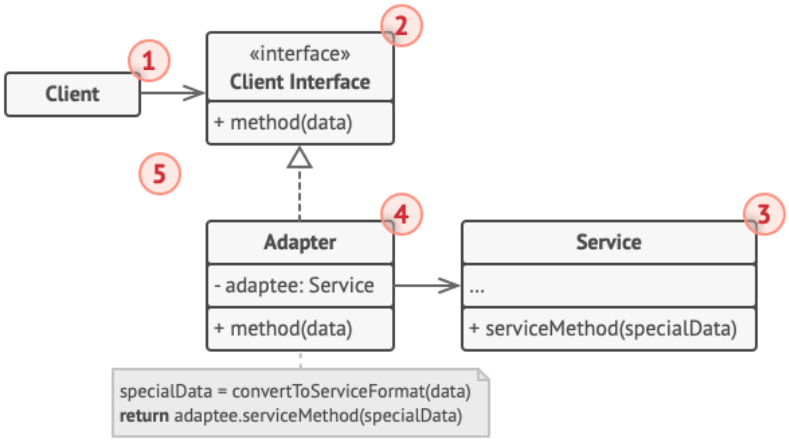
结构型模式

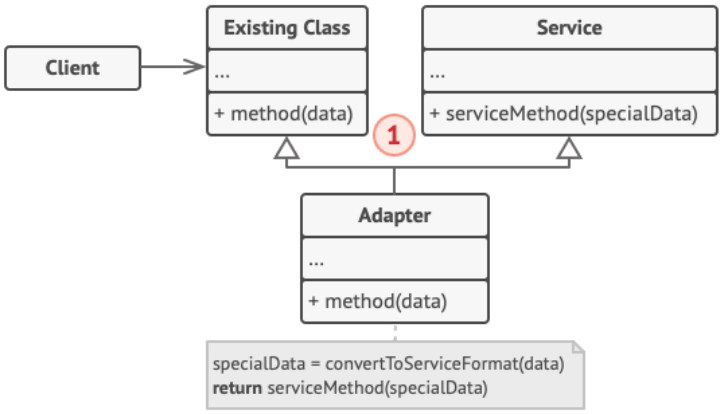
Adapter 适配器

适配器模式是一种结构型设计模式， 它能使接口不兼容的对象能够相互合作。

The adapter pattern is a structural design pattern that enables objects with incompatible interfaces to cooperate.



1. **客户端** （Client） 是包含当前程序业务逻辑的类。
2. **客户端接口** （Client Interface） 描述了其他类与客户端代码合作时必须遵循的协议。
3. **服务** （Service） 中有一些功能类 （通常来自第三方或遗留系统）。 客户端与其接口不兼容， 因此无法直接调用其功能。
4. **适配器** （Adapter） 是一个可以同时与客户端和服务交互的类： 它在实现客户端接口的同时封装了服务对象。 适配器接受客户端通过适配器接口发起的调用， 并将其转换为适用于被封装服务对象的调用。
5. 客户端代码只需通过接口与适配器交互即可， 无需与具体的适配器类耦合。 因此， 你可以向程序中添加新类型的适配器而无需修改已有代码。 这在服务类的接口被更改或替换时很有用： 你无需修改客户端代码就可以创建新的适配器类。



场景

- 当你希望使用某个类， 但是其接口与其他代码不兼容时， 可以使用适配器类。
- 如果您需要复用这样一些类， 他们处于同一个继承体系， 并且他们又有了额外的一些共同的方法， 但是这些共同的方法不是所有在这一继承体系中的子类所具有的共性。

优缺点

- 单一职责原则你可以将接口或数据转换代码从程序主要业务逻辑中分离。
- 开闭原则。 只要客户端代码通过客户端接口与适配器进行交互， 你就能在不修改现有客户端代码的情况下在程序中添加新类型的适配器。

Dis

- 代码整体复杂度增加， 因为你需要新增一系列接口和类。 有时直接更改服务类使其与其他代码兼容会更简单。

关系

- 桥接模式通常会于开发前期进行设计， 使你能够将程序的各个部分独立开来以便开发。 另一方面， 适配器模式通常在已有程序中使用， 让相互不兼容的类能很好地合作。
- 适配器可以对已有对象的接口进行修改， 装饰模式则能在不改变对象接口的前提下强化对象功能。 此外， 装饰还支持递归组合， 适配器则无法实现。
- 适配器能为被封装对象提供不同的接口， 代理模式能为对象提供相同的接口， 装饰则能为对象提供加强的接口。
- 外观模式为现有对象定义了一个新接口， 适配器则会试图运用已有的接口。 适配器通常只封装一个对象， 外观通常会作用于整个对象子系统上。
- 桥接、 状态模式和策略模式 （在某种程度上包括适配器） 模式的接口非常相似。 实际上， 它们都基于组合模式——即将工作委派给其他对象， 不过也各自解决了不同的问题。 模式并不只是以特定方式组织代码的配方， 你还可以使用它们来和其他开发者讨论模式所解决的问题。

Code

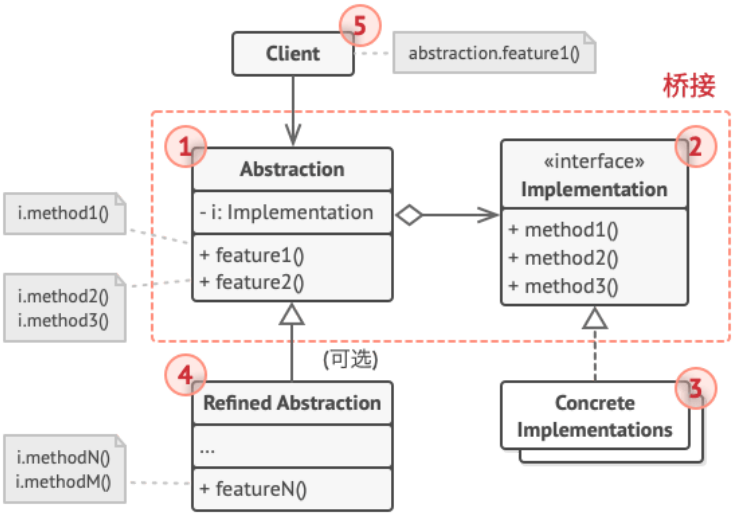
```
class Target:
    """
    The Target defines the domain-specific interface used by the client code.
    """

    def request(self) -> str:
        return "Target: The default target's behavior."


class Adaptee:
    """
    The Adaptee contains some useful behavior, but its interface is incompatible
    with the existing client code. The Adaptee needs some adaptation before the
    client code can use it.
    """

    def specific_request(self) -> str:
        return ".eetpadA eht fo roivaheb laicepS"


class Adapter(Target, Adaptee):
    """
    The Adapter makes the Adaptee's interface compatible with the Target's
```

```
    interface via multiple inheritance.
    """

    def request(self) -> str:
        return f"Adapter: (TRANSLATED) {self.specific_request()[::-1]}"


def client_code(target: "Target") -> None:
    """
    The client code supports all classes that follow the Target interface.
    """

    print(target.request(), end="")


if __name__ == "__main__":
    print("Client: I can work just fine with the Target objects:")
    target = Target()
    client_code(target)
    print("\n")

    adaptee = Adaptee()
    print("Client: The Adaptee class has a weird interface. "
          "See, I don't understand it:")
    print(f"Adaptee: {adaptee.specific_request()}", end="\n\n")

    print("Client: But I can work with it via the Adapter:")
    adapter = Adapter()
    client_code(adapter)
```

Bridge 桥接

桥接模式是一种结构型设计模式， 可将一个大类或一系列紧密相关的类拆分为抽象和实现两个独立的层次结构， 从而能在开发时分别使用。

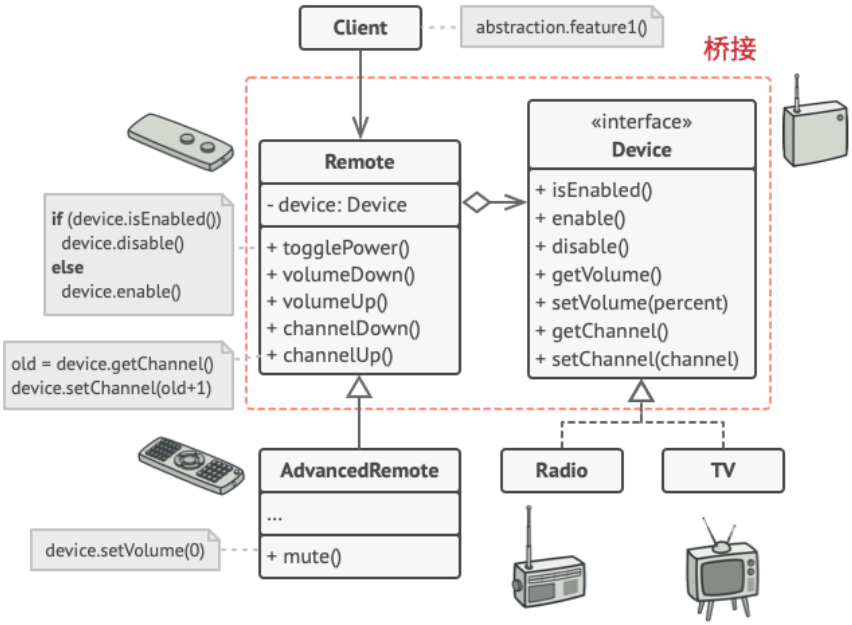
The bridging pattern is a structural design pattern that splits a large class or series of closely related classes into two separate hierarchies of abstraction and implementation that can be used separately at development time.



1. **抽象部分** （Abstraction） 提供高层控制逻辑， 依赖于完成底层实际工作的实现对象。
2. **实现部分** （Implementation） 为所有具体实现声明通用接口。 抽象部分仅能通过在这里声明的方法与实现对象交互。

抽象部分可以列出和实现部分一样的方法， 但是抽象部分通常声明一些复杂行为， 这些行为依赖于多种由实现部分声明的原语操作。

3. **具体实现** （Concrete Implementations） 中包括特定于平台的代码。
4. **精确抽象** （Refined Abstraction） 提供控制逻辑的变体。 与其父类一样， 它们通过通用实现接口与不同的实现进行交互。
5. 通常情况下， **客户端** （Client） 仅关心如何与抽象部分合作。 但是， 客户端需要将抽象对象与一个实现对象连接起来。



场景

- 如果你想要拆分或重组一个具有多重功能的庞杂类 （例如能与多个数据库服务器进行交互的类），可以使用桥接模式。
- 如果你希望在几个独立维度上扩展一个类， 可使用该模式。
- 如果你需要在运行时切换不同实现方法， 可使用桥接模式。

优缺点

- 你可以创建与平台无关的类和程序。
- 客户端代码仅与高层抽象部分进行互动， 不会接触到平台的详细信息。
- 开闭原则。 你可以新增抽象部分和实现部分， 且它们之间不会相互影响。
- 单一职责原则。 抽象部分专注于处理高层逻辑， 实现部分处理平台细节。

D：

对高内聚的类使用该模式可能会让代码更加复杂。

关系

What is the difference between the bridge and adapter patterns?

- 桥接模式通常会于开发前期进行设计， 使你能够将程序的各个部分独立开来以便开发。 另一方面， 适配器模式通常在已有程序中使用， 让相互不兼容的类能很好地合作。

Code ⭐

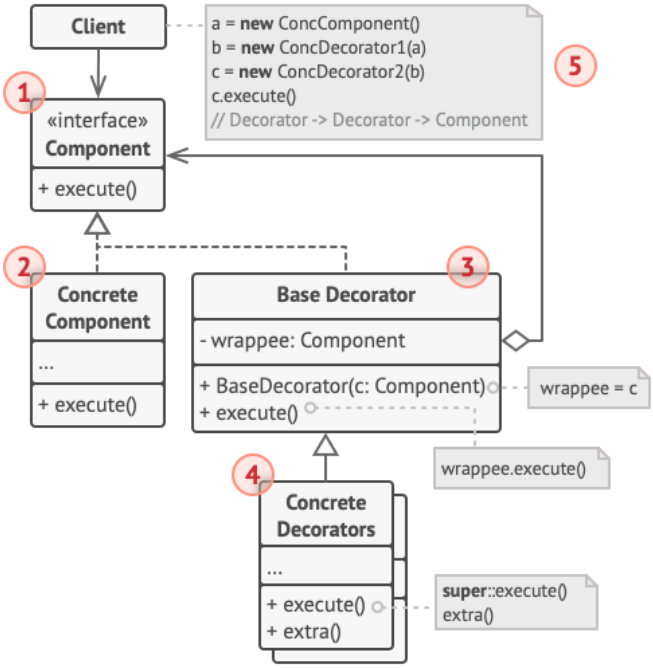
```
from __future__ import annotations
from abc import ABC, abstractmethod


class Phone:
    def __init__(self, implementation: Implementation) -> None:
        self.implementation = implementation

    def run(self) -> str:
        return (f"Abstraction Phone: Base operation with:\n"
                f"{self.implementation.run()}")


class OPPO(Phone):
    def run(self) -> str:
        return (f"OPPO: Extended operation with OPPO:\n"
                f"{self.implementation.run()}")


class ViVo(Phone):
    def run(self) -> str:
        return (f"ViVo: Extended operation with ViVo:\n"
                f"{self.implementation.run()}")


class Implementation(ABC):

    @abstractmethod
    def run(self) -> str:
        pass


class SoftWare(Implementation):
    def run(self) -> str:
        return "SoftWare: Here's the result"


class Appstore(Implementation):
    def run(self) -> str:
        return "Appstore: Here's the result"


class Camera(Implementation):
    def run(self) -> str:
        return "Camera: Here's the result"


if __name__ == "__main__":
    implementation = SoftWare()
    abstraction = Phone(implementation)
    print(abstraction.run())
    implementation = Camera()
    abstraction = ViVo(implementation)
    print(abstraction.run())
    abstraction = OPPO(implementation)
```

```
    print(abstraction.run())
```

Decorator 装饰器

装饰模式是一种结构型设计模式， 允许你通过将对象放入包含行为的特殊封装对象中来为原对象绑定新的行为。

Decorator is a structural design pattern that allows you to bind new behavior to an object by putting it into a special wrapper that contains behavior.



1. **部件** （Component） 声明封装器和被封装对象的公用接口。
2. **具体部件** （Concrete Component） 类是被封装对象所属的类。 它定义了基础行为， 但装饰类可以改变这些行为。
3. **基础装饰** （Base Decorator） 类拥有一个指向被封装对象的引用成员变量。 该变量的类型应当被声明为通用部件接口， 这样它就可以引用具体的部件和装饰。 装饰基类会将所有操作委派给被封装的对象。
4. **具体装饰类** （Concrete Decorators） 定义了可动态添加到部件的额外行为。 具体装饰类会重写装饰基类的方法， 并在调用父类方法之前或之后进行额外的行为。
5. **客户端** （Client） 可以使用多层装饰来封装部件， 只要它能使用通用接口与所有对象互动即可。

场景

- 如果你希望在无需修改代码的情况下即可使用对象， 且希望在运行时为对象新增额外的行为， 可以使用装饰模式。
- 如果用继承来扩展对象行为的方案难以实现或者根本不可行， 你可以使用该模式。

优缺点

A:

- 你无需创建新子类即可扩展对象的行为。
- 你可以在运行时添加或删除对象的功能。
- 你可以用多个装饰封装对象来组合几种行为。
- 单一职责原则。 你可以将实现了许多不同行为的一个大类拆分为多个较小的类。

D:

- 在封装器栈中删除特定封装器比较困难。
- 实现行为不受装饰栈顺序影响的装饰比较困难。
- 各层的初始化配置代码看上去可能会很糟糕。

关系

- 适配器模式可以对已有对象的接口进行修改， 装饰模式则能在不改变对象接口的前提下强化对象功能。 此外， 装饰还支持递归组合， 适配器则无法实现。
- 适配器能为被封装对象提供不同的接口， 代理模式能为对象提供相同的接口， 装饰则能为对象提供加强的接口。
- 装饰和代理有着相似的结构， 但是其意图却非常不同。 这两个模式的构建都基于组合原则， 也就是说一个对象应该将部分工作委派给另一个对象。 两者之间的不同之处在于代理通常自行管理其服务对象的生命周期， 而装饰的生成则总是由客户端进行控制。

Code ★

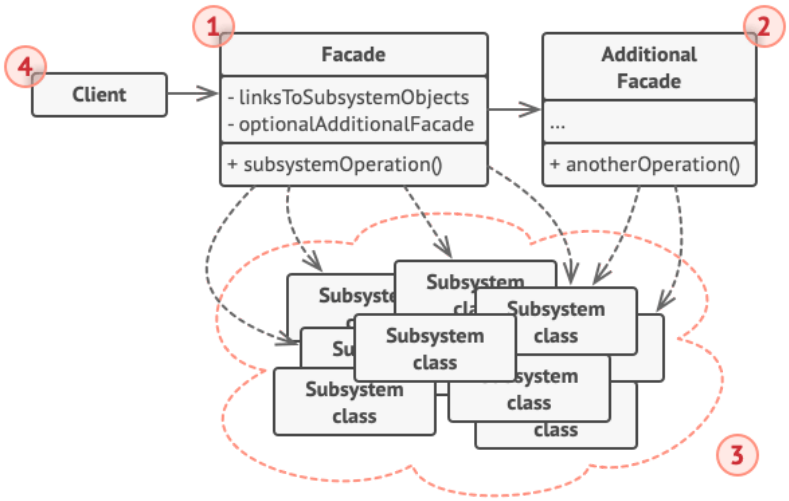
```
class Component():
    def operation(self) -> str:
        pass


class ConcreteComponent(Component):
    def operation(self) -> str:
        return "ConcreteComponent"


class Decorator(Component):
    _component: Component = None

    def __init__(self, component: Component) -> None:
        self._component = component

    @property
    def component(self) -> str:
        return self._component

    def operation(self) -> str:
        return self._component.operation()


class ConcreteDecoratorA(Decorator):
    def operation(self) -> str:
        return f"ConcreteDecoratorA({self.component.operation()})"


class ConcreteDecoratorB(Decorator):
    def operation(self) -> str:
        return f"ConcreteDecoratorB({self.component.operation()})"


def client_code(component: Component) -> None:

    # ...

    print(f"RESULT: {component.operation()}", end="")
```

```
    # ...


if __name__ == "__main__":
    simple = ConcreteComponent()
    print("Client: I've got a simple component:")
    client_code(simple)
    print("\n")

    decorator1 = ConcreteDecoratorA(simple)
    decorator2 = ConcreteDecoratorB(decorator1)
    print("Client: Now I've got a decorated component:")
    client_code(decorator2)
```

Facade 外观

外观模式是一种结构型设计模式， 能为程序库、 框架或其他复杂类提供一个简单的接口。

The facade pattern is a structural design pattern that provides a simple interface to libraries, frameworks, or other complex classes.



1. **外观** （Facade） 提供了一种访问特定子系统功能的便捷方式， 其了解如何重定向客户端请求， 知晓如何操作一切活动部件。
2. 创建**附加外观** （Additional Facade） 类可以避免多种不相关的功能污染单一外观， 使其变成又一个复杂结构。 客户端和其他外观都可使用附加外观。
3. **复杂子系统** （Complex Subsystem） 由数十个不同对象构成。 如果要用这些对象完成有意义的工作， 你必须深入了解子系统的实现细节， 比如按照正确顺序初始化对象和为其提供正确格式的数据。

   子系统类不会意识到外观的存在， 它们在系统内运作并且相互之间可直接进行交互。
4. **客户端** （Client） 使用外观代替对子系统对象的直接调用。

场景

- 如果你需要一个指向复杂子系统的直接接口， 且该接口的功能有限， 则可以使用外观模式。
- 如果需要将子系统组织为多层结构， 可以使用外观。

优缺点

- 你可以让自己的代码独立于复杂子系统。
- 外观可能成为与程序中所有类都耦合的上帝对象。God object

关系

- 外观与代理模式的相似之处在于它们都缓存了一个复杂实体并自行对其进行初始化。 代理与其服务对象遵循同一接口， 使得自己和服务对象可以互换， 在这一点上它与外观不同
- 外观模式为现有对象定义了一个新接口， 适配器模式则会试图运用已有的接口。 适配器通常只封装一个对象， 外观通常会作用于整个对象子系统上。
- 当只需对客户端代码隐藏子系统创建对象的方式时， 你可以使用抽象工厂模式来代替外观。

Code

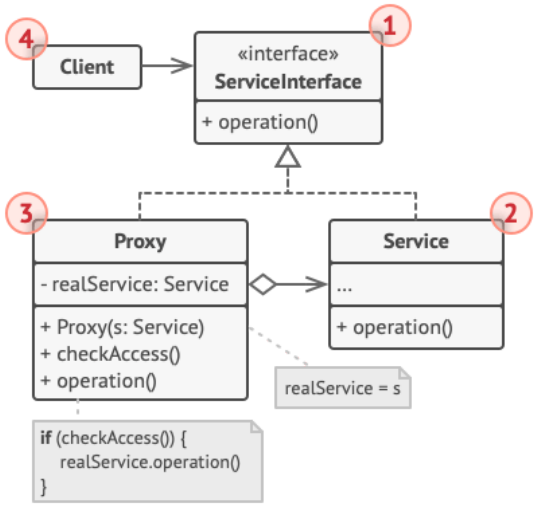
```
class Facade:
    """
    The Facade class provides a simple interface to the complex logic of one or
    several subsystems. The Facade delegates the client requests to the
    appropriate objects within the subsystem. The Facade is also responsible for
    managing their lifecycle. All of this shields the client from the undesired
    complexity of the subsystem.
    """

    def __init__(self, subsystem1: Subsystem1, subsystem2: Subsystem2) -> None:
        """
        Depending on your application's needs, you can provide the Facade with
        existing subsystem objects or force the Facade to create them on its
        own.
        """

        self._subsystem1 = subsystem1 or Subsystem1()
        self._subsystem2 = subsystem2 or Subsystem2()

    def operation(self) -> str:
        """
        The Facade's methods are convenient shortcuts to the sophisticated
        functionality of the subsystems. However, clients get only to a fraction
        of a subsystem's capabilities.
        """

        results = []
        results.append("Facade initializes subsystems:")
        results.append(self._subsystem1.operation1())
        results.append(self._subsystem2.operation1())
        results.append("Facade orders subsystems to perform the action:")
        results.append(self._subsystem1.operation_n())
        results.append(self._subsystem2.operation_z())
        return "\n".join(results)


class Subsystem1:
    """
    The Subsystem can accept requests either from the facade or client directly.
    In any case, to the Subsystem, the Facade is yet another client, and it's
    not a part of the Subsystem.
```

```
    """

    def operation1(self) -> str:
        return "Subsystem1: Ready!"

    # ...

    def operation_n(self) -> str:
        return "Subsystem1: Go!"


class Subsystem2:
    """
    Some facades can work with multiple subsystems at the same time.
    """

    def operation1(self) -> str:
        return "Subsystem2: Get ready!"

    # ...

    def operation_z(self) -> str:
        return "Subsystem2: Fire!"


def client_code(facade: Facade) -> None:
    """
    The client code works with complex subsystems through a simple interface
    provided by the Facade. When a facade manages the lifecycle of the
    subsystem, the client might not even know about the existence of the
    subsystem. This approach lets you keep the complexity under control.
    """

    print(facade.operation(), end="")


if __name__ == "__main__":
    # The client code may have some of the subsystem's objects already created.
    # In this case, it might be worthwhile to initialize the Facade with these
    # objects instead of letting the Facade create new instances.
    subsystem1 = Subsystem1()
    subsystem2 = Subsystem2()
    facade = Facade(subsystem1, subsystem2)
    client_code(facade)
```

Proxy 代理模式

代理模式是一种结构型设计模式， 让你能够提供对象的替代品或其占位符。 代理控制着对于原对象的访问， 并允许在将请求提交给对象前后进行一些处理。

Proxy pattern is a structural design pattern that allows you to provide substitutes for objects or placeholders for them. The proxy controls access to the original object and allows some processing before and after the request is submitted to the object.



1. **服务接口** （Service Interface） 声明了服务接口。 代理必须遵循该接口才能伪装成服务对象。
2. **服务** （Service） 类提供了一些实用的业务逻辑。
3. **代理** （Proxy） 类包含一个指向服务对象的引用成员变量。 代理完成其任务 （例如延迟初始化、 记录日志、 访问控制和缓存等） 后会将请求传递给服务对象。 通常情况下， 代理会对其服务对象的整个生命周期进行管理。
4. **客户端** （Client） 能通过同一接口与服务或代理进行交互， 所以你可在一切需要服务对象的代码中使用代理。

场景

- 延迟初始化 （虚拟代理）。 如果你有一个偶尔使用的重量级服务对象， 一直保持该对象运行会消耗系统资源时， 可使用代理模式。
- 访问控制 （保护代理）。 如果你只希望特定客户端使用服务对象， 这里的对象可以是操作系统中非常重要的部分， 而客户端则是各种已启动的程序 （包括恶意程序）， 此时可使用代理模式。
- 本地执行远程服务 （远程代理）。 适用于服务对象位于远程服务器上的情形。
- 记录日志请求 （日志记录代理）。 适用于当你需要保存对于服务对象的请求历史记录时。 代理可以在向服务传递请求前进行记录。

优缺点

A:

- 你可以在客户端毫无察觉的情况下控制服务对象。
- 如果客户端对服务对象的生命周期没有特殊要求， 你可以对生命周期进行管理。
- 即使服务对象还未准备好或不存在， 代理也可以正常工作。
- 开闭原则。 你可以在不对服务或客户端做出修改的情况下创建新代理。

D:

- 代码可能会变得复杂， 因为需要新建许多类。
- 服务响应可能会延迟。

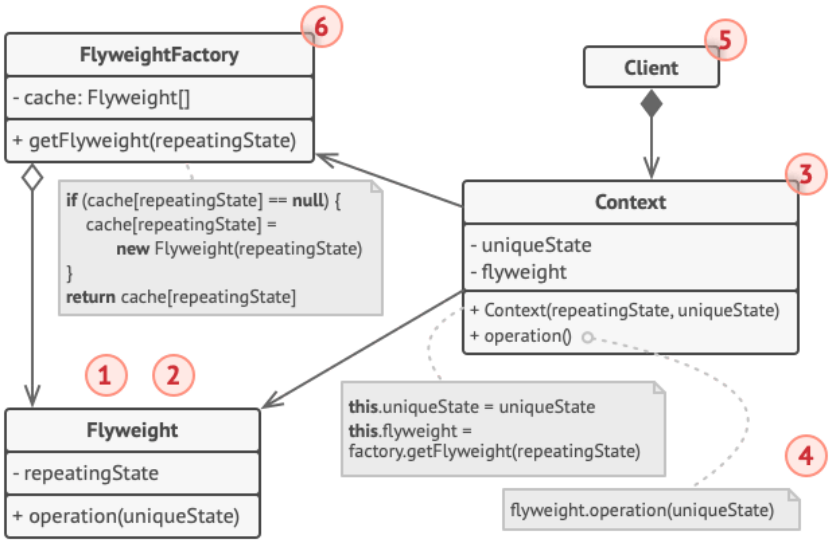
关系

- 适配器模式能为被封装对象提供不同的接口， 代理模式能为对象提供相同的接口， 装饰模式则能为对象提供加强的接口。
- 外观模式与代理的相似之处在于它们都缓存了一个复杂实体并自行对其进行初始化。 代理与其服务对象遵循同一接口， 使得自己和服务对象可以互换， 在这一点上它与外观不同。
- 装饰和代理有着相似的结构， 但是其意图却非常不同。 这两个模式的构建都基于组合原则， 也就是说一个对象应该将部分工作委派给另一个对象。 两者之间的不同之处在于**代理通常自行管理其服务对象的生命周期， 而装饰的生成则总是由客户端进行控制。**

Flyweight 享元

享元模式是一种结构型设计模式， 它摒弃了在每个对象中保存所有数据的方式， 通过共享多个对象所共有的相同状态， 让你能在有限的内存容量中载入更多对象。

Flyweight is a structural design pattern that eliminates the need to store all data in each object and allows you to load more objects into limited memory by sharing the same state shared by multiple objects.



1. 享元模式只是一种优化。 在应用该模式之前， 你要确定程序中存在与大量类似对象同时占用内存相关的内存消耗问题， 并且确保该问题无法使用其他更好的方式来解决。
2. **享元** （Flyweight） 类包含原始对象中部分能在多个对象中共享的状态。 同一享元对象可在许多不同情景中使用。 享元中存储的状态被称为 “内在状态”。 传递给享元方法的状态被称为 “外在状态”。
3. **情景** （Context） 类包含原始对象中各不相同的外在状态。 情景与享元对象组合在一起就能表示原始对象的全部状态。
4. 通常情况下， 原始对象的行为会保留在享元类中。 因此调用享元方法必须提供部分外在状态作为参数。 但你也可将行为移动到情景类中， 然后将连入的享元作为单纯的数据对象。
5. **客户端** （Client） 负责计算或存储享元的外在状态。 在客户端看来， 享元是一种可在运行时进行配置的模板对象， 具体的配置方式为向其方法中传入一些情景数据参数。
6. **享元工厂** （Flyweight Factory） 会对已有享元的缓存池进行管理。 有了工厂后， 客户端就无需直接创建享元， 它们只需调用工厂并向其传递目标享元的一些内在状态即可。 工厂会根据参数在之前已创建的享元中进行查找， 如果找到满足条件的享元就将其返回； 如果没有找到就根据参数新建享元。

场景

- 仅在程序必须支持大量对象且没有足够的内存容量时使用享元模式。

优缺点

A：如果程序中有很多相似对象， 那么你将可以节省大量内存。

D：

- 你可能需要牺牲执行速度来换取内存， 因为他人每次调用享元方法时都需要重新计算部分情景数据。
- 代码会变得更加复杂。

关系

- 你可以使用享元模式实现组合模式树的共享叶节点以节省内存。
- 享元展示了如何生成大量的小型对象， 外观模式则展示了如何用一个对象来代表整个子系统。

Code ⭐

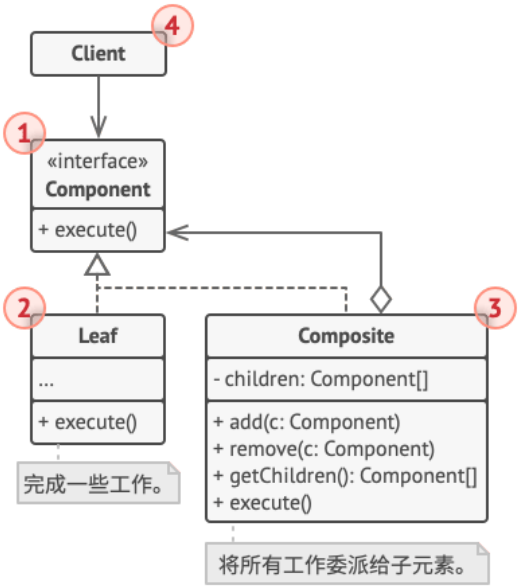
```
class Vehicle:
    def __init__(self, engine, color):
        self.engine = engine
        self.color = color

    def start(self):
        return 'It start running'

    def stop(self):
        return "It didn't running"

    def get_color(self):
        return self.color


class Car(Vehicle):
    def start(self):
        return 'Car start running'

    def stop(self):
        return "Car didn't running"

    def get_color(self):
        return f"Car's color is {self.color}"


class VehicleFactory:
    def __init__(self):
        self.hashtable = dict()

    def get_cars(self, key) -> Car:
        # Avoid creating cars of the same color repeatedly
        if key not in self.hashtable:
            self.hashtable[key] = Car('default Engine', key)
        return self.hashtable[key]

    def get_car_number(self):
        return len(self.hashtable.keys())


if __name__ == '__main__':
    factory = VehicleFactory()
    car1 = factory.get_cars('red')
    car2 = factory.get_cars('green')
    car3 = factory.get_cars('green')

    print(car1.start())
    print(car1.get_color())
    print(car1.stop())
    print(car2.start())
```

```
    print(car2.get_color())
    print(car2.stop())

    print(factory.get_car_number())  # 2 car
```

Composite 组合模式

组合模式是一种结构型设计模式， 你可以使用它将对象组合成树状结构， 并且能像使用独立对象一样使用它们。

The composite pattern is a structural design pattern that allows you to combine objects into a tree structure and use them as if they were individual objects.



1. **组件** （Component） 接口描述了树中简单项目和复杂项目所共有的操作。
2. **叶节点** （Leaf） 是树的基本结构， 它不包含子项目。

   一般情况下， 叶节点最终会完成大部分的实际工作， 因为它们无法将工作指派给其他部分。
3. **容器** （Container）——又名 “组合 （Composite）”——是包含叶节点或其他容器等子项目的单位。 容器不知道其子项目所属的具体类， 它只通过通用的组件接口与其子项目交互。

   容器接收到请求后会将工作分配给自己的子项目， 处理中间结果， 然后将最终结果返回给客户端。
4. **客户端** （Client） 通过组件接口与所有项目交互。 因此， 客户端能以相同方式与树状结构中的简单或复杂项目交互。

场景

- 如果你需要实现树状对象结构， 可以使用组合模式。
- 组合模式中定义的所有元素共用同一个接口。 在这一接口的帮助下， 客户端不必在意其所使用的对象的具体类。

优缺点

- 你可以利用多态和递归机制更方便地使用复杂树结构。
- 开闭原则。 无需更改现有代码， 你就可以在应用中添加新元素， 使其成为对象树的一部分。

D：

- 对于功能差异较大的类， 提供公共接口或许会有困难。 在特定情况下， 你需要过度一般化组件接口， 使其变得令人难以理解。

关系

- 桥接模式、 状态模式和策略模式 （在某种程度上包括适配器模式） 模式的接口非常相似。 实际上， 它们都基于组合模式——即将工作委派给其他对象， 不过也各自解决了不同的问题。 模式并不只是以特定方式组织代码的配方， 你还可以使用它们来和其他开发者讨论模式所解决的问题。
- 你可以在创建复杂组合树时使用生成器模式， 因为这可使其构造步骤以递归的方式运行。
- 责任链模式通常和组合模式结合使用。 在这种情况下， 叶组件接收到请求后， 可以将请求沿包含全体父组件的链一直传递至对象树的底部。

Code

```
from abc import ABC, abstractmethod
from typing import List


class Component(ABC):
    @property
    def parent(self) -> Component:
        return self._parent

    @parent.setter
    def parent(self, parent: Component):
        """
        Optionally, the base Component can declare an interface for setting and
        accessing a parent of the component in a tree structure. It can also
        provide some default implementation for these methods.
        """

        self._parent = parent

    """
    In some cases, it would be beneficial to define the child-management
    operations right in the base Component class. This way, you won't need to
    expose any concrete component classes to the client code, even during the
    object tree assembly. The downside is that these methods will be empty for
    the leaf-level components.
    """

    def add(self, component: Component) -> None:
        pass

    def remove(self, component: Component) -> None:
        pass

    def is_composite(self) -> bool:
        """
        You can provide a method that lets the client code figure out whether a
        component can bear children.
        """

        return False

    @abstractmethod
```

```
    def operation(self) -> str:
        """
        The base Component may implement some default behavior or leave it to
        concrete classes (by declaring the method containing the behavior as
        "abstract").
        """

        pass


class Leaf(Component):
    """
    The Leaf class represents the end objects of a composition. A leaf can't
    have any children.

    Usually, it's the Leaf objects that do the actual work, whereas Composite
    objects only delegate to their sub-components.
    """

    def operation(self) -> str:
        return "Leaf"


class Composite(Component):
    """
    The Composite class represents the complex components that may have
    children. Usually, the Composite objects delegate the actual work to their
    children and then "sum-up" the result.
    """
    def __init__(self) -> None:
        self._children: List[Component] = []

    """
    A composite object can add or remove other components (both simple or
    complex) to or from its child list.
    """

    def add(self, component: Component) -> None:
        self._children.append(component)
        component.parent = self

    def remove(self, component: Component) -> None:
        self._children.remove(component)
        component.parent = None

    def is_composite(self) -> bool:
        return True

    def operation(self) -> str:
        """
        The Composite executes its primary logic in a particular way. It
        traverses recursively through all its children, collecting and summing
        their results. Since the composite's children pass these calls to their
        children and so forth, the whole object tree is traversed as a result.
        """

        results = []
        for child in self._children:
```

```
            results.append(child.operation())
        return f"Branch({'+'.join(results)})"


def client_code(component: Component) -> None:
    """
    The client code works with all of the components via the base interface.
    """

    print(f"RESULT: {component.operation()}", end="")


def client_code2(component1: Component, component2: Component) -> None:
    """
    Thanks to the fact that the child-management operations are declared in the
    base Component class, the client code can work with any component, simple or
    complex, without depending on their concrete classes.
    """

    if component1.is_composite():
        component1.add(component2)

    print(f"RESULT: {component1.operation()}", end="")


if __name__ == "__main__":
    # This way the client code can support the simple leaf components...
    simple = Leaf()
    print("Client: I've got a simple component:")
    client_code(simple)
    print("\n")

    # ...as well as the complex composites.
    tree = Composite()

    branch1 = Composite()
    branch1.add(Leaf())
    branch1.add(Leaf())

    branch2 = Composite()
    branch2.add(Leaf())

    tree.add(branch1)
    tree.add(branch2)

    print("Client: Now I've got a composite tree:")
    client_code(tree)
    print("\n")

    print("Client: I don't need to check the components classes even when
managing the tree:")
    client_code2(tree, simple)
```