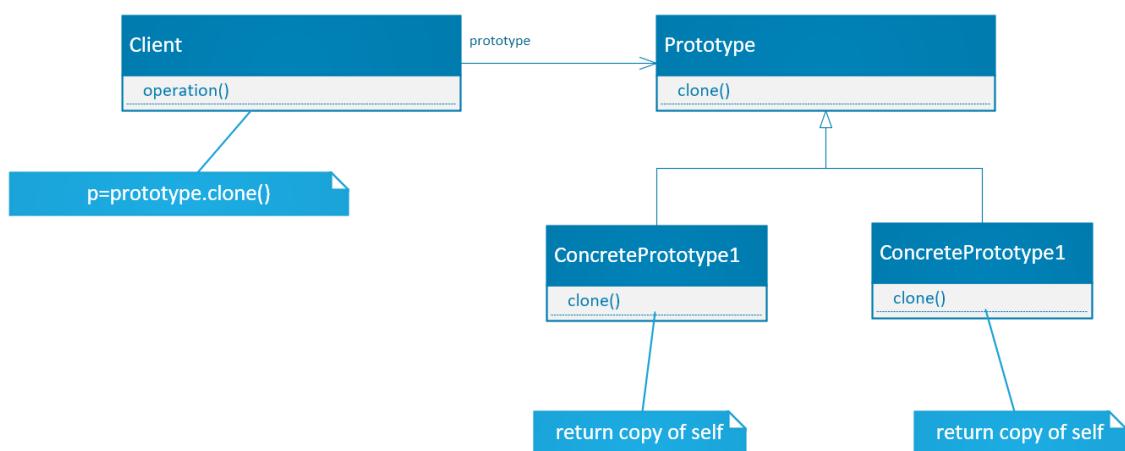
یا

- هنگامی که می‌خواهید از ایجاد یک سلسله مراتب کلاس کارخانه‌ها که با سلسله مراتب محصولات موازی است، اجتناب کنید.

یا

- هنگامی که نمونه‌های یک کلاس می‌تواند یک حالت از تنها چندین ترکیب محدود از حالت را داشته باشد. شاید ایجاد تعدادی از نمونه‌های اولیه‌ی متناظر و شبیه سازی آن‌ها، به جای نمونه‌سازی دستی از کلاس و هر بار با حالت مناسب، راحت‌تر باشد.

ساختار



class diagram of Prototype pattern - ۲۱

Prototype

یک واسط برای شبیه‌سازی خود اعلام می‌کند

ConcretePrototype

عملیات موردنیاز برای شبیه‌سازی خود را پیاده‌سازی می‌کند

Client

یک شیء جدید با درخواست شبیه‌سازی یک نمونه‌اولیه ایجاد می‌کند

مثال

فرض کنید که ما یک کلاس ScreenSaver داریم که واسطی را برای ایجاد محافظت صفحه‌های مختلف ارائه می‌کند. این کلاس آرایه از رنگ‌ها و متدهای برای طراحی آن می‌پذیرد. محافظه‌سازی صفحه ای با طرح مینیمالیست و رنگ‌های سبز و زرد ساخته ایم (شیء ای با این مشخصات از کلاس ScreenSaver ایجاد کردیم). حال ما می‌خواهیم یک محافظه‌سازی دیگر با همان طرح اما با رنگ‌های سبز و زرد و آبی، طراحی کنیم. پس باید از شیء محافظه‌سازی ابتدایی یک کپی تهیه کنیم و سپس رنگ آبی را به رنگ‌های موجود بیفزاییم. شاید بتوانیم بگوییم دلیل آنکه این کار را بدون استفاده از کلاس بندی انجام می‌دهیم، آن است که تنها قرار است یک شیء از این محافظه‌سازی ساخته شود و نمی‌خواهیم با کلاس‌بندی سیستم را بدون دلیل گسترش کنیم. با توجه با توضیحات داده شده، باید از الگوی Prototype استفاده کنیم.

در پایتون برای کپی کردن یک شیء، مازول کپی فراهم آورده شده و ما می‌توانیم به راحتی از آن استفاده کنیم.

پیاده‌سازی سیستم به صورت زیر خواهد بود:

```
class Prototype():
    def clone(self):
        return copy.deepcopy(self)

class ScreenSaver(Prototype):
    _design = None
    _colors: List[str] = None

    def __init__(self, designFunction, colors):
        self._design = designFunction
        self._colors = colors

    def addColor(self, color):
        self._colors.append(color)

    def draw(self):
        self._design(self._colors)
```

شکل ۲۲ - پیاده‌سازی کلاس‌های ScreenSaver و Prototype

برنامه را تست و اجرا می کنیم:

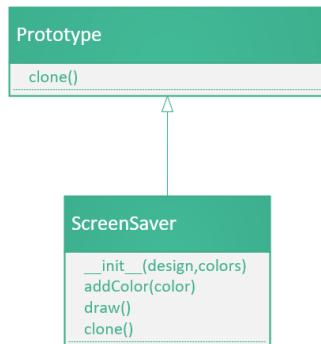
```
screenSaver1 = ScreenSaver(lambda colors: [print(
    f"The design is made with the color {i}") for i in colors], ["green", "yellow"])
screenSaver1.draw()
print("-----")
screenSaver2 = screenSaver1.clone()
screenSaver2.addColor("blue")
screenSaver2.draw()
```

شکل ۲۳ - تست و اجرا برنامه

```
The design is made with the color green
The design is made with the color yellow
-----
The design is made with the color green
The design is made with the color yellow
The design is made with the color blue
```

شکل ۲۴ - خروجی کد بالا

ساختار برنامه به شکل زیر است:



شکل ۲۵ - class diagram برنامه

الگوی Singleton

تعریف

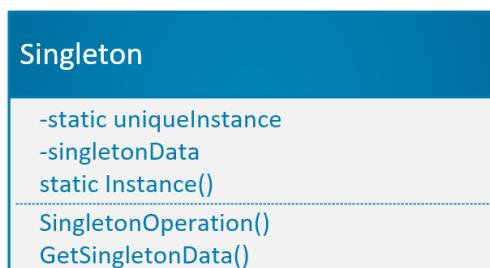
الگوی است که تنها اجازه‌ی ساخت یک نمونه از یک کلاس را می‌دهد و دسترسی همگانی^{۱۸} به آن را ارائه می‌کند.

هدف

می‌خواهیم یک کلاس را طوری پیاده سازی کنیم که تنها یکبار قابلیت نمونه سازی داشته باشد و در صورت تکرار نمونه سازی، همان نمونه‌ی قبلی را برگرداند. اینکار با استفاده از خصوصی کردن^{۱۹} تابع سازنده‌ی کلاس^{۲۰} و متده استاتیک انجام می‌شود.

موارد استفاده

- تنها باید یک نمونه از یک کلاس وجود داشته باشد و آن نمونه در نقطه دسترسی مشهودی قابل دسترسی باشد.
- زمانی که نمونه یگانه باید با قابل توسعه باشد و کلاینت‌ها باید بتوانند از یک نمونه توسعه یافته، بدون تغییر کد خود استفاده کنند.



class diagram of Singelton Design Pattern - ۲۶ شکل

ساختار

کلاس Singleton

کلاسی است که می‌توان تنها یک نمونه از آن ساخت.

^{۱۸} Public

^{۱۹} Private

^{۲۰} Constructor

مثال

فرض کنید که کامپیووتری در اختیار داریم که تنها یک درایو نوری دارد. ما می‌خواهیم سیستمی طراحی کنیم که درایو نوری، سی دی یا دی وی را بخواند. برای خواندن سی دی بعدی باید درایو نوری خالی باشد.

ابتدا باید به ویژگی‌های کلیدی این سیستم دقت کنیم:

۱. تنها یک درایو نوری وجود دارد.
۲. باید چک کنیم هنگام گذاشتن سی دی بعدی درایو نوری خالی باشد.
۳. درایو نوری برای همه در دسترس باشد.

با توجه به توضیحات بالا، می‌توانیم از الگوی Singleton استفاده کنیم. واقعیت این است که این الگو مخالفان بسیاری دارد. یکی از ایراداتی که به این الگو وارد است این است که شاید برنامه نویس در شناخت نیازمندی‌های سیستم دچار اشتباه شده و در آینده به بیش از یک نمونه از کلاس مذکور باشد. با این حال، برای اهداف آموزشی به این الگو می‌پردازیم.

چون در پایتون امکان خصوصی کردن متدهای وجود ندارد ، باید یک متابلاس برای اینکار ایجاد کنیم. برای پیاده سازی این الگو در پایتون دو راه وجود دارد. راه اول آسان تر است اما در پردازش موازی و یا چندپردازشی درست عمل نمی‌کند.

راه اول

ابتدا یک متابلاس SingletonMeta می‌سازیم که در متدهای `call` و `__init__` بررسی می‌کنیم که آیا نمونه‌ای از این کلاس ساخته شده است یا خیر. اگر نمونه‌ای موجود بود، برگردانده و اگر نمونه‌ای وجود نداشت، یک نمونه ساخته و برگردانده شود. سپس کلاس OpticalDrive را می‌سازیم که دارای یک متغیر `IsEmpty` و دو متدهای `read` و `eject` است.

```

class SingletonMeta(type):
    _classInstances = {}

    def __call__(cls, *args, **kwargs):
        if(cls not in cls._classInstances):
            instance = super().__call__(*args, **kwargs)
            cls._classInstances[cls] = instance

        return cls._classInstances[cls]

class OpticalDrive(metaclass=SingletonMeta):
    isEmpty = True

    def read(self, CD):
        if(self.isEmpty):
            self.isEmpty = False
            print(CD)
        else:
            print("The drive is full.")

    def eject():
        self.isEmpty = True

```

شکل ۲۷ – پیاده سازی کلاس *OpticalDrive* و *SingeltonMeta*

حال دو نمونه از این کلاس می سازیم:

```

drive1 = OpticalDrive()
drive1.read("Ahang ghadimi")

drive2 = OpticalDrive()
drive2.read("Ahang Jadid")

```

شکل ۲۸ – ساخت دو نمونه از کلاس *opticalDrive*

```

Ahang ghadimi
The drive is full.

```

شکل ۲۹ – خروجی کد بالا

همانطور که مشاهده می شود، تنها یک شئ ساخته شده است. زیرا هنگامی که سعی داشتیم یک شئ دیگر بسازیم و یک سی دی دیگر را بخوانیم، پیام "دراایو پر است" نمایش داده شد. اگر دو متغیر *drive1* و *drive2*^۱ را چاپ کنیم، میبینیم که هر دو به یک شئ اشاره می کنند.

```

<__main__.OpticalDrive object at 0x00000259529C1130>
<__main__.OpticalDrive object at 0x00000259529C1130>

```

شکل ۳۰ – خروجی چاپ دو متغیر *drive1* و *drive2*

اما اشکال این پیاده سازی اینجاست که اگر ما در نخ های ^{۲۱} مختلف از این کلاس نمونه سازی انجام دهیم، این امکان وجود دارد که نمونه ها مختلف باشند. زیرا ممکن است دو نخ همزمان اجرا شوند و دو نمونه مختلف را تولید کنند.

^۱ Threads

```

def createObject():
    drive = OpticalDrive()
    print(f"\n{drive}")

if(__name__ == "__main__"):
    thread1 = Thread(target=createObject, args=())
    thread2 = Thread(target=createObject, args=())
    thread1.start()
    thread2.start()
    thread1.join()
    thread2.join()

```

شکل ۳۱ – اجرای کد بالا به صورت چند نخی

راه دوم

پس باید این الگو را طوری پیاده سازی کنیم که در نخ های مختلف نیز به درستی عمل کند. برای اینکار باید یک قفل تعریف کنیم تا در لحظه فقط یک نخ بتواند وارد بخش نمونه سازی شود.

```

class SingletonMeta(type):
    _classInstances = {}
    _lock: Lock = Lock()

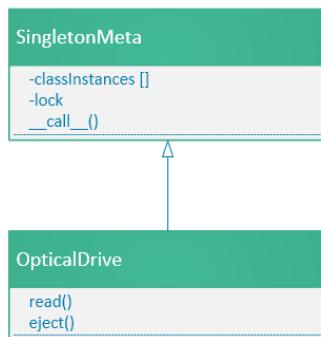
    def __call__(cls, *args, **kwargs):
        with cls._lock:
            if(cls not in cls._classInstances):
                instance = super().__call__(*args, **kwargs)
                cls._classInstances[cls] = instance

        return cls._classInstances[cls]

```

شکل ۳۲ – پیاده سازی الگوی Singleton به صورت Thread-proof

همچنین ساختار جدید سیستم به صورت زیر است:



شکل ۳۳ – سیستم پیاده سازی شده class diagram

الگوی **Builder**

تعریف

الگوی است که فرآیند ساخت یک شئ پیچیده را از نمایش آن جدا می‌کند تا بتوان با همان فرآیند ساخت شئ، نمایش های گوناگونی را نیز ایجاد کرد.

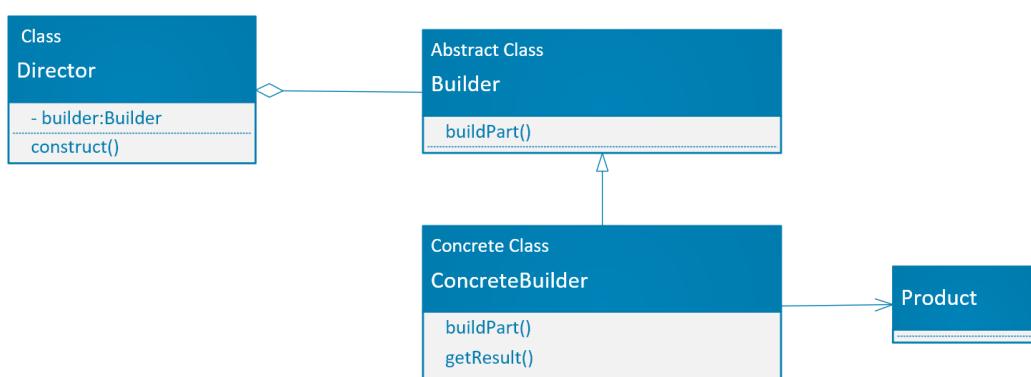
هدف

می‌خواهیم نحوه ساخت یک شئ پیچیده را مرحله به مرحله با ساخت شئ های کوچک‌تر انجام دهیم تا کنترل بیشتری بر روی چگونگی ساخت آن شئ داشته باشیم. همچنین بتوانیم در لحظه تصمیم بگیریم که آن شئ پیچیده چه شئ ای باشد و چگونه ساخته شود.

موارد استفاده

- الگوریتم ساخت یک شئ پیچیده باید مجزا از دیگر بخش‌های کد که وظیفه نمایش و سرهم‌بندی اشیاء را دارند، باشد.
- فرآیند ساخت باید اجازه ایجاد نمایش‌های مختلف برای شئ ساخته شده را بدهد.

ساختار



class diagram of Builder Pattern – ۳۴

کلاس Director

کلاسی است که در خود یک شئ از Builder را ذخیره می کند و با استفاده از یک شئ پیچیده را تولید می کند. وجود این کلاس ضروری نیست اما باعث افزایش خوانایی کد و طراحی سیستم می شود.

کلاس انتزاعی Builder

کلاسی انتزاعی است که چهارچوبی برای ساخت شئ product تعریف می کند

کلاس عینی ConcreteBuilder

کلاسی است که وظیفه ساخت و سرهم بندی و نمایش شئ ها را برعهده دارد

کلاس Product

شئ های نهایی هستند که کلاس ConcreteBuilder آن ها را می سازند

مثال

فرض کنید می خواهیم اپلیکیشنی طراحی کنیم که متنی را به زبان انگلیسی به عنوان ورودی می گیرد. این برنامه چندین کار انجام می دهد:

- متن را به زبان فارسی یا ایتالیایی ترجمه کرده و آن را به صورت متن نمایش دهد.
- ترجمه آن متن را بخواند و یک فایل صوتی را خروجی کند.

می دانیم که دو زبان مقصد وجود دارد که دو عملیات بالا باید برای هر زبان به صورت جداگانه پیاده سازی شود. پس باید یک کلاس انتزاعی داشته باشیم که هر زبان از آن ارث بری کند.

```
class TargetLanguage(ABC):
    @abstractmethod
    def translate():
        pass

    @abstractmethod
    def narrate():
        pass
```

شکل ۳۵ - کلاس TargetLanguage

خروجی دو متدهای `Text` و `Audio` به ترتیب متن و صدا است که باید برای آنها نیز کلاس جدآگانه تشکیل دهیم

```
class Text:  
    text: str  
  
    def __init__(self, txt):  
        self.text = txt  
  
    def download(self):  
        return self.text  
  
  
class Audio:  
    filename: str  
    fileformat: str  
    narrator: str  
  
    def __init__(self, name, fileformat, narrator):  
        self.filename = name  
        self.fileformat = fileformat  
        self.narrator = narrator  
  
    def download(self):  
        return self.text + self.fileformat + "\n Narration by:" + self.narrator
```

شکل ۳۶ - پیاده‌سازی کلاس‌های `Text` و `Audio`

حال باید کلاس‌های زبان‌های ایتالیایی و فارسی و دو متدهای `translate` و `narrate` را در آنها پیاده‌سازی کنیم.

```
class ToFarsi:  
    def translate(self, txt):  
        self.translation = Text("متن ترجمه شد!")  
        return self.translation.download()  
  
    def narrate():  
        pass  
        self.narration = Audio("Farsi-narration", "mp3", "Hootan Shakiba")  
        return self.narration.download()  
  
  
class ToItalian:  
    def translate(self, txt):  
        self.translation = Text("testo tradotto")  
        return self.translation.download()  
  
    def narrate():  
        pass  
        self.narration = Audio("Italian-narration", "mp3", "Carlo Sabatini")  
        return self.narration.download()
```

شکل ۳۷ - پیاده‌سازی کلاس‌های `ToItalian` و `ToFarsi`

حال کلاسی برای این اپلیکیشن طراحی و پیاده‌سازی می‌کنیم:

```
class TranslatorApp:  
    builder: TargetLanguage  
  
    def __init__(self, builder):  
        self.builder = builder  
  
    def translate(self, txt):  
        print(self.builder.translate(txt))  
  
    def narrate(self, txt):  
        print(self.builder.narrate(txt))  
  
    def changeBuilder(self, builder):  
        self.builder = builder
```

شکل ۳۸ - پیاده‌سازی کلاس `TranslationApp`

از برنامه اجرا می‌گیریم:

```

FarsiTranslationBuilder = ToItalian()
translationDirector = TranslatorApp(FarsiTranslationBuilder)
translationDirector.translate("translated text")

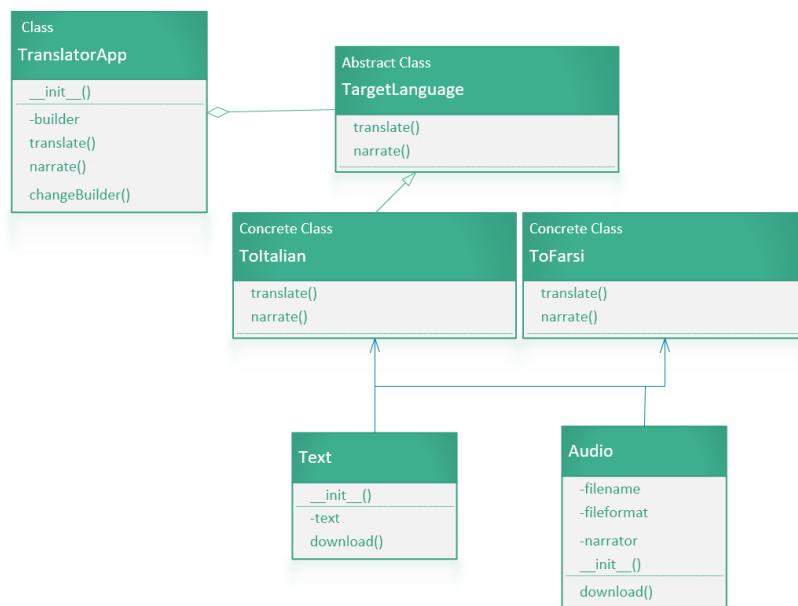
```

شکل ۳۹ - /جرای کد نهایی

testo tradotto

شکل ۴۰ - خروجی کد بالا

پس ساختار این اپلیکیشن به صورت زیر خواهد بود:



class diagram of translator app- شکل ۴۱

الگوهای ساختاری

الگوهای ساختاری مربوط به نحوه ترکیب اشیاء و کلاس‌ها به منظور ایجاد ساختارهای بزرگتر هستند. الگوهای ساختاری کلاس از ارثبری برای ترکیب واسطه‌ها یا پیاده‌سازی‌ها استفاده می‌کنند. به طور مثال به این توجه کنید که چگونه ارثبری چندگانه دو یا چند کلاس را در هم می‌آمیزد که نتیجه آن کلاسی است که ویژگی‌های کلاس والد را ترکیب کرده است. این الگو در موقعی که می‌خواهیم کتابخانه‌هایی از کلاس‌های مستقل از هم ایجاد کنیم، بسیار کاربردی است. یک مثال دیگر می‌تواند ساختار کلاس در الگوی Adapter باشد. به طور کلی، یک adapter باعث می‌شود که یک واسطه مانند واسطه دیگری باشد. بنابراین یک انتزاع همسان برای همه‌ی واسطه‌ها به وجود می‌آید.

اما الگوهای ساختاری شیء راه‌های مختلف ترکیب اشیاء را برای رسیدن به کارایی جدید معرفی می‌کنند. انعطاف‌پذیری افروده شده در ترکیب اشیاء از قابلیت تغییر ترکیب در زمان اجرا نشأت می‌گیرد که در ترکیب کلاس‌ها امری ناشدنیست. الگوی Composite یکی از این الگوهاست. این الگو چگونگی ساخت یک سلسله مراتب کلاسی که از کلاس‌هایی برای دو نوع شیء ساخته شده اند را معرفی می‌کند: شیء نخستین و شیء مرکب. اشیاء مرکب شما را قادر می‌سازد تا اشیاء نخستین^{۲۲} و دیگر اشیاء مرکب^{۲۳} را در قالب یک ساختار پیچیده دلخواه ترکیب کنید.

^{۲۲} Primitive

^{۲۳} Composite

الگوی Adapter

تعریف

الگوی Adapter ساختار یک کلاس را به شکل ^{۲۴} که مشتری ^۵ نیاز دارد تبدیل می‌کند. این الگو این امکان را فراهم می‌آورد که کلاس هایی با شکل های مختلف بتوانند با یکدیگر کار کنند. الگوی Adapter دو نوع است: الگوی Adapter کلاس و الگوی Adapter شی.

هدف

گاهی اوقات یک کلاس نمی‌تواند به شکلی که هست مورد استفاده قرار گیرد و ما باید شکل آن را تغییر دهیم تا بتوانیم از آن استفاده کنیم. اگر بخواهیم الگوی Adapter را با مثالی در دنیای واقعی توضیح دهیم، تبدیل دوشاخه مثال خوبی می‌تواند باشد. می‌دانیم پریزها در کشور های مختلف، استاندارد و شکل های مختلفی دارند. در بعضی از کشور ها، مانند ایران، کابل برق دو شاخه و در بعضی سه شاخه و با شکل ها و فواصل مختلف هستند.



شکل ۴۲ - انواع پریزها و سری ها کابل برق

حال فرض کنید یک شهروند آمریکایی به ایران سفر کرده است. او متوجه می‌شود که شکل پریزها در ایران با شکل پریزها در آمریکا متفاوت است. پس نمی‌تواند از کابل لپ تاپ خود به تنها یی استفاده کند. او برای حل این مشکل، باید از یک اداپتور تبدیل سه شاخه به دوشاخه استفاده کند. این همان کاری است که الگوی Adapter انجام می‌دهد.



شکل ۴۳ - اداپتور تبدیل سه شاخه به دوشاخه

^{۲۴} Interface

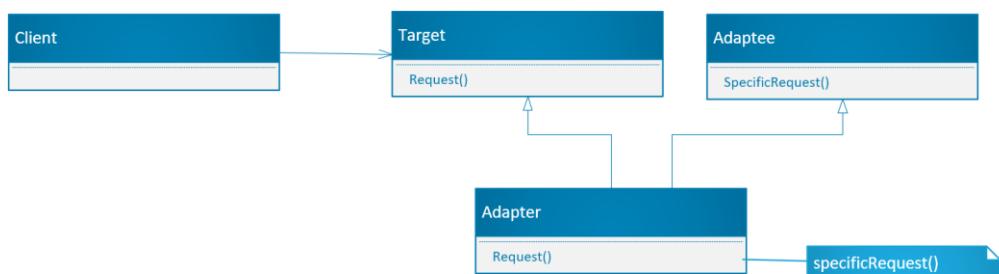
^{۲۵} Client

موارد استفاده

- هنگامی که می‌خواهید از یک کلاس استفاده کنید اما شکل آن طوری است که با آن کلاسی که شما نیاز دارید سازگار نیست.
- هنگامی که می‌خواهید یک کلاس طراحی کنید که قابلیت استفاده مجدد^{۲۶} داشته و بتواند با هر کلاس دیگری، فارغ از سازگار بودن شکل، کار کند.
- (فقط برای شئ ادپتر) نیاز دارید تا از چندین کلاس زیرین استفاده کنید اما سازگار کردن شکل این کلاس‌ها با زیرکلاس‌سازی، امری دشوار و ناکارآمد است. یک شئ ادپتور می‌تواند شکل کلاس والد را به خود بگیرد.

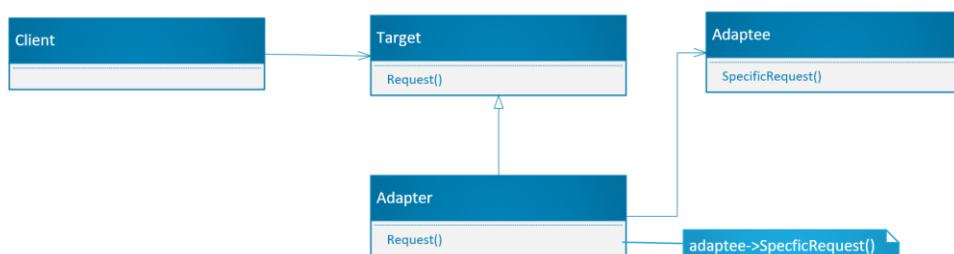
ساختار

ساختار الگوی Adapter کلاس:



class diagram of Adapter(Class)- ۴۴ شکل

ساختار الگوی Adapter شئ:



class diagram of Adapters(object) - ۴۵ شکل

^{۲۶} Reusable

:Target کلاس

ساختاری را که مشتری استفاده می‌کند مشخص می‌کند

:Client کلاس

با اشیائی که با Taget همخوانی دارند، سروکار دارد

:Adaptee کلاس

کلاسی است که پیاده سازی یک عملیات در آن انجام شده و کلاس Adapter از آن ارث بری می‌کند و یا شئ ای از آن را در خود می‌سازد.

:Adapter کلاس

کلاسی است که از کلاس Target و Adaptee ارث بری می‌کند و یا شئ ای از کلاس Adaptee را می‌سازد. این کلاس هنگامی که متده می‌شود، متده specificRequest() از کلاس Adaptee را اجرا می‌کند.

مثال

می‌خواهیم سیستمی طراحی کنیم که در قسمت ثبت نام آن، یک کد تایید به شماره موبایل کاربر فرستاده شود. ما از یک پنل پیامک استفاده می‌کنیم که از هر دو روش SOAP و REST پشتیبانی می‌کند. ما تصمیم گرفته‌ایم تا در فاز یک، از SOAP استفاده کنیم و اگر سرعت آن خوب نبود، از REST بهره می‌گیریم. برای هر یک از این روش‌ها، یک اسکریپت جداگانه وجود دارد که باید از آن استفاده کنیم. متده ارسال پیامک در این دو اسکریپت به ترتیب sendSmsRestNumbePN() و sendSmsBySoapByNumber() است. ما می‌توانیم در کد خود هر یک از کلاس‌های این دو روش را نمونه سازی کرده و سپس متده مورد نظر را صدا بزنیم اما اگر بخواهیم تغییری در روش اتصال به پنل پیامک به وجود آوریم، مجبوریم تمامی قسمت‌های برنامه اصلی که متده مربوطه را صدا زده‌اند را ویرایش کنیم که این برخلاف قانون بسته برای ویرایش است. پس ما می‌خواهیم کلاسی طراحی کنیم که قابلیت استفاده مجدد را داشته باشد و بتواند با هر اسکریپت اتصال به پنل پیامکی کار کند. راه حل، استفاده از الگوی Adapter است.

اگر بخواهیم الگوی Adapter کلاس را پیاده سازی کنیم، ابتدا یک واسطه TargetSMS می‌سازیم که متد SendSms() در آن تعریف شده است. سپس یک کلاس اداتپر به نام SMSAdapter می‌سازیم که از کلاس TargetSms و SoapSms ارث بری می‌کند.

یک کلاس هم به نام APPLogin می‌سازیم که در آن یک شی از SMSAdapter می‌سازیم.

```
class TargetSMS(ABC):
    @abstractmethod
    def SendSms(self, number):
        pass

class SoapSms():
    def sendSmsBySoapByNumber(self, number):
        print(f"Sending {randint(1000,9999)} to {number} ")

class SMSAdapter(TargetSMS, SoapSms):
    def SendSms(self, number):
        self.sendSmsBySoapByNumber(number)

class APPLogin:
    # ...
    # some codes here
    def __init__(self):
        self.SMSAdapter = SMSAdapter()
        self.SMSAdapter.SendSms("09305667042")

    # some codes there

login = APPLogin()
```

شکل ۴۶ - پیاده سازی Adapter(class)

sending 2009 to 09305667042

شکل ۴۷ - خروجی کد بالا

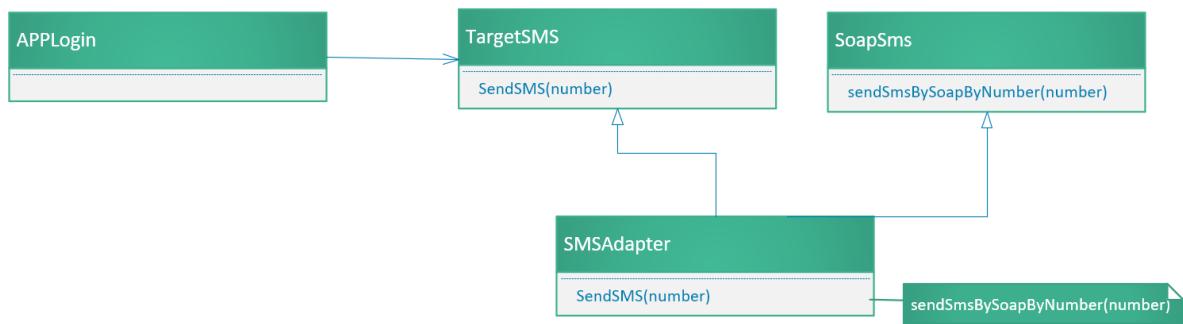
و اگر بخواهیم الگوی Adapter برای شی را پیاده سازی کنیم، باید در کلاس SMSAdapter یک شی از SoapSms بسازیم.

```
class SMSAdapter(TargetSMS):
    def SendSms(self, number):
        self.smsApproach = SoapSms()
        self.smsApproach.sendSmsBySoapByNumber(number)
```

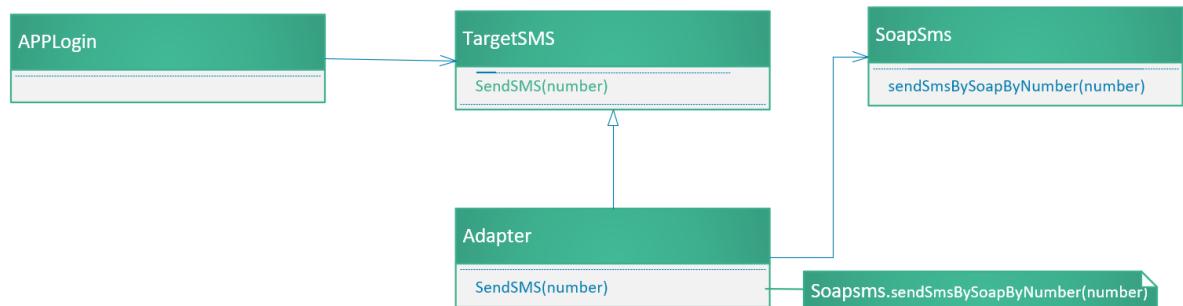
شکل ۴۸ - پیاده سازی Adapter(Object)

پس ساختار برنامه ما به شکل زیر خواهد بود:

Adapter(Class)



Adapter(Object)



class diagrams of Adapter(class)& Adapter(object) - ٤٩ شک

الگوی Bridge

تعريف

الگوی Bridge انتزاع^{۳۷} را از پیاده سازی^{۳۸} جدا می کند تا بتوانند مستقل از یکدیگر متفاوت باشند.

هدف

هنگامی که یک انتزاع می تواند چندین پیاده سازی داشته باشد، راه معمول انجام این کار ارث بری است. یک کلاس انتزاعی ساختار انتزاعی را تعریف می کند و کلاس های عینی آن را پیاده سازی می کنند. اما این روش همیشه به اندازه کافی انعطاف پذیر نیست. ارث بری یک پیاده سازی را برای همیشه به یک انتزاع مقید و محدود می کند که این، کار را برای ویرایش، توسعه و بازاستفاده ای انتزاع ها و پیاده سازی ها به طور مسقل سخت می کند. الگوی Bridge مانند ضرب کارتیزانی عمل می کند و از دو مجموعه انتزاع و پیاده سازی ترکیب های متنوعی می سازد.

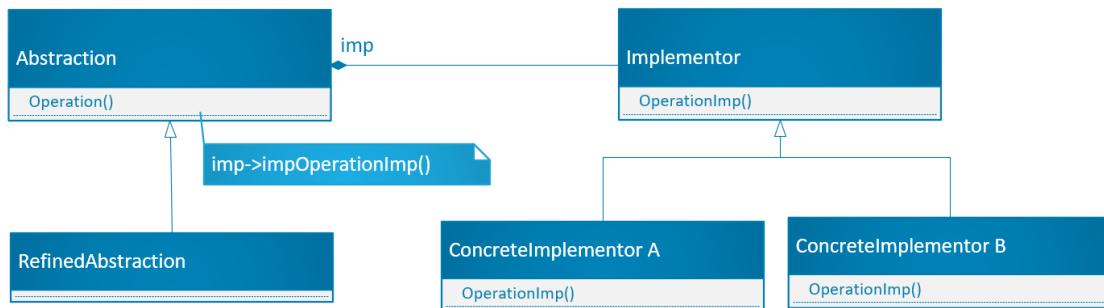
موارد استفاده

- هنگامی که نمی خواهید پیاده سازی ها و انتزاع ها برای همیشه به یکدیگر محدود ووابسته باشند. به طور مثال ممکن است سیستم باید طوری طراحی شود که پیاده سازی باید در زمان اجرا انتخاب شود یا تغییر کند.
- از طریق زیر کلاس سازی باید بتوان هم انتزاع ها و هم پیاده سازی های آنها را گسترش داد.
- تغییرات در پیاده سازی یک انتزاع نباید تاثیری روی مشتری ها داشته باشد. این به این معناست که کد آنها نیازی به کامپایل مجدد نداشته باشد.
- تعداد کلاس ها رو به افزایش هستند.
- می خواهید یک پیاده سازی را بین چندین شئ مختلف به اشتراک بگذارید (احتمالاً با استفاده از تکنیک شمارش ارجاعات^{۳۹}) و این کار باید از مشتری مخفی بماند.

^{۳۷} Abstraction

^{۳۸} Implementation

^{۳۹} Reference Counting Technique



شکل ۵۰

Abstraction

این کلاس یک واسطه برای انتزاع تعریف می‌کند و همچنین دارای یک نمونه از شئ ای از جنس Implementor است.

RefinedAbstraction

این کلاس واسطه Abstraction را توسعه می‌دهد.

Implementor

یک واسطه برای پیاده‌سازی کلاس‌ها تعریف می‌کند. این واسطه نیازی به این ندارد که کاملاً با واسطه Abstraction هماهنگ باشد. در حقیقت این دو می‌توانند کاملاً متفاوت از هم باشند. به طور معمول، واسطه Implementor فقط چندین عمل اولیه را پوشش می‌دهد و Abstraction عملیات سطح بالاتری از این چندین عمل اولیه را تعریف می‌کند.

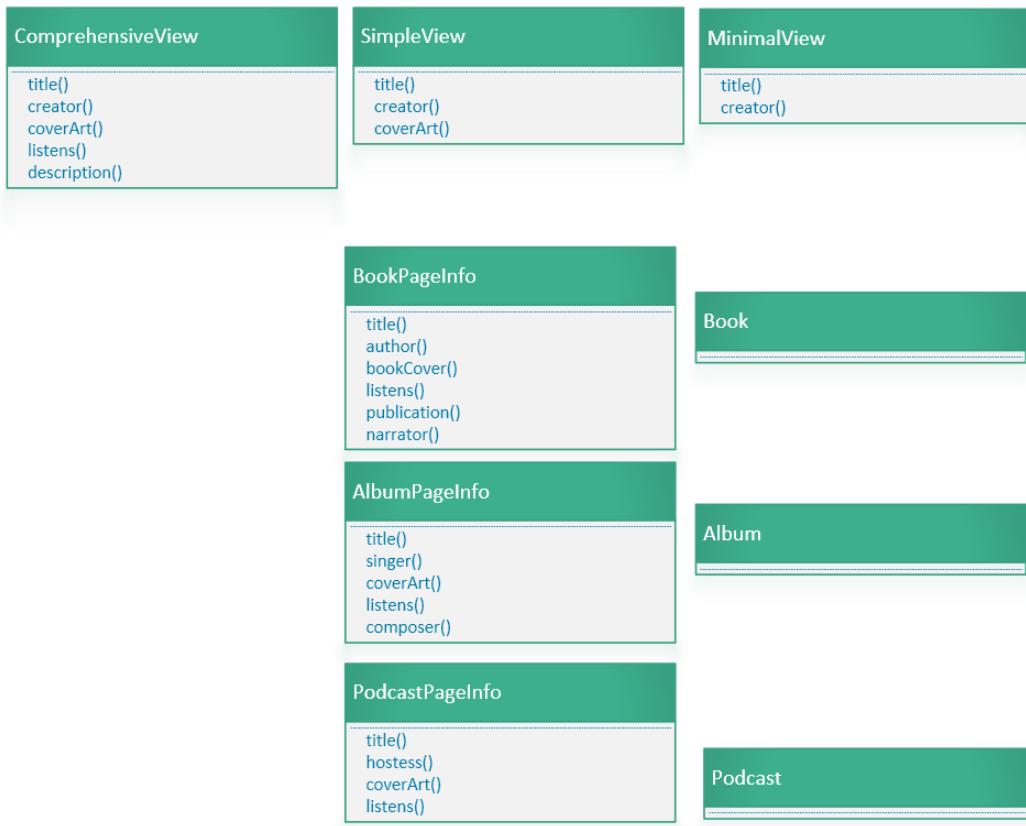
ConcreteImplementor

واسطه Implementor را پیاده‌سازی می‌کند.

^{۱۰} Higher-level

فرض کنید که می‌خواهیم برنامه‌ای شبیه برنامه اسپاتیفای^{۱۳} طراحی کنیم. می‌خواهیم سه نوع محتوا کتاب، آلبوم و پادکست داشته باشیم. هنگامی که روی آن‌ها کلیک می‌کنیم، وارد صفحه اختصاصی هر محتوا شده و اطلاعات آن را مشاهده می‌کنیم. همچنین می‌خواهیم سه شکل نمایش داشته باشیم: جامع، ساده و مینیمال. در نمایش جامع، کاور، نام هنرمند، عنوان، تعداد شنیده شدن، توضیحات و ...، در نمایش ساده کاور، عنوان، نام هنرمند و در نمایش مینیمال فقط عنوان و نام هنرمند نمایش داده می‌شود. ما می‌خواهیم هر کدام از انواع محتوای کتاب، آلبوم و پادکست دارای این سه نمایش باشند و با توجه به سایز صفحه کاربر نمایش مناسب به او نشان داده شود. تا اینجای کار می‌دانیم که باید سه کلاس برای نمایش و سه کلاس برای نوع محتوا و سه کلاس برای اطلاعات صفحه‌ی اصلی هر نوع محتوا بسازیم زیرا اطلاعات هر محتوا بسته به نوع آن فرق می‌کند. به طور مثال اطلاعات کتاب شامل عنوان کتاب، جلد پشت و روی کتاب، گوینده، بیوگرافی نویسنده و ... است ولی اطلاعات آلبوم شامل خواننده، آهنگساز، دیسکوگرافی و ... است. توجه کنید دلیل اینکه برای صفحه اطلاعات کتاب یک کلاس جدا ساخته ایم، وجود صفاتی مانند بیوگرافی، گوینده و ... است که جزئی از کتاب نیستند ولی در مورد آن کتاب هستند. پس کلاس‌های ما به صورت زیر خواهند بود.

^{۱۳} Spotify



شکل ۵۱ – کلاس های اولیه

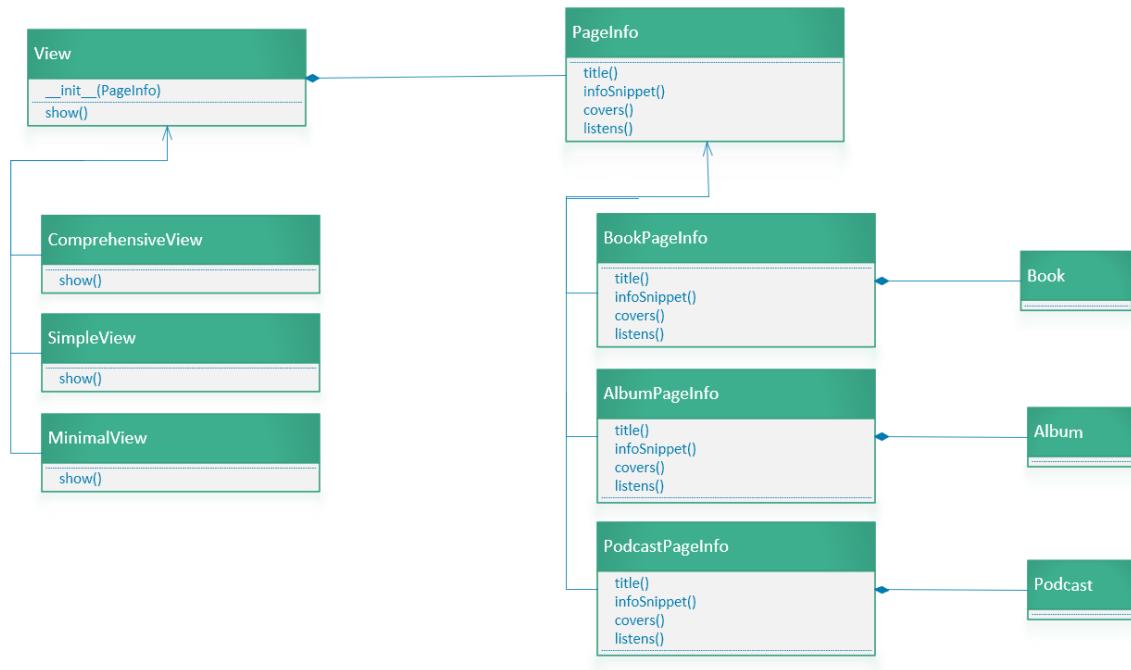
اولین راهی که برای پیاده‌سازی نمایش‌های مختلف برای محتواهای مختلف به ذهن می‌رسد این است که برای هر ترکیب صفحات اطلاعات محتوا و نمایش یک کلاس بسازیم. زیرا صفحات اطلاعات محتواهای مختلف با یکدیگر فرق می‌کنند و نمی‌توانیم با یک کلاس نمایش ساده اطلاعات درست را نمایش دهیم. به طور مثال اگر در کلاس نمایش ساده متدهای داشته باشیم که نام خواننده را نشان دهد، این متدها در مورد محتوای کتاب و پادکست به خوبی کار نمی‌کند. پس باید هر کلاس صفحه اطلاعات با هر کلاس نمایش با هم ترکیب شوند. برای درک بهتر دو مجموعه زیر را در نظر می‌گیریم. مجموعه S۱ نمایش‌ها و مجموعه S۲ اطلاعات صفحات هر محتوا هستند.

$$S_1 = \{ComprehensiveView, SimpleView, MinimalView\}$$

$$S_2 = \{BookPageInfo, AlbumPageInfo, PodcastPageInfo\}$$

حال اگر بخواهیم به روش اشاره شده در بالا، عمل کنیم و این دو مجموعه را با هم ترکیب کنیم، ۹ کلاس خواهیم داشت! اما این کار نمی‌تواند گزینه مناسبی باشد زیرا با افزایش انواع محتوا و انواع نمایش تعداد کلاس‌ها به شدت افزایش پیدا می‌کند و این کار را برای نگهداری یک سیستم به شدت سخت می‌کند.

راه حل استفاده از الگوی Bridge است. باید سطوح انتزاع و سطوح پیاده‌سازی را از هم جدا کنیم. پس ابتدا باید نمایش و صفحه اطلاعات را از پیاده‌سازی هر یک جدا کنیم:



شکل ۴۲ سیستم جدید class diagram

ابتدا یک کلاس به نام PageInfo می‌سازیم که تمامی کلاس‌های BookPageInfo, AlbumPageInfo, PodcastPageInfo از آن ارث بری می‌کنند. سپس یک کلاس انتزاعی به نام View می‌سازیم که دارای دو متده‌سازنده و () show باشد. این کلاس در متده‌سازنده خود یک شیء از PageInfo را دارا می‌باشد. سپس کلاس‌های ComprehensiveView, SimpleView, MinimalView را می‌سازیم که از کلاس View ارث بری می‌کنند. حال نوبت آن است که کلاس‌ها را پیاده سازی کنیم. (برای کاهش پیچیدگی‌ها، کلاس‌ها ساده سازی می‌کنیم)

```

class Book:
    def __init__(self, title, author, narrator, fCover, bCover):
        self.title = title
        self.author = author
        self.narrator = narrator
        self.fCover = fCover
        self.bCover = bCover

class Album:
    def __init__(self, title, singer, composer):
        self.title = title
        self.singer = singer
        self.composer = composer

class Podcast:
    def __init__(self, title, hostess, episode, sponser):
        self.title = title
        self.hostess = hostess
        self.episode = episode
        self.sponser = sponser

```

شکل ۵۳ - پیاده‌سازی کلاس‌های کتاب، آلبوم و پادکست

```

class PageInfo:
    def title():
        pass

    def creator():
        pass

    def metaInfo():
        pass

    def cover():
        pass

class BookPageInfo(PageInfo):
    def __init__(self, book: Book):
        self.book = book

    def title(self):
        print(f"***{self.book.title}***")

    def creator(self):
        print(f"\nAuthor:{self.book.author}")

    def metaInfo(self):
        self.narrator()
        self.biography()

    def cover(self):
        print(
            f"\nfront cover:{self.book.fCover} | back cover: {self.book.bCover}")

    def biography(self):
        print("\nHere comes the biography!")

    def narrator(self):
        print(f"\nThe narrator is {self.book.narrator}")

```

```

class AlbumPageInfo(PageInfo):
    def __init__(self, album: Album):
        self.album = album

    def title(self):
        print(f"***{self.album.title}***")

    def creator(self):
        print(f"\nSinger:{self.album.singer}")

    def metaInfo(self):
        self.discography()
        self.composer()

    def cover(self):
        print(f"\nAlbum cover:{self.album.cover} ")

    def discography(self):
        print("\nHere comes the discography!")

    def composer(self):
        print(f"\nThe composer is {self.album.composer}")


class PodcastPageInfo(PageInfo):
    def __init__(self, podcast: Podcast):
        self.podcast = podcast

    def title(self):
        print(f"***{self.podcast.title}***")

    def creator(self):
        print(f"\nHostess:{self.podcast.hostess}")

    def metaInfo(self):
        self.sponser()
        self.episode()

    def cover(self):
        print(f"\nPodcast cover:{self.podcast.cover} ")

    def sponser(self):
        print(f"\nToday's podcast's sponser is {self.podcast.sponser}")

    def episode(self):
        print(f"\nToday's episode is about {self.podcast.episode}")

```

شکل ۵۴ - پیاده سازی کلاس های *BookPageInfo* ، *PageInfo* ، *AlbumPageInfo* و *PodcastPageInfo*

```

class View(ABC):
    def __init__(self, pageInfo: PageInfo):
        self.pageInfo = pageInfo

    @abstractmethod
    def show(self):
        pass


class ComprehensiveView(View):
    def show(self):
        self.pageInfo.title()
        self.pageInfo.creator()
        self.pageInfo.cover()
        self.pageInfo.metaInfo()


class SimpleView(View):
    def show(self):
        self.pageInfo.title()
        self.pageInfo.creator()
        self.pageInfo.cover()


class MinimalView(View):
    def show(self):
        self.pageInfo.title()
        self.pageInfo.creator()

```

شکل ۵۵ - پیاده سازی کلاس های *View*، *ComprehensiveView*، *SimpleView*، *MinimalView*

در نهایت از برنامه تست می‌گیریم:

```
myBook = BookPageInfo(Book("Book Title", "Book Author",
                           "Book Narrator", "Front Cover", "Back Cover"))
bigscreen = ComprehensiveView(myBook)
print("Comprehensive View-----")
bigscreen.show()
smallscreen = SimpleView(myBook)
print("Simple View-----")
smallscreen.show()
myAlbum = AlbumPageInfo(Album("Album Title", "Album Singer", "Song Composer"))
minimalistic = MinimalView(myAlbum)
print("Minimal View-----")
minimalistic.show()
```

شکل ۵۶ - تست برنامه

```
Comprehensive View-----
***Book Title***
Author:Book Author
front cover:Front Cover | back cover: Back Cover
The narrator is Book Narrator
Here comes the biography!

Simple View-----
***Book Title***
Author:Book Author
front cover:Front Cover | back cover: Back Cover

Minimal View-----
***Album Title***
Singer:Album Singer
```

شکل ۵۷ - خروجی کد بالا

همانطور که در بالا نشان داده شد، توانستیم با استفاده از الگوی bridge و جدا کردن سطوح انتزاع و پیاده‌سازی کلاس‌های View و PageInfo تعداد کلاس‌ها و وابستگی بین آن‌ها را نیز کاهش دهیم.

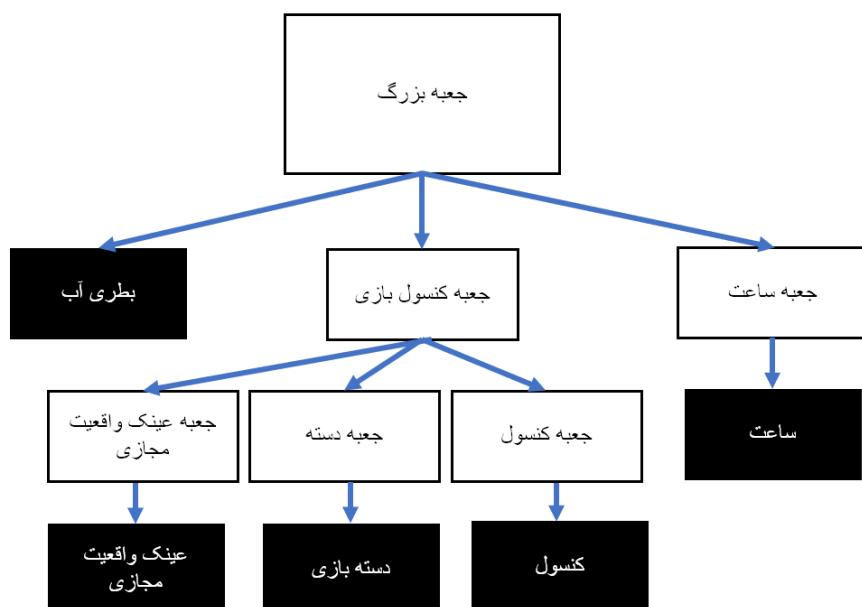
الگوی Composite

تعریف

الگوی Composite از اشیاء یک ساختار درختی می‌سازد که سلسله مراتب جزء و کل را نشان می‌دهد. این الگو به مشتری اجازه می‌دهد تا رفتار یکسانی با اشیاء منفرد و یا ترکیبی از اشیاء داشته باشد.

هدف

سیستم داریم که هر بخش آن از بخش‌های دیگری تشکیل شده است و اجرای یک عملیات روی یک شئ نیازمند اجرای آن عملیات روی تمامی شئ‌های فرزند آن است. به طور مثال فرض کنید که شما از سایتی خرید اینترنتی انجام داده‌اید و یک جعبه بزرگ حاوی کالا‌های شما به دست شما می‌رسد. کالا‌های شما یک ساعت، کنسول بازی و بطری آب هستند که در این جعبه بزرگ قرار گرفته‌اند. ساعت خود درون جعبه ای دیگر است. بطری آب جعبه‌ای ندارد. کنسول بازی داخل جعبه‌ای است که درون آن کنسول، دسته بازی، شارژر و عینک واقعیت مجازی قرار دارد که هر یک داخل جعبه‌ای مخصوص قرار داده شده‌اند (شکل ۵۳). فرض کنید ما قصد داریم تا قیمت کالا‌ها را چک کنیم. پس باید ابتدا جعبه بزرگ را باز کنیم و سپس این کار را برای هر جعبه ادامه دهیم تا به کالا‌ها برسیم. پس باید عملیات چک کردن قیمت روی هر جعبه و یا کالا اعمال شود. این همان کاری است که الگوی Composite انجام می‌دهد.

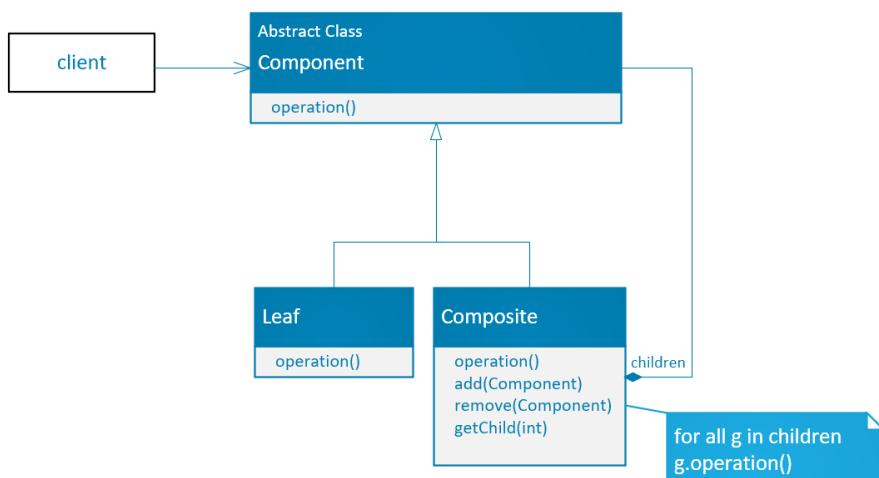


شکل ۵۳ - نمایش درختی جعبه و کالا ها

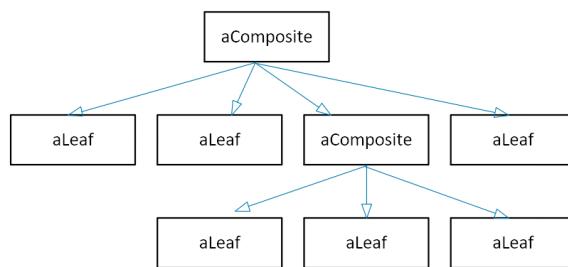
موارد استفاده

- هنگامی که می خواهید سلسله مراتب جزء و کل اشیاء را نمایش دهید.
- هنگامی که برنامه شما می تواند ساختار درختی داشته باشد.
- هنگامی که می خواهید مشتری قادر باشد تا تفاوت بین ترکیبی از اشیاء و اشیاء منفرد را نادیده بگیرد. در این ساختار مشتری می تواند با تمامی ترکیب های اشیاء به یک شکل رفتار کند.

ساختار



شکل ۵۹ - class diagram of Composite Pattern



شکل ۶۰ - نمایش درخت یک ترکیب اشیاء

Component

یک واسطه برای اشیاء در ترکیب تعریف می کند که هر شیء می تواند با استفاده از آن به شیء های فرزند خود دسترسی پیدا کرده و آن ها را مدیریت کند. همچنین رفتار های مشترک میان کلاس ها را پیاده سازی می کند. در مواد خاص این کلاس می تواند یک واسطه برای دسترسی شیء فرزند به والد نیز تعریف کند.

Leaf

نشان دهنده یک شئ برگ است. شئ برگ هیچ فرزندی ندارد. در این کلاس تنها رفتار های اولیه در ترکیب پیاده سازی می شوند.

Composite

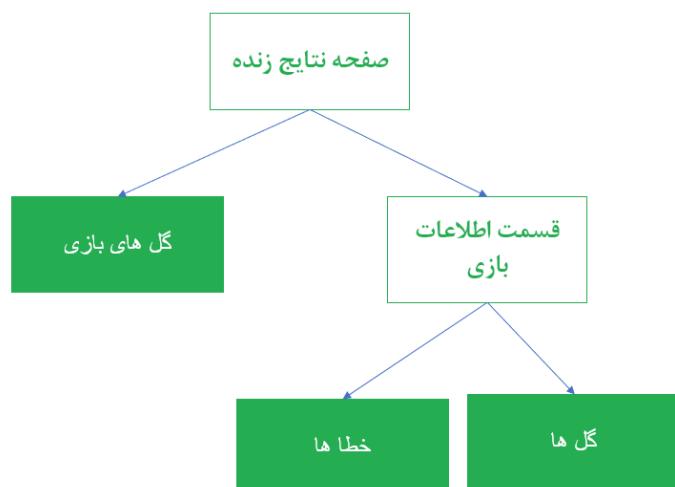
رفتار های اشیائی را که دارای فرزند هستند تعریف می کند. همچنین فرزندان را در خود ذخیره کرده و عملیات مربوط به فرزندان در کلاس Component را پیاده سازی می کند.

Client

با اشیاء موجود در ترکیب، توسط واسطه Component کار می کند.

مثال

می خواهیم برنامه طراحی کنیم که نتایج زنده مسابقات فوتبال را نمایش دهد. در صفحه هر مسابقه، نام دو تیم، گل ها، زنده گل، خطاهای ویدیویی گل های بازی نمایش داده می شود. در هر بروز رسانی همه اطلاعات به جز نام دو تیم بروزرسانی می شوند. ما تصمیم گرفته ایم که قسمت اطلاعات زنده را به صورت مازولات طراحی کنیم تا در ادامه بتوانیم مازول های جدیدی اضافه کنیم و یا کاربر خود قسمت هایی را که می خواهد انتخاب کند. نمایش درخت سیستم ما به صورت زیر خواهد بود:



شکل ۱۶ - نمایش درخت سیستم

عملیات بروزرسانی در هر بخش بالا متفاوت از دیگری است. به طور مثال برای بروزرسانی گل‌ها باید از جدول گل‌ها جستار گرفته شود و برای خطاهای از جدول خطاهای آن‌ها دارای یک متده بروزرسانی هستند.

```
class Component(ABC):
    @abstractmethod
    def refresh(self):
        pass

class Tab(Component):
    def __init__(self, tabName, modules: list[Component]):
        self.tabName = tabName
        self.modules = modules

    def add(self, module: Component):
        self.modules.append(module)

    def remove(self, module: Component):
        self.modules.remove(module)

    def refresh(self):
        list(map(lambda module: module.refresh(), self.modules))

class GoalsStats(Component):
    def refresh(self):
        print("Update Goals ⚽")

class FoulsStats(Component):
    def refresh(self):
        print("Update Fouls ⚪")

class HighlightVideo(Component):
    def refresh(self):
        print("Update Highlight videos 💥⚽")
```

شکل ۶۲ - پیاده‌سازی سیستم نتایج زنده

همانطور که مشاهده می‌شود کلاس Tab ارث بری می‌کند، متده refresh به نحوه پیاده‌سازی شده است که متده refresh تمامی اشیاء فرزند آن صدا زده شود. حال کد بالا را تست می‌کنیم:

```
print("LiveScore - Juventus vs Inter ")
goalModule = GoalsStats()
foulModule = FoulsStats()
highlight = HighlightVideo()
StatsTab = Tab("StatsTab", [goalModule, foulModule])

StatsTab.refresh()
highlight.refresh()
```

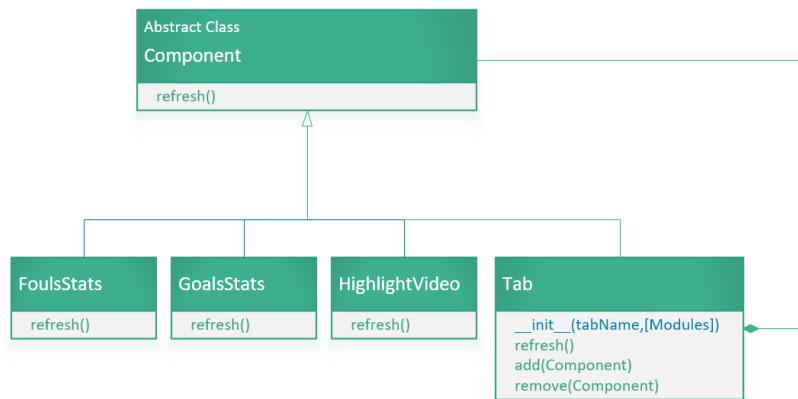
شکل ۶۳ - اجرای و تست کد بالا

```
LiveScore - Juventus vs Inter
Update Goals ⚽
Update Fouls ⚪
Update Highlight videos 💥⚽
```

شکل ۶۴ - خروجی کد بالا

برنامه به درستی کار می‌کند.

همچنین ساختار این برنامه به شکل زیر است:



شکل ۶۵ برنامه طراحی شده class diagram -

الگوی Decorator

تعریف

الگوی Decorator به طور پویا وظایف جدیدی به یک شئ اضافه می‌کند. این الگو یک راه جایگزین و انعطاف‌پذیر را، نسبت به ارثبری، برای افزایش کارایی ارائه می‌کند.

هدف

گاهی اوقات می‌خواهیم که وظایف جدیدی به یک یا گروهی از اشیاء، و نه یک کلاس، اضافه کنیم. یک راه اضافه کردن این وظایف، بهره بردن از ارثبری و راه بهتر آن استفاده از ترکیب و الگوی Decorator است. فرض کنید که برای شخصی یک جفت جوراب هدیه گرفتید اما نمی‌خواهید آن را بدون هیچ تزئینی به او بدهید. می‌خواهید ابتدا آن را در جعبه‌ای بگذارید. سپس آن را دور کاغذ کادو بپیچید و روی آن ربان بزنید. استفاده از ارثبری برای اینکار مانند مراجعه مجدد به فروشگاه و خرید یک جفت جوراب که درون جعبه است و کاغذ کادو و ربان دارد، است. ولی اینکار می‌تواند توسط خودتان انجام شود. جعبه، کاغذ کادو و ربان خود هر سه هدیه محسوب می‌شوند اما کاری به هدیه قبلی اضافه می‌کنند و آن را در بر می‌گیرند. به همین دلیل است که نام دیگر این الگو Wrapper است. شکل زیر به فهم بهتر موضوع کمک می‌کند:



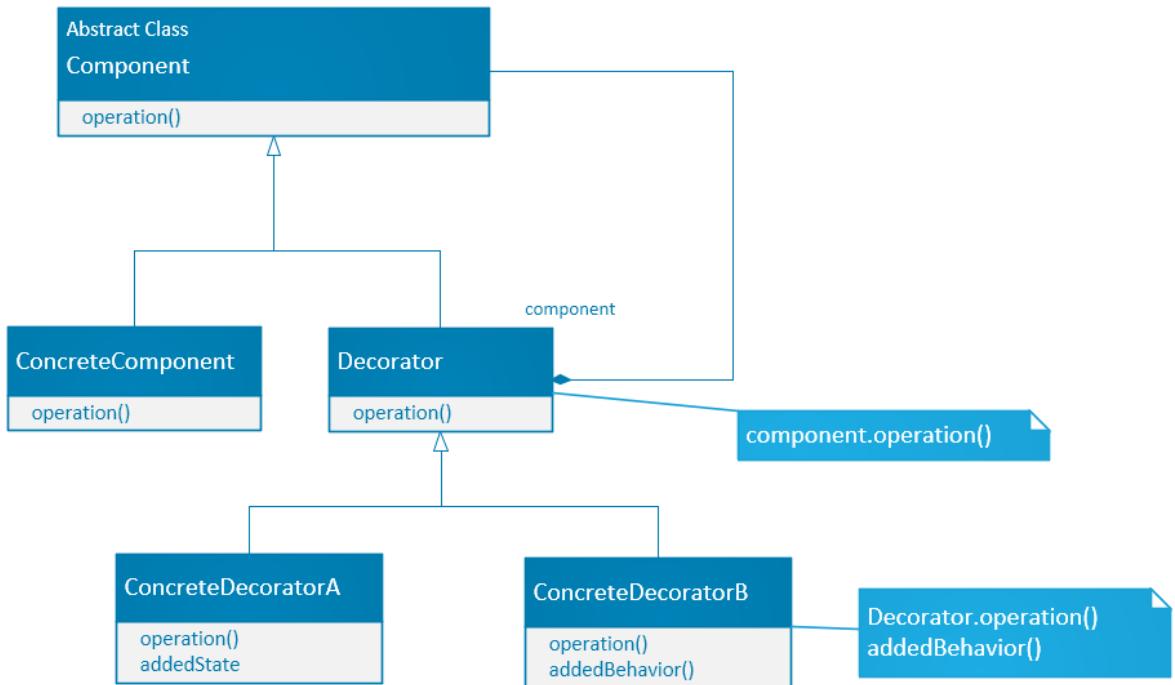
شکل ۶۶- شکل بصری مثال ملا

موارد استفاده

- هنگامی که می‌خواهید به چندین شئ به طور پویا^{۳۲} و شفاف^{۳۳}، که به این معنی است که شئ‌های دیگر را تغییر ندهید، وظایف جدیدی اضافه کنید.
- آن وظایف می‌توانند از شئ حذف شوند.
- هنگامی که افزایش کارایی توسط ارثبری غیرعملی است.

^{۳۲} Dynamic

^{۳۳} Transparent



شکل ۶۷ class diagram of Decorator Pattern

Component

یک رابط برای اشیائی که می‌توانند مسئولیت‌های جدید بپذیرند است.

ConcreteComponent

یک شیء را تعریف می‌کند که قابلیت افزودن مسئولیت‌های جدید به آن‌ها وجود دارد.

Decorator

یک رابط است که از کلاس **Component** ارث بری می‌کند و همچنین یک شیء از آن را نیز در خود دارد.

ConcreteDecorator

رابط **Decorator** را پیاده سازی می‌کند و مسئولیت جدیدی به شیء **Component** اضافه می‌کند.

مثال

می خواهیم مرورگری بنویسیم که بتواند متن و تصویر را نمایش دهد. ابتدا یک کلاس ElementView می سازیم که هر نمایش مانند نمایش تصویر یا نمایش متن باید از آن ارث بری کند و متدهای show() را پیاده سازی کند سپس دو کلاس PictureView و TextView را می سازیم.

```
class ElementView(ABC):
    @abstractmethod
    def show(self):
        pass

class TextView(ElementView):
    def show():
        print("Text is being shown")

class PictureView(ElementView):
    def show():
        print("Picture is being shown")
```

شکل ۶۱ - پیاده سازی اولیه کلاس های PictureView و TextView از ElementView

حال می خواهیم که این نمایش ها حاشیه و اسکرول داشته باشند. ساده ترین کار اضافه کردن چهار متده است که خروجی هر کدام به ترتیب نمایش ساده، نمایش حاشیه دار، نمایش اسکرول دار، نمایش حاشیه دار اسکرول دار است. این کار بهینه نیست زیرا هر بار با اضافه شدن یک قابلیت، به طور مثال قابلیت انتخاب، چندین متده نمایش جدید باید اضافه شود. راه بعدی استفاده از ارث بری است که این راه هم بهینه نیست زیرا به تعداد ترکیب های ممکن، مانند مثال متدها، باید کلاس ساخت. به طور مثال کلاس TextViewWithBorder ...

اما با استفاده از الگوی Decorator می توانیم هر قابلیت را به شکل کلاسی طراحی کنیم و سپس نمایش ها را به عنوان آرگومان ورودی به آنها بدهیم. پس ابتدا یک رابط به نام Decorator ایجاد می کنیم که از کلاس ElementView ارث بری می کند. سپس دو کلاس Scroll و Border را پیاده سازی می کنم.

```
class Decorator(ElementView, ABC):
    def __init__(self, elementView):
        self.elementView = elementView

class Border(Decorator):
    def show(self):
        self.border()

    def border(self):
        print("-----")
        self.elementView.show()
        print("-----\n")

class Scroll(Decorator):
    def show(self):
        self.scroll()

    def scroll(self):
        print("Scroller Added ++++++")
        self.elementView.show()
        print("\n")
```

شکل ۶۲ - پیاده سازی کلاس های Border و ScrollDecorator

```
textWithBorder = Border(TextView())
textWithBorder.show()

textwithScroll = Scroll(TextView())
textwithScroll.show()

pictureWithBorderAndScroll = Scroll(Border(PictureView()))
pictureWithBorderAndScroll.show()
```

شکل ۷۰ - اجرای برنامه

```
-----
Text is being shown
-----
Scroller Added ++++++
Text is being shown

Scroller Added ++++++
-----
Picture is being shown
-----
```

شکل ۷۱ - خروجی کد بالا

همانطور که مشاهده می شود توانستیم با استفاده از الگوی **Decorator** بدون ایجاد کلاس های اضافی، قابلیت اسکرول و حاشیه را به نمایش متن و نمایش تصویر اضافه کنیم.

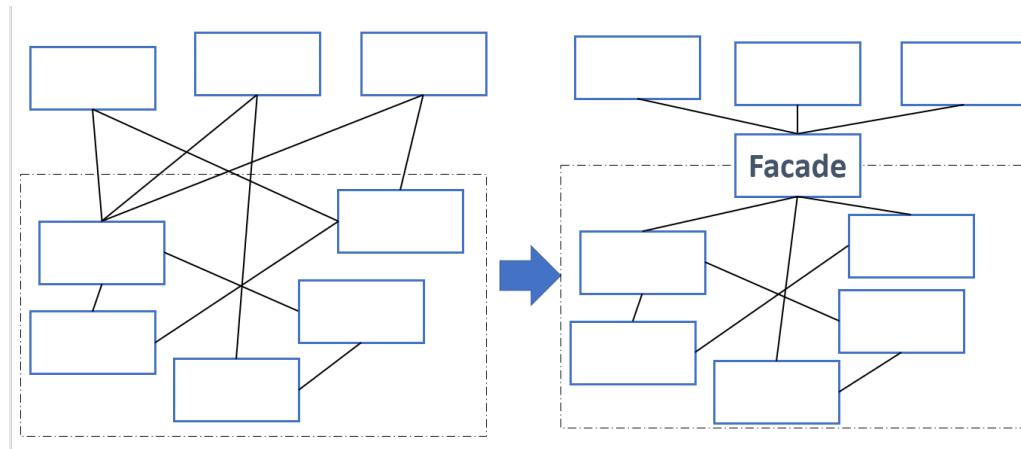
الگوی Facade

تعریف

این الگو یک رابط یکسان برای مجموعه‌ای از رابط‌ها در یک زیرسیستم تعریف می‌کند. این رابط سطح‌بالا استفاده از این زیرسیستم را تسهیل می‌کند.

هدف

تقسیم ساختار یک سیستم به چندین زیرسیستم باعث کاهش پیچیدگی می‌شود. یک هدف معمول در طراحی این است که بتوانیم ارتباطات ووابستگی‌ها را بین زیرسیستم‌ها کاهش دهیم. یک راه رسیدن به این هدف، استفاده از یک Facade است که یک واسط ساده برای امکانات کلی یک زیرسیستم ارائه می‌کند.



شکل ۷۲ - کاهش پیچیدگی ارتباطات کلاس‌ها با استفاده از الگوی Facade

همچنین در طراحی زیرسیستم‌ها بهتر است به قاعده حداقل دانش^{۳۴} توجه کنیم. این قاعده بیان می‌کند که متدى از یک شئ فقط باید در چهار حالت زیر استفاده شود:

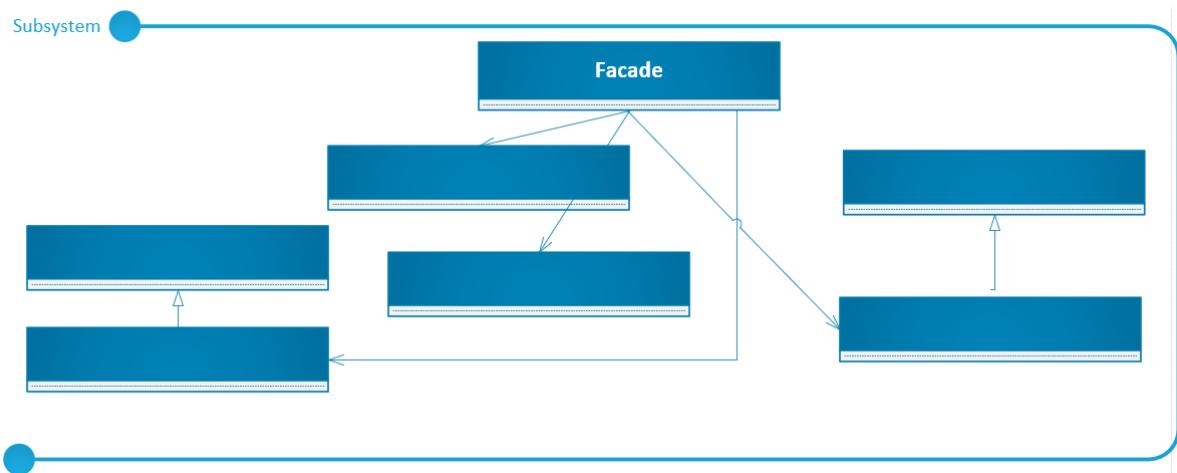
۱. متى به خود شئ باشد
۲. متى متعلق به شئ اى باشد که به عنوان پaramتر ورودى به متى داده شده است
۳. متى متعلق به هر شئ اى که متى فعلی می‌سازد باشد
۴. متى متعلق به هر جزئى از شئ باشد (مانند صدا زدن متى از یک شئ ذخیره شده در کلاس)

^{۳۴} Principle of Least Knowledge

موارد استفاده

- هنگامی که می خواهید یک رباط ساده برای یک زیرسیستم پیچیده ارائه کنید. هر چه زیرسیستم ها بیشتر توسعه داده شوند، پیچیدگی آن ها نیز بیشتر می شود. اغلب الگوهای طراحی باعث ایجاد کلاس های کوچکتر و بیشتر می شوند. این قابلیت استفاده مجدد زیرسیستم را افزایش می دهد و شخصی سازی آن را آسان تر می کند اما کار را برای استفاده مشتری ای که نیازی به شخصی سازی آن ندارد، مشکل می کند. یک نما - Facade - یک نمایش پیشفرض ساده از زیرسیستم ارائه می کند که برای اغلب مشتری ها کافی است. تنها مشتری هایی که نیاز به شخصی سازی بیشتر دارند باید از چیزی بیشتر از نما استفاده کنند.
- هنگامی که تعداد وابستگی ها بین مشتری و کلاس های پیاده سازی یک انتزاع زیاد است، باید از یک نما برای جدای کردن زیرسیستم ها از مشتری ها و دیگر زیرسیستم ها استفاده کرد که باعث بهبود استقلال و جابه جایی پذیری زیرسیستم می شود.
- هنگامی که می خواهید زیرسیستمان را لایه بندی کنید، از یک نما استفاده کنید تا یک نقطه ورود ^{۳۵} برای هر سطح زیرسیستم ایجاد کنید. اگر زیرسیستم ها به یکدیگر وابسته باشند، شما به راحتی می توانید این وابستگی را از با محدود کردن ارتباط بین آن ها فقط از طریق نمایشان، ساده سازی کنید.

ساختار



class diagram of Façade Pattern - ۷۳

^{۳۵} Entry Point

Facade

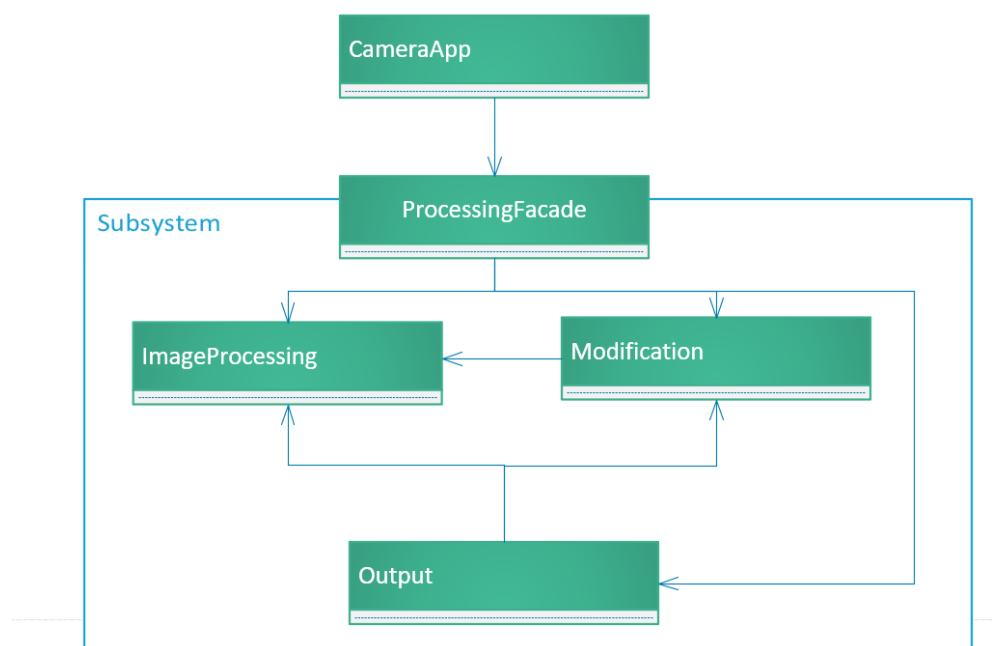
این واسط می‌داند که کدام کلاس‌های یک زیرسیستم مسئول چه کاری هستند و درخواست‌های مشتری را به آن‌ها محو می‌کند.

Subsystem classes

کلاس‌هایی هستند که کارایی یک زیرسیستم را پیاده‌سازی می‌کنند. آن‌ها هم می‌توانند مستقیماً کار مشتری و هم کاری که توسط شی Facade به آن محو گردیده را انجام دهند. با این حال، هیچ دانشی از ندارند، پس هیچ ارجاعی^{۳۶} از آن را در خود نگه نمی‌دارند.

مثال

می‌خواهیم برنامه‌ای طراحی کنیم که اطلاعات ورودی یک دوربین را بگیریم، بتوانیم آن را پردازش، ویرایش و خروجی (در فرمت jpg) بگیریم. واضح است که مشتری ما برنامه یک دوربین است. می‌خواهیم سیستم را طور طراحی کنیم که برنامه دوربین خود تصمیم بگیرد که آیا می‌خواهد تمامی این مراحل را طی کند و یا فقط می‌خواهد تصویر پردازش شده و به کاربر نشان داده شود. اما در اغلب اوقات نیاز است هر سه مورد بالا انجام شود. همچنین می‌خواهیم در آینده بتوانیم قابلیت‌های جدید به آن اضافه کنیم. از توضیحات این سیستم متوجه می‌شویم که الگوی مورد نظر Facade است. طراحی و پیاده‌سازی سیستم ما به شکل زیر است.



شکل ۷۴ - سیستم طراحی شده class diagram

^{۳۶} Reference

```

class CameraApp:
    def __init__(self, data):
        self.data = data

    def takePicture(self):
        self.facade = ProcessingFacade(self.data)
        print(self.facade.output())


class ImageProcessing:
    def __init__(self, rawData):
        self.rawData = rawData

    def process(self):
        self.result = f"Processed Data+-{self.rawData}"
        return self


class Modification:
    def __init__(self, imageProcessing):
        self.imageProcessing = imageProcessing

    def modify(self):
        self.result = f"Modified Data**{self.imageProcessing.result}"
        return self


class Output:
    def __init__(self, finalData):
        self.finalData = finalData
    def export(self):
        return "Image Ready! output.jpg"


class ProcessingFacade:
    def __init__(self, rawData):
        self.processed = ImageProcessing(rawData).process()
        self.modified = Modification(self.processed).modify()
        self.result = Output(self.modified).export()

    def output(self):
        return self.result


camera = CameraApp("1010100111101110011110110010110111010110")
camera.takePicture()

onlyProcessedImage = ImageProcessing(
    "1010100111101110011110110010110111010110").process().result

print(onlyProcessedImage)

```

شکل ۷۵ - پایه سازی سیستم با

```

Image Ready! output.jpg
Processed Data+-1010100111101110011110110010110111010110

```

شکل ۷۶ - خروجی کد ۷۴

الگوی Proxy

تعریف

الگوی Proxy یک جانشین برای یک شئ دیگر، به منظور کنترل دسترسی به آن، فراهم می‌کند.

هدف

یکی از دلایلی که می‌خواهیم دسترسی به به یک شئ را کنترل کنیم، به تاخیر انداختن هزینه‌ی ایجاد آن شئ، تا زمانی که واقعاً به آن نیاز داریم، است. به طور مثال فرض کنید که می‌خواهیم ویرایشگر سند^{۳۷} طراحی کنیم. ساخت بعضی شئ‌های گرافیکی، مانند عکس‌های رستر، زمان زیادی می‌برد ولی ما می‌خواهیم که برنامه به سرعت باز شود، پس باید از ساخت شئ‌های پرهزینه در ابتدای کار اجتناب کنیم. در هر صورت ساخت تمامی اشیاء گرافیکی در همان ابتدا معقول نیست زیرا قرار نیست همه‌ی آن اشیاء در یک زمان نشان داده شوند.

با توجه به این محدودیت‌ها، تنها هنگامی که یک شئ نیاز به دیده شدن دارد، باید آن را بسازیم. راه حل استفاده از یک پروکسی –Proxy- است که به عنوان یک جانشین برای عکس واقعی عمل می‌کند. این پروکسی تنها مانند یک عکس رفتار می‌کند و در هنگام نیاز به عکس، آن را نمونه‌سازی می‌کند.

موارد استفاده

الگوی Proxy هنگامی مورد استفاده قرار می‌گیرد که به یک ارجاع پیچیده یا چندکاره، به جای یک ارجاع ساده، نیاز داریم. چند موقعیت که در آن، این الگو قابل استفاده است در زیر آمده است:

- پروکسی دور^{۳۸}، نمایشی محلی از یک شئ را در یک فضای آدرس متفاوت ارائه می‌کند.
- پروکسی مجازی^{۳۹}، اشیاء پرهزینه را در زمان نیاز می‌سازد. پروکسی در مثال بالا از همین نوع است.
- پروکسی حافظتی^{۴۰}، دسترسی به شئ اصلی را کنترل می‌کند. پروکسی‌های حافظتی هنگامی پرکاربرد هستند که اشیاء باید سطوح دسترسی متفاوتی داشته باشند. پروکسی کرنل در سیستم عامل Choices، دسترسی‌های حافظت شده‌ای به اشیاء سیستم عامل می‌دهد.

^{۳۷} Document Editor

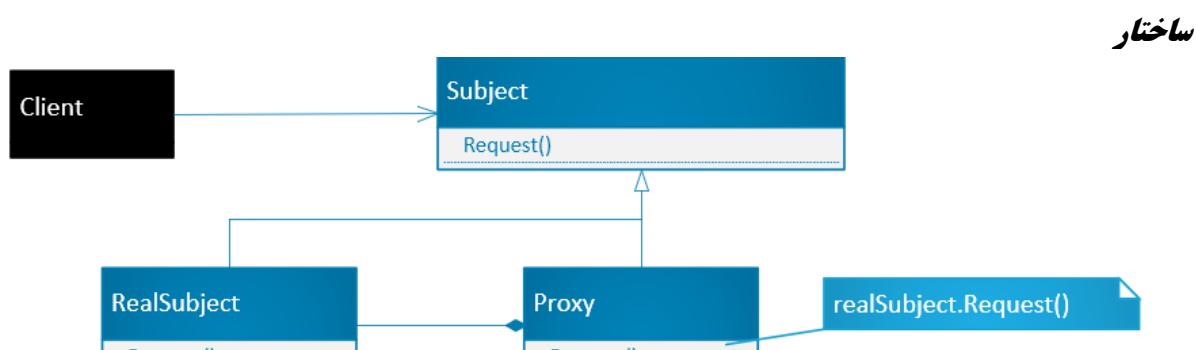
^{۳۸} Remote Proxy

^{۳۹} Virtual Proxy

^{۴۰} Protection Proxy

- اشاره‌گر هوشمند^{۴۱}، جایگزینی برای یک اشاره‌گر است که هنگامی که به یک شی دسترسی پیدا می‌شود، عملیات اضافه‌ای را انجام می‌دهد. از کاربردهای رایج آن می‌توان به موارد زیر اشاره کرد:

- شمارش تعداد اشاره‌گرهایی که به شیء اصلی اشاره می‌کنند تا هنگامی که دیگر هیچ ارجاعی وجود ندارد، آن را از حافظه آزاد کند.
- بارگذاری یک شیء همیشگی در حافظه هنگامی که برای اولین بار ارجاع داده می‌شود
- بررسی اینکه شیء اصلی پیش از دسترسی به آن، قفل شده است تا شیء دیگری نتواند آن را تغییر دهد.



شكل ۷۷ Class diagram of Proxy Pattern – ۷۷



شكل ۷۸ Object diagram of Proxy Pattern - ۷۸

Proxy

یک ارجاع از شیء اصلی را در خود نگه می‌دارد که به آن اجازه دسترسی به شیء را می‌دهد. Proxy ممکن است به یک Subject اشاره داشته باشد اگر RealSubject و Subject یکی باشند. همچنین واسطی است که با واسط Subject یکسان است پس یک پروکسی می‌تواند جایگزینی برای شیء واقعی باشد. این کلاس دسترسی به شیء اصلی را نیز کنترل می‌کند و مسئولیت ایجاد و حذف آن را بر عهده دارد.

^{۴۱} Smart reference

مسئولیت های دیگر این کلاس می تواند به شکل زیر باشد:

- پروکسی های دور وظیفه رمزگذاری^{۴۲} یک درخواست و آرگومان های آن، و ارسال آن درخواست رمزگذاری شده به شیء اصلی در یک فضای آدرس متفاوت دارد.
- پروکسی های مجازی باید اطلاعات اضافی در مورد شیء اصلی را در حافظه کش^{۴۳} ذخیره کند تا بتوانند دسترسی به آن را به تاخیر بیندازند. در مثال گفته شده در قسمت "هدف"، پروکسی ابعاد یک عکس واقعی را در حافظه کش ذخیره می کند.
- پروکسی های حفاظتی بررسی می کنند که آیا صدا زننده اجازه دسترسی برای اجرای یک درخواست را دارد یا خیر.

Subject

یک واسط برای RealSubject و Proxy تعریف می کند تا یک Proxy بتواند در هر جایی به جای RealSubject استفاده شود.

RealSubject

شیء اصلی که پروکسی آن را نمایندگی می کند، تعریف می کند.

مثال

همانطور که در مثال قسمت "هدف" گفته شد، می خواهیم یک ویراشگر متن بنویسیم که هر موقع به عکس های رستر برخورد، آن را بارگذاری کند. پروکسی یک تصویر باید ابعاد آن را داشته باشد تا ویرایشگر بتواند هنگام بارگذاری سند، فاصله ها و صفحه بندی را به درستی انجام دهد.

برای شروع پیاده سازی، ابتدا یک کلاس انتزاعی Element می سازیم که در آن تابع render() تعریف شده است.

```
class Element(ABC):
    @abstractmethod
    def render():
        pass
```

شکل ۷۹ - پیاده سازی کلاس Element

^{۴۲} Encrypt

^{۴۳} Cache

سپس کلاس های PictureProxy و Picture.Text را پیاده سازی می کنیم:

```
class Text(Element):
    def __init__(self, text):
        self.text = text

    def render(self):
        print(f"Rendered Text: **** \n {self.text}")


class Picture(Element):
    def __init__(self, pic, height, width):
        self.pic = pic
        self.height = height
        self.width = width

    def render(self):
        print(
            f"\nRendered Picture:*** \n {self.pic} -{self.height} * {self.width}")


class PictureProxy(Element):
    def __init__(self, picture):
        self.picture = picture

    def render(self, xFactor):
        if(self.check_access(xFactor)):
            self.picture.render()
        else:
            print(
                f"\nProxy ! : \n there's a picture with the title {self.picture.pic} and the dimentions of {self.picture.height} * {self.picture.width} ")

    def check_access(self, xFactor):
        # Proxy: Checking access prior to firing a real request
        if(xFactor):
            return True
        else:
            return False
```

شکل ۸۰ - کلاس های PictureProxy و Picture.Text

متده است PictureProxy در کلاس checkAccess بررسی می کند که آیا الان عکس اصلی رندر شود یا پروکسی آن. آرگومان xFactor تعیین کننده ی این شرط است. xFactor می توانند هر عامل تعیین کننده ای مانند شماره هی صفحه، تاییدیه نیاز به عکس و غیره باشد. ما برای سادگی این فاکتور یک مقدار دودویی ^{۴۴} در نظر گرفتیم.

حال برنامه را تست و اجرا می کنیم:

```
someText = Text(
    "hey this sis a sample text. you can ignore it but i wouldn't recommend ou to do so.")

aPicture = Picture("Picture of a Cow", 1080, 720)

aPictureProxy = PictureProxy(aPicture)

someText.render()
aPictureProxy.render(False)
aPictureProxy.render(True)
```

شکل ۸۱ - تست و اجرای برنامه

```
Rendered Text: ****
hey this sis a sample text. you can ignore it but i wouldn't recommend ou to do so.

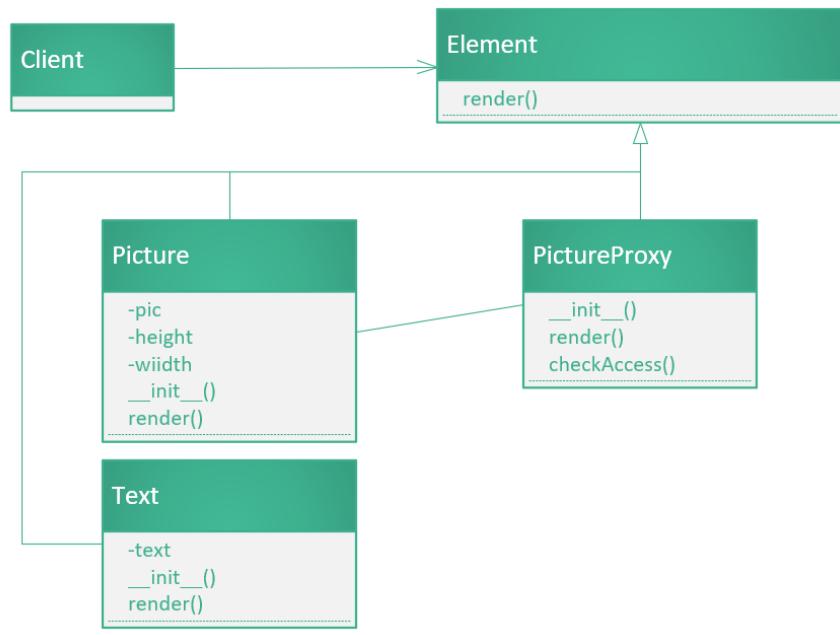
Proxy ! :
there's a picture with the title Picture of a Cow and the dimentions of 1080 * 720

Rendered Picture:***
Picture of a Cow -1080 * 720
```

شکل ۸۲ - خروجی کد بالا

^{۴۴} Boolean

ساختار برنامه طراحی شده به شکل زیر است:



شکل class diagram – ۸۳

الگوی Flyweight

تعریف

الگوی Flyweight از اشتراک‌گذاری برای پشتیبانی بهینه از تعداد زیادی شئ با جزئیات زیاد، استفاده می‌کند.

هدف

در بعضی از اپلیکیشن‌ها نیاز است تا از اشیاء در سراسر برنامه استفاده شود اما پیاده‌سازی ساده‌انگارانه می‌تواند سنگین تمام شود. به طور مثال ممکن است در یک ویرایشگر سند، هر حرف به عنوان یک شئ در نظر گرفته شود. با اینکه این کار انعطاف‌پذیری فوق‌العاده‌ای به برنامه می‌دهد اما انجام آن بسیار پرهزینه است. الگوی Flyweight نحوه‌ی اشتراک‌گذاری اشیاء را به منظور استفاده آن‌ها در سطوح مختلف داده^{۴۵} بدون هزینه سنگین را توضیح می‌دهد.

یک سبک وزن – flyweight – یک شئ اشتراکی است که می‌تواند در چندین موقعیت، به طور همزمان، استفاده شود. شئ سبک‌وزن در هر موقعیت به عنوان یک شئ مستقل عمل می‌کند – مانند یک نمونه از شئ ای که به اشتراک گذاشته نشده است. شئ سبک‌وزن نمی‌تواند در مورد موقعیتی که از آن استفاده می‌شود، پیش‌بینی داشته باشد. نکته‌ی کلیدی در اینجا، تفاوت بین حالت درونی^{۴۶} و برونی^{۴۷} است. حالت درونی اطلاعاتی است که بین تمامی شئ‌ها مشترک است اما حالت بیرونی اطلاعات منحصر به فرد هر شئ است. حالت درونی در شئ سبک‌وزن ذخیره می‌شود. این حالت شامل اطلاعاتی است که از موقعیت سبک‌وزن مستقل است و همین ویژگی آن را قابل اشتراک‌گذاری می‌کند. حالت خارجی به موقعیت سبک‌وزن وابسته است؛ در نتیجه نمی‌تواند به اشتراک گذاشته شود. اشیاء مشتری وظیفه‌ی انتقال حالت خارجی به سبک‌وزن را در هنگام نیاز دارند. این موارد در قسمت مثال به تفصیل توضیح داده شده است.

موارد استفاده

کارایی الگوی Flyweight تا حد زیادی به اینکه کجا و چگونه استفاده شده است، وابسته است. تنها زمانی از این الگو استفاده کنید که تمامی ۵ شرط زیر برقرار باشد:

۱. یک برنامه شامل تعداد زیادی شئ باشد.

۲. هزینه‌ی ذخیره‌سازی، به دلیل تعداد بالای اشیاء، زیاد باشد.

۳. اکثر اشیاء بتوانند بیرونی سازی شوند.

^{۴۵} Granularities

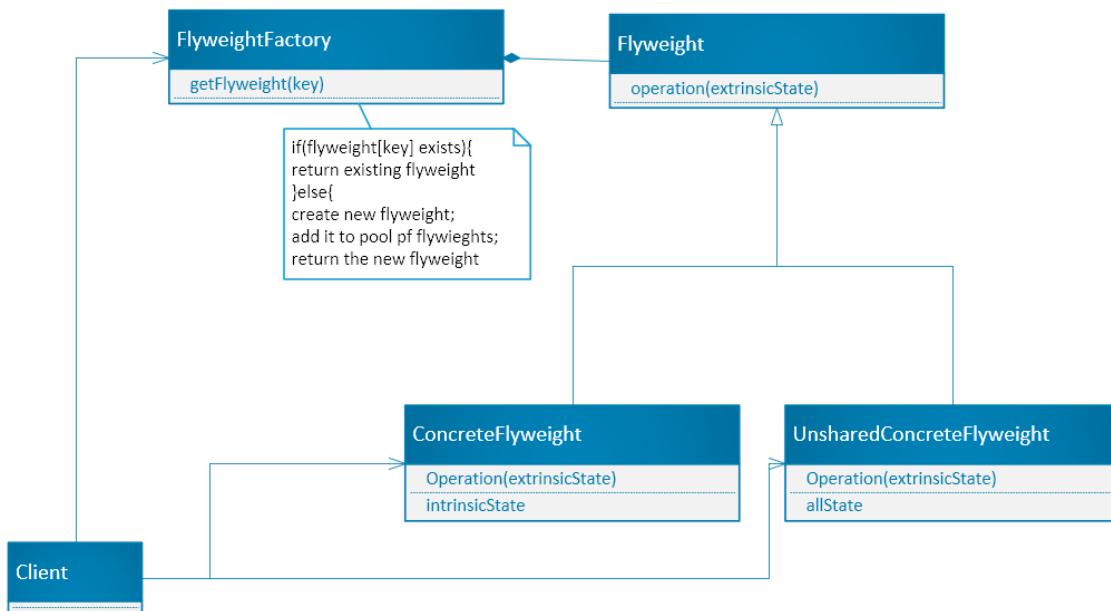
^{۴۶} Intrinsic

^{۴۷} Extrinsic

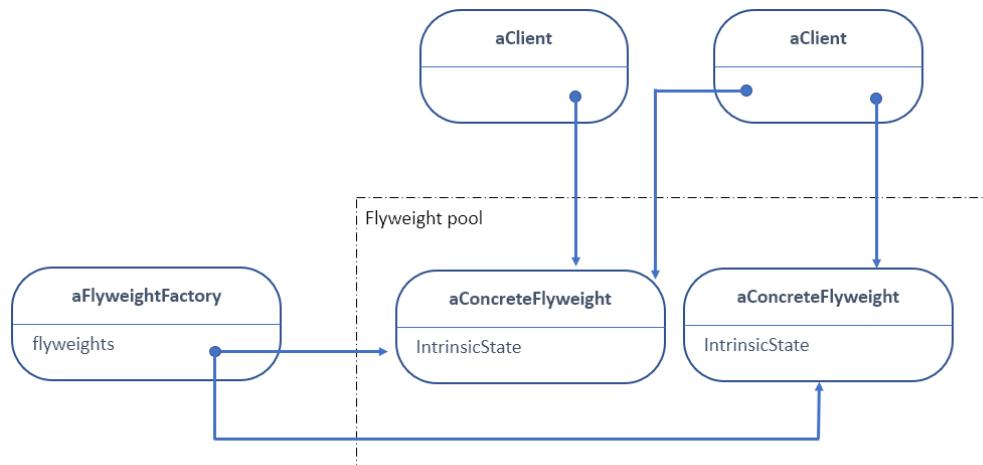
۴. گروه زیادی از اشیاء ، در هنگامی که حالت بیرونی حذف شده است، بتوانند با چندین شی اشتراکی جایگزین شوند.

۵. برنامه به هویت شی وابسته نباشد. از آنجاییکه اشیاء سبک وزن باید به اشتراک گذاشته شوند، تست‌های هویت همیشه مقدار True را برای اشیاء متفاوت برمی‌گرداند

ساختار



class diagram of Flyweight pattern - ۸۴



object diagram of Flyweight pattern-۸۵

Flyweight

یک واسط تعریف می کند که سبک وزن ها با استفاده از آن می توانند حالت بیرونی را دریافت و روی آن عملیاتی انجام دهند.

ConcreteFlyweight

واسط Flyweight را پیاده سازی می کند و فضای ذخیره سازی حالت درونی احتمالی را فراهم می کند. یک شی Flyweight باید قابل اشتراک گذاری باشد و هر حالتی که ذخیره می کند، باید درونی باشد که به ConcreteFlyweight این معناست که این حالت باید مستقل از موقعیت شی ConcreteFlyweight باشد.

UnsharedConcreteFlyweight

نیازی نیست همه زیر کلاس های Flyweight به اشتراک گذاشته شوند. واسط Flyweight این کار را اجبار نمی کند. داشتن شی UnsharedConcreteFlyweight، به عنوان فرزند برای ConcreteFlyweight در بعضی از سطوح ساختاری، معمول است.

FlyweightFactory

شی های سبک وزن را ساخته و مدیریت می کند. همچنین از به اشتراک گذاشته شدن سبک وزن ها اطمینان حاصل می کند. هنگامی که یک مشتری، یک سبک وزن درخواست می کند، شی FlyweightFactory یک نمونه موجود در غیر این صورت یک نمونه جدید، را فراهم می کند.

Client

دارای یک ارجاع به سبک وزن هاست. همچنین حالت بیرونی آن ها را محاسبه و ذخیره می کند.

مثال

می خواهیم ویرایشگر متنی بنویسیم که در آن هر حرف یک شی باشد که اطلاعات نظیر کد اسکی^{۴۸}، تایپوگرافی، رنگ، اندازه و مختصات را در خود ذخیره کند. فرض کنید می خواهیم فایلی را باز کنیم که ۱۰۰.۰۰۰.۰۰۰ کلمه دارد. فرض می کنیم هر کلمه به طور متوسط از ۵ حرف تشکیل شده است. پس با این شرایط ۵ هزار حرف داریم که برای تک تک آن ها باید شی ای بسازیم. ما اینکار را با استفاده از یک حلقه for شبیه سازی کرده ایم.

^{۴۸} ASCII code

```

class Letter:
    def __init__(self, ASCIICode, typography, color, size, coordinate):
        self.ASCIICode = ASCIICode
        self.typography = typography
        self.color = color
        self.size = size
        self.coordinate = coordinate

    def draw(self):
        pass

start_time = time.time()
for i in range(500000):
    globals()[f"object{i}"] = Letter("69", "A", "black", "12", [i, i])

process = psutil.Process(os.getpid())
print(
    colored(f"--- CPU TIME ---{(time.time() - start_time)} seconds", "red"))

# in Megabytes
print(
    colored(f"--- Memory Used--- {process.memory_info().rss/1048576} MB", "red"))

```

شکل ۸۶ - شبیه سازی نمونه سازی حروف با استفاده از حلقه `for`

CPU TIME
1.5897488594055176 seconds
Memory Used
220.6484375 MB

شکل ۸۷ - منابع استفاده شده برای اجرای این کد

همانطور که مشاهده می شود ساخت این تعداد شیء بسیار گران است و ۲۲۰ مگابایت از حافظه رم را اشغال می کند. برای حل این مشکل ابتدا باید توجه کرد که بسیاری از داده های این اشیاء، مانند تایپوگرافی، رنگ و اندازه و کد اسکی یکسان هستند. در عین حال بعضی از داده ها مانند مختصات متغیر هستند. کاری که باید انجام دهیم این است که مجموعه ای از اشیاء را که داده های یکسان زیادی دارند، را به عنوان یک شیء جدید در نظر بگیریم. به طور مثال حرف A ای که رنگ آن مشکی، اندازه آن ۱۴، تایپوگرافی آن Times new Roman است را به عنوان یک نمونه در نظر می گیریم. تنها داده ای که برای اعضای این مجموعه متفاوت است، مختصات آن ها است. پس صفات کد اسکی، تایپوگرافی، اندازه و رنگ حالت درونی و مختصات حالت بیرونی است. حال باید یک شیء سبک وزن طراحی کنیم که نمونه هایی مانند این نمونه را در خود ذخیره کند و هر زمان که نیاز به فراخوانی و استفاده از آن نمونه داشتیم، حالت بیرونی را از به شکل آرگومان به متد مناسب آن کلاس، که در اینجا متد `draw` می باشد، انتقال می دهیم.

پس ابتدا یک کلاس سبک وزن به نام `Glyph` می سازیم که حالات درونی در آن ذخیره شده و متد `draw` در آن تعریف شده باشد.

```

class Glyph(ABC):
    def __init__(self, ASCIICode, typography, color, size):
        self.ASCIICode = ASCIICode
        self.typography = typography
        self.color = color
        self.size = size

    @abstractmethod
    def draw(self, coordinate):
        pass

```

شکل ۸۸ - پیاده سازی کلاس *Glyph*

کلاس *Charcater* را پیاده سازی می کنیم:

```

class Charcater(Glyph):
    def draw(self, coordinate):
        #print(f"{chr(int(self.ASCIICode))} at {coordinate} ({self.typography})")
        pass

```

شکل ۸۹ - پیاده سازی کلاس *Charcater*

حال نوبت آن است که کلاس *GlyphFactory* را طراحی کنیم. این کلاس هنگامی که نرم افزار باید به طور مثال حرف B مشکی با فونت Times و سایز ۱۴ در مختصات (۱۰۱) را نشان دهد، ابتدا بررسی می کند که آیا شئ ای با چنین مشخصاتی وجود دارد یا نه. اگر وجود داشت، متده *draw* آن را صدا می زند و مختصات را به آن تحويل می دهد. اما اگر شئ ای با این مشخصات وجود نداشت، ابتدا یک شئ *Glyph* ساخته، آن را در *glyphContext* ذخیره کرده و آن شئ را برمی گرداند.

```

class GlyphFactory:
    _glyphContext: Dict[str, Charcater] = {}

    def getCharacter(self, ASCIICode, typography, color, size):
        if self._glyphContext.get(f"{ASCIICode}{typography}{color}{size}"):
            return self._glyphContext.get(f"{ASCIICode}{typography}{color}{size}")
        else:
            newCharacter = Charcater(ASCIICode, typography, color, size)
            self._glyphContext[f"{ASCIICode}{typography}{color}{size}"] = newCharacter
            return newCharacter

```

شکل ۹۰ - پیاده سازی کلاس *GlyphFactory*

در انتها تابعی به نام *draw* تعریف می کنیم که از *glyphFactory* استفاده کرده و با هر بار صدا زدن، یک شئ را، چه شئ جدید باشد و چه شئ تکراری، در یک متغیر که نام آن با توجه به مقدار *i* تغییر می کند، ذخیره می کند.

```

glyphFactory = GlyphFactory()

def draw(ASCIICode, typography, color, size, coordinate, i):
    globals()[f"object{i}"] = glyphFactory.getCharacter(
        ASCIICode, typography, color, size)
    globals()[f"object{i}"].draw(coordinate)

```

شکل ۹۱ - پیاده سازی تابع *draw*

حال با استفاده از حلقه *for* شبیه سازی را انجام می دهیم:

```

start_time = time.time()
for i in range(500000):
    draw("69", "Times", "black", "14", (i, i), i)

process = psutil.Process(os.getpid())
print(
    colored(f"CPU TIME \n{(time.time() - start_time)} seconds", "red"))

# in Megabytes
print(
    colored(f"Memory Used \n {process.memory_info().rss/1048576} MB", "red"))

```

شکل ۹۲ - پیاده سازی حلقه for

```

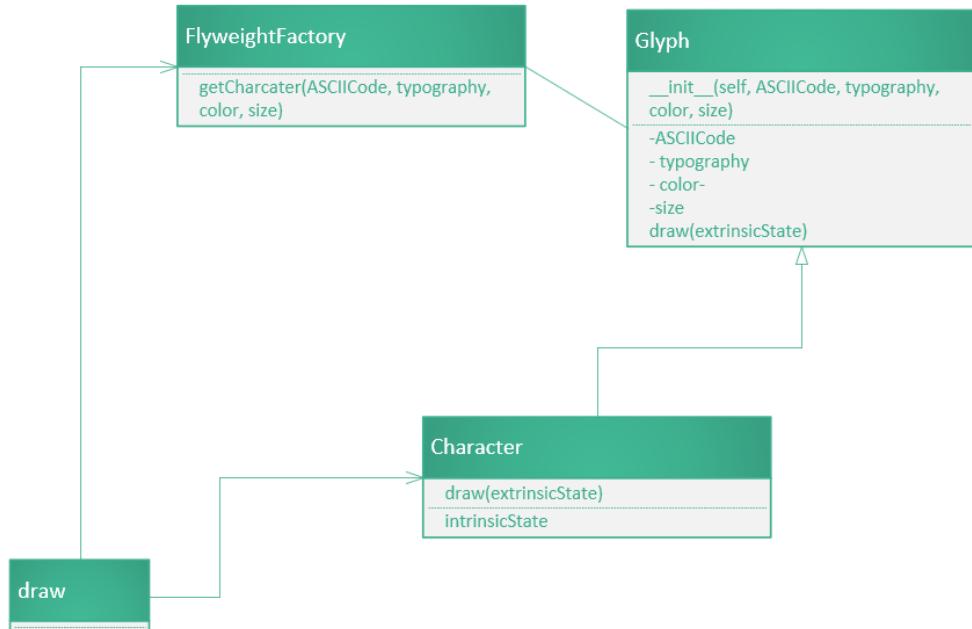
CPU TIME
1.881305456161499 seconds
Memory Used
89.0 MB

```

شکل ۹۳ - منابع استفاده شده برای اجرای این کد

همانطور که مشاهده می شود، حافظه مصرف شده در برنامه جدید سه برابر کمتر از حافظه مصرف شده در برنامه اولیه است.

ساختار برنامه جدید به شکل زیر است:



شکل ۹۴ - برنامه جدید class diagram

الگوهای رفتاری

الگوهای رفتاری به الگوریتم‌ها و تخصیص مسئولیت‌ها بین اشیاء مربوط هستند. الگوهای رفتاری نه تنها الگوهای اشیاء و کلاس‌ها را بلکه الگوهای ارتباط میان آن‌ها را نیز توصیف می‌کند. این الگوهای جریان کنترل پیچیده را، که دنبال کردن آن در زمان اجرا دشوار است، توصیف می‌کنند. آن‌ها تمرکزان را از روی جریان کنترل بر می‌دارند و به شما اجازه می‌دهند تا تنها روی چگونگی ارتباط اشیاء تمرکز کنید.

الگوهای رفتاری کلاس از ارث بری برای توزیع رفتار میان کلاس‌ها استفاده می‌کنند. الگوهای رفتاری شئ از ترکیب اشیاء برای این هدف بهره می‌برد. بعضی از آن‌ها چگونگی همکاری اشیاء به منظور اجرای عملیاتی که به تنها‌ی توسط هیچ شئ ای ممکن نبود را توصیف می‌کنند.

الگوی Chain Of Responsibility

تعریف

این الگو از جفت شدگی^{۴۹} ارسال کننده یک درخواست و دریافت کننده آن جلوگیری می‌کند. این کار با استفاده از اجازه دادن به چندین شیء برای مدیریت کردن یک درخواست صورت می‌گیرد. بدین صورت که اشیاء دریافت کننده به یکدیگر زنجیر شده اند و درخواست را در طول زنجیر انتقال می‌دهند. این کار تا زمانی ادامه پیدا می‌کند که یک شیء در زنجیر بتواند آن درخواست را مدیریت کند.

هدف

یک دستیار کمکی حساس به متن^{۵۰} برای رابط کاربری گرافیکی را در نظر بگیرید. کاربر می‌تواند با کلیک کردن روی هر قسمتی از این رابط، توضیحاتی در مورد آن قسمت بدست آورد. این توضیحات در قسمت‌های مختلف متغیر است. به طور مثال یک دکمه در باکس دیالوگ توضیحات متفاوتی از یک دکمه مشابه در صفحه اصلی دارد. اگر آن قسمت دارای هیچ توضیحاتی نبود، باید اطلاعات کلی تری از محیطی که کاربر در آن قرار دارد، نشان داده شود. (در این مثال صفحه اصلی)

بنابراین، مرتب کردن توضیحات با توجه به عمومیت آن‌ها - از اختصاصی ترین تا عمومی ترین آن‌ها - امری طبیعی است. علاوه بر این، روشی است که یک درخواست کمک توسط یکی از چندین اشیاء رابط کاربری مدیریت می‌شود؛ درخواستی که به شرایط و میزان اختصاصی بودن توضیحات موجود وابسته است.

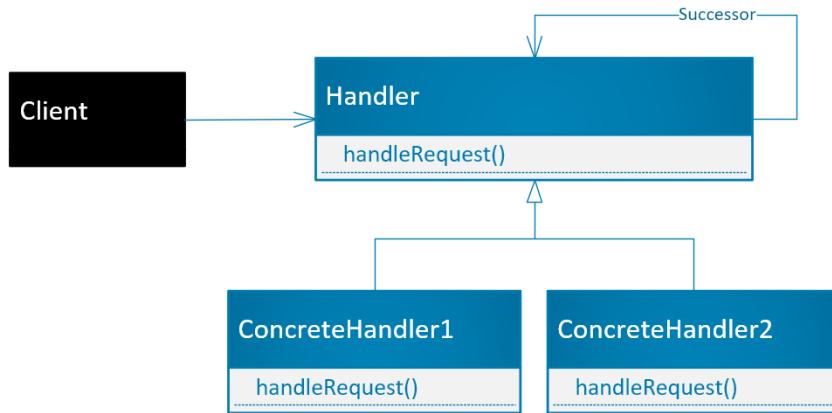
اما مشکل اینجاست که شیء ای که توضیحات کمکی را ارائه می‌دهد، برای شیء ای که درخواست کمک را آغاز می‌کند (مانند یک دکمه) شناخته شده نیست. پس باید راهی برای جداسازی دکمه ای که درخواست توضیحات کمکی را ارسال می‌کند و شیء ای که آن توضیحات را ارائه می‌دهد، پیدا کنیم. الگوی Chain of Responsibility (زنجیره‌ی مسئولیت) نحوه‌ی انجام این کار را توضیح می‌دهد.

موارد استفاده

- هنگامی که ممکن است بیش از یک شیء درخواستی را مدیریت کنند و مدیریت کننده مشخص نیست. مدیریت کننده باید به صورت خودکار پیدا شود.
- هنگامی که می‌خواهید بدون آنکه دریافت کننده را به طور واضح مشخص کنید، درخواستی را به یکی از چندین اشیاء بفرستید
- هنگامی که مجموعه از اشیاء که می‌توانند یک درخواست را مدیریت کنند، باید به طور پویا مشخص شوند.

^{۴۹} Coupling

^{۵۰} Context-sensitive



شکل ۹۵ class diagram of Chain Of Responsibility Pattern



شکل ۹۶ Object diagram of Chain Of Responsibility Pattern

Handler

یک رابط برای مدیریت درخواست‌ها ایجاد می‌کند. همچنین می‌تواند ارتباط جانشین را نیز پیاده سازی کند.

ConcreteHandler

درخواستی که وظیفه او است را مدیریت می‌کند. همچنین می‌تواند به جانشین‌های خود دسترسی داشته باشد. اگر **ConcreteHandler** بتواند درخواست را مدیریت کند، این کار را انجام خواهد داد. در غیر اینصورت درخواست را به جانشین خود منتقل می‌کند.

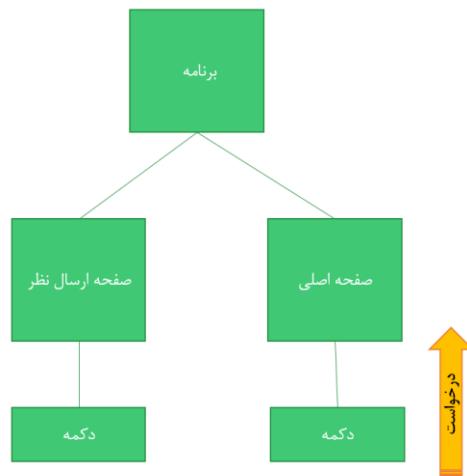
Client

یک درخواست را برای **ConcreteHandler** ایجاد می‌کند.

مثال

همانطور که در بخش "هدف" گفته شد، می‌خواهیم یک دستیار کمکی حساس به متن طراحی کنیم که هنگامی که کاربر روی هر شئ ای که کلیک می‌کند، توضیحاتی در مورد آن داده شود. یک شئ در موقعیت های مختلف می‌تواند توضیحات مختلفی داشته باشد. مثلاً یک دکمه در صفحه اصلی توضیح متفاوتی از یک دکمه در صفحه ارسال نظر دارد. همچنین اگر شئ ای دارای توضیحات نبود، باید توضیحات والد آن نمایش داده شود که در این مثال والد می‌تواند صفحه اصلی یا صفحه ارسال نظر باشد.

پس ما یک ساختار زنجیری (درختی) داریم که در آن یک درخواست به ترتیب به گره‌های بعدی فرستاده شده و در گره مناسب اجرا و مدیریت می‌شود.



شکل ۹۷ - ساختار درختی نرم‌افزار

هر کدام از اجزای بالا یک مدیریت کننده درخواست محسوب می‌شوند. پس نیاز به کلاسی داریم که همه‌ی این اجزا، کلاس‌ها، از آن ارث بری کنند:

```
class HelpHandler(ABC):
    _successor: HelpHandler

    def __init__(self, successor=None):
        self._successor = successor

    @abstractmethod
    def showInfo(self):
        pass

    def handleHelpInfo(self):
        if(self.condition()):
            self.showInfo()
        else:
            if(self._successor):
                self._successor.showInfo()
            return False

    def condition(self):
        if(hasattr(self, 'info')):
            return True
        else:
            return False

    def setInfo(self, info):
        self.info = info
```

شکل ۹۸ - پیاده سازی کلاس HelpHandler

متدهای `handleHelpInfo` و `showInfo` مدیریت درخواست را بر عهده دارد. بدین صورت که اگر ویژگی `info` در شی ای وجود نداشت، متدهای `showInfo` و `handleHelpInfo` جانشین (والد) آن صدای زده شود. این متدهای در هر کدام از زیرکلاس های این کلاس، که در این مثال `CommentPage`، `Application` و `Button` هستند، پیاده‌سازی شده است. متدهای `setInfo` برای تخصیص توضیحات به شی مورد نظر در کلاس `HelpInfo` پیاده‌سازی شده است.

پیاده‌سازی کلاس های `CommentPage`، `Application` و `Button` به صورت زیر است:

```
class Application(HelpHandler):
    def showInfo(self):
        print(f"Application: {self.info}")

class CommentPage(HelpHandler):
    def showInfo(self):
        print(f"CommentPage: {self.info}")

class Button(HelpHandler):
    def showInfo(self):
        print(f"Button: {self.info}")
```

شکل ۹۹ - پیاده‌سازی کلاس های `CommentPage`، `Application` و `Button`

حال برنامه را تست می‌کنیم. ابتدا یک شی برنامه می‌سازیم که هیچ والد ندارد. سپس صفحه‌ی نظرات و بعد از آن دو دکمه در صفحه نظرات می‌سازیم که دکمه اول بدون توضیحات و دکمه دوم دارای توضیحات است.

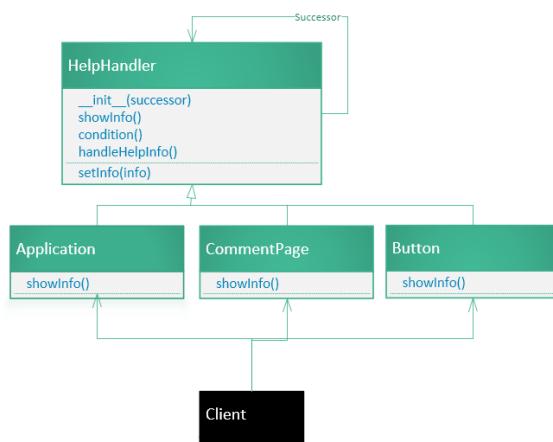
```
application = Application()
application.setInfo("This is just a test application")
commentPage = CommentPage(application)
commentPage.setInfo("You can send your comment in this page")
button1 = Button(commentPage)
button2 = Button(commentPage)
button2.setInfo("this is the second button which includes help info")
button1.handleHelpInfo()
button2.handleHelpInfo()
```

شکل ۱۰۰ - تست و اجرا برنامه

```
CommentPage: You can send your comment in this page
Button: this is the second button which includes help info
```

شکل ۱۰۱ - خروجی کد بالا

همانطور که انتظار داشتیم، هنگامی که روی دکمه اول کلیک می‌کنیم، توضیحات والد آن یعنی صفحه نظرات و هنگامی که روی دکمه دوم کلیک می‌کنیم، توضیحات خود دکمه نمایش داده می‌شود. ساختار برنامه به شکل زیر است:



شکل ۱۰۲ - برنامه class diagram

الگوی Command

تعریف

الگوی Command یک درخواست را در قالب یک شئ کپسوله‌سازی می‌کند. این کار به شما اجازه می‌دهد تا مشتری‌ها را با درخواست‌ها، صفات یا لاغ درخواست‌ها پارامتریندی^{۵۱} و از عملیات برگشت‌پذیر پشتیبانی کنید.

هدف

گاهی اوقات نیاز است تا درخواست‌هایی به اشیاء فرستاده شوند بدون آنکه از آن درخواست و یا شئ گیرنده آن اطلاعاتی داشته باشیم. به طور مثال، جعبه‌ابزارهای رابط کاربری^{۵۲} شامل اشیائی مانند دکمه‌ها و منوها هستند که یک درخواست را با توجه به ورودی کاربر اجرا می‌کنند. اما جعبه‌ابزار نمی‌تواند درخواست را به طور صریح در دکمه یا منو پیاده‌سازی کند؛ چرا که تنها برنامه‌ای که از آن جعبه‌ابزار استفاده کرده است از وظیفه‌ی آن دکمه یا منو باخبر است. ما به عنوان طراحان این جعبه‌ابزار راهی برای پی‌بردن به اطلاعات درخواست و گیرنده آن نداریم.

الگوی Command به جعبه‌ابزار این اجازه را می‌دهد که درخواست‌هایی از برنامه‌های نامشخص بسازد. در این روش، درخواست خود به یک شئ تبدیل می‌شود. این شئ می‌تواند ذخیره و یا منتقل شود. ایده‌ی اصلی این الگو یک کلاس انتزاعی Command است که یک رابط برای اجرای عملیات‌ها تعریف می‌کند. در ساده‌ترین شکل، این رابط دارای یک متده انتزاعی execute است. زیرکلاس‌های Concrete Command یک جفت گیرنده – عمل را مشخص می‌کنند که این کار با ذخیره کردن گیرنده در یک متغیر نمونه و پیاده‌سازی execute برای فرآخوانی آن درخواست انجام می‌دهد. گیرنده دانش مورد نیاز برای اجرای درخواست را دارد.

موارد استفاده

- هنگامی که می‌خواهید اشیاء را با یک عملیات اجرایی پارامتریندی کنید. شما می‌توانید چنین پارامتریندی‌هایی را در یک زبان روندی با استفاده از یک تابع برگشتی، که تابعی است که جایی ثبت

^{۵۱} Parameterize

^{۵۲} User interface toolkits

می شود تا در نقطه زمانی دیگری فراخوانی شود، انجام دهد. Command ها همان توابع برگشتی در برنامه نویسی شئ گرا هستند.

• هنگامی که می خواهید درخواست ها را در زمان های مختلف مشخص، صفت بندی، و اجرا کنید. یک شئ Command می تواند زیست ^{۵۳} مستقلی از درخواست اصلی داشته باشد. اگر دریافت کننده یک درخواست بتواند به صورت یک آدرس مستقل از فاصله نمایش داده شود، شما می توانید یک شئ درخواست آن درخواست به یک پردازش دیگر منتقل کنید و درخواست را آنجا انجام دهید.

• هنگامی که می خواهید از برگشت به قبل ^{۵۴} پشتیبانی کنید. عمل Execute در Command می تواند حالت را برای معکوس سازی اثرات آن در شئ Command ذخیره سازی کند. رابط Command باید یک عمل Unexecute داشته باشد که اثرات فراخوانی قبلی Execute را معکوس کند. دستوارت اجرا شده در یک لیست تاریخچه ذخیره می شوند. با حرکت به جلو و عقب در این لیست می توان به undo و redo بی نهایت دست یافت.

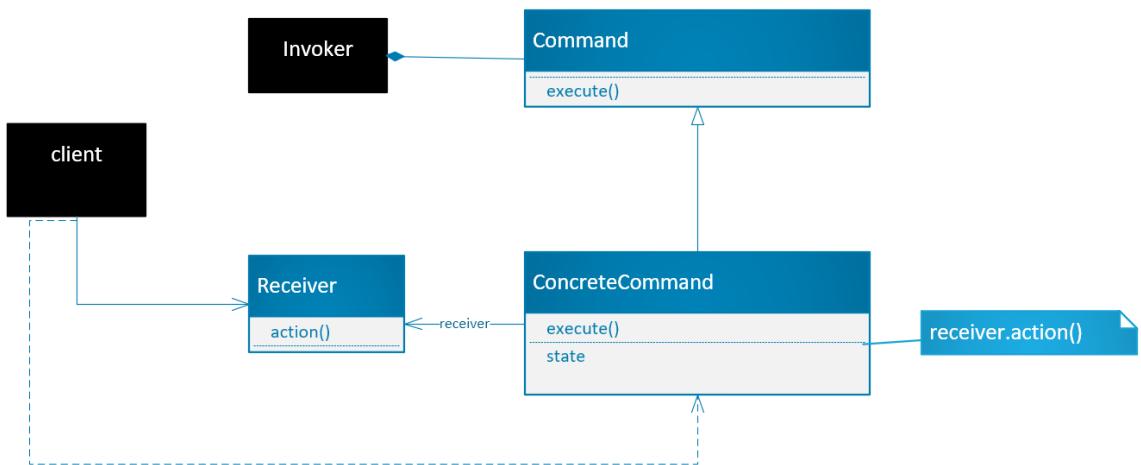
• هنگامی که می خواهید از تغییرات لاغ پشتیبانی کنید تا در صورت خرابی سیستم، مجدداً اعمال شوند. ، می توانید با تقویت رابط Command با عملیات بارگذاری و ذخیره یک گزارش دائمی از تغییرات اعمال شده نگه دارید. فرآیند بازیابی پس از خرابی، شامل بارگذاری مجدد دستورات ثبت شده از دیسک و اجرای مجدد آنها با عملیات Execute است.

• هنگامی که می خواهید یک سیستم را حول عملیات سطح بالا، که بر اساس عملیات اولیه ساخته شده است، ساختار بندی کنید. چنین ساختاری در سیستم های اطلاعاتی که از تراکنش ها پشتیبانی می کنند رایج است. یک تراکنش مجموعه ای از تغییرات در داده ها را دربر می گیرد. الگوی Command راهی برای مدل سازی تراکنش ها ارائه می دهد. Commands دارای یک رابط مشترک هستند که به شما این امکان را می دهد که همه تراکنش ها را به یک شکل فراخوانی کنید. این الگو همچنین گسترش سیستم را با تراکنش های جدید آسان می کند.

^{۵۳} Lifetime

^{۵۴} Undo

ساختار



- class diagram of Command Pattern ۱۰۳

Command

یک رابط برای اجرای یک عملیات تعریف می‌کند

ConcreteCommand

یک انقبiad^{۵۵} بین شیء گیرنده و یک عمل تعریف می‌کند. همچنین متدهای execute() را با فراخوانی عملیات متناظر روی شیء گیرنده، پیاده‌سازی می‌کند.

Client

یک شیء ConcreteCommand می‌سازد و گیرنده آن را مشخص می‌کند

Invoker

از Command می‌خواهد تا درخواست را اجرا کند.

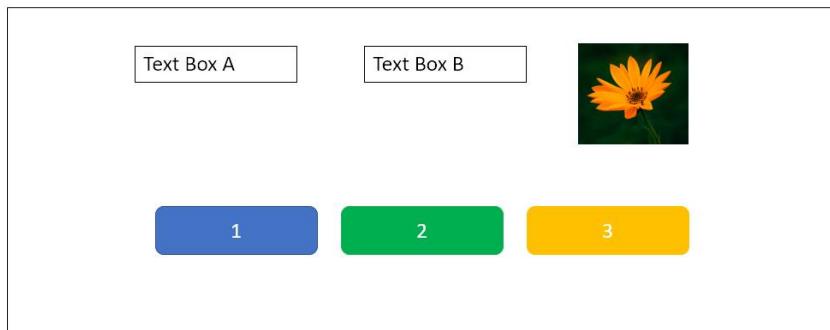
^{۵۵} Binding

Receiver

از چگونگی اجرای عملیات مرتبط با اجرای یک درخواست باخبر است. هر کلاسی می‌تواند به عنوان یک گیرنده عمل کند.

مثال

فرض کنید که می‌خواهیم وبسایتی طراحی کنیم که کاربران بتوانند با استفاده از آن وبسایت خودشان را طراحی کنند. در قسمتی از این وبسایت، کاربر می‌تواند دکمه‌هایی اضافه کرده و عملیاتی که هر دکمه انجام می‌دهد را تعیین کند. همچنین کاربر باید تعیین کند که این عملیات روی کدام عنصر اعمال خواهد شد. فرض می‌کنیم کاربر سه دکمه را ایجاد می‌کند. در صفحه دو جعبه متن و یک تصویر وجود دارد. (شکل ۹۹) عملیات موجود در وبسایت ما، در حال حاضر شامل زوم کردن، حاشیه دار کردن، بلوری کردن و پاک کردن عنصر دریافت کننده آن عمل است. کاربر می‌تواند یک یا چند عملیات را به هر کدام از این دکمه‌ها اضاف کند.



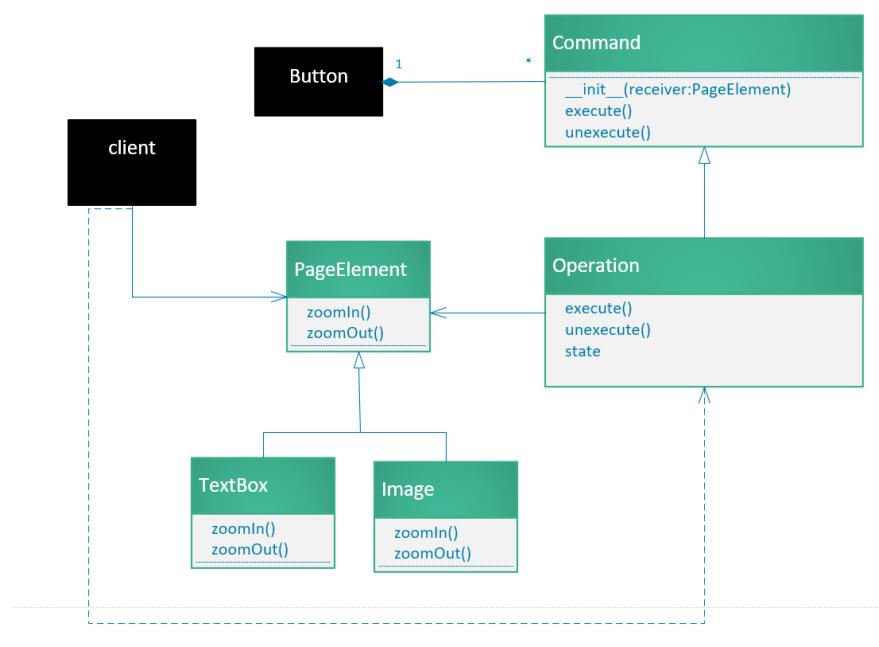
شکل ۱۰۴ - صفحه پیاده‌سازی شده توسط کاربر

ما باید وب سایتمان را طوری طراحی کنیم که کاربر بتواند هر عملی را به هر دکمه‌ای و روی هر عنصری در صفحه اجرا کند. این یعنی که صدازننده‌ی عمل، دکمه‌ها، هیچ اطلاعاتی در مورد عمل ندارند. اما همه‌ی این عملیات ذاتاً عمل هستند. پس باید یک کلاس انتزاعی Command وجود داشته باشد. همچنین این عمل‌ها در مورد عناصری که باید روی آن‌ها اجرا شوند، عناصری که عمل را دریافت می‌کنند، اطلاعاتی ندارند. ما همچنین می‌خواهیم کاربر بتواند از این دکمه‌ها به صورت تاگل، استفاده کند. این به این معناست که اگر دوباره روی آن کلیک کند، معکوس عمل قبلی اجرا شود. به طور مثال اگر دکمه ۱ عمل تار کردن را انجام می‌دهد و کاربر یکبار روی آن کلیک کرده است، با کلیک دوباره روی آن عمل شفاف‌سازی انجام شود.

با توجه به نیازمندی‌های سیستم و توضیحات بالا، الگوی مورد نظر ما الگوی Command است که هر کدام از چهار عمل ذکر شده یک receiver هستند. عناصر صفحه invoker و دکمه‌ها نیز هستند. توجه

کنید اگر بخواهیم بدون استفاده از الگوی Command این برنامه را توسعه دهیم، با توجه به اینکه عناصر صفحه متفاوت هستند، نحوه‌ی پیاده سازی عملیات ذکر شده نیز متفاوت خواهد بود و باید در هر کجا که قصد استفاده از این عملیات را داریم، متد‌های مناسب شئ عنصر را مستقیماً صدا بزنیم. و اگر به طور مثال یک متد را در کلاس عناصر حذف یا اضافه کنیم، باید کدامان در تمامی دکمه‌ها را تغییر دهیم. مزایای استفاده از الگوی Command غیرجفت‌سازی شئ صدازننده از عملیات، شئ بودن عملیات و بهره برداری از مزایای شئ بودن است.

ساختار برنامه توضیح داده شده به شکل زیر است:



شکل ۱۰۵ برنامه class diagram -

برنامه را پیاده‌سازی می‌کنیم:

```

class PageElement(ABC):
    @abstractmethod
    def zoomIn(self):
        pass

    @abstractmethod
    def zoomOut(self):
        pass

class Command(ABC):
    def __init__(self, receiver: PageElement):
        self.receiver = receiver

    @abstractmethod
    def execute(self):
        pass

    @abstractmethod
    def unexecute(self):
        pass

class TextBox(PageElement):
    def zoomIn(self):
        print("font size ++ ")

    def zoomOut(self):
        print("font size -- ")
  
```

```

class Button():
    do = True

    def setAction(self, operation: Command):
        self.operation = operation

    def click(self):
        if(self.do):
            self.operation.execute()
        else:
            self.operation.unexecute()

        self.do = not(self.do)

class ZoomCommand(Command):
    def execute(self):
        self.receiver.zoomIn()

    def unexecute(self):
        self.receiver.zoomOut()

textBox = TextBox()
image = Image()
imageZoom = ZoomCommand(image)
textZoom = ZoomCommand(textBox)
btn1 = Button()
btn2 = Button()
btn1.setAction(imageZoom)
btn2.setAction(textZoom)
btn1.click()
btn2.click()
btn2.click()

```

شکل ۱۰۶ - پیاده سازی و اجرای برنامه

```

Zoomed In
Zoomed Out
font size ++
font size --

```

شکل ۱۰۷ - خروجی کد بالا

الگوی Observer

تعریف

الگوی است که یک وابستگی یک به چند، بین اشیاء ایجاد می‌کند تا زمانی که یک شی تغییر کرد، بقیه شی‌ها نیز از این تغییر آگاه شوند و به طور خودکار بروزرسانی شوند.

هدف

یکی از عوارض جانبی تقسیم‌بندی یک سیستم به کلاس‌های مرتبط به یکدیگر، نیاز به حفظ سازگاری میان اشیاء مرتبط است و این سازگاری نباید باعث کوپلینگ بالا^{۵۶} شود زیرا بازکاربردپذیری^{۵۷} آن را کاهش می‌دهد. به طور مثال در نرم افزار اکسل دیتا هم به صورت نمودار و هم به صورت صفحه گسترده^{۵۸} تمایش داده می‌شود. صفحه گسترده و نمودار یکدیگر را نمی‌شناسند اما رفتارشان خلاف این است زیرا وقتی کاربر تغییراتی در صفحه گسترده انجام می‌دهد، داده تغییر کرده و در نتیجه نمودار نیز تغییر می‌کند. در الگوی Observer به این داده subject و هر مشاهده گری، مانند صفحه گسترده یا نمودار در مثال اکسل، Observer می‌گویند.

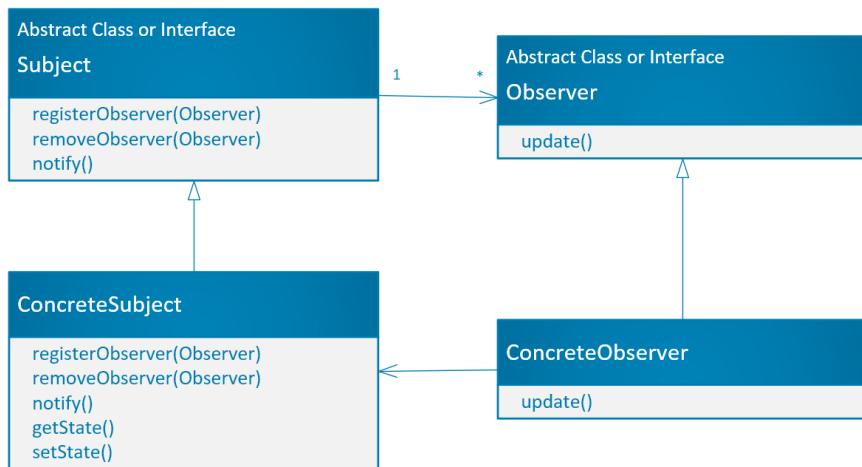
موارد استفاده

- هنگامی که یک انتزاع دو جنبه دارد که هر یک به دیگری وابسته است. کپسوله سازی این جنبه‌ها در شی‌های مختلف به ما این اجازه را می‌دهد که تنوع و بازکاربردپذیری را افزایش دهیم.
- هنگامی که تغییر در یک شی به معنای تغییر در شی‌های دیگر نیز هست و ما از تعداد شی‌هایی که به این واسطه تغییر کنند بی‌خبریم.
- هنگامی که یک شی باید تغییرات اعمال شده در خود را به شی‌های دیگر اطلاع دهد بدون آنکه بداند آن شی‌ها کدام شی‌ها هستند. به عبارت دیگر، این شی‌ها با هم کوپلینگ پایین داشته باشند.

^{۵۶} Tightly coupled

^{۵۷} Reusability

^{۵۸} Spread sheet



شکل ۱۰۸ class diagram in Observer Pattern

کلاس انتزاعی :Subject

یک کلاس انتزاعی است که مشاهده کننده های خود را می شناسد. شیء های Observer می توان هر تعداد باشند. این کلاس یک واسطه برای اضافه و یا حذف کردن شیء های Observer ارائه می کند.

واسطه :Observer

یک واسطه است که واسطی برای بروزرسانی شیء های مورد نظر تعریف می کند.

کلاس عینی :ConcreteSubject

کلاس عینی ای است که داده های مورد نظر در آن ذخیره می شوند و در هنگام تغییر آن ها، به تمامی مشاهده گر ها اعلام می فرستد.

کلاس عینی :ConcreteObserver

کلاسی است که دارای یک ارجاع از شیء ConcreteObject است و اطلاعات گرفته شده از این شیء را ذخیره می کند و با استفاده از متدهای update این اطلاعات را بروزرسانی می کند.

فرض کنید که یک ایستگاه هواشناسی وجود دارد که داده های هواشناسی را با استفاده از سنسور های خود دریافت و ذخیره می کند. تا کنون، چند نرم افزار هواشناسی به این ایستگاه متصل شده اند و از داده های آن استفاده می کنند. هر نرم افزار اطلاعات خاصی را نمایش می دهد. نرم افزار CurrentWeather دما، فشار هوا و رطوبت فعلی، نرم افزار AverageWeather میانگین دما، فشار و رطوبت، و نرم افزار WeatherForecast پیش بینی از وضعیت هوا در سه روز آینده را ارائه می کند. همچنین پیش بینی می شود با توجه به دقت بالای این ایستگاه هواشناسی، نرم افزار WeatherHistory نیز به این ایستگاه متصل شود که تاریخچه دما را در منطقه نشان خواهد داد. هنگامی که داده های ایستگاه هواشناسی تغییر می کنند، این نرم افزار ها باید از این تغییرات مطلع گردند و اطلاعات آن ها بروزرسانی شود. پس در این مثال ایستگاه هواشناسی یک Subject و نرم افزار Obervser هستند.

می خواهیم سیستمی برای این ایستگاه هواشناسی طراحی کنیم. ابتدا باید به ویژگی های کلیدی این ایستگاه دقت کنیم:

۱. چندین نرم افزار به این ایستگاه متصل اند اما ایستگاه از تعداد آن ها بی اطلاع است.
۲. ممکن است در آینده بعضی نرم افزار ها استفاده از داده های این ایستگاه را شروع یا متوقف کنند.
۳. تغییر داده های ایستگاه هواشناسی، باید به تمامی نرم افزار های متصل اطلاع رسانی شود.

طراحی اشتباہ

```
class WeatherStation(ABC):
    temperature:float;
    barometricPressure: float;
    humidity: float;

    def setData():
        #aquires the new data from the sensors
        pass;

    def notify():
        CurrentWeather.update(temperature,barometricPressure,humidity);
        AverageWeather.update(temperature,barometricPressure,humidity);
        WeatherForecast.update(temperature,barometricPressure,humidity);
```

شکل ۱۰.۹ - طراحی اولیه و اشتباہ سیستم

این طراحی و پیاده سازی دو ایراد دارد:

۱. برای حذف و اضافه کردن observer باید کد را تغییر دهیم و به صورت دستی باید هر observer را به کدمان اضافه کنیم.

۲. ارسال اطلاعات از طریق آرگومان های متدهای `update` می تواند مشکل ساز شود. زیرا ممکن است این ایستگاه سنسور های جدیدی اضافه کند و داده های جدیدی تولید کند (به طور مثال میزان UV). در این صورت باید کدام را هم در `notify` و هم در `observer` ها تغییر دهیم که نگهداری کد را مشکل می کند.

پس باید سیستم را دوباره طراحی کنیم. ابتدا یک کلاس انتزاعی یا یک واسط به نام `Subject` می سازیم که سه متدهای `registerObserver()`، `removeObserver()` و `notify()` را تعریف می کند که به ترتیب وظیفه ثبت، حذف و اطلاع رسانی `Observer` را برعهده دارد.

```
class Subject(ABC):
    @abstractmethod
    def registerObserver(self, observer: Observer):
        return

    @abstractmethod
    def removeObserver(self, observer: Observer):
        return

    @abstractmethod
    def notifyObserver(self, observer: Observer):
        return
```

شکل ۱۱۰ - پیاده سازی کلاس انتزاعی `Subject`

سپس یک کلاس عینی به نام `WeatherStation` تعریف می کنیم که از کلاس `Subject` ارث بری می کند و متدهای آن را پیاده سازی می کند. علاوه بر آن، چهار متدهای `getTemperature()`، `getPressure()`، `setData()` و `getHumidity()` نیز وجود دارد که به ترتیب وظیفه دادن اطلاعات به ایستگاه هواشناسی (گرفتن اطلاعات از سنسورها)، برگرداندن دما، فشار و رطوبت را برعهده دارند.

```
class WeatherStation(Subject):
    __temperature: float
    __barometricPressure: float
    __humidity: float

    def __init__(self):
        self.__observers = []

    def registerObserver(self, observer: Observer):
        self.__observers.append(observer)

    def removeObserver(self, observer: Observer):
        self.__observers.remove(observer)

    def notifyObserver(self):
        for obj in self.__observers:
            obj.update()

    def setData(self, temperature: float, barometricPressure: float, humidity: float):
        self.__temperature = temperature
        self.__barometricPressure = barometricPressure
        self.__humidity = humidity
        self.notifyObserver()

    def getTemperature(self):
        return self.__temperature

    def getPressure(self):
        return self.__barometricPressure

    def getHumidity(self):
        return self.__humidity
```

شکل ۱۱۱ - پیاده سازی کلاس `WeatherStation`

حال باید کلاس های انتزاعی DisplayPage و Observer را پیاده سازی کنیم.

```
class Observer(ABC):
    @abstractmethod
    def update():
        return

class displayPage(ABC):
    @abstractmethod
    def display():
        pass
```

شکل ۱۱۲ - پیاده سازی کلاس های DisplayPage و Observer

کلاس DisplayPage یک واسط است که رفتار display را اعلام می کند و کلاس های عینی باید آن را پیاده سازی کنند.

حال برای هر کدام از نرم افزار ها یک کلاس تشکیل می دهیم که از کلاس های DisplayPage و Observer ارث بری می کند.

```
class CurrentWeatherSoftware(Observer, displayPage):
    __temperature: float
    __barometricPressure: float
    __humidity: float
    __weatherStation: WeatherStation

    def __init__(self, weatherStationObj):
        self.__weatherStation = weatherStationObj
        self.__weatherStation.registerObserver(self)

    def update(self):
        self.__temperature = self.__weatherStation.getTemperature()
        self.__barometricPressure = self.__weatherStation.getPressure()
        self.__humidity = self.__weatherStation.getHumidity()

    def display(self):
        print("The current Weather Info: \nTemperature:{0} \nPressure:{1} \nHumidity:{2}".format(
            self.__temperature, self.__barometricPressure, self.__humidity))

class AverageWeatherSoftware(Observer, displayPage):
    __temperature: float
    __barometricPressure: float
    __humidity: float
    __weatherStation: WeatherStation

    def __init__(self, weatherStationObj):
        self.__weatherStation = weatherStationObj
        self.__weatherStation.registerObserver(self)

    def update(self):
        self.__temperature = self.__weatherStation.getTemperature()
        self.__barometricPressure = self.__weatherStation.getPressure()
        self.__humidity = self.__weatherStation.getHumidity()

    def display(self):
        print("displays the average Temperature, Pressure and Humidity")
```

```

class WeatherForecast(Observer, displayPage):
    __temperature: float
    __barometricPressure: float
    __humidity: float
    __weatherStation: WeatherStation

    def __init__(self, weatherStationObj):
        self.__weatherStation = weatherStationObj
        self.__weatherStation.registerObserver(self)

    def update(self):
        self.__temperature = self.__weatherStation.getTemperature()
        self.__barometricPressure = self.__weatherStation.getPressure()
        self.__humidity = self.__weatherStation.getHumidity()

    def display(self):
        print("displays the weather forecast")

```

شکل ۱۱۳ - پیاده سازی کلاس های نرم افزار ها

همانطور که در شکل بالا مشخص است، متدهای update و display آرگومانی را به عنوان ورودی نمی‌گیرد. بلکه کلاس های نرم افزار ها در هنگام نمونه سازی شئ ای از WeatherStation می‌گیرند و هنگام بروزرسانی، داده ها را مستقیماً از همان شئ، با صدا زدن متدهای مربوطه، دریافت می‌کنند.
و در آخر از سیستم طراحی شده خروجی می‌گیریم:

```

RashtWeatherStation = WeatherStation()
CurrentWeatherSoftware = CurrentWeatherSoftware(RashtWeatherStation)
RashtWeatherStation.setData(9, 1021, 70)
CurrentWeatherSoftware.display()
RashtWeatherStation.setData(7, 1018.5, 80)
CurrentWeatherSoftware.display()

```

شکل ۱۱۴ - اجرای کد نهایی

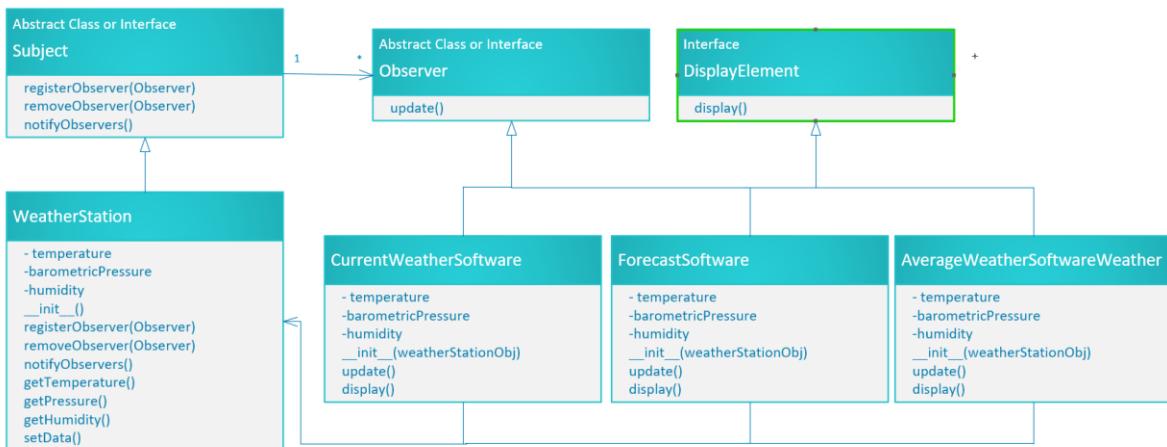
```

The current Weather Info:
Temperature:9
Pressure:1021
Humidity:%70
The current Weather Info:
Temperature:7
Pressure:1018.5
Humidity:%80

```

شکل ۱۱۵ - خروجی کد بالا

ساختار برنامه جدید به شکل زیر خواهد بود:



شکل ۱۱۶ سیستم طراحی شده class diagram – ۱۱۶

الگوی Iterator

تعریف

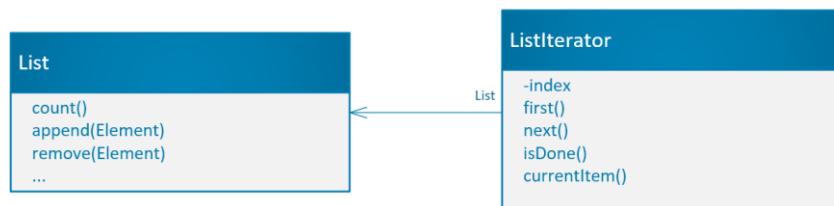
الگوی Iterator راهی برای دسترسی متوالی به عناصر یک شی تجمعی^{۵۹} بدون افشاگری نمایش اساسی آن را نهاده می‌کند.

هدف

یک شی تجمعی مانند یک لیست باید راهی برای دسترسی به عناصر آن بدون افشاگری ساختار داخلی آن در اختیار شما قرار دهد. علاوه بر این، بسته به آنچه می‌خواهید انجام دهید، ممکن است بخواهید از راه‌های مختلف لیست را طی کنید. اما حتی اگر بتوانید موارد مورد نیاز خود را پیش‌بینی کنید، احتمالاً نمی‌خواهید رابط List را با عملیات برای پیمایش‌های مختلف پر کنید. همچنین ممکن است نیاز داشته باشید که بیش از یک پیمایش در انتظار روی یک لیست داشته باشید.

الگوی Iterator به شما امکان می‌دهد همه این کارها را انجام دهید. ایده کلیدی در این الگو این است که مسئولیت دسترسی و پیمایش را از شی لیست خارج می‌شود و در یک شی تکرارکننده – Iterator – قرار داده می‌شود. کلاس Iterator یک رابط برای دسترسی به عناصر لیست تعریف می‌کند. یک شی تکرارشونده مسئول پیگیری عنصر فعلی است. یعنی می‌داند که از کدام عناصر قبلًا عبور کرده است.

به عنوان مثال، یک کلاس List یک ListIterator را، با رابطه زیر بین آنها، صدا می‌زنند:



شکل ۱۱۷ - ارتباط بین List و ListIterator

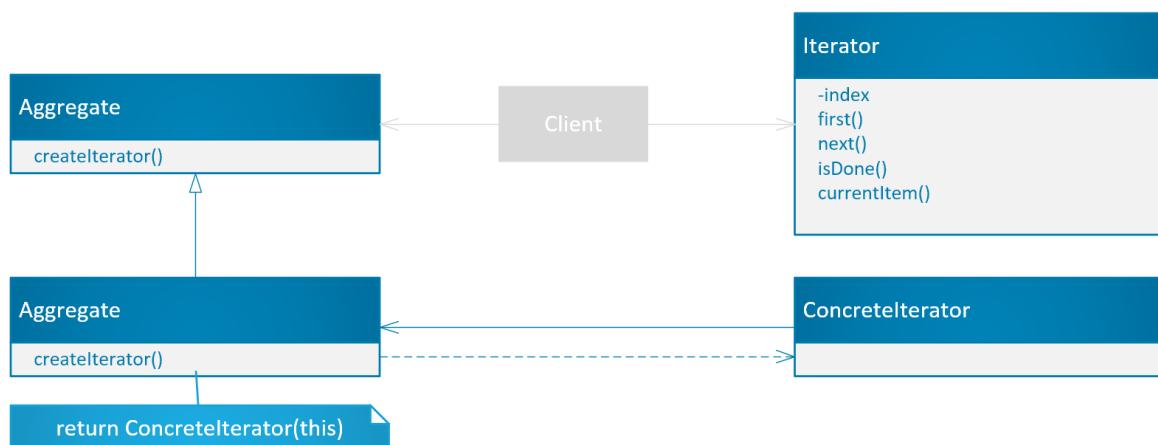
قبل از آنکه شی ای از ListIterator بسازید، باید یک List برای پیمایش بسازید. سپس می‌توانید با یک شی عناصر لیست را به ترتیب پیمایش کنید. متدهای First و Next عنصر فعلی و متدهای CurrentItem و isDone عنصر اول را بر می‌گردانند. متدهای isDone و Next بزرگی می‌کنند که آیا به پایان لیست رسیده ایم یا خیر.

^{۵۹} Aggregate object

موارد استفاده

- هنگامی که می‌خواهید به محتویات یک شی تجمعی، بدون افشا نمایش داخلی آن، دسترسی داشته باشد.
- هنگامی که می‌خواهید برنامه‌نگاری متعدد از اشیاء تجمعی پشتیبانی کند.
- هنگامی که به یک یک رابط یک شکل برای پیمایش ساختارهای تجمعی مختلف (یعنی برای پشتیبانی از تکرار چندریختی^۶) نیاز دارید.

ساختار



- class diagram of Iterator Pattern ۱۱۸ شکل

Iterator

یک واسط برای دسترسی به و پیمایش عناصر ارائه می‌دهد.

ConcreteIterator

واسط **Iterator** را پیاده سازی می‌کند و موقعیت فعلی پیمایش شی تجمعی را نگه می‌دارد.

Aggregate

یک رابط برای ایجاد یک شی **Iterator** فراهم می‌کند

ConcreteAggregate

رابط **Aggregate** را پیاده سازی کرده و شی‌ای از تکرارکننده مورد نظر می‌سازد.

^۶ Polymorphic iteration

مثال

می خواهیم پیمایش پیش فرض یک مجموعه کلمات به شکلی باشد که یکی در میان کلمات برگردانده شوند.

در پایتون کلاس دو Iterable و Iterator وجود دارد که به ترتیب برای پیاده سازی یک تکرار کننده و تکرار شونده استفاده می شوند. کلاس تکرار کننده باید متدهای next و iter را پیاده سازی کند.

برنامه ما به شکل زیر خواهد بود:

```
class SkipTraverse(Iterator):
    _position = 0

    def __init__(self, collections):
        self.collections = collections

    def __next__(self):
        try:
            value = self.collections[self._position]
            self._position += 2
        except IndexError:
            raise StopIteration()

        return value

class WordsCollection(Iterable):
    def __init__(self, collection: List[str]):
        self._collection = collection

    def __iter__(self):
        return SkipTraverse(self._collection)

words = WordsCollection(["this", "doesn't", "make", "any", "sense"])
print("\n".join(words))
```

شکل ۱۱۹ - پیاده سازی و اجرای برنامه

```
this  
make  
sense
```

شکل ۱۲۰ - خروجی کد بالا

الگوی State

تعريف

الگوی State به یک شی اجازه می‌دهد تا بتواند در صورت تغییر حالت درونی خود، رفتار خود را تغییر دهد.

هدف

یک کلاس TCPConnection را در نظر بگیرید که نشان دهنده اتصال شبکه است. یک شی-TCPConnection می‌تواند در یکی از چندین حالت مختلف باشد: برقرار^۱، در حال گوش دادن و بسته. هنگامی که یک شی TCPConnection درخواست‌هایی را از اشیاء دیگر دریافت می‌کند، بسته به وضعیت فعلی خود پاسخ‌های متفاوتی می‌دهد. به عنوان مثال، تأثیر درخواست باز^۲، بستگی به این دارد که آیا اتصال در حالت بسته یا برقرار است. الگوی State توضیح می‌دهد که TCPConnection چگونه می‌تواند رفتارهای متفاوتی را در هر حالت از خود نشان دهد.

ایده اصلی در این الگو، معروفی یک کلاس انتزاعی به نام TCPState برای نمایش حالت‌های اتصال شبکه است. کلاس TCPState یک رابط مشترک برای همه کلاس‌هایی که حالت‌های عملیاتی مختلف را نشان می‌دهند، اعلام می‌کند. زیر کلاس‌های TCPState رفتار مخصوص یک حالت را اجرا می‌کنند. به عنوان مثال، کلاس‌های TCPEstablished و TCPClosed در حالت‌های Established و Closed پیاده‌سازی می‌کنند. TCPConnection

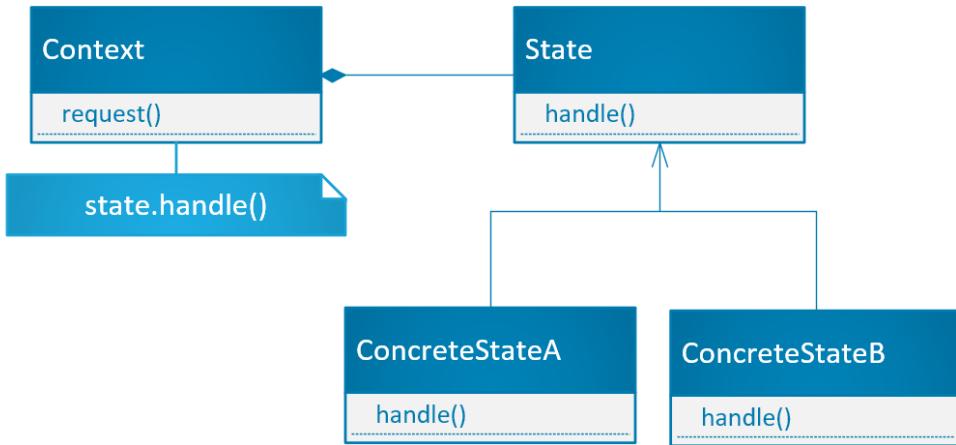
موارد استفاده

- هنگامی که رفتار یک شی به حالت آن بستگی دارد و باید رفتار خود را در زمان اجرا بسته به آن حالت تغییر دهد.
- هنگامی که عملیات دارای عبارات شرطی بزرگ و چند بخشی است که به وضعیت شی بستگی دارد. این حالت معمولاً با یک یا چند ثابت شمارشی^۳ نشان داده می‌شود. اغلب، چندین عملیات حاوی همین ساختار شرطی هستند. الگوی State هر شاخه از شرط را در یک کلاس جداگانه قرار می‌دهد. این کار به شما این امکان را می‌دهد که وضعیت شی را به عنوان یک شی تنها در نظر بگیرید که می‌تواند مستقل از اشیاء دیگر متفاوت باشد.

^۱ Established

^۲ Open request

^۳ Enumerate constant



شکل ۱۲۱ class diagram of State Pattern - ۱۲۱

Context

رابط مورد علاقه مشتریان را تعریف می کند. همچنین نمونه ای از یک زیر کلاس ConcreteState را نگه می دارد که وضعیت فعلی را تعریف می کند.

State

یک رابط برای کپسوله کردن رفتار مرتبط با یک حالت خاص از Context تعریف می کند.

ConcreteState

هر زیر کلاس یک رفتار مرتبط با حالتی از Context را پیاده سازی می کند.

مثال ۱

فرض کنید که در تیم توسعه واتس اپ هستیم و می خواهیم با توجه به وضعیت اتصال فرد عکسها و ویدیو ها را بارگذاری کنیم. سه حالت wifi, offline و data داریم. ممکن است در آینده حالات lowWifi و lowData را نیز اضافه کنیم. بارگذاری فیلم و بارگذاری عکس متد هایی هستند که باید پیاده سازی کنیم. در هر یک از حالات ذکر شده، پیاده سازی این متد ها با یکدیگر متفاوت است.

پیاده سازی و اجرای این برنامه به شکل زیر است:

```

class ConnectionState(ABC):
    @abstractmethod
    def loadImages(self):
        pass

    @abstractmethod
    def loadVideos(self):
        pass

class Offline(ConnectionState):
    def loadImages(self):
        print("don't even try")

    def loadVideos(self):
        print("don't even try")

class Wifi(ConnectionState):
    def loadImages(self):
        print("download all images ")

    def loadVideos(self):
        print("download all videos ")

class Data(ConnectionState):
    def loadImages(self):
        print("download half of images ")

    def loadVideos(self):
        print("download no videos ")

class Connection:
    _connectionState = Offline()

    def setConnectionState(self, connectionState: ConnectionState):
        self._connectionState = connectionState

    def load(self):
        self._connectionState.loadImages()
        self._connectionState.loadVideos()

connection = Connection()
connection.load()
data = Data()

connection.setConnectionState(data)
connection.load()

```

شکل ۱۲۲ - پیاده‌سازی و اجرای برنامه

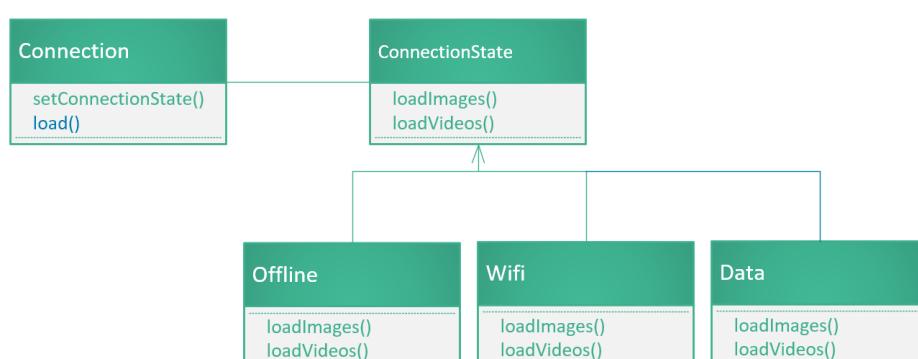
```

don't even try
don't even try
download half of images
download no videos

```

شکل ۱۲۳ - خروجی کد پلا

ساختار برنامه به شکل زیر است:

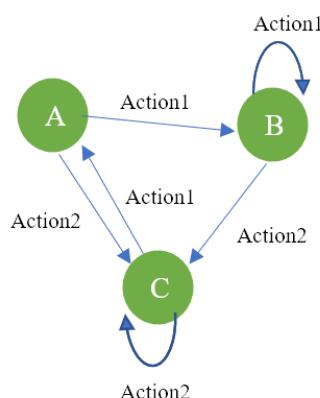


شکل ۱۲۴ - class diagram برنامه

مثال ۲

این بار می‌خواهیم مثالی انتزاعی اما کمی سخت‌تر را پیاده سازی کنیم. فرض کنید ماشین حالتی ^{۶۴} به شکل

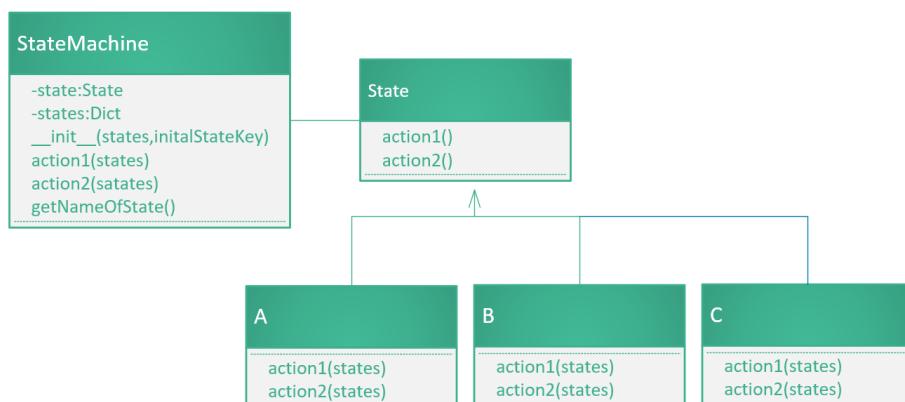
زیر داریم:



شکل ۱۲۵ - Sample State Machine

برای هر حالت دو عمل وجود دارد. پس در کلاس انتزاعی State این دو عمل را تعریف می‌کنیم که کلاس های A,B و C آن‌ها را پیاده‌سازی می‌کنند. اجرای هر یک از این اعمال حالت دستگاه را تغییر می‌دهد. پس هر بار باید شیء حالتی برگردانده شود. برای جلوگیری از ایجاد نمونه‌های اضافی و تکراری حالت‌ها، باید ابتدا تمامی اشیاء حالت‌ها را در جایی ذخیره کنیم. در کلاس StateMachine تابع سازنده را به نحوی طراحی می‌کنیم که ابتدا یک دیکشنری از اشیاء حالت‌ها و سپس کلید حالت اولیه را دریافت می‌کند. و در هر باری که متده از آن حالت صدای زده می‌شود، به طور مثال ^۱، دیکشنری حاوی اشیاء نیز به آن متده فرستاده می‌شود. آن متده اجرا شده و شیء حالتی را از همان دیکشنری بر می‌گرداند که در متده state در کلاس StateMachine ذخیره می‌شود.

برای ذخیره شدن StateMachine



شکل ۱۲۶ - برنامه طراحی شده class diagram

^{۶۴} State machine

برنامه را به شکل زیر پیاده‌سازی می‌کنیم:

```
class State(ABC):
    @abstractmethod
    def action1(self, states):
        pass

    @abstractmethod
    def action2(self, states):
        pass

class A(State):
    def action1(self, states):
        return states["B"]

    def action2(self, states):
        return states["C"]

class B(State):
    def action1(self, states):
        return states["B"]

    def action2(self, states):
        return states["C"]

class C(State):
    def action1(self, states):
        return states["A"]

    def action2(self, states):
        return states["C"]

class StateMachine():
    _states: Dict[State]
    _state: State = None

    def __init__(self, states: Dict[State], initialStateKey: str):
        self._states = states
        self._state = states[initialStateKey]

    def action1(self):
        print(self.getNameofState()+"---->A")
        self._state = self._state.action1(self._states)
        print(self.getNameofState())

    def action2(self):
        print(self.getNameofState()+"---->B")
        self._state = self._state.action2(self._states)
        print(self.getNameofState())

    def getNameofState(self):
        name = str(type(self._state))
        return [name[name.find(".")+1:name.find(".")+2]]
```

شکل ۱۲۷ - پیاده‌سازی برنامه

```
myStateMachine = StateMachine({"A": A(), "B": B(), "C": C()}, "A")

myStateMachine.action1()
myStateMachine.action1()
myStateMachine.action2()
myStateMachine.action2()
myStateMachine.action1()
myStateMachine.action2()
```

شکل ۱۲۸ - تست و اجرای برنامه

```
A---->B
B---->B
B---->C
C---->C
C---->A
A---->C
```

شکل ۱۲۹ - خروجی کد بالا

همانطور که مشاهده می شود، انجام اعمال 1^{st} و 2^{nd} action در هر حالت، ما را به حالت مورد انتظارمان می برد. پس برنامه به درستی کار می کند.

مزیت الگوی State نسبت به عبارات شرطی نگهداری آسان تر و قابلیت گسترش آن است. فرض کنید اگر می خواستیم مثال بالا را تنها با عبارات شرطی پیاده سازی کنیم، نیاز به ۶ عبارت شرطی داشتیم که با افزایش حالت ها بیشتر هم می شدند و کار نگهداری و گسترش را بسیار سخت می کردند.

الگوی Strategy

تعریف

الگوی Strategy خانواده‌ای از الگوریتم‌ها را تعریف کرده، هر کدام را کپسوله‌سازی می‌کند و این امکان را فراهم می‌آورد که بتوانند به جای یکدیگر استفاده شوند. این الگو به الگوریتم‌ها اجازه می‌دهد تا مستقل از مشتری‌ها متفاوت باشند.

هدف

الگوریتم‌های بسیار زیادی برای تبدیل جریانی از متن به خطوط مختلف وجود دارد. پیاده‌سازی آن‌ها در کلاس‌هایی که به آن‌ها نیاز دارند به دلایل زیر گزینه مناسبی نیست:

- مشتری‌هایی که نیاز به شکستن خط^{۶۵} دارند، باید پیچیدگی بیشتری را برای توسعه کد شکستن خط متحمل شوند. این کار نگهداری مشتری‌ها را، به ویژه هنگامی که از چندین الگوریتم شکستن خط پشتیبانی کنند، سخت‌تر می‌کند.
- الگوریتم مناسب در زمان‌های مختلف، مختلف است. همچنین نمی‌خواهیم از الگوریتم‌های شکستن خطی که به آن‌ها نیازی نداریم، پشتیبانی کنیم.
- افزودن الگوریتم‌های جدید و تغییر الگوریتم‌های موجود در زمانی که شکستن خط درون مشتری پیاده‌سازی شده است، کار بسیار دشواری است.
- ما می‌توانیم از این مشکلات را با تعریف کلاس‌هایی که الگوریتم‌های مختلف شکستن خط را کپسوله‌سازی می‌کنند، اجتناب کنیم. الگوریتمی که به این صورت کپسوله‌سازی شده است، یک استراتژی – strategy – نام دارد.

موارد استفاده

- هنگامی که بسیاری از کلاس‌های مرتبط فقط در رفتارشان با هم تفاوت دارند. استراتژی‌ها راهی برای پیکربندی یک کلاس با یکی از رفتارهای متعدد ارائه می‌دهند.

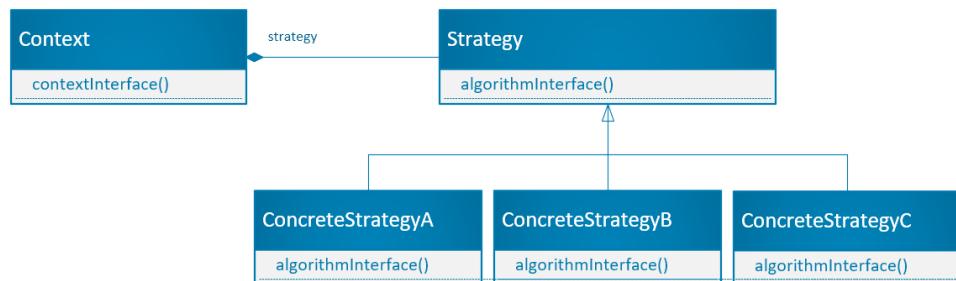
^{۶۵} Line breaking

- هنگامی که به انواع مختلفی از یک الگوریتم نیاز دارید. به عنوان مثال، شما ممکن است الگوریتم هایی را تعریف کنید که منعکس کننده مبادلات مکان/زمان مختلف هستند. زمانی می توان از استراتژی ها استفاده کرد که این گونه ها به عنوان سلسله مراتب کلاسی از الگوریتم ها پیاده سازی شوند.

- هنگامی که یک الگوریتم از داده هایی استفاده می کند که مشتریان نباید درباره آنها بدانند. از الگوی Strategy برای جلوگیری از افشای ساختارهای داده پیچیده و خاص الگوریتم استفاده کنید.

- هنگامی که کلاس رفتارهای زیادی را تعریف می کند، و اینها به صورت عبارات شرطی متعدد در عملیات آن ظاهر می شوند. به جای بسیاری از شرطی ها، شاخه های شرطی مرتبط را به کلاس خود منتقل کنید.

ساختار



class diagram of Strategy pattern - ۱۳۰

Strategy

یک واسط مشترک برای تمامی الگوریتم های پشتیبانی شده اعلام می کند. Context از این واسط برای صدا زدن الگوریتمی که توسط یک ConcreteStrategy تعریف شده است، استفاده می کند.

ConcreteStrategy

با استفاده از واسط Strategy الگوریتم را پیاده سازی می کند.

Context

با یک شی ConcreteStrategy پیکربندی شده است. همچنین دارای یک ارجاع به شیء Strategy است. این کلاس ممکن است رابطی را برای Strategy به منظور دسترسی به دیتا های خود فراهم کند.

مثال

می خواهیم برنامه ای برای یک اکوالایزر بنویسیم. این اکوالایزر دو سبک کلاسیک و راک را پشتیبانی می کند. کاربر در این برنامه می تواند سبک موسیقی در حال پخش را انتخاب کرده، و دکمه ای پخش را بزند. همچنین می تواند در هر لحظه سبک موسیقی را عوض کند. پیاده سازی این برنامه با استفاده از متدهای سبک های موسیقی ممکن است اما برای اضافه کردن یک سبک جدید یا ویرایش تنظیمات یک سبک مجبور به تغییر کد در کلاس برنامه هستیم. با توجه به توضیحات داده شده، باید الگوی Strategy را پیاده سازی کنیم پس برنامه ما به شکل زیر خواهد بود:

```
class EqualizerApp():
    equalizerGenre: EqualizerGenre = None

    def __init__(self, defaultGenre: EqualizerGenre):
        self.equalizerGenre = defaultGenre

    def play(self):
        self.equalizerGenre.adjust()
        name = str(type(self.equalizerGenre))
        name = name[name.find(".") + 1: name.rfind(".")]
        print(f'the music is playing. Equalizer On:{name}\n')

    def changeGenre(self, genre: EqualizerGenre):
        self.equalizerGenre = genre
        self.play()

class EqualizerGenre(ABC):
    @abstractmethod
    def adjust(self):
        pass

class Classical(EqualizerGenre):
    def adjust(self):
        print("Tuned for classical genre")

class Rock(EqualizerGenre):
    def adjust(self):
        print("Tuned for Rock genre")

class Pop(EqualizerGenre):
    def adjust(self):
        print("Tuned for Pop genre")
```

شکل ۱۳۱ - پیاده سازی برنامه

```
rock = Rock()
classical = Classical()
pop = Pop()
app = EqualizerApp(pop)
app.play()
app.changeGenre(rock)
```

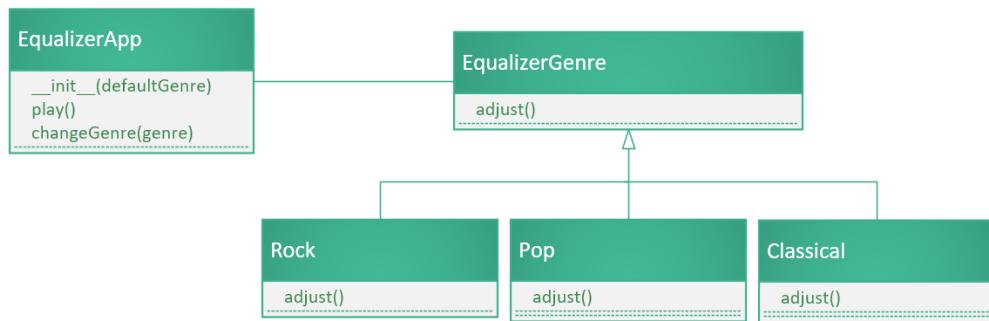
شکل ۱۳۲ - نتست و اجرای برنامه

```
Tuned for Pop genre
the music is playing. Equalizer On:Pop

Tuned for Rock genre
the music is playing. Equalizer On:Rock
```

شکل ۱۳۳ - خروجی کد بالا

ساختار برنامه به شکل زیر است:



شکل ۱۳۴ class diagram - براما

الگوی Template Method

تعريف

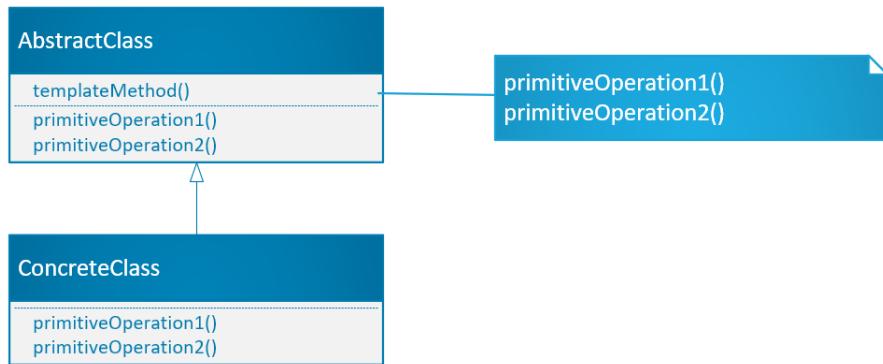
الگوی Template Method اسکلتی از یک الگوریتم در یک عمل را تعریف می‌کند که بخش‌های مختلف آن را به زیرکلاس‌های خود متحول می‌کند. این الگو به زیرکلاس‌ها اجازه می‌دهد تا بخش‌های مشخصی از یک الگوریتم را، بدون تغییر دادن ساختار کلی یک الگوریتم، تغییر دهند.

هدف

الگوی Template Method برای تعریف ساختاری استفاده می‌شود که ما کاملاً مطمئنیم که هرگز آن را تغییر نخواهیم داد. استفاده معمول این الگو در فریمورک‌ها است. جایی که ساختار یک فریمورک ثابت بوده و تغییر نخواهد کرد. اگر بخواهیم برای این الگو مثالی از دنیای واقعی بزنیم، می‌توانیم آن را به فرم بازکردن حساب تشبيه کنیم. فرم دارای متون پیش‌فرض و ثابت خود و مکان‌هایی برای وارد کردن اطلاعات کاربر است. هنگامی که کاربر اطلاعات خود را وارد کرد، آن فرم را، که شامل متن‌های چاپ شده و اطلاعات خود است، به متصلی می‌دهد. متصلی نیز اطلاعات و امضای خود را وارد فرم می‌کند. بدین ترتیب فرآیند بازکردن حساب به پایان می‌رسد.

موارد استفاده

- هنگامی که می‌خواهیم بخش‌های تغییرناپذیر یک الگوریتم را در یک زمان پیاده‌سازی کنیم و پیاده‌سازی بخش‌های متغیر آن را بر عهده زیرکلاس‌ها بگذاریم.
- هنگامی که رفتارهای مشترک میان زیرکلاس‌ها باید در یک کلاس مشترک محلی شود تا از کد تکراری جلوگیری شود. ابتدا تفاوت‌های که موجود را شناسایی می‌کنید و سپس تفاوت‌ها را به شکل عملیات‌های جدید جدا می‌کنید. در نهایت، کد متفاوت را با یک متد قالب - template - جایگزین می‌کنید که یکی از این عملیات جدید را فراخوانی می‌کند.
- هنگامی که باید گسترش زیرکلاس‌ها را کنترل کنید. می‌توانید یک متد قالب تعریف کنید که در زمان‌های مشخص عملیات قلاب را صدا می‌زند. بنابراین گسترش زیرکلاس‌ها تنها در این زمان‌ها ممکن است.



شکل ۱۳۵ - class diagram of Template Method pattern

AbstractClass

عملیات نخستین انتزاعی را تعریف می‌کند که زیرکلاس‌های عینی آن را برای پیاده‌سازی مراحل یک الگوریتم تعریف می‌کنند. همچنین متدهای انتزاعی را تعریف می‌کند. متدهای انتزاعی مراحل و عملیات تعریف شده در AbstractClass و عملیاتی که در اشیاء دیگر است را صدا می‌زنند.

ConcreteClass

عملیات مقدماتی را برای اجرای مراحل مختلف الگوریتم پیاده‌سازی می‌کند.

مثال

فرض کنید در حال توسعه فریمورکی برای ایجاد فرم نظر سنجی هستیم. هر نظرسنجی حداقل از یک فیلد متن و یک دکمه ارسال تشکیل شده است. مشتری می‌تواند فیلدهای دیگری نیز به این فرم اضافه کند. پس فیلد متن و دکمه ارسال همیشه ثابت و فیلدهای مشتری متغیر است. با توجه به شرایط، انتخاب ما الگوی Template Method است. پس ابتدا یک کلاس انتزاعی SurveyForm می‌سازیم که سه متدهای create، reviewBox و submitButton در آن پیاده‌سازی و متدهای anotherTextBox و reviewBox در آن تعریف شده است:

```

class SurveyForm(ABC):
    def create(self):
        self.anotherTextBox()
        self.reviewBox()
        self.submitButton()

    def reviewBox(self):
        print("a review box □")

    def submitButton(self):
        print("a submit button ⬤")

    @abstractmethod
    def anotherTextBox(self):
        pass
  
```

شکل ۱۳۶ - پیاده‌سازی کلاس SurveyForm

سپس کلاس mySurvey را می‌سازیم که یک فیلد نام نیز دارد.

```
class MySurvey(SurveyForm):
    def anotherTextBox(self):
        print("name box AB")
```

شکل ۱۳۷ - پیاده‌سازی کلاس MySurvey

برنامه را اجرا و تست می‌کنیم:

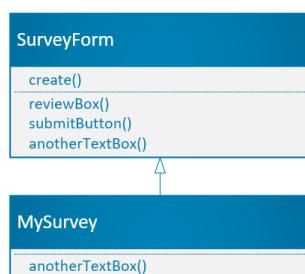
```
mySurvey = MySurvey()
mySurvey.create()
```

شکل ۱۳۸ - تست و اجرای برنامه

name box AB
a review box□
a submit button○

شکل ۱۳۹ - خروجی کد بالا

ساختار برنامه به شکل زیر است:



شکل ۱۴۰ - class diagram برنامه

فصل پنجم : منابع

- ١.Gamma, Eric: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional; 1st edition (November 1, 1994)
- ٢.Freeman, Eric et al: Headfirst Design Patterns. O'Reilly Media; 1st edition (October 1, 2004)
٣. Alexander Shvets, Design Patterns, <https://refactoring.guru/design-patterns>
- ٤.Alexander Shvets, Design Patterns in Python
<https://refactoring.guru/design-patterns/python>
٥. Christopher Okhravi, Design Patterns Series,
<https://www.youtube.com/c/ChristopherOkhravi>