设 计 模 式

可复用的面向对象软件的基础

Pionpill [1]

本文为设计模式[2] 一书的简单笔记

2022 年 12 月 13 日

[1]笔名：北岸，电子邮件：673486387@qq.com，Github：https://github.com/Pionpill

[2]英文原名: Desgin Patterns Elements of Reusable Object-Oriented Software

前言：

在阅读本文之前，默认已经掌握基本的面向对象思想，对 Java/Python... 语法有基础的了解 (至少看得懂)，且有一定的项目经验。

由于原书出版在上世纪，有些当时新颖的理念在现在已经成为一些主流的设计方案，因此本人不会对一些基础的内容进行过多说明，如有需要还请查看原文。

原文使用 C++/Smalltalk 语言，即使 C++ 仍具有很高的热度，但已不及 Python/Java... 这些新的高级语言，此外 C++ 的语法毋庸置疑比较复杂，我们应更加关注设计模式而不是语言实现本身，因此本人使用的例子更多参考了 CSDN 等技术社区的其它例子。

一般地，本文的 Java 例子使用连接，Python 则直接贴出代码，主要是因为 Python 代码简短，可读性强。这些 Python 例子源自于 python-patterns 项目，有稍许修改。受限于长度，这些代码往往并不完全符合模式结构，就例子本身实现的作用来看，有些设计模式甚至显得多余冗长，毕竟在一个几百行只有一个文件的脚本中使用设计模式本身就显得多余。因此希望读者仅将示例用作理解设计模式，而不是按部就班。

本文只介绍性地说明了各个设计模式，并没有深入挖掘设计模式，原因如下:

- 本人能力与实战经验有限。
- 由于各语言的特性不同，无法给出统一的回答。比如，如果要在 Java/Python 中实现 Singleton 模式，在 Java 中考虑到线程安全问题，往往要写出比较复杂的代码，如果再涉及到多个 JVM 虚拟机，问题变得更加复杂，而且需要一些 Java 高级特性的知识。而在 Python 中只需要结合装饰器用简单的代码就可以完成。
- 详细解读一个设计模式，往往需要项目作为例子，这并不是一两个文件，几百行代码就能解释清楚的。

此外，本文的结构和原文大不相同，请将本文当作字典查阅相关知识而不是和原书一般循序渐进地阅读。

本人的书写环境:

- Window10

参考文献:

- python-patterns: `https://github.com/faif/python-patterns`

2022 年 12 月 13 日

目录

设计模式简介

警告 0.1. 本章都是笔者自己的总结。笔者能力有限，请读者自行斟酌，欢迎任何建议。此外，本章也是对后面章节的一个概述，如果读者有设计模式基础，可以跳过。

设计模式的目的

设计模式，即在特定场景下解决一般设计问题的类和相互通信的对象的描述。通俗的说，掌握一门语言仅是在语法与特性上学会了一种工具，而设计模式则是使用工具的一类方法，帮助我们更加方便，简洁地完成我们的项目。在程序设计领域，则是增加代码的可复用度，写出高内聚低耦合的代码。

接口是面向对象编程中的一个重要概念，接口的存在让我们将注意力转移到操作而不是实例本身。熟练使用接口，熟悉动态绑定与多态有助于我们更好地理解设计模式。在学习设计模式的过程中，我们应该更加关注对接口编程，而不是针对类编程。此外，设计模式的对象是类与实例，这其中还包含抽象类，混入类等，读者需要对这些概念有一定的了解。

引用 CSDN 博主「割韭韭」的一句话[1]: 在现实情况下（至少是我所处的环境当中），很多人往往沉迷于设计模式，他们使用一种设计模式时，从来不去认真考虑所使用的模式是否适合这种场景，而往往只是想展示一下自己对面向对象设计的驾驭能力。编程时有这种心理，往往会发生滥用设计模式的情况。所以，在学习设计模式时，一定要理解模式的适用性。必须做到使用一种模式是因为了解它的优点，不使用一种模式是因为了解它的弊端；而不是使用一种模式是因为不了解它的弊端，不使用一种模式是因为不了解它的优点。

		目的		
		创建型	结构型	行为型
范围	类	Factory Method	Adapter	Interpreter Template Method
	对象	Abstract Factory Builder Prototye Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

[1] 原文链接:https://blog.csdn.net/zhengzhb/article/details/7489639

设计模式的分类

按作用范围，模式可分为对类/实例起作用。按照模式目的，则可分为创建型，结构型，行为型三种。顾名思义，创建型与对象的创建有关；结构性处理类或对象的组合；行为型对类或对象怎样交互分配职责进行描述。

目的	设计模式	可变的方面
创建	Abstract Factory	产品对象家族
	Builder	如何创建一个组合对象
	Factory Method	被实例化的子类
	Prototype	被实例化的类
	Singleton	一个类的唯一实例
结构	Adapter	对象的接口
	Bridge	对象的实现
	Composite	一个对象的结构和组成
	Decorator	对象的职责，不生成子类
	Facade	一个子系统的接口
	Flyweight	对象的存储开销
	Proxy	如何访问一个对象，该对象的位置
行为	Chain of Responsibility	满足一个请求的对象
	Command	合适，怎样满足一个请求
	Interpreter	一个语言的文法及解释
	Iterator	如何遍历，访问一个聚合的各元素
	Mediator	对象间怎样交互，和谁交互
	Memento	一个对象中哪些私有信息存放在该对象之外，以及在什么时候进行存储
	Observer	多个对象依赖于另外一个对象，而这些对象又如何保持一致
	State	对象的状态
	Strategy	算法
	Template Method	算法中的某些步骤
	Visitor	某些可作用于一个 (组) 对象上的操作，但不修改这些对象的类

1 创建型模式

1.1 Abstract Factory (抽象工厂)

意图

提供一个接口以创建一系列相关或互相依赖的对象，而无须指定它们具体的类。

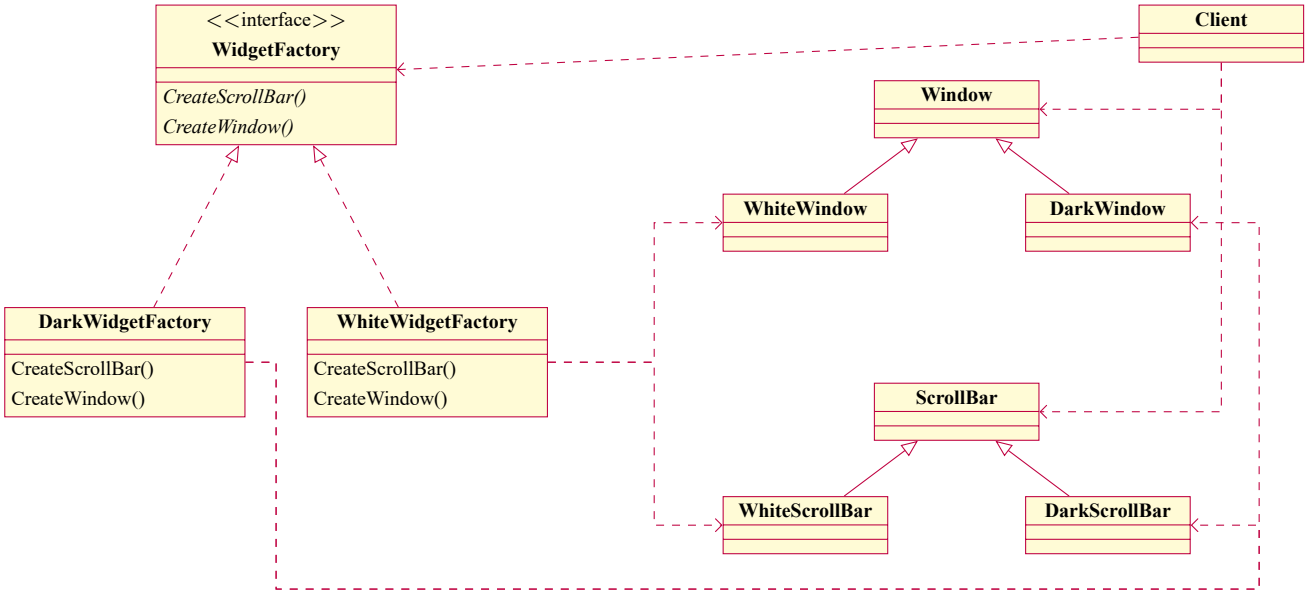
抽象工厂模式将对象的**创建**与**获取**隔离开，使对象的**获取不依赖于它的具体类**，而是依赖于一个封装具体对象创建的工厂类，使用者不需要关心这些工厂类创建具体对象的细节。

别名: Kit

动机

假设我们在编写 GUI，此时由于客户需要，我们将编写多套组件 (滚动条，按钮等)，相同的组件套中美术风格一致。那么很显然，我们需要在已有 GUI 的基础上集成修改，假设有 White,Dark 两种风格，则每个组件类需要派生出两个对应的类。如果没有设计模式，我们会直接调用构造函数来实例化这些小组件，这是可行的，但如果我们以后需要修改某个类，如果对类名进行了修改，或者要弃用之前的 `WhiteAWidget` 改用 `WhiteBWidget`, 我们往往需要修改所有的构造函数，这使得维护变得一团糟且很容易犯错，并且我们的两套组件之间并没有明显的区别，仅能通过类名中的 White/Dark 这些单词区别。

为了解决这个问题，我们可以定义抽象 `WidgetFactory` 类，这个类声明了一个用来创建每一类基本组建的接口。这些接口将负责获取组件实例，而客户不需要知道正在使用的是哪些具体类。结构如下所示：



经此改动，客户仅与抽象类定义的接口交互，而不需要特定的具体类的接口。我们后期的维护也仅需要修改抽象类接口中的内容。

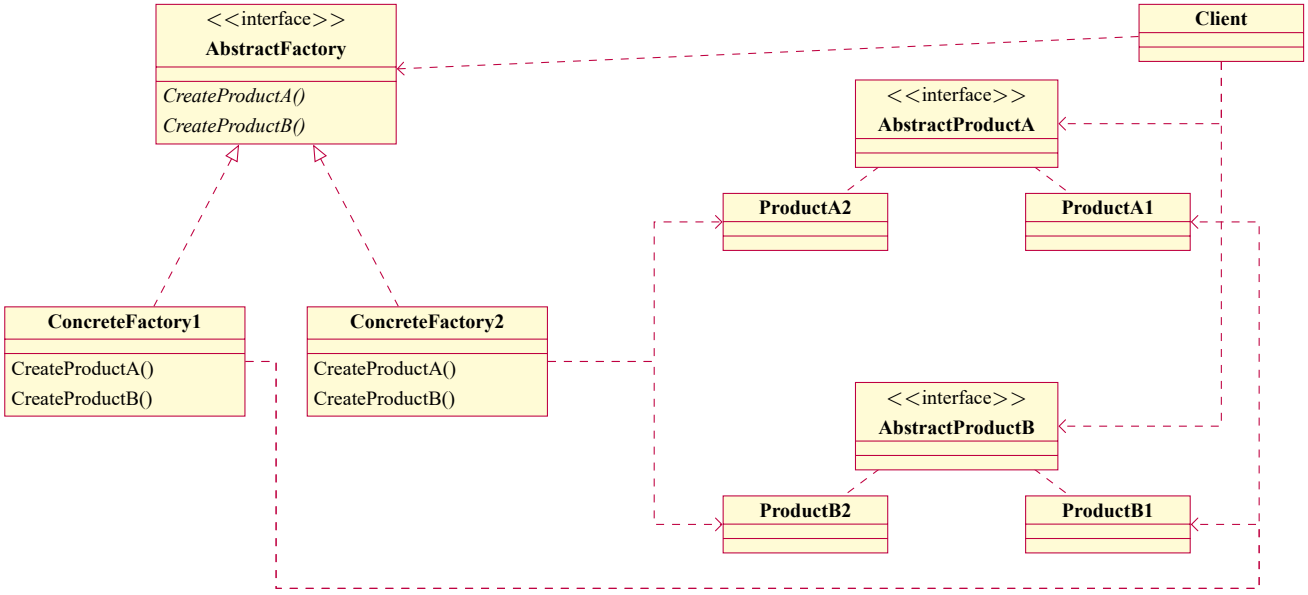
适用性

以下情况适用 AbstractFactory 模式:

- 系统独立于它的产品的创建，组合和表示。
- 系统要由多个产品嗅裂中的一个来配置。
- 创建一系列相关的产品对象的设计以便进行联合使用。
- 提供一个产品类库，但只想显示它们的接口而不是实现。

结构

此模式的结构如下所示:



参与者

- AbstractFactory: 声明一个创建抽象产品对象的操作接口。
- ConcreteFactory: 实现创建具体产品对象的操作。
- AbstractProduct: 为一类产品对象声明一个接口。
- ConcreteProduct: 定义一个将被相应的具体工厂创建的产品对象。实现 AbstractProduct 接口。
- Client: 仅使用由 AbstractFactory 和 AbstractProduct 类声明的接口。

协作

- 通常在运行时创建一个 ConcreteFactory 类的实例。为创建不同的产品对象，应使用不同的具体工厂。
- AbstractFactory 将产品对象的创建延迟到它的 ConcreteFactory 子类。

优缺点

- **分离了具体的类**
- **易于交换产品系列**
- **保持了产品的一致性**
- **难以支持新种类的产品**: AbstractFactory 接口确定了可以被创建的产品集合，如果需要增加新的产品，组需要修改 AbstractFactory 类及其子类。在实现中给出了解决方案。

实现

下面是实现 AbstractFactory 模式的一些有用的技术:

- **将工厂作为单件**: 一个应用中一般每个产品系列只需要一个 ConcreteFactory 实例，因此工厂最好实现为一个 Singleton。
- **创建产品**: AbstractFactory 仅声明一个创建产品的接口,真正创建产品是由 ConcreteProduct 子类实现的。最通常的办法是为每一个产品定义一个工厂方法 (Factory Method)。如果有多个可能的产品系列，也可以使用 Prototype 模式来实现。
- **定义可扩展的工厂**: AbstractFactory 通常为每一个种它可以生产的产品定义一个操作。增加一种新的产品要求改变它的接口和它相关的类。一个更灵活但不安全的方案是为创建对象的操作添加一个参数。参数制定了将被创建的对象的种类。可以是字符串，整数，标识符.....

相关模式

- Factory Method: AbstractFactory 通常用 Factory Method 实现。
- Prototype: AbstractFactory 也可以用 Prototype 实现。
- Singleton: 一个具体的工厂通常是一个单件。

例子

- Java 实现 GUI: `https://zhuanlan.zhihu.com/p/499672744`

```python
# Reference:
    https://github.com/faif/python-patterns/blob/master/patterns/creational/abstract_factory.py

import random
from typing import Type


# AbsrtactProduct
class Pet:
    def __init__(self, name: str) -> None:
        self.name = name

    def speak(self) -> None:
        raise NotImplementedError

    def __str__(self) -> str:
        raise NotImplementedError


# ConcreteProduct
class Dog(Pet):
    def speak(self) -> None:
        print("woof")

    def __str__(self) -> str:
        return f"Dog<{self.name}>"

```

```python

# ConcreteProduct
class Cat(Pet):
    def speak(self) -> None:
        print("meow")

    def __str__(self) -> str:
        return f"Cat<{self.name}>"


# ConcreteFactory
class PetShop:
    """A pet shop"""
    def __init__(self, animal_factory: Type[Pet]) -> None:
        """pet_factory is our abstract factory. We can set it at will."""
        self.pet_factory = animal_factory

    def buy_pet(self, name: str) -> Pet:
        """Creates and shows a pet using the abstract factory"""
        pet = self.pet_factory(name)
        print(f"Here is your lovely {pet}")
        return pet


# Factory Method
def random_animal(name: str) -> Pet:
    """Let's be dynamic!"""
    return random.choice([Dog, Cat])(name)


if __name__ == "__main__":
    catShop = PetShop(Cat) # 只能购买"猫"的宠物店
    cat = catShop.buy_pet("Kitty")
    petShop = PetShop(random_animal)
    for name in ["Anddy", "DouDou", "Mua"]:
        pet = petShop.buy_pet(name)
        pet.speak()
```

1.2 Builder (生成器)

意图

将一个复杂对象的构建与表示分离，使得同样的构建过程可以创建不同的表示。

动机

假设我们要实例化一个机器人对象，机器人有手，眼睛，传感器等等参数需要构造。对于不同的机器人有些参数是必要的，有些是不必要的，有些可以使用默认参数。为了实现机器人对象的实例化，我们的构造方法往往会十分复杂，需要加入很多参数，甚至于需要书写不止一个构造函数。

为了避免上述情况，我们可以在机器人内部创建一个 builder 对象，调用 builder 对象不同的方法用于设置不同的参数。在机器人的构造函数中使用 builder 来对这些参数进行赋值[2]。因此，含有 builder 的类实例化代码往往是这样的:

```
CompanyClient client = new CompanyClient.Builder()
    .setCompanyName("百度")
    .setCompanyAddress("海定区百度大厦")
    .setCompanyRegfunds(5)
    .setmPerson("1000人以上")
    .build();
```

在上述代码中，我们可以有选择性地为 company 设置属性，这样设置属性地方式更加灵活。其中 company 通常被称为导向器 (director)。

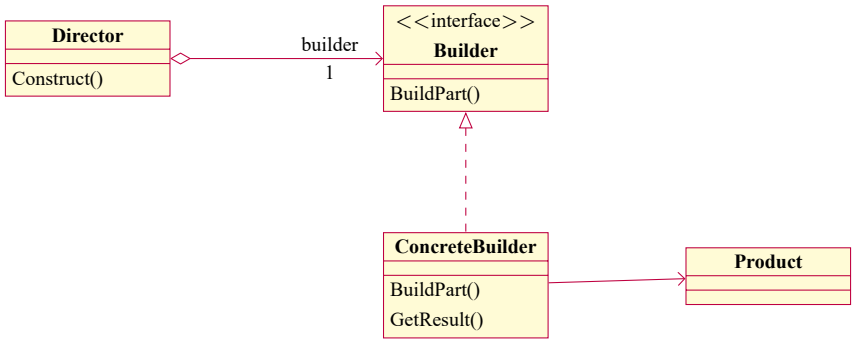
适用性

以下情况使用 Builder 模式:

- 当创建复杂对象地算法应该独立于该对象地组成部分以及它们地装配方式时。
- 当构造过程必须允许被构造的对象有不同的表示时。

结构

Builder 模式的结构如下图所示



参与者

- **Builder**: 为创建一个 Product 对象的各个部件指定抽象接口。

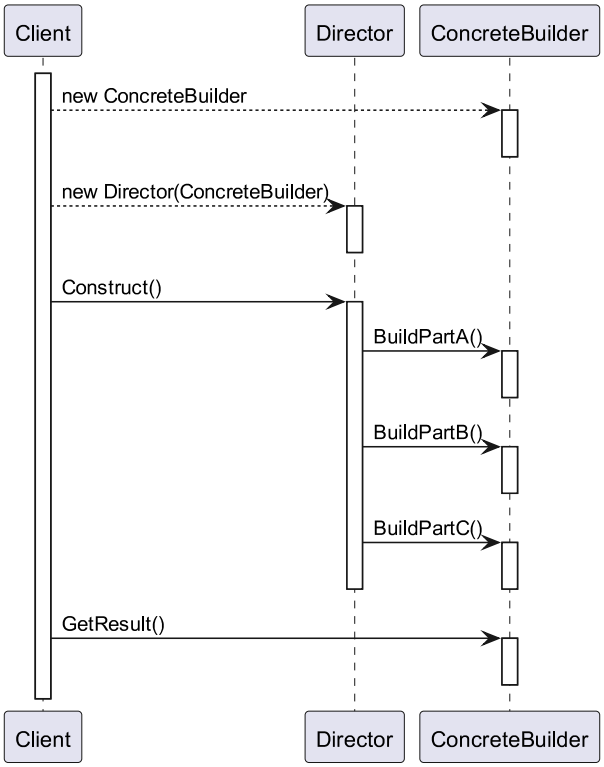
[2]比较抽象，建议结合例子链接理解。

- **ConcreteBuilder**: 实现接口以构造和装配该产品的各个部件。
- **Director**: 构造一个使用 Builder 接口的对象。
- **Product**: 被构造的复杂对象。

协作

- 用户创建 Director 对象，并用他所想要的 Builder 对象进行配置。
- 一旦生成了产品部件，导向器就会通知生成器。
- 生成器处理导向器的请求，并将部件添加到该产品中。
- 客户从生成器中检索产品。

下图的交互说明了 Builder 和 Director 是如何与一个客户协作的。



优缺点

- **可以改变一个产品的内部表示**
- **对构造过程进行更精细的控制**
- **构造代码变得更加复杂，并不美观**

实现

- **装配和构造接口**: 生成器逐步地构造他们的产品。因此 Builder 类接口必须足够普遍，以便为各种类型的具体生成器构造产品。

相关模式

- Abstract Factory: 与 Builder 类似，不过 Abstract Factory 注重于多个系列地产品对象。Builder 注重于一步步构造一个复杂的对象。
- Composite: 通常用 Builder 生成。

例子

- Java 实现 Builder 设计模式: https://blog.csdn.net/qq_17678217/article/details/86507693

```python
# Reference:
    https://github.com/faif/python-patterns/blob/master/patterns/creational/builder.py


# Abstract Building
class Building:
    def __init__(self):
        self.build_floor()
        self.build_size()

    def build_floor(self):
        raise NotImplementedError

    def build_size(self):
        raise NotImplementedError

    def __repr__(self):
        return "Floor: {0.floor} | Size: {0.size}".format(self)


# Concrete Buildings
class House(Building):
    def build_floor(self):
        self.floor = "One"

    def build_size(self):
        self.size = "Big"


# Concrete Buildings
class Flat(Building):
    def build_floor(self):
        self.floor = "More than One"

    def build_size(self):
        self.size = "Small"


class ComplexBuilding:
    def __repr__(self):
        return "Floor: {0.floor} | Size: {0.size}".format(self)


class ComplexHouse(ComplexBuilding):
    def build_floor(self):
        self.floor = "One"

    def build_size(self):
```

```
        self.size = "Big and fancy"


# construct
def construct_building(cls):
    building = cls()
    building.build_floor()
    building.build_size()
    return building


if __name__ == "__main__":
    house = House()
    print(house) # Floor: One | Size: Big
    flat = Flat()
    print(flat) # Floor: More than One | Size: Small
    complex_house = construct_building(ComplexHouse)
    print(complex_house) # Floor: One | Size: Big and fancy
```

1.3 Factory Method (工厂方法)

意图

定义一个用于创建对象的接口，让子类决定实例化哪一个类。Factory Method 使一个类的实例化延迟到子类。

动机

假设我们有一个产品 A，随着用户需求变化，需要生产 B 类产品。改变原有的配置非常困难，假设用户下一次再发生变化，再次改变将增大成本。因此我们可以为每类产品创建对应的工厂函数。

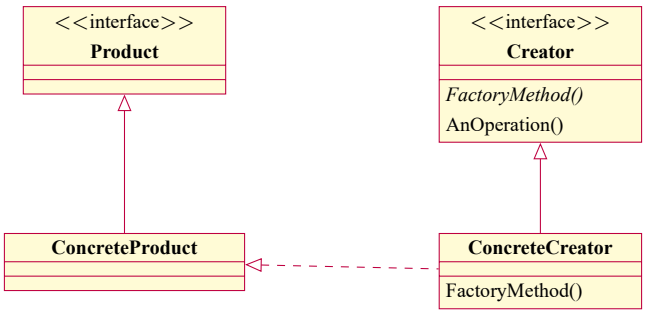
工厂模式是 new 的一个替代品，他有诸多好处，但与此同时，他会增加代码复杂度，需要读者慎重考虑。

适用性

以下情况可以使用 Factory Method 模式:

- 当一个类不知道它所必须创建的对象的类的时候。
- 当一个类希望由它的子类来指定它所创建的对象的时候。
- 当类将创建对象的职责委托给多个帮助子类中的一个，并且你希望将哪一个帮助子类是代理者这一信息局部化的时候。

结构



参与者

- **Product**: 定义工厂方法所创建的对象的接口。
- **ConcreteProduct**: 实现 Product 接口。
- **Creator**: 声明工厂方法，返回一个 Product 类型的对象。Creator 也可以定义一个工厂方法的缺省实现，返回一个缺省的 ConcreteProduct 对象。可以调用工厂方法以创建一个 Product 对象。
- **ConcreteCreator**: 重定义工厂方法以返回一个 ConcreteProduct 实例。

协作

- Creator 依赖于它的子类来定义工厂方法，所以它返回一个适当的 ConcreteProduct 实例。

优缺点

- **为子类提供钩子**: 用工厂方法再一个类的内部创建对象通常比直接创建对象更灵活。Factory Method 给子类一个钩子以提供对象的扩展版本。
- **连接平行的类层次**
- **代码复杂**

实现

- **参数化工厂方法**: 采用一个标识要被创建的对象种类的参数，以创建多种产品。
- **使用模板以避免创建子类**: 我们可能会为了创建适当的 Product 对象而被迫创建 Creator 子类。我们可以提供 Creator 的一个模板子类，使用 Product 类作为模板参数。

代码

- Java 实现 Factory Method: `https://blog.csdn.net/varyall/article/details/82355964`

```python
# Reference:
    https://github.com/faif/python-patterns/blob/master/patterns/creational/factory.py


# ConcreteProduct
class GreekLocalizer:
    """A simple localizer a la gettext"""
    def __init__(self) -> None:
        self.translations = {"dog": "σκύλος", "cat": "γάτα"}

    def localize(self, msg: str) -> str:
        """We'll punt if we don't have a translation"""
        return self.translations.get(msg, msg)


# ConcreteProduct
class EnglishLocalizer:
    """Simply echoes the message"""
    def localize(self, msg: str) -> str:
        return msg


# Factory Method
def get_localizer(language: str = "English") -> object:
    localizers = {
        "English": EnglishLocalizer,
        "Greek": GreekLocalizer,
    }
    return localizers[language]()


if __name__ == "__main__":
    e, g = get_localizer(language="English"), get_localizer(language="Greek")
    for msg in "dog parrot cat bear".split():
        print(e.localize(msg), g.localize(msg))
```

1.4 Prototype (原型)

意图

用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

动机

如果我们需要创建同一个类的多个实例，这需要给这些实例分配大量的内存，消耗 CPU 资源。而在有些情况下，往往这些实例的属性是相同或者只有极小的差别。

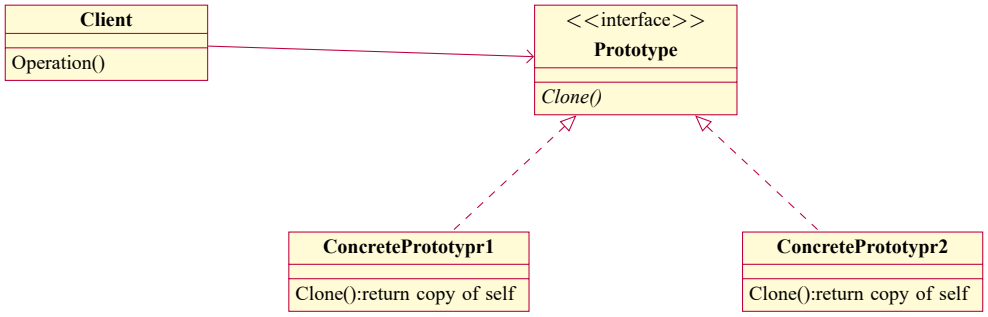
为了释放内存资源，提高性能，我们采用原型模式，直接在内存中拷贝已有的实例。

适用性

下列情况可以使用 Prototype 模式:

- 一个系统独立于它的产品创建，构成和表示时。
- 要实例化的类是在运行时指定的，例如动态转载。
- 避免创建一个与产品类层次平行的工厂层次时。
- 一个类的实例只有几个不同状态中的一种时。建立原型并克隆他们。

结构



参与者

- **Prototype**: 声明一个克隆自身的接口
- **ConcretePrototype**: 实现一个克隆自身的操作
- **Client**: 让原型克隆自身从而创建一个新的对象

协作

- 客户请求一个原型克隆自身。

优缺点

- **运行时增加和删除产品**: 允许客户注册原型实例就将一个新的具体产品类并入系统。比其它创建型模式更灵活。
- **减少子类构造**
- **动态配置应用**

实现

- **使用原型管理器**: 当一个系统中原型数目不固定时，要保持一个可用原型的注册表。

在注册表中存储和检索原型，我们通常称之为原型管理器 (Prototype Manager)。

- **实现克隆操作**: Prototype 模式最难实现的地方在于实现 Clone 操作。当对象包含循环时，者尤其棘手。

相关模式

- **Abstract Factory**: 在某些方面，这两个模式是相互竞争的，但他们也可以一起使用。Abstract Factory 可以存储一个被克隆的原型的集合，并返回产品对象。
- **Composite/Decorator**: 这两个设计模式可以从 Prototype 中获益。

代码

- Java 实现 Prototype: `https://blog.csdn.net/qq_38526573/article/details/87633257`

```python
# Reference:
    https://github.com/faif/python-patterns/blob/master/patterns/creational/prototype.py

from __future__ import annotations

from typing import Any


class Prototype:
    def __init__(self, value: str = "default", **attrs: Any) -> None:
        self.value = value
        self.__dict__.update(attrs)

    def clone(self, **attrs: Any) -> Prototype:
        """Clone a prototype and update inner attributes dictionary"""
        # Python in Practice, Mark Summerfield
        # copy.deepcopy can be used instead of next line.
        obj = self.__class__(**self.__dict__)
        obj.__dict__.update(attrs)
        return obj


class PrototypeDispatcher:
    def __init__(self):
        self._objects = {}

    def get_objects(self) -> dict[str, Prototype]:
        """Get all objects"""
        return self._objects

    def register_object(self, name: str, obj: Prototype) -> None:
        """Register an object"""
        self._objects[name] = obj

    def unregister_object(self, name: str) -> None:
        """Unregister an object"""
        del self._objects[name]


```

```
if __name__ == "__main__":
    dispatcher = PrototypeDispatcher()
    prototype = Prototype()
    d = prototype.clone()
    a = prototype.clone(value='a-value', category='a')
    b = a.clone(value='b-value', is_checked=True)
    dispatcher.register_object('objecta', a)
    dispatcher.register_object('objectb', b)
    dispatcher.register_object('default', d)
    print(b.category, b.is_checked) # a True
```

1.5 Singleton (单件)

意图

保证一个类仅有一个实例，并提供一个访问它的全局访问点。

动机

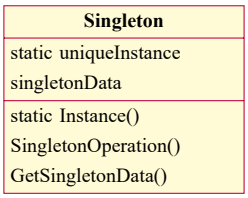
对于一些类来说，只有一个实例是十分重要的，怎样保证只有一个实例并且这个实例易于被访问呢? 全局变量使得一个对象可以被访问，但它不能防止你实例化多个对象。

一个更好的办法是，让类自身负责保存它的唯一实例。这个类可以保证没有其它实例可以被创建 (通过截取创建新对象的请求)，并且它可以通过一个访问该实例的方法。

适用性

- 类只能有一个实例并且客户可以从一个总所周知的访问点访问它时。
- 当这个唯一实例应该是通过子类化可扩展的，并且用户应该无需更改代码就能使用一个扩展的实例时。

结构



参与者

- **Singleton**: 定义一个 Instance 操作，允许客户访问它的唯一实例。

协作

- 用户只能通过 Singleton 的 Instance 操作访问一个 Singleton 实例。

优缺点

- **对唯一实例的受控访问**: 因为 Singleton 类封装它的唯一实例，所以可以严控客户怎样以及何时访问它。
- **缩小名字空间**: Singleton 模式是对全局变量的一种改进，它避免了那些存储唯一实例的全局变量污染名字空间。
- **允许对操作和表示的精化**: Singleton 类可以有子类，而且用这个扩展类的实例来配置一个应用是很容易的。
- **允许可变数目的实例**: 可以改变方法限制实例的个数。

实现

- **保证一个唯一的实例**
- **创建 Singleton 类的子类**

例子

- Java: https://blog.csdn.net/wo_shi_ltb/article/details/78773957
- Python: https://blog.csdn.net/u010569893/article/details/104264281

使用 Python/Java 实现 Singleton 模式难免会遇到线程安全，并发问题... 在上面两个例子中有对应的解决方案，下面给出的是一个基础的例子。

```python
# Reference: https://blog.csdn.net/lqxqust/article/details/51910007


class Singleton(object):
    instance = None

    def __new__(cls, *args, **kwargs):
        if cls.instance is None:
            cls.instance = super().__new__(cls, *args, **kwargs)
        return cls.instance


if __name__ == "__main__":
    t1 = Singleton()
    t2 = Singleton()
    print(id(t1) == id(t2))
```

2 结构模式

2.1 Adapter (适配器)

意图

将一个类的接口转换成客户希望的另一个接口。Adapter 模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

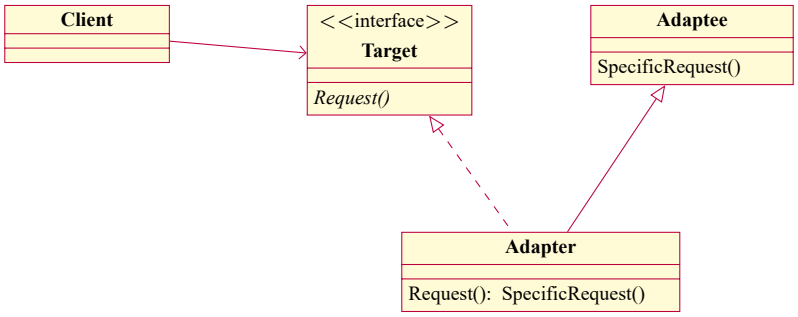
别名: 包装器 (wrapper)

适用性

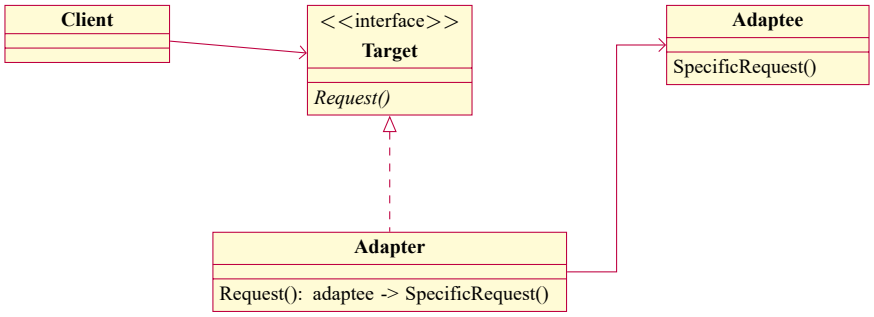
- 想使用一个已经存在的类，但它的接口不符合需求。
- 想创建一个可复用的类，该类需要与其它不相关/不可预见的类协同工作。
- (对象) 想使用一些已经存在的子类，但是不可能对每一个都进行子类化以匹配它们的接口。对象适配器可以适配它们的父类接口。

结构

类适配器使用多重继承对一个接口与另一个接口进行匹配:



对象适配器依赖于对象组合:



参与者

- **Target**: 定义 CLient 使用的与特定领域相关的接口。
- **Client**: 与符合 Target 接口的对象协调。
- **Adaptee**: 定义一个已经存在的接口，这个接口需要适配。
- **Adapter**: 对 Adaptee 的接口与 Target 接口进行适配。

协作

• Client 在 Adpater 实例上调用了一些操作。接着适配器调用 Adaptee 的操作实现这个请求。

优缺点

类适配器:

• 用一个具体的 Adapter 类对 Adaptee 和 Target 进行匹配。结果是当我们想要匹配一个类以及所有它的子类时，类 Adapter 将不能胜任工作。

对象适配器:

• 允许一个 Adapter 与多个 Adaptee 同时工作。Adapter 也可以一次给所有的 Adaptee 添加功能。

例子

• Java: https://blog.csdn.net/qq_38974638/article/details/124149535

```python
# Reference:
    https://github.com/faif/python-patterns/blob/master/patterns/structural/adapter.py

from typing import Callable, TypeVar

T = TypeVar("T")


class Dog:
    def __init__(self) -> None:
        self.name = "Dog"

    def bark(self) -> str:
        return "woof!"


class Cat:
    def __init__(self) -> None:
        self.name = "Cat"

    def meow(self) -> str:
        return "meow!"


class Human:
    def __init__(self) -> None:
        self.name = "Human"

    def speak(self) -> str:
        return "'hello'"


class Car:
    def __init__(self) -> None:
        self.name = "Car"
```

```
35
36      def make_noise(self, octane_level: int) -> str:
37          return f"vroom{'!' * octane_level}"
38
39
40  class Adapter:
41      def __init__(self, obj: T, **adapted_methods: Callable):
42          """We set the adapted methods in the object's dict."""
43          self.obj = obj
44          self.__dict__.update(adapted_methods)
45
46      def __getattr__(self, attr):
47          """All non-adapted calls are passed to the object."""
48          return getattr(self.obj, attr)
49
50      def original_dict(self):
51          """Print original object dict."""
52          return self.obj.__dict__
53
54
55  if __name__ == "__main__":
56      objects = []
57      dog = Dog()
58      print(dog.__dict__) # {'name': 'Dog'}
59      objects.append(Adapter(dog, make_noise=dog.bark))
60      print(objects[0].original_dict()) # {'name': 'Dog'}
61
62      cat = Cat()
63      objects.append(Adapter(cat, make_noise=cat.meow))
64      human = Human()
65      objects.append(Adapter(human, make_noise=human.speak))
66      car = Car()
67      objects.append(Adapter(car, make_noise=lambda: car.make_noise(3)))
68      for obj in objects:
69          print("A {0} goes {1}".format(obj.name, obj.make_noise()))
70      # A Dog goes woof!
71      # A Cat goes meow!
72      # A Human goes 'hello'
73      # A Car goes vroom!!!
```

2.2 Bridge (桥接)

意图

将抽象部分与它的实现部分分离，使他们可以独立地变化。

别名: Handle/Body

动机

当一个抽可能有多个实现时，通常用继承来协调它们。抽象类定义对改抽象的接口，而具体的子类则用不同方式加以是实现。但是此方法有时候不够灵活。继承机制将抽象部分与它的实现部分固定在一起，使得难以对抽象部分和实现部分独立地进行修改，扩充和复用。

假设需要设计 3 种颜色 4 类形状的图案，那么就需要在图案类下派生出 12 个不同的子类以实现全部的图案样式。而如果现在又新增了一种颜色，就又需要派生出 4 个新的类，这显然非常复杂。

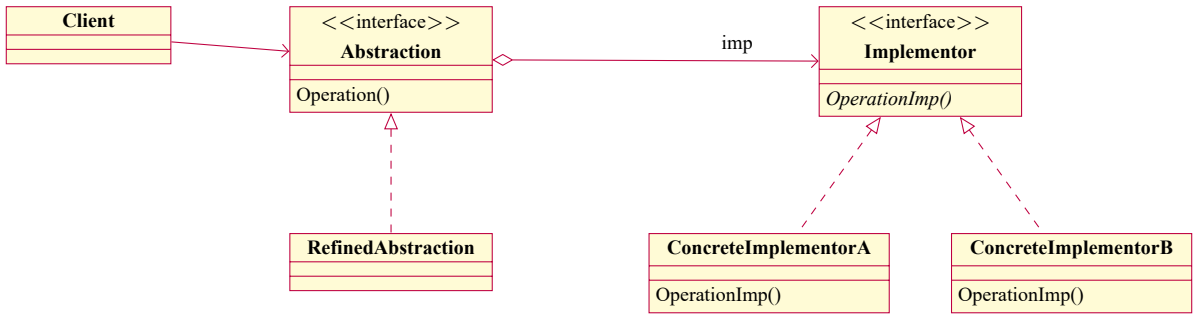
一种好的方式是将颜色和图案非为不同的类，让图案类包含颜色元素。这样我们就只需要编写 7 个类，再添加功能时也只需要添加一个基类。

桥接模式将继承关系转换成关联关系，从而降低了类与类之间的耦合，减少了代码的编写量。

适用性

- 不希望再抽象和实现部分之间有一个固定的关系。
- 类的抽象和实现都应该可以通过生成子类的方法加以扩充。

结构



参与者

- **Abstraction**: 定义抽象类接口，维护指向 Implementor 类型的指针。
- **RefinedAbstraction**: 扩充由 Abstraction 定义的接口。
- **Implementor**: 定义实现类的接口。
- **ConcreteImplementor**: 实现 Implementot 接口并定义它的具体实现。

协作

- Abstraction 将 client 的请求转发给它的 Implementor 对象。

优缺点

- **分离接口与实现部分**: 实现未必要绑定在接口上，抽象类的实现可以再运行时进行配置，一个对象甚至可以在运行时改变它的实现。
- **提高可扩充性**: 可以独立地对 Abstraction 和 Implementor 层次结构进行扩充。

例子

- Java: https://blog.csdn.net/en_joker/article/details/82839813
- Video: https://www.bilibili.com/video/BV1Pp4y167DG

```python
# Reference: https://github.com/faif/python-patterns/blob/master/patterns/structural/bridge.py


# ConcreteImplementor
class DrawingAPI1:
    def draw_circle(self, x, y, radius):
        print(f"API1.circle at {x}:{y} radius {radius}")


# ConcreteImplementor
class DrawingAPI2:
    def draw_circle(self, x, y, radius):
        print(f"API2.circle at {x}:{y} radius {radius}")


# Refined Abstraction
class CircleShape:
    def __init__(self, x, y, radius, drawing_api):
        self._x = x
        self._y = y
        self._radius = radius
        self._drawing_api = drawing_api

    # low-level i.e. Implementation specific
    def draw(self):
        self._drawing_api.draw_circle(self._x, self._y, self._radius)

    # high-level i.e. Abstraction specific
    def scale(self, pct):
        self._radius *= pct


if __name__ == "__main__":
    shapes = (CircleShape(1, 2, 3,
                          DrawingAPI1()), CircleShape(5, 7, 11, DrawingAPI2()))
    for shape in shapes:
        shape.scale(2.5) # API1.circle at 1:2 radius 7.5
        shape.draw() # API2.circle at 5:7 radius 27.5
```

2.3 Composite (组合)

意图

将对象组合成树形结构以表示 “整体-部分” 的层次结构。Composite 使得用户对单个对象和组合对象的使用具有一致性。

动机

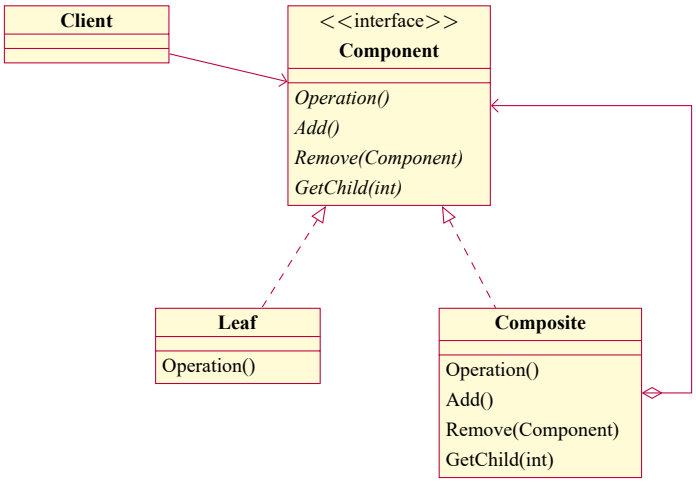
Composite 对应 UML 中的组合关系，注意 UML 中组合 (composition) 和聚合 (aggregation) 的区别:

- 聚合: has-a 关系，部分可以脱离整体存在，如学校有老师，学校不存在了，老师依然存在。
- 组合: part-of 关系，部分不可以脱离主体存在，如脸上有嘴巴，没有脸，嘴巴不能单独存在。

适用性

- 想表达部分-整体层次。
- 希望用户忽略组合对象与单个对象的不同。用户将统一地使用组合结构中的所有对象。

结构



参与者

- **Component**: 为组合中的对象声明接口，声明接口用于访问和管理 Component 子组件。
- **Leaf**: 在组合中表示叶节点对象，定义行为。
- **Composite**: 定义有子部件的那些部件的行为；存储子部件；在 Component 接口中实现与子部件有关的操作。
- **Client**: 通过 Component 接口操纵组合部件的对象。

协作

- 用户使用 Component 类接口与组合结构中的对象进行交互。如果接收者是一个叶节点，则直接处理请求。如果接受者是 Composite，它通过将请求发送给它的子部件在转

发请求之前和/或之后可能执行一些辅助操作。

优缺点

- 定义了包含基本对象和组合对象的类层次结构
- 使得更容易增加新型的组件
- 使你的设计变得更加一般化

实现

- **显式的父部件引用**: 保持子部件到父部件的引用能简化组合结构的便利和管理。通常在 Component 类中定义父部件引用。Leaf 和 Composite 类可以继承这个引用以及管理这个引用的哪些操作。
- **共享组件**: 共享组件可以减少对存储的需求。一个组件应该有多个父组件，但这会导致二义性问题，Flyweight 模式讨论了如何修改设计。
- **最大化 Component 接口**: Composite 模式的目的之一是使得用户不知道他们正在使用的具体的 Leaf 和 Composite 类。为了达到这一目的，Composite 类应为 Leaf 和 Composite 类尽可能多定义一些公共操作。

例子

- 视频: `https://www.bilibili.com/video/BV1854y147ez`

```python
# Reference:
    https://github.com/faif/python-patterns/blob/master/patterns/structural/composite.py

from abc import ABC, abstractmethod
from typing import List


class Graphic(ABC):
    @abstractmethod
    def render(self) -> None:
        raise NotImplementedError("You should implement this!")


class CompositeGraphic(Graphic):
    def __init__(self) -> None:
        self.graphics: List[Graphic] = []

    def render(self) -> None:
        for graphic in self.graphics:
            graphic.render()

    def add(self, graphic: Graphic) -> None:
        self.graphics.append(graphic)

    def remove(self, graphic: Graphic) -> None:
        self.graphics.remove(graphic)


```

```python
class Ellipse(Graphic):
    def __init__(self, name: str) -> None:
        self.name = name

    def render(self) -> None:
        print(f"Ellipse: {self.name}")


if __name__ == "__main__":
    ellipse1 = Ellipse("1")
    ellipse2 = Ellipse("2")
    ellipse3 = Ellipse("3")
    ellipse4 = Ellipse("4")
    graphic1 = CompositeGraphic()
    graphic2 = CompositeGraphic()
    graphic1.add(ellipse1)
    graphic1.add(ellipse2)
    graphic1.add(ellipse3)
    graphic2.add(ellipse4)
    graphic = CompositeGraphic()
    graphic.add(graphic1)
    graphic.add(graphic2)
    graphic.render()
    # Ellipse: 1
    # Ellipse: 2
    # Ellipse: 3
    # Ellipse: 4
```

2.4 Decorator (装饰)

意图

动态地给一个对象添加一些格外的职责。就增加功能来说，Decorator 模式相比生成子类更为灵活。

别名: 包装机 (wrapper)

动机

假定我们有一篇已经填入了全部内容的纸质文章，现在我们需要在文章的开头加上公司名称，结尾加上公司 logo。

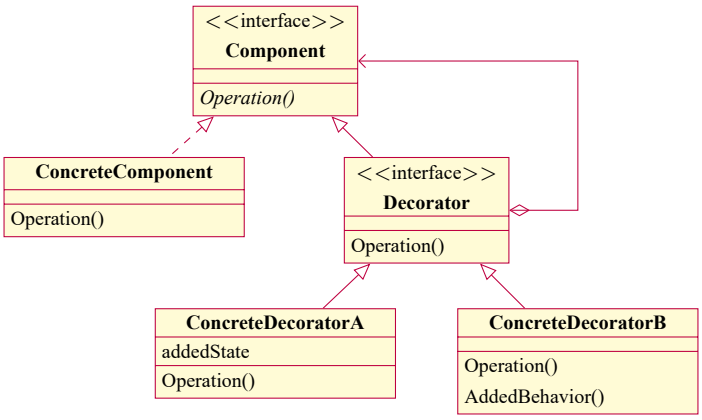
应该怎么办呢，传统的方式是在文章类的基础上派生出一个新的公司文章类，并添加新的功能，但这些功能和基类并没有逻辑上的联系以及数据交换，因此可以想办法避免类派生。

装饰模式通过将组件嵌入另一个对象的模式为上述问题提供了一个解决方案。

适用性

- 在不影响其他对象的情况下，以动态透明的方式给单个对象添加职责。
- 处理那些可以撤销的职责。
- 当不能采用子生成类的方式进行扩充时。

结构



参与者

- **Component**: 定义一个对象接口，可以给这些对象动态地添加职责。
- **ConcreteComponent**: 定义一个对象，可以给这个对象添加一些职责。
- **Decorator**: 维持一个指向 Component 对象的指针，并定义一个与 Component 接口一致的接口。
- **ConcreteDecorator**: 向组件添加职责。

协作

- Decorator 将请求转发给它的 Component 对象，并有可能在转发请求前后执行一些附加的动作。

优缺点

- 比静态继承更灵活
- 避免在层次结构高层的类有太多的特征
- **对象问题**: Decorator 和 Component 不一样，Decorator 是一个透明的包装。我们从对象标识的角度出发，被装饰的对象并不能视作新的对象。如果一个系统采用 Decorator 模式，在运行时可能会出现很多个看上去类似的对象，对于不了解系统实现的人来说，学习系统与排错将变得很困难。

实现

- **接口的一致性**: 装饰对象的接口必须与它所装饰的 Component 的接口时一致的，因此，所有的 ConcreteDecorator 类必须有一个公共的父类。
- **省略抽象的 Decorator 类**: 当只需要添加一个职责时，没有必要定义抽象 Decorator 类。
- **保持 Component 类的简单性**: 为了保证接口的一致性，组件和装饰必须有一个公共的 Component 父类。因此保持这个类的简单性是很重要的，即它应集中于定义接口而不是存储数据。
- **改变对象外壳与内核**: 装饰模式应该只改变对象的外壳，内核应该由其他模式控制。

实例

- Java: `https://blog.csdn.net/xiaofeng10330111/article/details/105608235`
- Video: `https://www.bilibili.com/video/BV1hp4y1D7MP`

```python
# Reference:
    https://github.com/faif/python-patterns/blob/master/patterns/structural/decorator.py


class TextTag:
    """Represents a base text tag"""
    def __init__(self, text):
        self._text = text

    def render(self):
        return self._text


class BoldWrapper(TextTag):
    """Wraps a tag in <b>"""
    def __init__(self, wrapped):
        self._wrapped = wrapped

    def render(self):
        return f"<b>{self._wrapped.render()}</b>"


class ItalicWrapper(TextTag):
    """Wraps a tag in <i>"""
    def __init__(self, wrapped):
        self._wrapped = wrapped
```

```
26
27      def render(self):
28          return f"<i>{self._wrapped.render()}</i>"
29
30
31  def main():
32      simple_hello = TextTag("hello, world!")
33      special_hello = ItalicWrapper(BoldWrapper(simple_hello))
34      print("before:", simple_hello.render()) # before: hello, world!
35      print("after:",
36            special_hello.render()) # after: <i><b>hello, world!</b></i>
```

2.5 Facade (外观)

意图

为子系统中的一组接口提供一个一致的界面，Facade 模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

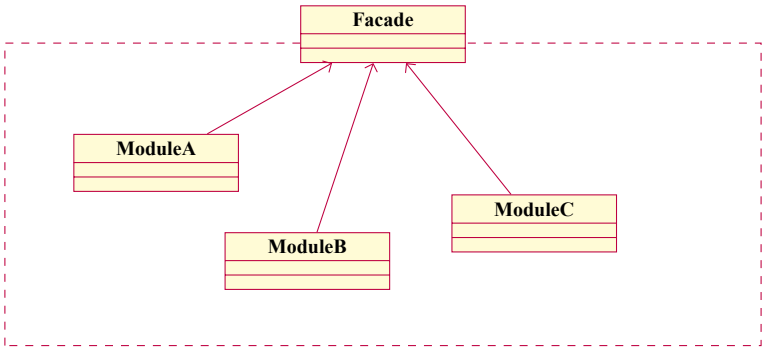
动机

将一个系统划分成若干个子系统有利于降低系统的复杂性。一个常见的设计目标时使子系统间的通信和相互依赖关系达到最小。达到该目标的途径之一就是引入一个外观对象，它为子系统中较一般的设施提供了一个单一而简单的界面。

适用性

- 要为一个复杂子系统提供一个简单接口时。子系统往往因为不断演化而变得越来越复杂，大多数模式使用时都会产生更多更小的类。这使得子系统更具可复用性，也更容易对系统进行定制。
- 客户程序与抽象类的实现部分之间存在着很大的依赖性。引入 Facade 将这个子系统与客户以及其他的子系统分离，可以提高子系统的独立性与可移植性。
- 当你需要构建一个层次结构的子系统时，使用 Facade 模式定义子系统中每层的入口点。如果子系统之间是相互依赖的，可以让它们仅通过 Facade 进行通信，从而简化了它们之间的依赖关系。

结构



参与者

- **Facade**: 知道哪些子系统类负责处理请求；将客户的请求代理给适当的子系统对象。
- **Subsystem classed**: 实现子系统的功能；处理 Facade 对象指派的任务；没有 Facade 的任何相关信息，即没有指向 Facade 的指针。

协作

- 客户程序通过发送请求黑 Facade 的方式与子系统通信，Facade 将这些消息转发给适当的子系统对象。尽管是子系统中的有关对象在做实际工作，但 Facade 模式本身也必须将它的接口转换成子系统的接口。
- 使用 Facade 的客户端程序不需要直接访问子系统对象。

优缺点

- 对客户屏蔽了子系统组件，减少了客户处理对象的数目并使得系统用起来更加方便。
- 是西安了子系统与客户之间的松耦合关系，而子系统内部的功能组件往往是紧耦合的。

实现

- **降低客户-子系统之间的耦合度**: 用抽象类实现 Facade 而它的具体子类对应于不同的子系统是心啊，这可以进一步降低客户与子系统的耦合度。
- **公共子系统与私有子系统**: 一个子系统与一个类的相似之处是，它们都有接口并且都封装了一些东西。而子系统封装了一些类。子系统的公共接口包含所有客户程序都可以访问的类，私有接口仅用于对子系统进行扩充。

例子

- Java: https://blog.csdn.net/qq_45034708/article/details/114972361
- Video: https://www.bilibili.com/video/BV1xz4y1X7HW

```python
# Reference: https://github.com/faif/python-patterns/blob/master/patterns/structural/facade.py


class CPU:
    def freeze(self):
        print("Freezing processor.")

    def jump(self, position):
        print("Jumping to:", position)

    def execute(self):
        print("Executing.")


class Memory:
    def load(self, position, data):
        print(f"Loading from {position} data: '{data}'.")


class SolidStateDrive:
    def read(self, lba, size):
        return f"Some data from sector {lba} with size {size}"


class ComputerFacade:
    def __init__(self):
        self.cpu = CPU()
        self.memory = Memory()
        self.ssd = SolidStateDrive()

    def start(self):
        self.cpu.freeze()
        self.memory.load("0x00", self.ssd.read("100", "1024"))
        self.cpu.jump("0x00")
        self.cpu.execute()
```

```
36
37
38 def main():
39     computer_facade = ComputerFacade()
40     computer_facade.start()
41     # Freezing processor.
42     # Loading from 0x00 data: 'Some data from sector 100 with size 1024'.
43     # Jumping to: 0x00
44     # Executing.
```

2.6 Flyweight (享元)

意图

运用共享技术有效地支持大量细粒度的对象。

动机

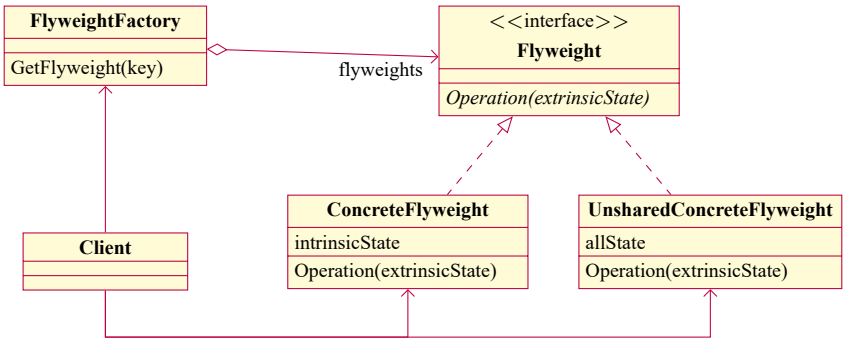
享元即: 共享元素，复用元素。例如在云盘中上传影片，在上传之前，云盘会对影片进行检查，如果发现影片在服务器中已经存在了，就不需要重新上传一份，即保存一份影片的源文件，在用户云盘中添加已有的影片即可。

在享元模式中，有外部状态和内部状态。内部状态是不会改变的，例如影片本身的数据。外部状态会改变，例如影片的上传时间，拥有者昵称。

适用性

- 一个程序使用了大量的对象。
- 完全由于使用大量的对象造成很大的开销。
- 对象的大多数状态都可变为外部状态。
- 如果删除对象的外部状态，那么可以用相对较少的共享对象取代很多组对象。
- 应用程序不依赖于对象标识。

结构



参与者

- **Flyweight**: 描述一个接口，通过这个接口 flyweight 可以接受并作用于外部状态。
- **ConcreteFlyweight**: 实现 Flyweight 接口，并为内部状态添加存储空间。对象必须是共享的。
- **UnsharedConcreteFlyweight**: 并非所有 Flyweight 子类都需要被共享。
- **FlyweightFactory**: 创建并管理 flyweight 对象。确保合理地共享 flyweight。当用户请求一个 flyweight 时 Flyweight 会创建一个实例或者一共一个已有的实例。
- **Client**: 维持一个对 flyweight 的引用。计算或者存储一个 (多个) flyweight 的外部状态。

协作

- flyweight 执行时所需的状态必定是内部的或外部的，内部状态存储于 ConcreteFlyweight 对象之中，而外部对象则是由 Client 对象存储或计算。当用户调用 flyweight 对象的操作时，将该状态传递给它。

- 用户不应直接对 ConcreteFlyweight 类进行实例化，而只能从 FlyweightFactory 对象得到 ConcreteFlyweight 对象，这可以保证对它们适当地进行共享。

优缺点

使用 Flyweight 模式时，传输，查找或计算内外部状态都会产生运行时的开销，尤其当 flyweight 原先被存储为内部状态时。然而，空间上的节省抵消了这些开销。共享的 flyweight 越多，节省空间就越大。

存储节约由以下几个因素决定:

- 由于共享带来的实例总数减少的数目
- 对象内部状态的平均数目
- 外部状态是计算的还是存储的

共享的 flyweight 越多，存储节约也就越多。节约量随着共享状态的增多而增大。当对象使用大量的内部及外部状态，并且外部状态都是计算出来的而非存储的时候，节约量将达到最大。所以，可以用两种方法来节约存储: 用共享减少内部状态的消耗，用计算时间换取对外部状态的存储。

实现

- **删除外部状态**: 该模式的可用性在很大程度上取决于是否容易识别外部状态并将它从共享对象中删除。理想情况是: 外部状态可以由一个单独的对象结构计算得到，且该结构的存储需求非常小。
- **管理共享对象**

实例

- Java: `https://blog.csdn.net/weixin_40980639/article/details/115287157`
- Video: `https://www.bilibili.com/video/BV1Ka4y1L7jg`

```python
# Reference:
    https://github.com/faif/python-patterns/blob/master/patterns/structural/flyweight.py

import weakref


class Card:
    """The Flyweight"""
    _pool: weakref.WeakValueDictionary = weakref.WeakValueDictionary()

    def __new__(cls, value, suit):
        obj = cls._pool.get(value + suit)
        if obj is None:
            obj = object.__new__(Card)
            cls._pool[value + suit] = obj
            obj.value, obj.suit = value, suit
        return obj

    def __repr__(self):
        return f"<Card: {self.value}{self.suit}>"
```

```
20
21
22 if __name__ == "__main__":
23     c1 = Card('9', 'h')
24     c2 = Card('9', 'h')
25     c1, c2 # (<Card: 9h>, <Card: 9h>)
26     c1 == c2 # True
27     c1 is c2 # True
28     c1.new_attr = 'temp'
29     c3 = Card('9', 'h')
30     hasattr(c3, 'new_attr') # True
31     Card._pool.clear()
32     c4 = Card('9', 'h')
33     hasattr(c4, 'new_attr') # False
```

2.7 Proxy (代理)

意图

为其他对象提供一种代理以控制对这个对象的访问。

别名: Surrogate

动机

当一个对象进行访问控制的一个原因是: 只有我们确实需要这个对象时才对它进行创建和初始化。

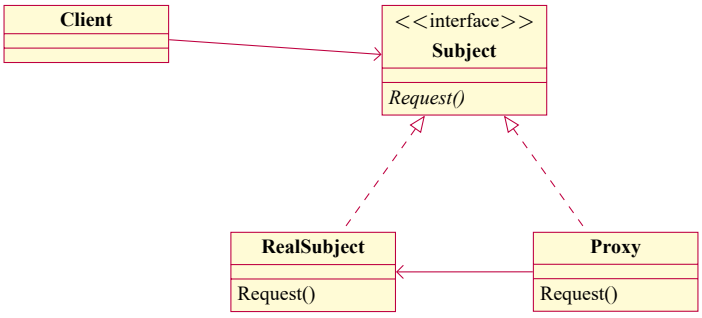
例如我们需要访问谷歌，但由于不可名状的原因，我们无法通过浏览器直接访问。但是我们可以通过一些代理软件简介访问服务器。

总而言之，仅通过代理类才能访问到背后的对象。

适用性

- **远程代理**: 为一个对象在不同的地址空间提供局部代表。
- **虚代理**: 根据需要创建开销很大的对象。
- **保护代理**: 控制对原始对象的访问。保护代理用于对象应该有不同的访问权限的时候。
- **智能代理**: 取代了简单的指针，再访问对象时执行一些附加操作。

结构



参与者

- **Proxy**: 保存一个引用使得代理可以访问实体；控制对实体的存取，并可能负责创建和删除它。
- **Subject**: 定义 RealSubject 和 Proxy 的共用接口，这样就在任何使用 RealSubject 的地方都可以使用 Proxy
- **RealSubject**: 定于 Proxy 所代表的实体。

协作

- 代理根据其种类，再适当的时候向 RealSubject 转发请求。

优缺点

- Peoxy 模式再访问对象时引入了一定程度的间接性。

实例

- Java: https://blog.csdn.net/xiaofeng10330111/article/details/105633821
- Video: https://www.bilibili.com/video/BV15V411z7nD

```python
# Reference: https://github.com/faif/python-patterns/blob/master/patterns/structural/proxy.py

from typing import Union


class Subject:
    def do_the_job(self, user: str) -> None:
        raise NotImplementedError()


class RealSubject(Subject):
    def do_the_job(self, user: str) -> None:
        print(f"I am doing the job for {user}")


class Proxy(Subject):
    def __init__(self) -> None:
        self._real_subject = RealSubject()

    def do_the_job(self, user: str) -> None:
        print(f"[log] Doing the job for {user} is requested.")
        if user == "admin":
            self._real_subject.do_the_job(user)
        else:
            print("[log] I can do the job just for `admins`.")


def client(job_doer: Union[RealSubject, Proxy], user: str) -> None:
    job_doer.do_the_job(user)


if __name__ == "__main__":
    proxy = Proxy()
    real_subject = RealSubject()
    client(proxy, 'admin')
    # [log] Doing the job for admin is requested.
    # I am doing the job for admin
    client(proxy, 'anonymous')
    # [log] Doing the job for anonymous is requested.
    # [log] I can do the job just for `admins`.
    client(real_subject, 'admin')
    # I am doing the job for admin
    client(real_subject, 'anonymous')
    # I am doing the job for anonymous
```

3 行为模式

3.1 ChainofResponsibility (职责链)

意图

使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连城一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。

动机

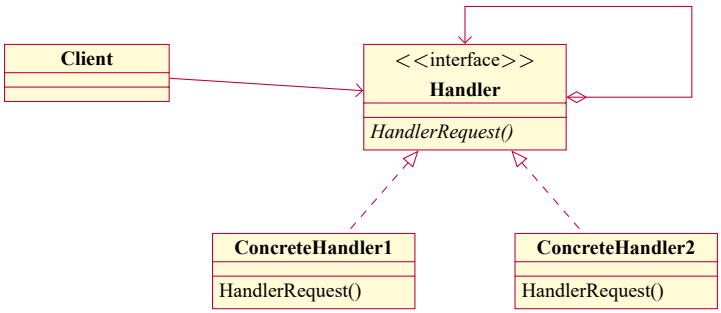
假设某大学生需要请假出校，向学院客户端发送了请假申请。如果当天去当天回，则由辅导员审批即可；如果需要请假 1-7 天，则由辅导员交给学院领导审批；如果需要休学，则由学院向学校申请审批。

在上述过程中，辅导员，学院，学校都可以对出校申请进行审批，这三者对请假处理事件构成了责任链。

适用性

- 有多个对象可以处理一个请求，哪个对象处理该请求运行时自动确定。
- 你想在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。
- 可处理一个请求的对象集合应被动态指定。

结构



参与者

- **Handler**: 定义一个处理请求的接口；实现后继链。
- **ConcreteHandler**: 处理负责的请求；访问后继者。
- **Client**: 向链上的具体处理者对象提交请求。

协作

- 当客户提交一个请求时，请求沿链传递直至有一个 ConcreteHandler 对象负责处理它。

优缺点

- **降低耦合度**: 使得一个对象无需知道是其他哪一个对象处理其请求。对象仅需要知道该请求会被 “正确” 处理。
- **增强了给对象指派职责的灵活性**: 当在对象中分派职责时，职责链给你更多的灵活性。
- **不保证被接受**: 既然一个请求没有明确的接收者，那么就不保证它一定会被处理 (可

能直到末端都不会被处理)。

实现

- **实现后继者链**: 有两种方法可以实现后继者链。1. 定义新的链接。2. 使用已有的链接。
- **连接后继者**: 如果没有已有的引用可定义一个链，那么你必须自己引入它们。这种情况下 Handler 不仅定义该请求的接口，通常也维护后继者。

例子

- Java: https://blog.csdn.net/qq359605040/article/details/122718721
- Video: https://www.bilibili.com/video/BV1uk4y127hG

```python
# Refernence:
    https://github.com/faif/python-patterns/blob/master/patterns/behavioral/chain_of_responsibility.py

from abc import ABC, abstractmethod
from typing import Optional, Tuple


class Handler(ABC):
    def __init__(self, successor: Optional["Handler"] = None):
        self.successor = successor

    def handle(self, request: int) -> None:
        res = self.check_range(request)
        if not res and self.successor:
            self.successor.handle(request)

    @abstractmethod
    def check_range(self, request: int) -> Optional[bool]:
        return NotImplementedError


class ConcreteHandler0(Handler):
    @staticmethod
    def check_range(request: int) -> Optional[bool]:
        if 0 <= request < 10:
            print(f"request {request} handled in handler 0")
            return True
        return None


class ConcreteHandler1(Handler):
    """... With it's own internal state"""

    start, end = 10, 20

    def check_range(self, request: int) -> Optional[bool]:
        if self.start <= request < self.end:
            print(f"request {request} handled in handler 1")
            return True
        return None
```

```
40
41
42  class ConcreteHandler2(Handler):
43      """... With helper methods."""
44      def check_range(self, request: int) -> Optional[bool]:
45          start, end = self.get_interval_from_db()
46          if start <= request < end:
47              print(f"request {request} handled in handler 2")
48              return True
49          return None
50
51      @staticmethod
52      def get_interval_from_db() -> Tuple[int, int]:
53          return (20, 30)
54
55
56  class FallbackHandler(Handler):
57      @staticmethod
58      def check_range(request: int) -> Optional[bool]:
59          print(f"end of chain, no handler for {request}")
60          return False
61
62
63  if __name__ == "__main__":
64      h0 = ConcreteHandler0()
65      h1 = ConcreteHandler1()
66      h2 = ConcreteHandler2(FallbackHandler())
67      h0.successor = h1
68      h1.successor = h2
69      requests = [2, 5, 14, 22, 18, 3, 35, 27, 20]
70      for request in requests:
71          h0.handle(request)
```

3.2 Command (命令)

意图

将一个请求封装为一个对象，从而使你可以用不同的请求对客户进行参数化，对请求排队或记录请求日志，以及支持可撤销的操作。

别名: 动作 (action), 事务 (transaction)

动机

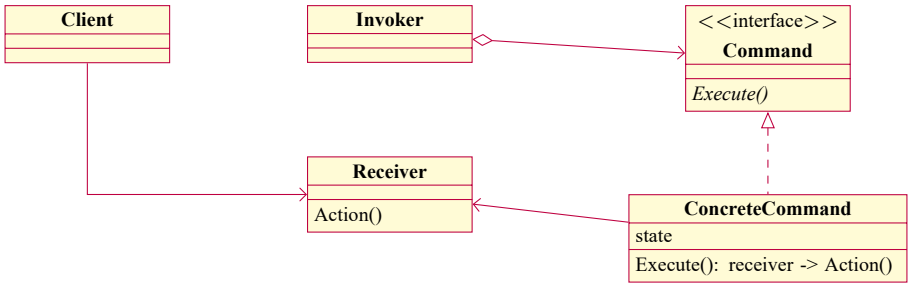
假设我们在开发一款文本编辑软件，我们需要通过按钮实现保存功能。那么我们可以将保存的逻辑直接写在按钮的 UI 类中。但是我们也可以使用右键-> 保存，Ctrl+S 的方式保存文件。显然我们没有必要在这三种方式中都添加保存的逻辑代码。

命令模式即是将某些方法参数化，作为回调函数以达到复用的效果。

适用性

- 抽象出执行的动作以参数化某对象。可用回调函数表示这种参数化机制。
- 在不同的时刻指定，排列和执行请求。
- 支持取消操作。Command 的 Execute 操作可在实施操作前将状态存储起来，在取消操作时这个状态用来消除该操作的影响。

结构

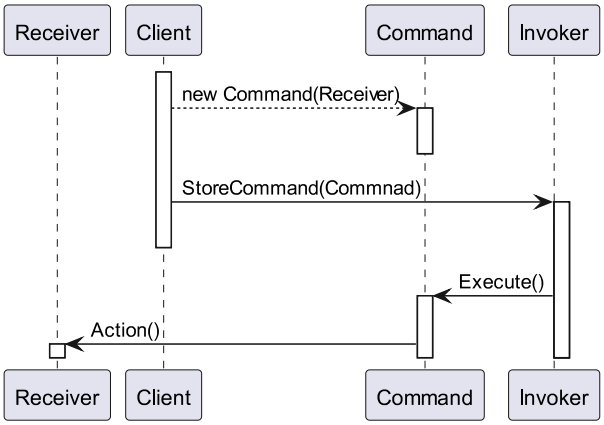


参与者

- **Command**: 声明执行操作的接口。
- **ConcreteCommand**: 将一个接收者对象绑定与一个动作；调用接收者相应的操作，以实现 Execute。
- **Client**: 创建一个具体命令对象并设定它的接收者。
- **Invoker**: 要求该命令执行这个请求。
- **Receiver**: 知道如何实施与执行一个请求相关的操作。

协作

- Client 创建一个 ConcreteCommand 对象并指定它的 Receiver 对象。
- 某 Invoker 对象存储该 ConcreteCommand 对象。
- 该 Invoker 通过调用 Command 对象的 Execute 操作来提交一个请求。
- ConcreteCommand 对象调用它的 Receiver 的一些操作以执行该请求。



优缺点

- Command 模式将调用操作的对象与知道如何实现该操作的对象解耦。
- Command 是头等的对象。它们可像其他的对象一样被操纵和扩展。
- 可以将多个命令装配成一个组合命令。
- 增加新的 Command 很容易，因为着无须改变已有的类。

实现

- **支持撤销 (undo) 和重做 (redo)**: 如果 Command 提供方法逆转它们操作的执行，就可支持撤销和重做功能。为达到这个目的，ConcreteConnamd 类可能需要格外的状态信息。

例子

- Java: `https://blog.csdn.net/weixin_40980639/article/details/123218117`
- Video: `https://www.bilibili.com/video/BV1G5411L7qg`

```python
# Reference:
    https://github.com/faif/python-patterns/blob/master/patterns/behavioral/command.py

from typing import List, Union


class HideFileCommand:
    def __init__(self) -> None:
        self._hidden_files: List[str] = []

    def execute(self, filename: str) -> None:
        print(f"hiding {filename}")
        self._hidden_files.append(filename)

    def undo(self) -> None:
        filename = self._hidden_files.pop()
        print(f"un-hiding {filename}")


class DeleteFileCommand:
    def __init__(self) -> None:
        self._deleted_files: List[str] = []
```

```
22
23      def execute(self, filename: str) -> None:
24          print(f"deleting {filename}")
25          self._deleted_files.append(filename)
26
27      def undo(self) -> None:
28          filename = self._deleted_files.pop()
29          print(f"restoring {filename}")
30
31
32  class MenuItem:
33      """
34      The invoker class. Here it is items in a menu.
35      """
36      def __init__(self, command: Union[HideFileCommand,
37                                        DeleteFileCommand]) -> None:
38          self._command = command
39
40      def on_do_press(self, filename: str) -> None:
41          self._command.execute(filename)
42
43      def on_undo_press(self) -> None:
44          self._command.undo()
45
46
47  if __name__ == "__main__":
48      item1 = MenuItem(DeleteFileCommand())
49      item2 = MenuItem(HideFileCommand())
50      test_file_name = 'test-file'
51      item1.on_do_press(test_file_name)
52      item1.on_undo_press()
53      item2.on_do_press(test_file_name)
54      item2.on_undo_press()
```

3.3 Interpreter (解释器)

意图

给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

动机

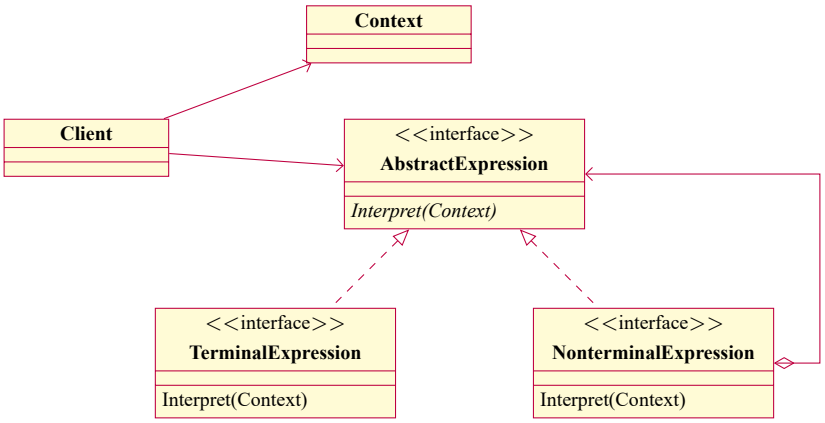
如果一种特定类型的问题发生的频率足够高，那么可能就值得将这个问题的各个实例表述为一个简单语言中的句子。这样就可以构建一个解释器，该解释器通过解释这些句子来解决该问题。例如正则表达式。

适用性

当有一个语言需要解释执行，并且你可将该语言中的句子表示为一个抽象语法树，可以使用解释器模式。而当存在以下情况时该模式效果最好。

- **文法简单**: 对于复杂的文法，文法的类层次变得庞大而无法管理。
- **效率不是关键问题**: 最高效的解释器通常不是通过直接解释语法分析树实现的，而是将它们转换成另一种形式。例如，正则表达式通常被转换成状态机。

结构



参与者

- **AbstractExpression**: 声明一个抽象的解释操作，这个接口为抽 i 想语法树中所有的节点所共享。
- **TerminalExpression**: 实现与文法中的终结符相关联的解释操作。一个句子中的每个终结符需要该类的一个实例。
- **NonterminalExpression**: 对文法中的每一条规则都需要一个对应的类。
- **Context**: 包含解释器之外的一些全局信息

解释器模式应用场景比较少，这里不做过多介绍。

3.4 Iterator (迭代器)

意图

提供一种方法顺序访问一个聚合对象中的各个元素，而又不需要暴露该对象的内部表示。

别名: cursor(游标)

动机

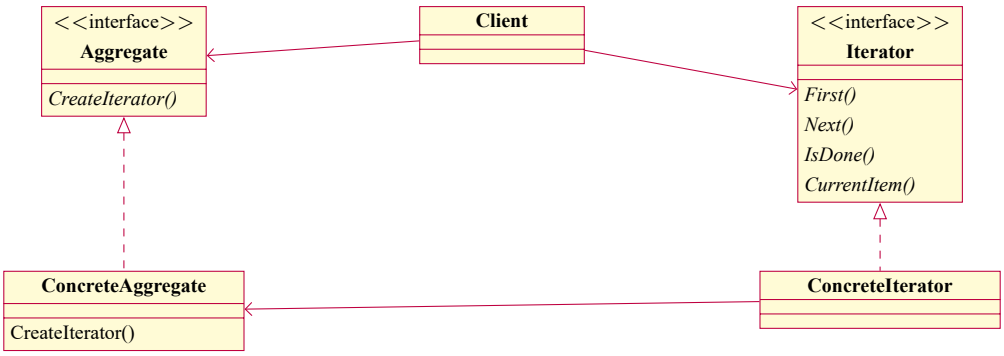
一个聚合对象，如列表 (list)，应该提供一种方法来让别人可以访问它的元素，而又不需要暴露该对象的内部表示。

简单来说，我们以某种规则 (算法) 获取某集合对象中的元素，这种算法就可以被单独写成一个迭代器对象，以获得对应集合中的元素。

适用性

- 访问一个聚合对象的内容而无需暴露它的内部表示。
- 支持对聚合对象的多种遍历。
- 为遍历不同的聚合结构提供一个统一的接口。

结构



参与者

- **Iterator**: 迭代器定义访问和遍历元素的接口。
- **ConcreteIterator**: 具体迭代器实现迭代器接口。对该聚合遍历时跟踪当前位置。
- **Aggregate**: 聚合定义创建相应迭代器对象的接口。
- **ConcreteAggregate**: 具体聚合实现创建相应迭代器的接口，该操作返回 ConcreteIterator 的一个适当的实例。

协作

- ConcreteIterator 跟踪聚合中的当前对象，并能够计算出待遍历的后继对象。

优缺点

- 支持以不同的方式遍历一个聚合
- 简化了聚合的接口
- 在同一个聚合上可以有多个遍历

例子

- Java: https://blog.csdn.net/zhengzhb/article/details/7610745
- Video: https://www.bilibili.com/video/BV1wK411V791

```python
# Reference:
    https://github.com/faif/python-patterns/blob/master/patterns/behavioral/iterator_alt.py

from __future__ import annotations


class NumberWords:

    _WORD_MAP = ("one", "two", "three", "four", "five")

    def __init__(self, start: int, stop: int) -> None:
        self.start = start
        self.stop = stop

    def __iter__(self) -> NumberWords: # this makes the class an Iterable
        return self

    def __next__(self) -> str: # this makes the class an Iterator
        if self.start > self.stop or self.start > len(self._WORD_MAP):
            raise StopIteration
        current = self.start
        self.start += 1
        return self._WORD_MAP[current - 1]


# Test the iterator

if __name__ == "__main__":
    # Counting to two...
    for number in NumberWords(start=1, stop=2):
        print(number)
    # Counting to five...
    for number in NumberWords(start=1, stop=5):
        print(number)
```

3.5 Mediator (中介者)

意图

用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

动机

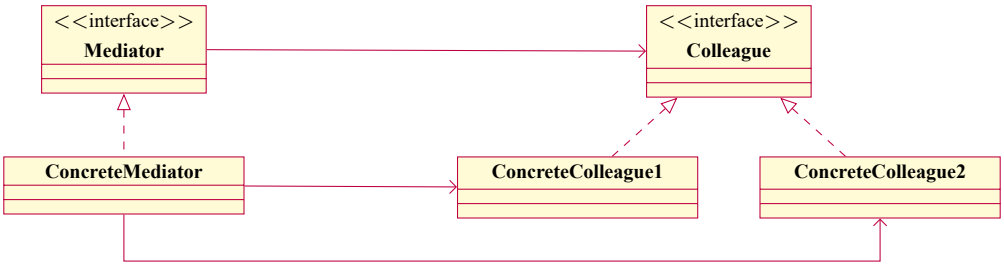
假设有一个机场，有很多飞机需要起飞降落。那么如何安排这些飞机的行动呢。飞机 A 如果要起飞，则需要通知剩下的所有飞机，并且一一确定在同一时间没有其他飞机会起飞，这个过程显然是非常繁琐的。

我们可以构建一个塔台，由塔台来和飞机交互替代飞机与飞机之间的交互。这样飞机的行动就统一由塔台者一个对象负责。

适用性

- 一组对象以定义良好切复杂的方式进行通信。产生的相互依赖关系结构混乱且难以理解。
- 一个对象引用其他很多对象并且直接与这些对象通信，导致难以复用该对象。
- 想定制一个分布在多个类中的行为，而又不想生成太多的子类。

结构



参与者

- **Mediator**: 定义一个接口用于与同事对象通信
- **ConcreteMediator**: 具体中介者通过各同事对象实现协作行为; 了解并维护它的各个同事。
- **Colleague**: 每一个同事类都需要知道它的中介者对象; 每一个同事对象在需要与其他同事通信的时，与它的终结者通信。

协作

同事向一个中介者对象发送和接收请求。中介者在各同事间适当地转发请求以实现协作行为。

优缺点

- **减少了子类的生成**: Mediator 将原本分布于多个对象间的行为集中在一起。改变这些行为只需要生成 Mediator 子类即可。
- **将各 Colleague 解耦**

- 简化了对象协议
- 对对象如何协作进行了抽象
- 使控制集中化

实现

- **忽略抽象的 Mediator 类**: 当各 Colleague 仅与一个 Mediator 一起工作时，没有必要定义一个抽象的 Mediator 类。
- **通信**: 当一个感兴趣的事件发生时，Colleague 必须与其 Mediator 通信。一种实现方法是使用 Observer 模式，将 Mediator 实现为一个 Observer，各 Colleague 作为 Subject，一旦其状态改变就发送给 Mediator。Mediator 做出的响应是将状态改变的结果传播给其他的 Colleague。

例子

- Java: `https://blog.csdn.net/qq_45515432/article/details/104041054`
- Video: `https://www.bilibili.com/video/BV1hK4y1L7zV`

```python
# Reference:
    https://github.com/faif/python-patterns/blob/master/patterns/behavioral/mediator.py

from __future__ import annotations


class ChatRoom:
    def display_message(self, user: User, message: str) -> None:
        print(f"[{user} says]: {message}")


class User:
    def __init__(self, name: str) -> None:
        self.name = name
        self.chat_room = ChatRoom()

    def say(self, message: str) -> None:
        self.chat_room.display_message(self, message)

    def __str__(self) -> str:
        return self.name


if __name__ == "__main__":
    molly = User('Molly')
    mark = User('Mark')
    ethan = User('Ethan')
    molly.say("Hi Team! Meeting at 3 PM today.")
    # [Molly says]: Hi Team! Meeting at 3 PM today.
    mark.say("Roger that!")
    # [Mark says]: Roger that!
    ethan.say("Alright.")
    # [Ethan says]: Alright.
```

3.6 Memento (备忘录)

意图

在不破坏封装性的前提下，捕捉一个对象的内部状态，并在该对象之外保存这个状态这样以后就可将该对象恢复到原先保存的状态。

别名: Token

动机

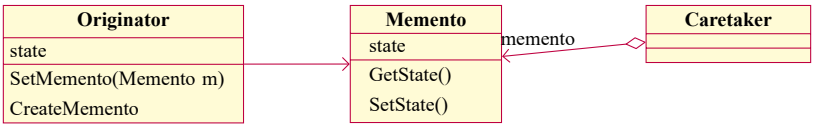
有时有必要保存一个对象的内部状态。为了允许用户取消不确定的操作或从错误中恢复过来，需要实现检查点和取消机制，而要实现这些机制，必须事先将状态信息保存在某处，这样才能将对象恢复到它们先前的状态。

一个备忘录 (memento) 是一个对象，它存储另一个对象在某个瞬间的内部状态，而后者称为备忘录的原发器。

适用性

- 必须保存一个对象在某个时刻的 (部分) 状态，这样以后需要时它才能恢复到先前的状态。
- 如果一个接口让其他对象直接得到这些状态，将会暴露对象的实现细节并破坏对象的封装性。

结构

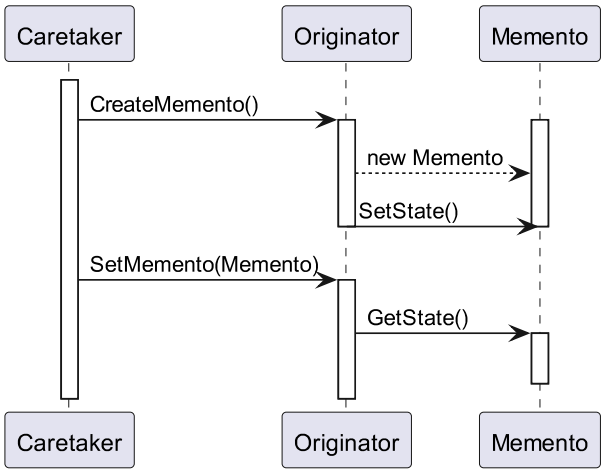


参与者

- **Memento**: 存储原发器对象的内部状态。原发起根据需要决定备忘录存放原发器的哪些内部状态；防止原发器以外的其他对象访问备忘录。
- **Originator**: 原发器创建一个备忘录，用以记录当前时刻它的内部状态；使用备忘录恢复内部状态。
- **Caretaker**: 负责保存好备忘录；不能对备忘录的内容进行操作或检查。

协作

- 管理者向原发器请求一个备忘录，保存一段时间后，将其送回给原发器，如下面的交互图所示:
- 备忘录是被动的。只有创建备忘录的原发器会对它的状态进行赋值和检索。



优缺点

- **保持封装边界**: 使用备忘录可以避免暴露一些只应由原发器管理却又必须存储在原发器之外的信息。
- **使用备忘录可能代价很高**: 如果用户频繁地创建备忘录和恢复原发器状态，且需要拷贝大量的信息，可能会导致非常大的开销。
- **维护备忘录的潜在代价**: 管理者负责删除它所维护的备忘录。然而，管理者不知道备忘录中有多少个状态。因此当存储备忘录时，一个本来很小的管理者可能会产生大量的存储开销。

例子

- Java: `https://blog.csdn.net/yuanchangliang/article/details/119211105`
- Video: `https://www.bilibili.com/video/BV1Sv41167o6`

```python
# Reference:
    https://github.com/faif/python-patterns/blob/master/patterns/behavioral/memento.py

from copy import copy, deepcopy
from typing import Callable, List


def memento(obj, deep=False):
    state = deepcopy(obj.__dict__) if deep else copy(obj.__dict__)

    def restore():
        obj.__dict__.clear()
        obj.__dict__.update(state)

    return restore


class Transaction:
    deep = False
    states: List[Callable[[], None]] = []

    def __init__(self, deep, *targets):
```

```
        self.deep = deep
        self.targets = targets
        self.commit()

    def commit(self):
        self.states = [memento(target, self.deep) for target in self.targets]

    def rollback(self):
        for a_state in self.states:
            a_state()


class Transactional:
    """Adds transactional semantics to methods. Methods decorated with
    @Transactional will rollback to entry-state upon exceptions.
    """
    def __init__(self, method):
        self.method = method

    def __get__(self, obj, T):
        """
        A decorator that makes a function transactional.
        :param method: The function to be decorated.
        """
        def transaction(*args, **kwargs):
            state = memento(obj)
            try:
                return self.method(obj, *args, **kwargs)
            except Exception as e:
                state()
                raise e

        return transaction


class NumObj:
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return f"<{self.__class__.__name__}: {self.value!r}>"

    def increment(self):
        self.value += 1

    @Transactional
    def do_stuff(self):
        self.value = "1111" # <- invalid value
        self.increment() # <- will fail and rollback


if __name__ == "__main__":
    num_obj = NumObj(-1)
```

```
print(num_obj)
# <NumObj: -1>
a_transaction = Transaction(True, num_obj)
try:
    for _ in range(3):
        num_obj.increment()
        print(num_obj)
    a_transaction.commit()
    print('-- committed')
    for _ in range(3):
        num_obj.increment()
        print(num_obj)
    num_obj.value += 'x' # will fail
    print(num_obj)
except Exception:
    a_transaction.rollback()
    print('-- rolled back')
# <NumObj: 0>
# <NumObj: 1>
# <NumObj: 2>
# -- committed
# <NumObj: 3>
# <NumObj: 4>
# <NumObj: 5>
# -- rolled back
print(num_obj)
# <NumObj: 2>
print('-- now doing stuff ...')
# -- now doing stuff ...
try:
    num_obj.do_stuff()
except Exception:
    print('-> doing stuff failed!')
    import sys
    import traceback
    traceback.print_exc(file=sys.stdout)
# -> doing stuff failed!
# Traceback (most recent call last):
...
# TypeError: ...str...int...
print(num_obj)
# <NumObj: 2>
```

3.7 Observer (观察者)

意图

定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

别名: 依赖 (dependent) 发布订阅 (publish-subscribe)

动机

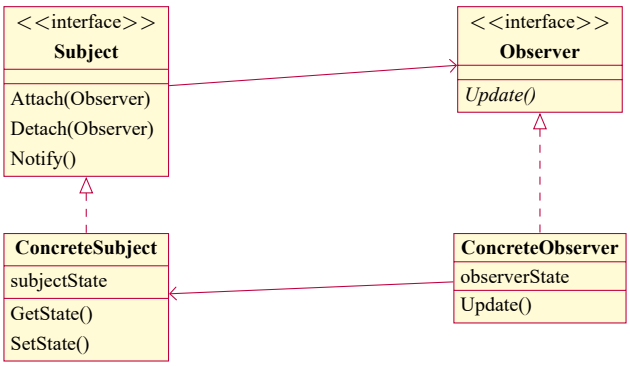
假设有一个人欠了很多钱，债主都盯着这个人的收入。如果这个人有钱了，债主会视该人的经济情况催债。

在这种情况下，这个人的经济情况一旦发生改动，多个债主就会做出相应的动作，这被称作观察者模式。其中关键对象是目标 (subject)，其他债主是观察者 (observer)。

适用性

- 一个抽象模型有两个方面，其中一个方面依赖于另一方面。将这两者封装在独立的对象中，以使它们可以各自独立地改变和复用。
- 对一个对象的改变需要同时改变其他对象，而不知道具体有多少对象有待改变。
- 一个对象必须通知其他对象，而它又不能假定其他对象是谁。换言之，你不希望这些对象是紧密耦合。

结构



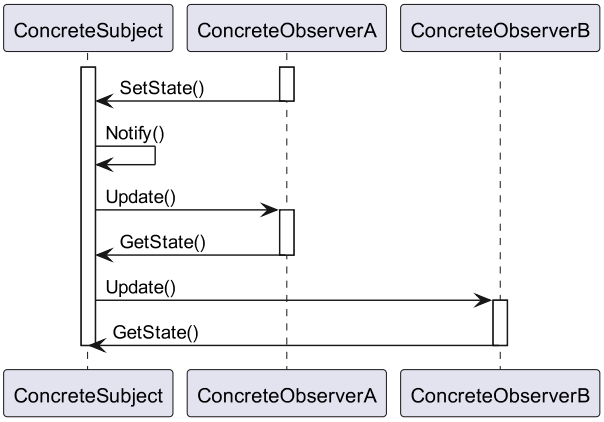
参与者

- **Subject**: 目标知道它的观察者。可以有任意多个观察者观察同一个目标; 提供注册和删除观察者对象的接口。
- **Observer**: 为哪些在目标发生改变时需要获得通知的对象定义一个更新接口。
- **ConcreteSubject**: 将有关状态存入各 ConcreteObserver 对象; 当它的状态发生改变时，向其各个观察者发出通知。
- **ConcreteObserver**: 维护一个指向 ConcreteSubject 对象的引用; 存储有关状态, 这些状态应与目标的状态保持一致; 实现 Observer 的更新接口，以使自身状态与目标的状态保持一致。

协作

• 当 ConcreteSubject 发生任何可能导致其观察者与本身状态不一致的改变时，它将通知它的各个观察者。

• 在得到一个具体目标的改变通知后，ConcreteObserver 对象可向目标对象查询信息。ConcreteObserver 使用这些信息使它的状态与目标对象的状态一致。



注意发出改变请求的 Observer 对象并不立即更新，而是将其推迟到它从目标得到一个通知之后。Notify 不总是由目标对象调用，它也可被一个观察者或其他对象调用。

优缺点

• **目标和观察者间的抽象耦合**: 一个目标所知道的仅仅是他有一系列观察者，每个都符合抽象的 Observer 类的简单接口。目标不知道任何一个观察者术语哪个具体的类。这样目标和抽象者之间的耦合是抽象的和最小的。

• **支持广播通信**: 不像通产的请求，目标发送的通知不需要指定它的接收者。通知被自动广播给所有已向该目标对象登记的对象。

• **意外的更新**: 由于一个观察者并不知道其他观察者的存在，它可能对改变目标的最终代价一无所知。

例子

• Java: `https://blog.csdn.net/itachi85/article/details/50773358`

• Video: `https://www.bilibili.com/video/BV1vg4y1v7V4`

```python
# Reference: http://code.activestate.com/recipes/131499-observer-pattern/

from __future__ import annotations

from contextlib import suppress
from typing import Protocol


# define a generic observer type
class Observer(Protocol):
    def update(self, subject: Subject) -> None:
        pass


```

```python
class Subject:
    def __init__(self) -> None:
        self._observers: list[Observer] = []

    def attach(self, observer: Observer) -> None:
        if observer not in self._observers:
            self._observers.append(observer)

    def detach(self, observer: Observer) -> None:
        with suppress(ValueError):
            self._observers.remove(observer)

    def notify(self, modifier: Observer | None = None) -> None:
        for observer in self._observers:
            if modifier != observer:
                observer.update(self)


class Data(Subject):
    def __init__(self, name: str = "") -> None:
        super().__init__()
        self.name = name
        self._data = 0

    @property
    def data(self) -> int:
        return self._data

    @data.setter
    def data(self, value: int) -> None:
        self._data = value
        self.notify()


class HexViewer:
    def update(self, subject: Data) -> None:
        print(f"HexViewer: Subject {subject.name} has data 0x{subject.data:x}")


class DecimalViewer:
    def update(self, subject: Data) -> None:
        print(f"DecimalViewer: Subject {subject.name} has data {subject.data}")


if __name__ == "__main__":
    data1 = Data('Data 1')
    data2 = Data('Data 2')
    view1 = DecimalViewer()
    view2 = HexViewer()
    data1.attach(view1)
    data1.attach(view2)
    data2.attach(view2)
    data2.attach(view1)
```

```
68
69      data1.data = 10
70      # DecimalViewer: Subject Data 1 has data 10
71      # HexViewer: Subject Data 1 has data 0xa
72
73      data2.data = 15
74      # HexViewer: Subject Data 2 has data 0xf
75      # DecimalViewer: Subject Data 2 has data 15
```

3.8 State (状态)

意图

允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它的类。

别名: 状态对象。

动机

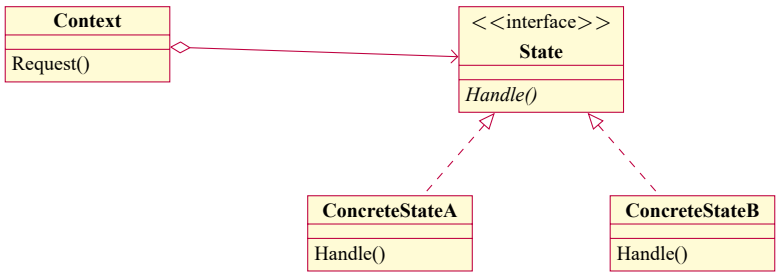
假设有一名大学生要去上课，正常情况下该学生专心致志听课。如果该学生前一天没有睡好，则上课效率较低。如果该学生分手了，则完全没有心思上课。

在上面例子中，学生前一天的状态会极大地影响上课事件。

适用性

- 一个对象的行为取决于它的状态，并且它必须在运行时根据状态改变它的行为。
- 一个操作中含有庞大的多分支条件语句，且这些分支依赖于该对象的状态。这个状态通常用一个或多个枚举常量表示。通常，有多个操作包含这一相同的条件结构。State 模式将每一个条件放入一个独立的类中。这使得你可以根据对象自身的情况将对象的状态作为一个对象。

结构



参与者

- **Context**: 定义客户感兴趣的接口; 维护一个 ConcreteState 子类的实例，这个实例定义当前状态。
- **State**: 定义一个接口以封装与 Context 的一个特定状态相关的行为。
- **ConcreteState**: 每一个子类实现一个与 Context 的一个状态相关的行为。

协作

- Context 将与状态相关的请求委托给当前的 ConcreteState 对象处理。
- Context 可将自身作为一个参数传递给处理该请求的状态对象。这使得状态对象必要时可访问 Context。
- Context 是客户使用的主要接口。客户可用状态对象来配置一个 Context，一旦一个 Context 配置完毕，它的客户不再需要直接与状态对象打交道。
- Context 或 ConcreteState 子类都可决定哪个状态是另一个的后继者，以及是在何种条件下进行状态转换。

优缺点

- 将特定状态相关的行为局部化，并且将不同状态的行为分割开来
- **State** 对象可被共享

例子

- Java: https://blog.csdn.net/weixin_39397471/article/details/82843404
- Video: https://www.bilibili.com/video/BV1oi4y1g7Nn

```python
# Reference: https://github.com/faif/python-patterns/blob/master/patterns/behavioral/state.py


class State:
    """Base state. This is to share functionality"""
    def scan(self):
        """Scan the dial to the next station"""
        self.pos += 1
        if self.pos == len(self.stations):
            self.pos = 0
        print(f"Scanning... Station is {self.stations[self.pos]} {self.name}")


class AmState(State):
    def __init__(self, radio):
        self.radio = radio
        self.stations = ["1250", "1380", "1510"]
        self.pos = 0
        self.name = "AM"

    def toggle_amfm(self):
        print("Switching to FM")
        self.radio.state = self.radio.fmstate


class FmState(State):
    def __init__(self, radio):
        self.radio = radio
        self.stations = ["81.3", "89.1", "103.9"]
        self.pos = 0
        self.name = "FM"

    def toggle_amfm(self):
        print("Switching to AM")
        self.radio.state = self.radio.amstate


class Radio:
    """A radio.    It has a scan button, and an AM/FM toggle switch."""
    def __init__(self):
        """We have an AM state and an FM state"""
        self.amstate = AmState(self)
        self.fmstate = FmState(self)
        self.state = self.amstate

```

```
    def toggle_amfm(self):
        self.state.toggle_amfm()

    def scan(self):
        self.state.scan()


if __name__ == "__main__":
    radio = Radio()
    actions = [radio.scan] * 2 + [radio.toggle_amfm] + [radio.scan] * 2
    actions *= 2
    for action in actions:
        action()
```

3.9 Strategy (策略)

意图

定义一系列的算法，把它们一个个封装起来的，并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。

别名: policy

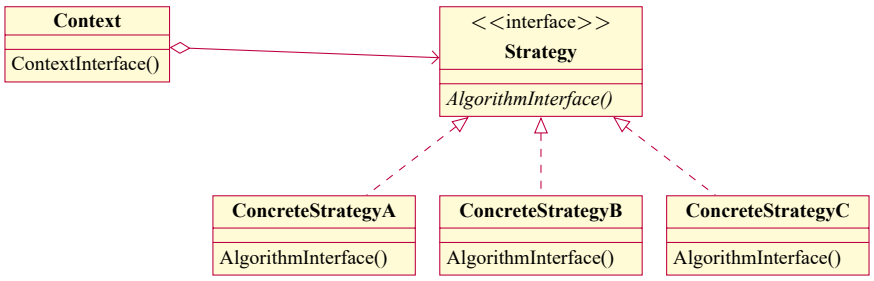
动机

在智能洗衣机上，我们可以选择快洗，慢洗，脱水，羊毛洗等等模式。这些模式即可以被称为做洗衣机的策略。

适用性

- 许多相关的类仅仅是行为有异。“策略” 提供了一种用多个行为中的一个行为来配置一个类的方法。
- 需要使用一个算法的不同变体。例如，你可能会定义一些反应不同的空间/时间权衡的算法。当这些变体实现为一个算法的类层次时，可以使用策略模式。
- 算法使用客户不应该知道的数据。可使用策略模式以避免暴露复杂的，与算法相关的数据结构。
- 一个类定义了多种行为，并且这些行为在这个类的操作中以多个条件语句的形式出现。将相关的条件分支移入它们各自的 Strategy 类中以替换这些条件语句。

结构



参与者

- **Strategy**: 定义所有支持的算法的公开接口。Context 使用这个接口来调用其 ConcreteStrategy 定义的算法。
- **ConcreteStrategy**: 以 Strategy 接口实现某具体算法。
- **Context**: 用一个 ConcreteStrategy 对象来配置; 维护一个对 Strategy 对象的引用; 可定义一个接口来让 Strategy 访问它的数据。

优缺点

- **策略可复用**
- **消除了一些条件语句**: 委托给了 Strategy 对象
- **增加了对象的数目**: Strategy 结合 Flyweight 可以有效地缓解这个问题

例子

- Java: https://blog.csdn.net/youanyyou/article/details/116931663
- Video: https://www.bilibili.com/video/BV1oi4y1g7Nn

```python
# Reference:
    https://github.com/faif/python-patterns/blob/master/patterns/behavioral/strategy.py

from __future__ import annotations

from typing import Callable


class DiscountStrategyValidator: # Descriptor class for check perform
    @staticmethod
    def validate(obj: Order, value: Callable) -> bool:
        try:
            if obj.price - value(obj) < 0:
                raise ValueError(
                    f"Discount cannot be applied due to negative price resulting.
                        {value.__name__}"
                )
        except ValueError as ex:
            print(str(ex))
            return False
        else:
            return True

    def __set_name__(self, owner, name: str) -> None:
        self.private_name = f"_{name}"

    def __set__(self, obj: Order, value: Callable = None) -> None:
        if value and self.validate(obj, value):
            setattr(obj, self.private_name, value)
        else:
            setattr(obj, self.private_name, None)

    def __get__(self, obj: object, objtype: type = None):
        return getattr(obj, self.private_name)


class Order:
    discount_strategy = DiscountStrategyValidator()

    def __init__(self,
                 price: float,
                 discount_strategy: Callable = None) -> None:
        self.price: float = price
        self.discount_strategy = discount_strategy

    def apply_discount(self) -> float:
        if self.discount_strategy:
            discount = self.discount_strategy(self)
        else:
```

```
            discount = 0

        return self.price - discount

    def __repr__(self) -> str:
        return f"<Order price: {self.price} with discount strategy:
            {getattr(self.discount_strategy,'__name__',None)}>"


def ten_percent_discount(order: Order) -> float:
    return order.price * 0.10


def on_sale_discount(order: Order) -> float:
    return order.price * 0.25 + 20


if __name__ == "__main__":
    order = Order(100, discount_strategy=ten_percent_discount)
    print(order)
    # <Order price: 100 with discount strategy: ten_percent_discount>
    print(order.apply_discount())
    # 90.0
    order = Order(100, discount_strategy=on_sale_discount)
    print(order)
    # <Order price: 100 with discount strategy: on_sale_discount>
    print(order.apply_discount())
    # 55.0
    order = Order(10, discount_strategy=on_sale_discount)
    # Discount cannot be applied due to negative price resulting. on_sale_discount
    print(order)
    # <Order price: 10 with discount strategy: None>
```

3.10 Template Method (模板方法)

意图

定义一个操作中的算法的骨架，而将一些步骤延迟到了子类中。TemplateMethod 使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

动机

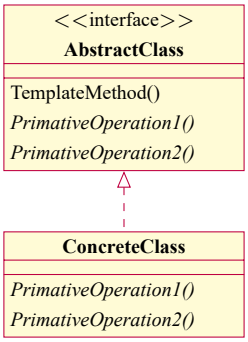
当我们需要做一道菜的时候，有以下几个步骤: 放油 -> 爆香 -> 放菜 -> 放盐 -> 翻炒 -> 放调味料 -> 关火 -> 起锅。这是做菜的通用步骤，但不同的菜每个步骤不尽相同，例如油盐放的多少，调味料的选择。

这种通用步骤就被称为模板方法，而具体实现需要自己负责。

适用性

- 一次性实现一个算法的不变部分，并将可变的行为留给子类来实现。
- 各子类中公共的行为应被提取出来并集中到一个公共父类中以避免代码重复。
- 控制子类扩展。模板方法只在特定点调用钩子操作，这样只允许在这些点进行扩展。

结构



参与者

- **AbstractClass**: 定义抽象的原语操作 (primitive operation)，具体的子类将重定义它们以实现一个算法的各步骤; 实现一个模板方法，定义一个算法的骨架。该模板方法不仅调用原语操作，也调用定义在 AbstractClass 或其他对象中的操作。
- **ConcreteClass**: 实现原语操作以完成算法中与特定子类相关的步骤。

协作

- ConcreteClass 靠 AbstractClass 来实现算法中不变的步骤。
- TemplateMethod 中以规定顺序调用各原语操作。

效果

- **钩子操作**: 提供了缺省的行为，子类可以在必要时进行扩展。钩子操作在缺省情况下通常是空操作。

实现

- **尽量减少原语操作**: 需要重定义的操作越多，代码就越冗长。

- **命名约定**: 可以给被重定义的操作名字加上一个前缀 (如 Do-) 以识别他们。

例子

- Java: https://blog.csdn.net/qq_26775359/article/details/109603752
- Video: https://www.bilibili.com/video/BV1kk4y117j5

```python
# Reference:
    https://github.com/faif/python-patterns/blob/master/patterns/behavioral/template.py


def get_text() -> str:
    return "plain-text"


def get_pdf() -> str:
    return "pdf"


def get_csv() -> str:
    return "csv"


def convert_to_text(data: str) -> str:
    print("[CONVERT]")
    return f"{data} as text"


def saver() -> None:
    print("[SAVE]")


def template_function(getter, converter=False, to_save=False) -> None:
    data = getter()
    print(f"Got `{data}`")
    if len(data) <= 3 and converter:
        data = converter(data)
    else:
        print("Skip conversion")
    if to_save:
        saver()
    print(f"`{data}` was processed")


if __name__ == "__main__":
    template_function(get_text, to_save=True)
    # Got `plain-text`
    # Skip conversion
    # [SAVE]
    # `plain-text` was processed
    template_function(get_pdf, converter=convert_to_text)
    # Got `pdf`
    # [CONVERT]
```

```
    # `pdf as text` was processed
    template_function(get_csv, to_save=True)
    # Got `csv`
    # Skip conversion
    # [SAVE]
    # `csv` was processed
```

3.11 Visitor (访问者)

意图

表示一个作用于某对象结构中的个元素的操作。它使你可以再不改变各元素的类的前提下定义作用于这些元素的新操作。

动机

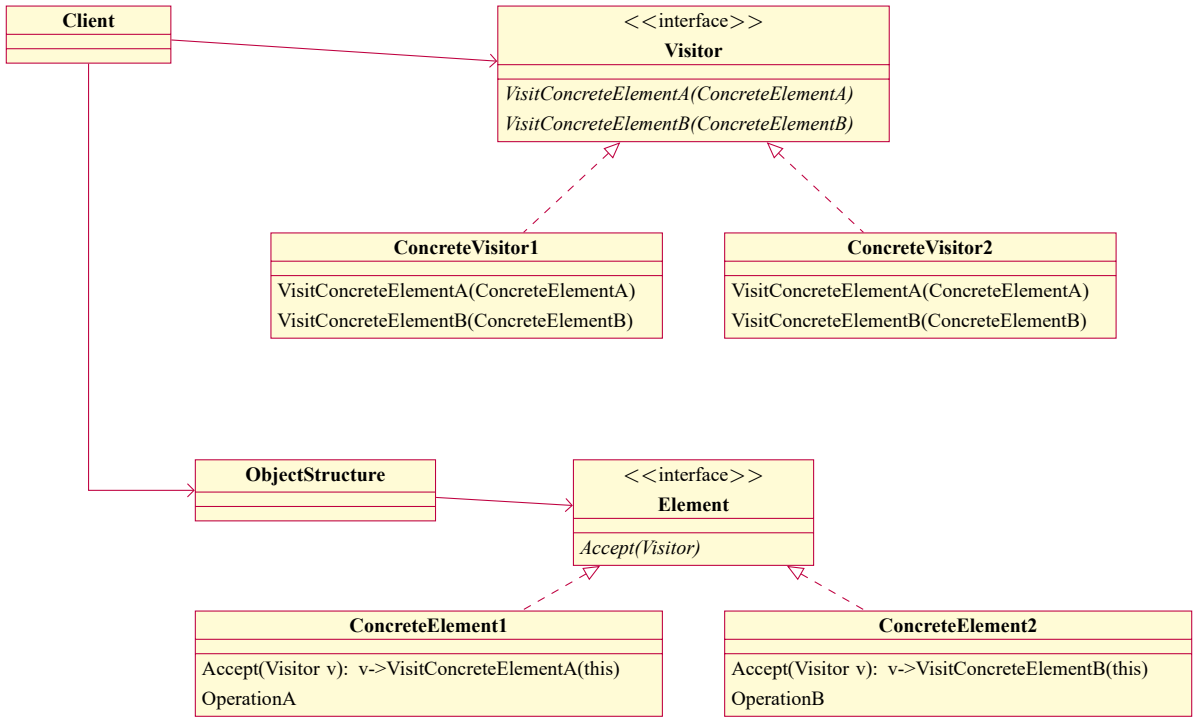
考虑这样一种情形，现在有机器人一代，它具有一些简单的功能。由于客户不满意一代机器人的功能，公司程序员在不改变硬件芯片的前提下，对机器人系统进行了更新，将所有机器人返厂并推出了二代机器人。

这种不改变原有机器人硬件，系统可以更新的模式；就可以视作访问者模式。

适用性

- 一个对象包括很多类的对象，它们有不同的接口，而你相对这些对象实施一些依赖于具体类的操作。
- 需要对一个对象结构中的对象进行很多不同并且不相关的操作，而你想避免让这些操作 “污染” 这些对象的类。Visitor 使得你可以将相关的操作集中起来定义在一个类中。
- 定义对象结构的类很少改变，但经常需要在此结构上定义新的操作。改变对象结构类需要重新定义所有访问者的接口，这可能需要很大的代价。如果对象结构类经常改变，那么可能还是在这些类中定义这些操作比较好。

结构



参与者

- **Visitor**: 为该对象结构中 ConcreteElement 的每一个类声明一个 Visit 操作。改操作的名字和特征标识了发送 Visit 请求给该访问者的类。这使得访问者可以确定正被访问元

素的具体的类。这样该访问者就可以通过该元素的特定接口直接访问它。

- **ConcreteVisitor**: 实现每个由 Visitor 声明的操作。
- **Element**: 定义一个 Accept 操作，它以一个访问者为参数。
- **ConcreteElement**: 实现 Accept 操作，该操作以一个访问者为参数。
- **ObjectStructure**: 提供一个高层的接口以允许该访问者为参数。

协作

一个使用 Visitor 模式的客户必须创建一个 ConcreteVisitor 对象，然后便利该对象结构，并用该访问者访问每一个元素。

当一个元素被访问时，它调用对应于它的类的 Visitor 操作。如果有必要，该元素将自身作为这个操作的一个参数，以便该访问者访问它的状态。

优缺点

- **易于增加新操作**
- **访问者集中相关的操作而分离无关的操作**
- **增加新的 ConcreteElement 类很困难**: 每添加一个 Element 子类就需要创建一个对应的 ConcreteVisitor 类。
- **破坏封装**: 访问者方法假定 ConcreteElement 接口的功能足够强，足以让访问者进行工作，但往往需要提供访问元素内部状态的公共操作，这可能会破坏它的封装性。

例子

- Java: `https://blog.csdn.net/zhengzhb/article/details/7489639`
- Video: `https://www.bilibili.com/video/BV1nt4y1k7j5`

```python
# Reference:
    https://github.com/faif/python-patterns/blob/master/patterns/behavioral/visitor.py


class Node:
    pass


class A(Node):
    pass


class B(Node):
    pass


class C(A, B):
    pass


class Visitor:
    def visit(self, node, *args, **kwargs):
        meth = None
```

```
        for cls in node.__class__.__mro__:
            meth_name = f"visit_{cls.__name__}"
            meth = getattr(self, meth_name, None)
            if meth:
                break
        if not meth:
            meth = self.generic_visit
        return meth(node, *args, **kwargs)

    def generic_visit(self, node, *args, **kwargs):
        print(f"generic_visit {node.__class__.__name__}")

    def visit_B(self, node, *args, **kwargs):
        print(f"visit_B {node.__class__.__name__}")


if __name__ == "__main__":
    a, b, c = A(), B(), C()
    visitor = Visitor()
    visitor.visit(a) # generic_visit A
    visitor.visit(b) # visit_B B
    visitor.visit(c) # visit_B C
```