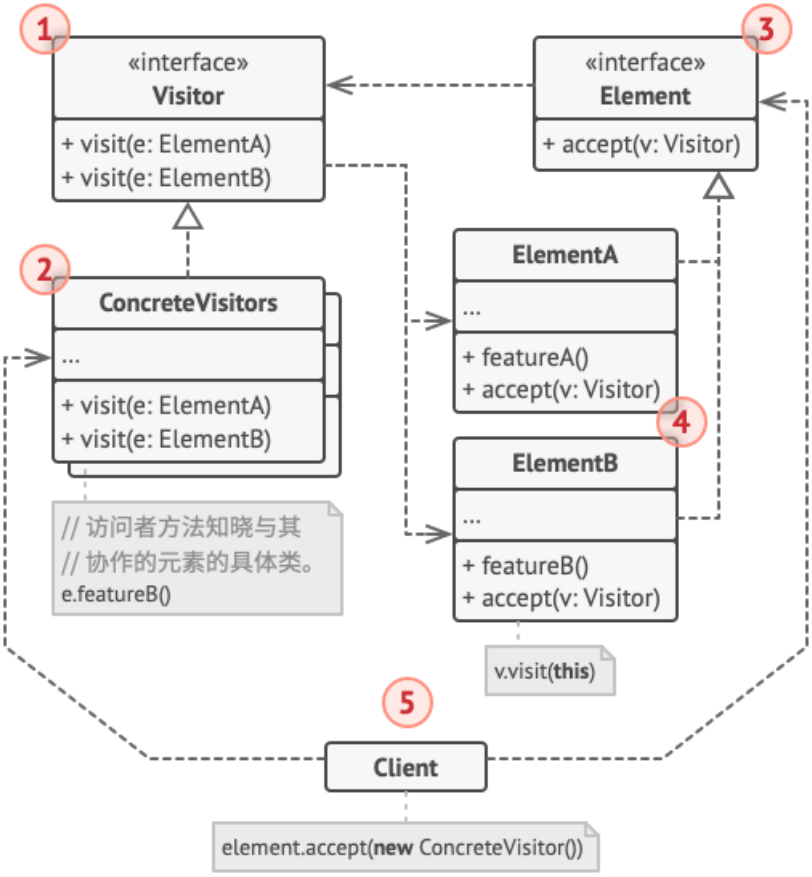
行为模式

Visitor 访问者

访问者模式是一种行为设计模式， 它能将算法与其所作用的对象隔离开来。

The visitor pattern is a behavior design pattern that **isolates** an algorithm from the object it works on.



1. **访问者** （Visitor） 接口声明了一系列以对象结构的具体元素为参数的访问者方法。 如果编程语言支持重载， 这些方法的名称可以是相同的， 但是其参数一定是不同的。
2. **具体访问者** （Concrete Visitor） 会为不同的具体元素类实现相同行为的几个不同版本。
3. **元素** （Element） 接口声明了一个方法来 “接收” 访问者。 该方法必须有一个参数被声明为访问者接口类型。
4. **具体元素** （Concrete Element） 必须实现接收方法。 该方法的目的是根据当前元素类将其调用重定向到相应访问者的方法。 请注意， 即使元素基类实现了该方法， 所有子类都必须对其进行重写并调用访问者对象中的合适方法。
5. **客户端** （Client） 通常会作为集合或其他复杂对象 （例如一个组合树） 的代表。 客户端通常不知晓所有的具体元素类， 因为它们会通过抽象接口与集合中的对象进行交互。

场景

- 如果你需要对一个复杂对象结构 （例如对象树） 中的所有元素执行某些操作， 可使用访问者模式。If you need to perform certain operations on all elements in a complex object structure, such as a tree of objects, use the Visitor pattern.
- 可使用访问者模式来清理辅助行为的业务逻辑。
- 当某个行为仅在类层次结构中的一些类中有意义， 而在其他类中没有意义时， 可使用该模式。

优缺点

- **开闭原则。** 你可以引入在不同类对象上执行的新行为， 且无需对这些类做出修改。
- **单一职责原则。** 可将同一行为的不同版本移到同一个类中。
- 访问者对象可以在与各种对象交互时收集一些有用的信息。 当你想要遍历一些复杂的对象结构（例如对象树）， 并在结构中的每个对象上应用访问者时， 这些信息可能会有所帮助。

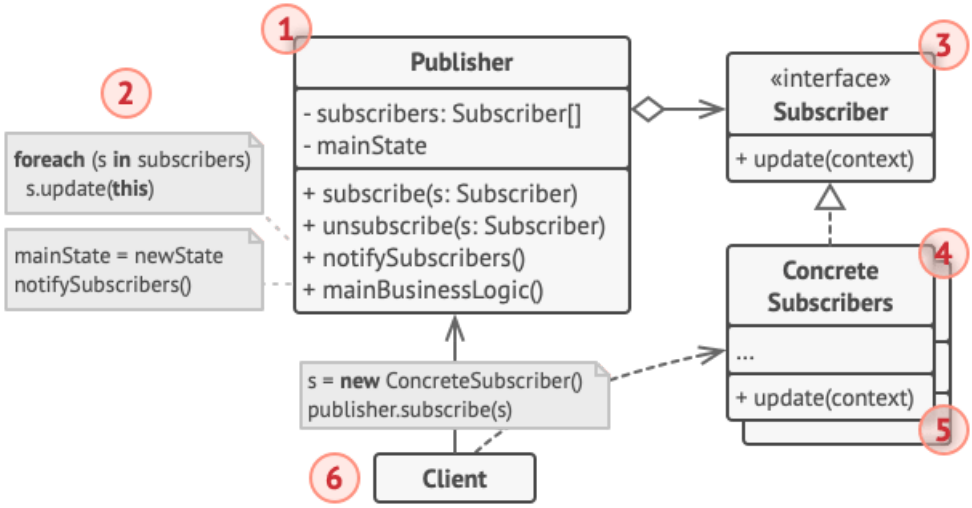
关系

- 你可以将访问者模式视为命令模式的加强版本， 其对象可对不同类的多种对象执行操作。
- 你可以使用访问者对整个组合模式树执行操作。
- 可以同时使用访问者和迭代器模式来遍历复杂数据结构， 并对其中的元素执行所需操作， 即使这些元素所属的类完全不同。

Command 命令模式

观察者模式是一种行为设计模式， 允许你定义一种订阅机制， 可在对象事件发生时通知多个 “观察” 该对象的其他对象。

The Observer pattern is a behavior design pattern that allows you to define a **subscription mechanism** that notifies multiple other objects that "**observe**" an object when an **event happen**.



1. **发布者** （Publisher） 会向其他对象发送值得关注的事件。 事件会在发布者自身状态改变或执行特定行为后发生。 发布者中包含一个允许新订阅者加入和当前订阅者离开列表的订阅构架。
2. 当新事件发生时， 发送者会遍历订阅列表并调用每个订阅者对象的通知方法。 该方法是在订阅者接口中声明的。
3. **订阅者** （Subscriber） 接口声明了通知接口。 在绝大多数情况下， 该接口仅包含一个 `update` 更新方法。 该方法可以拥有多个参数， 使发布者能在更新时传递事件的详细信息。
4. **具体订阅者** （Concrete Subscribers） 可以执行一些操作来回应发布者的通知。 所有具体订阅者类都实现了同样的接口， 因此发布者不需要与具体类相耦合。
5. 订阅者通常需要一些上下文信息来正确地处理更新。 因此， 发布者通常会将一些上下文数据作为通知方法的参数进行传递。 发布者也可将自身作为参数进行传递， 使订阅者直接获取所需的数据。
6. **客户端** （Client） 会分别创建发布者和订阅者对象， 然后为订阅者注册发布者更新。

场景

- 当一个对象状态的改变需要改变其他对象， 或实际对象是事先未知的或动态变化的时， 可使用观察者模式。

  The observer mode can be used when a change in the state of one object requires changes to other objects, or when the actual object is previously unknown or dynamically changing.

- 当应用中的一些对象必须观察其他对象时， 可使用该模式。 但仅能在有限时间内或特定情况下使用。

优缺点

- **开闭原则** ***The open closed principle***。 你无需修改发布者代码就能引入新的订阅者类 （如果是发布者接口则可轻松引入发布者类）。
- 你可以在运行时建立对象之间的联系。

D：

- 订阅者的通知顺序是随机的。

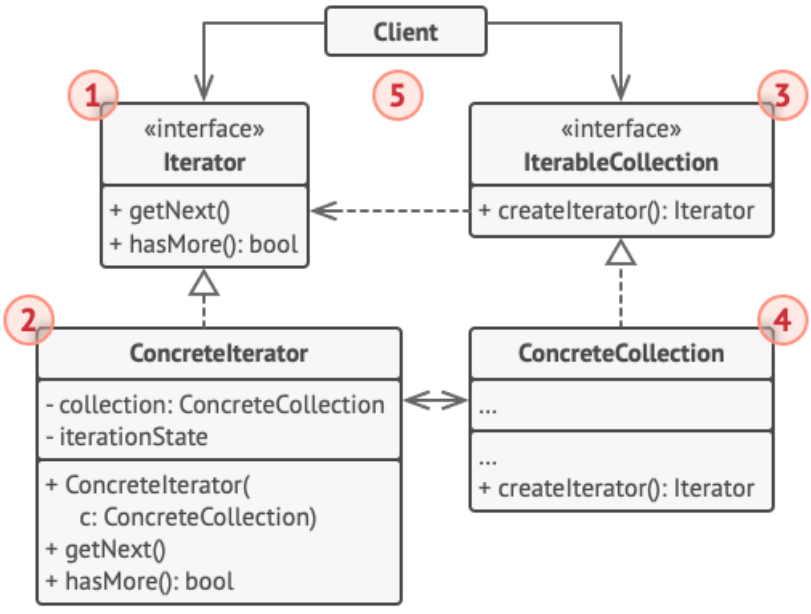
关系

What is the difference between the observer pattern and the chain of responsibility pattern in the notification delivery?

- **责任链** 按照顺序将请求动态传递给一系列的潜在接收者， 直至其中一名接收者对请求进行处理。 **Chain of responsibility** Requests are dynamically passed in order to a series of potential recipients until one of them processes the request.
- **观察者** 允许接收者动态地订阅或取消接收请求。Allows the receiver to dynamically subscribe or unsubscribe from receiving requests.

Iterator 迭代器

迭代器模式是一种行为设计模式， 让你能在不暴露集合底层表现形式 （列表、 栈和树等） 的情况下遍历集合中所有的元素。The iterator pattern is a behavioral design pattern that lets you iterate through all the elements of a collection **without exposing the underlying representation of the collection** (lists, stacks, trees, etc.).



1. **迭代器** （Iterator） 接口声明了遍历集合所需的操作： 获取下一个元素、 获取当前位置和重新开始迭代等。
2. **具体迭代器** （Concrete Iterators） 实现遍历集合的一种特定算法。 迭代器对象必须跟踪自身遍历的进度。 这使得多个迭代器可以相互独立地遍历同一集合。
3. **集合** （Collection） 接口声明一个或多个方法来获取与集合兼容的迭代器。 请注意， 返回方法的类型必须被声明为迭代器接口， 因此具体集合可以返回各种不同种类的迭代器。
4. **具体集合** （Concrete Collections） 会在客户端请求迭代器时返回一个特定的具体迭代器类实体。 你可能会琢磨， 剩下的集合代码在什么地方呢？ 不用担心， 它也会在同一个类中。 只是这些细节对于实际模式来说并不重要， 所以我们将其省略了而已。
5. **客户端** （Client） 通过集合和迭代器的接口与两者进行交互。 这样一来客户端无需与具体类进行耦合， 允许同一客户端代码使用各种不同的集合和迭代器。

   客户端通常不会自行创建迭代器， 而是会从集合中获取。 但在特定情况下， 客户端可以直接创建一个迭代器 （例如当客户端需要自定义特殊迭代器时）。

场景

- 当集合背后为复杂的数据结构， 且你希望对客户端隐藏其复杂性时 （出于使用便利性或安全性的考虑）， 可以使用迭代器模式。
- 使用该模式可以减少程序中重复的遍历代码。
- 如果你希望代码能够遍历不同的甚至是无法预知的数据结构， 可以使用迭代器模式。

优缺点

- 单一职责原则 Single responsibility principle。 通过将体积庞大的遍历算法代码抽取为独立的类， 你可对客户端代码和集合进行整理。
- 开闭原则 The open closed principle。 你可实现新型的集合和迭代器并将其传递给现有代码， 无需修改现有代码。
- 你可以并行遍历同一集合， 因为每个迭代器对象都包含其自身的遍历状态。
- 相似的， 你可以暂停遍历并在需要时继续。

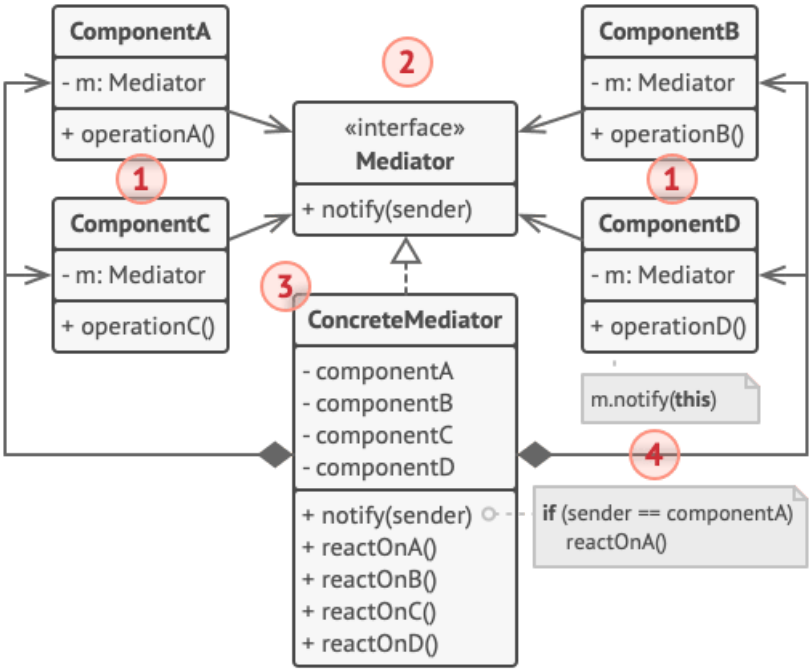
D:

- 如果你的程序只与简单的集合进行交互， 应用该模式可能会矫枉过正。
- 对于某些特殊集合， 使用迭代器可能比直接遍历的效率低。

Mediator 中介者

中介者模式是一种行为设计模式， 能让你减少对象之间混乱无序的依赖关系。 该模式会限制对象之间的直接交互， 迫使它们通过一个中介者对象进行合作。

The Mediator pattern is a behavior design pattern that allows you to reduce chaotic dependencies between objects. This pattern restricts direct interaction between objects, forcing them to collaborate through a mediator object.



1. **组件** （Component） 是各种包含业务逻辑的类。 每个组件都有一个指向中介者的引用， 该引用被声明为中介者接口类型。 组件不知道中介者实际所属的类， 因此你可通过将其连接到不同的中介者以使其能在其他程序中复用。
2. **中介者** （Mediator） 接口声明了与组件交流的方法， 但通常仅包括一个通知方法。 组件可将任意上下文 （包括自己的对象） 作为该方法的参数， 只有这样接收组件和发送者类之间才不会耦合。
3. **具体中介者** （Concrete Mediator） 封装了多种组件间的关系。 具体中介者通常会保存所有组件的引用并对其进行管理， 甚至有时会对其生命周期进行管理。
4. 组件并不知道其他组件的情况。 如果组件内发生了重要事件， 它只能通知中介者。 中介者收到通知后能轻易地确定发送者， 这或许已足以判断接下来需要触发的组件了。

   对于组件来说， 中介者看上去完全就是一个黑箱。 发送者不知道最终会由谁来处理自己的请求， 接收者也不知道最初是谁发出了请求。

场景

- 当一些对象和其他对象紧密耦合以致难以对其进行修改时， 可使用中介者模式。
- 当组件因过于依赖其他组件而无法在不同应用中复用时， 可使用中介者模式。
- 如果为了能在不同情景下复用一些基本行为， 导致你需要被迫创建大量组件子类时， 可使用中介者模式。

优缺点

- 单一职责原则。 你可以将多个组件间的交流抽取到同一位置， 使其更易于理解和维护。
- 开闭原则。 你无需修改实际组件就能增加新的中介者。
- 你可以减轻应用中多个组件间的耦合情况。
- 你可以更方便地复用各个组件。

D:

- 一段时间后， 中介者可能会演化成为上帝对象。

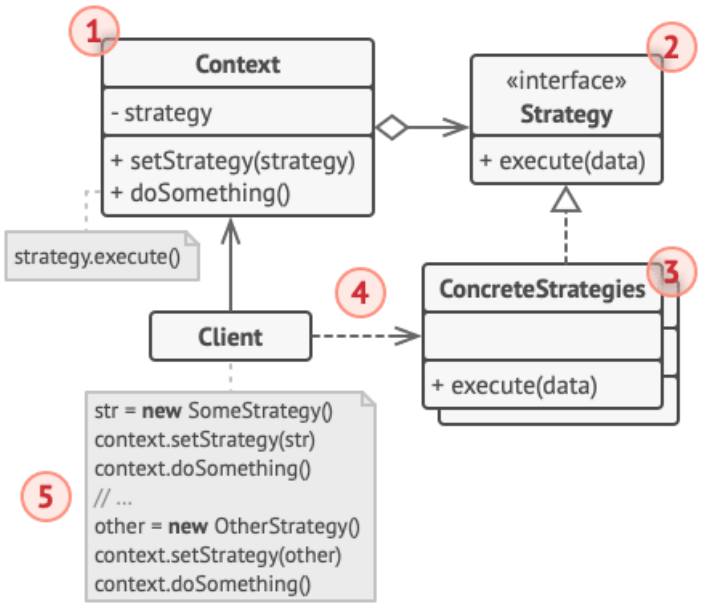
关系

- 责任链模式、 命令模式、 中介者模式和观察者模式用于处理请求发送者和接收者之间的不同连接方式：
  - 责任链按照顺序将请求动态传递给一系列的潜在接收者， 直至其中一名接收者对请求进行处理。
  - 命令在发送者和请求者之间建立单向连接。
  - 中介者清除了发送者和请求者之间的直接连接， 强制它们通过一个中介对象进行间接沟通。
  - 观察者允许接收者动态地订阅或取消接收请求。

Strategy 策略模式

策略模式是一种行为设计模式， 它能让你定义一系列算法， 并将每种算法分别放入独立的类中， 以使算法的对象能够相互替换。

The strategy pattern is a behavior design pattern that allows you to define a series of algorithms and place each algorithm in a separate class so that the objects of the algorithm are interchangeable.



1. **上下文** （Context） 维护指向具体策略的引用， 且仅通过策略接口与该对象进行交流。
2. **策略** （Strategy） 接口是所有具体策略的通用接口， 它声明了一个上下文用于执行策略的方法。
3. **具体策略** （Concrete Strategies） 实现了上下文所用算法的各种不同变体。
4. 当上下文需要运行算法时， 它会在其已连接的策略对象上调用执行方法。 上下文不清楚其所涉及的策略类型与算法的执行方式。
5. **客户端** （Client） 会创建一个特定策略对象并将其传递给上下文。 上下文则会提供一个设置器以便客户端在运行时替换相关联的策略。

场景

- 当你有许多仅在执行某些行为时略有不同的相似类时， 可使用策略模式。
- 如果算法在上下文逻辑中不是特别重要， 使用该模式能将类的业务逻辑与其算法实现细节隔离开来。
- 当类中使用了复杂条件运算符以在同一算法的不同变体中切换时， 可使用该模式。

优缺点

- 你可以在运行时切换对象内的算法。
- 你可以将算法的实现和使用算法的代码隔离开来。
- 你可以使用组合来代替继承。
- 开闭原则。 你无需对上下文进行修改就能够引入新的策略。

D:

- 如果你的算法极少发生改变， 那么没有任何理由引入新的类和接口。 使用该模式只会让程序过于复杂。
- 客户端必须知晓策略间的不同——它需要选择合适的策略。
- 许多现代编程语言支持函数类型功能， 允许你在一组匿名函数中实现不同版本的算法。 这样， 你使用这些函数的方式就和使用策略对象时完全相同， 无需借助额外的类和接口来保持代码简洁。

关系

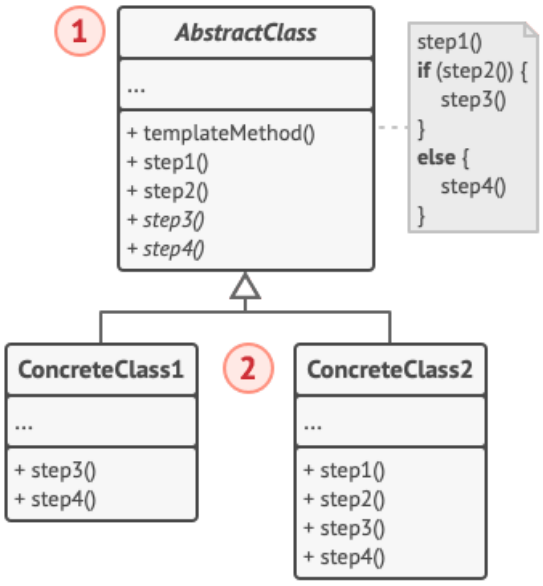
- **模板方法模式**基于继承机制： 它允许你通过扩展子类中的部分内容来改变部分算法。 **策略**基于组合机制： 你可以通过对相应行为提供不同的策略来改变对象的部分行为。 模板方法在类层次上运作， 因此它是静态的。 策略在对象层次上运作， 因此允许在运行时切换行为。

  The template method pattern is based on inheritance: it allows you to change part of the algorithm by extending part of the subclass. **Strategies** Based on composition: You can change part of an object's behavior by providing different strategies for corresponding behaviors. A template method operates at the class level, so it is static. Policies operate at the object level, thus allowing behavior to be switched at run time.

Template Method 模板方法

模板方法模式是一种行为设计模式， 它在超类中定义了一个算法的框架， 允许子类在不修改结构的情况下重写算法的特定步骤。

The **template method pattern** is a behavior design pattern that defines the framework of an algorithm in a superclass, allowing subclasses to override specific steps of the algorithm without modifying the structure.



1. **抽象类** （AbstractClass） 会声明作为算法步骤的方法， 以及依次调用它们的实际模板方法。 算法步骤可以被声明为 `抽象` 类型， 也可以提供一些默认实现。
2. **具体类** （ConcreteClass） 可以重写所有步骤， 但不能重写模板方法自身。

场景

- 当你只希望客户端扩展某个特定算法步骤， 而不是整个算法或其结构时， 可使用模板方法模式。
- 当多个类的算法除一些细微不同之外几乎完全一样时， 你可使用该模式。 但其后果就是， 只要算法发生变化， 你就可能需要修改所有的类。

优缺点

- 你可仅允许客户端重写一个大型算法中的特定部分， 使得算法其他部分修改对其所造成的影响减小。
- 你可将重复代码提取到一个超类中。

D:

- 部分客户端可能会受到算法框架的限制。
- 模板方法中的步骤越多， 其维护工作就可能会越困难。

关系

- 工厂方法模式是模板方法模式的一种特殊形式。 同时， 工厂方法可以作为一个大型模板方法中的一个步骤。

Code ★

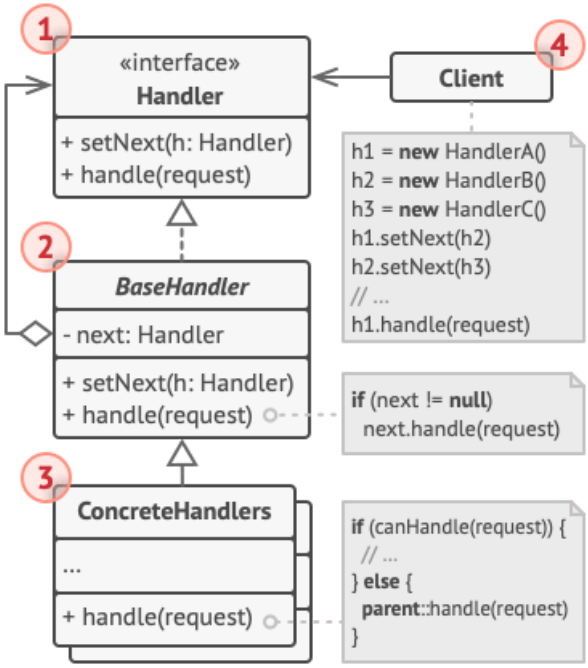
```
from abc import ABC, abstractmethod


class HouseTemplate(ABC):
    def buildHouse(self) -> None:
        self.buildFoundation()
        self.buildWalls()
        self.buildWindows()
        print("House is build")

    def buildFoundation(self) -> None:
        print("Building Glass Windows")

    @abstractmethod
    def buildWalls(self) -> None:
        pass

    def buildWindows(self) -> None:
        print("Building foundation with cement,iron rods and sand")


class WoodenHouseand(HouseTemplate):
    def buildWalls(self) -> None:
        print("Building Wooden Walls")


class GlassHouse(HouseTemplate):
    def buildWalls(self) -> None:
        print("Building Glass Walls")


def client_code(abstract_class: HouseTemplate) -> None:
    # ...
```

```
    abstract_class.buildHouse()
    # ...


if __name__ == "__main__":
    print("the result is as following:")
    client_code(WoodenHouseand())
    print("")

    print("****************")
    client_code(GlassHouse())
```

Chain of Responsibility 责任链

责任链模式是一种行为设计模式， 允许你将请求沿着处理者链进行发送。 收到请求后， 每个处理者均可对请求进行处理， 或将其传递给链上的下个处理者。

The chain of responsibility pattern is a behavior design pattern that allows you to send requests down a chain of handlers. Upon receipt of the request, each handler can either process it or pass it on to the next handler on the chain.



1. **处理者** （Handler） 声明了所有具体处理者的通用接口。 该接口通常仅包含单个方法用于请求处理， 但有时其还会包含一个设置链上下个处理者的方法。
2. **基础处理者** （Base Handler） 是一个可选的类， 你可以将所有处理者共用的样本代码放置在其中。

   通常情况下， 该类中定义了一个保存对于下个处理者引用的成员变量。 客户端可通过将处理者传递给上个处理者的构造函数或设定方法来创建链。 该类还可以实现默认的处理行为： 确定下个处理者存在后再将请求传递给它。
3. **具体处理者** （Concrete Handlers） 包含处理请求的实际代码。 每个处理者接收到请求后， 都必须决定是否进行处理， 以及是否沿着链传递请求。

   处理者通常是独立且不可变的， 需要通过构造函数一次性地获得所有必要地数据。
4. **客户端** （Client） 可根据程序逻辑一次性或者动态地生成链。 值得注意的是， 请求可发送给链上的任意一个处理者， 而非必须是第一个处理者。

场景

- 当程序需要使用不同方式处理不同种类请求， 而且请求类型和顺序预先未知时， 可以使用责任链模式。
- 当必须按顺序执行多个处理者时， 可以使用该模式。
- 如果所需处理者及其顺序必须在运行时进行改变， 可以使用责任链模式。

优缺点

- 你可以控制请求处理的顺序。
- 单一职责原则。 你可对发起操作和执行操作的类进行解耦。
- 开闭原则。 你可以在不更改现有代码的情况下在程序中新增处理者。

D：

- 部分请求可能未被处理

关系

- 责任链模式、 命令模式、 中介者模式和观察者模式用于处理请求发送者和接收者之间的不同连接方式：
  - 责任链按照顺序将请求动态传递给一系列的潜在接收者， 直至其中一名接收者对请求进行处理。
  - 命令在发送者和请求者之间建立单向连接。
  - 中介者清除了发送者和请求者之间的直接连接， 强制它们通过一个中介对象进行间接沟通。
  - 观察者允许接收者动态地订阅或取消接收请求。

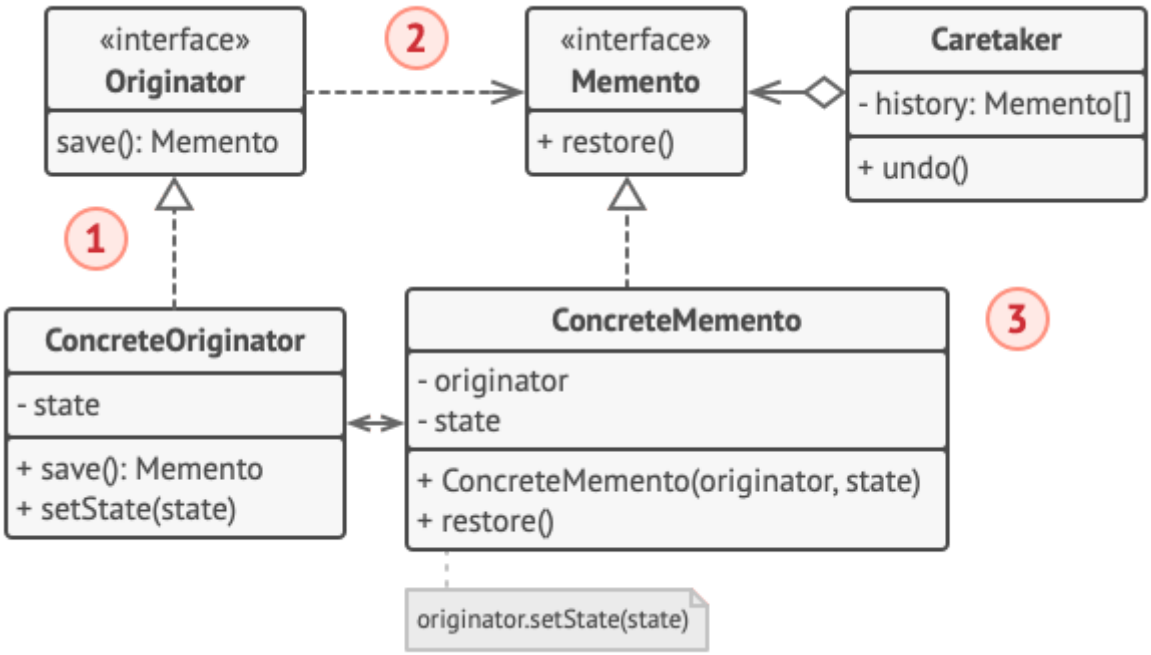
Memento 备忘录

备忘录模式是一种行为设计模式， 允许在不暴露对象实现细节的情况下保存和恢复对象之前的状态。

Memento pattern is a behavioral design pattern that allows an object to be saved and restored to its previous state without exposing its implementation details.

封装更加严格的实现

如果你不想让其他类有任何机会通过备忘录来访问原发器的状态， 那么还有另一种可用的实现方式。



1. 这种实现方式允许存在多种不同类型的原发器和备忘录。 每种原发器都和其相应的备忘录类进行交互。 原发器和备忘录都不会将其状态暴露给其他类。
2. 负责人此时被明确禁止修改存储在备忘录中的状态。 但负责人类将独立于原发器， 因为此时恢复方法被定义在了备忘录类中。
3. 每个备忘录将与创建了自身的原发器连接。 原发器会将自己及状态传递给备忘录的构造函数。 由于这些类之间的紧密联系， 只要原发器定义了合适的设置器 （setter）， 备忘录就能恢复其状态。

场景

- 当你需要创建对象状态快照来恢复其之前的状态时， 可以使用备忘录模式。
- 当直接访问对象的成员变量、 获取器或设置器将导致封装被突破时， 可以使用该模式。

优缺点

A:

- 你可以在不破坏对象封装情况的前提下创建对象状态快照。
- 你可以通过让负责人维护原发器状态历史记录来简化原发器代码。

D:

- 如果客户端过于频繁地创建备忘录， 程序将消耗大量内存。
- 负责人必须完整跟踪原发器的生命周期， 这样才能销毁弃用的备忘录。
- 绝大部分动态编程语言 （例如 PHP、 Python 和 JavaScript） 不能确保备忘录中的状态不被修改。

关系

- 你可以同时使用命令模式和备忘录模式来实现 “撤销”。 在这种情况下， 命令用于对目标对象执行各种不同的操作， 备忘录用来保存一条命令执行前该对象的状态。
- 你可以同时使用备忘录和迭代器模式来获取当前迭代器的状态， 并且在需要的时候进行回滚。
- 有时候原型模式可以作为备忘录的一个简化版本， 其条件是你需要在历史记录中存储的对象的状态比较简单， 不需要链接其他外部资源， 或者链接可以方便地重建。

Code ★

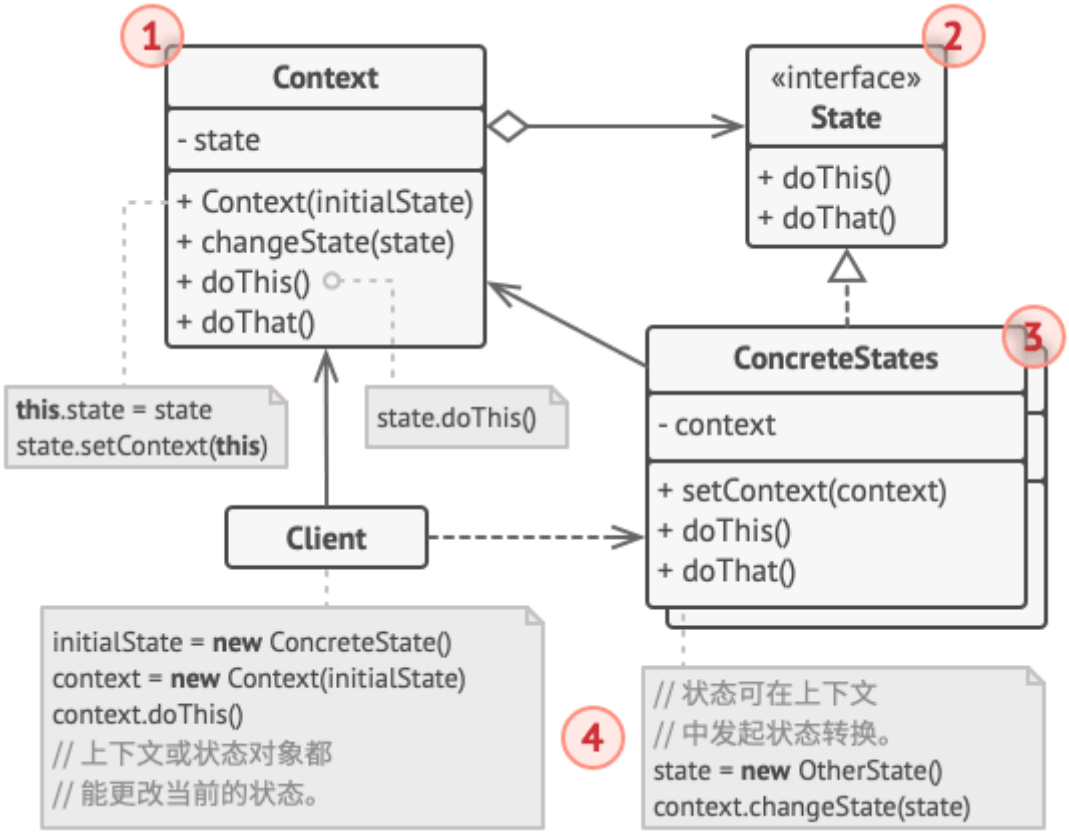
```python
import math
from abc import ABC, abstractmethod


class Memento(ABC):
    @abstractmethod
    def get_state(self) -> str:
        pass

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


class Article:
    _id = None
    _title = None
    _content = None
```

```
    def __init__(self, id: str, title: str, content: str) -> None:
        self._id = id
        self._title = title
        self._content = content

    def do_something(self) -> None:
        print("Originator: I'm doing something important.")
        self._content = str(format(math.radians(123) * 1000, '0.2f')) + "、" +
self._content
        print(f"Originator: and my state has changed to: {self._content}")

    def save(self):
        return ArticleMemento(self._id, self._title, self._content)

    def restore(self, article) -> None:
        self._content = article.get_state()
        print(f"Originator: My state has changed to: {self._content}")


class ArticleMemento(Memento):
    def __init__(self, id: str, title: str, content: str) -> None:
        self._id = id
        self._title = title
        self._content = content

    def get_state(self) -> str:
        return self._content

    def get_title(self) -> str:
        return self._title


class Caretaker:
    def __init__(self, article: Article) -> None:
        self._mementos = []
        self._originator = article

    def backup(self) -> None:
        print("\nCaretaker: Saving Originator's state...")
        self._mementos.append(self._originator.save())

    def undo(self) -> None:
        if not len(self._mementos):
            return

        memento = self._mementos.pop()
        print(f"Caretaker: Restoring state to: {memento.get_title()}")
        try:
            self._originator.restore(memento)
        except Exception:
            self.undo()

    def show_history(self) -> None:
        print("Caretaker: Here's the list of mementos:")
        for memento in self._mementos:
            print(memento.get_title())
```

```
# Test Class(function)
if __name__ == "__main__":
    originator = Article("1", "article1", "这是内容哦")
    caretaker = Caretaker(originator)

    caretaker.backup()
    originator.do_something()

    caretaker.backup()
    originator.do_something()

    caretaker.backup()
    originator.do_something()

    print()
    caretaker.show_history()

    print("\nClient: Now, let's rollback!\n")
    caretaker.undo()

    print("\nClient: Once more!\n")
    caretaker.undo()
```

State 状态

状态模式是一种行为设计模式， 让你能在一个对象的内部状态变化时改变其行为， 使其看上去就像改变了自身所属的类一样。



1. **上下文** （Context） 保存了对于一个具体状态对象的引用， 并会将所有与该状态相关的工作委派给它。 上下文通过状态接口与状态对象交互， 且会提供一个设置器用于传递新的状态对象。

2. **状态** （State） 接口会声明特定于状态的方法。 这些方法应能被其他所有具体状态所理解， 因为你不希望某些状态所拥有的方法永远不会被调用。
3. **具体状态** （Concrete States） 会自行实现特定于状态的方法。 为了避免多个状态中包含相似代码， 你可以提供一个封装有部分通用行为的中间抽象类。

   状态对象可存储对于上下文对象的反向引用。 状态可以通过该引用从上下文处获取所需信息， 并且能触发状态转移。
4. 上下文和具体状态都可以设置上下文的下个状态， 并可通过替换连接到上下文的状态对象来完成实际的状态转换。

场景

- 如果对象需要根据自身当前状态进行不同行为， 同时状态的数量非常多且与状态相关的代码会频繁变更的话， 可使用状态模式。
- 如果某个类需要根据成员变量的当前值改变自身行为， 从而需要使用大量的条件语句时， 可使用该模式。
- 当相似状态和基于条件的状态机转换中存在许多重复代码时， 可使用状态模式。

优缺点

A:

- 单一职责原则。 将与特定状态相关的代码放在单独的类中。
- 开闭原则。 无需修改已有状态类和上下文就能引入新状态。
- 通过消除臃肿的状态机条件语句简化上下文代码。

D:

- 如果状态机只有很少的几个状态， 或者很少发生改变， 那么应用该模式可能会显得小题大作。

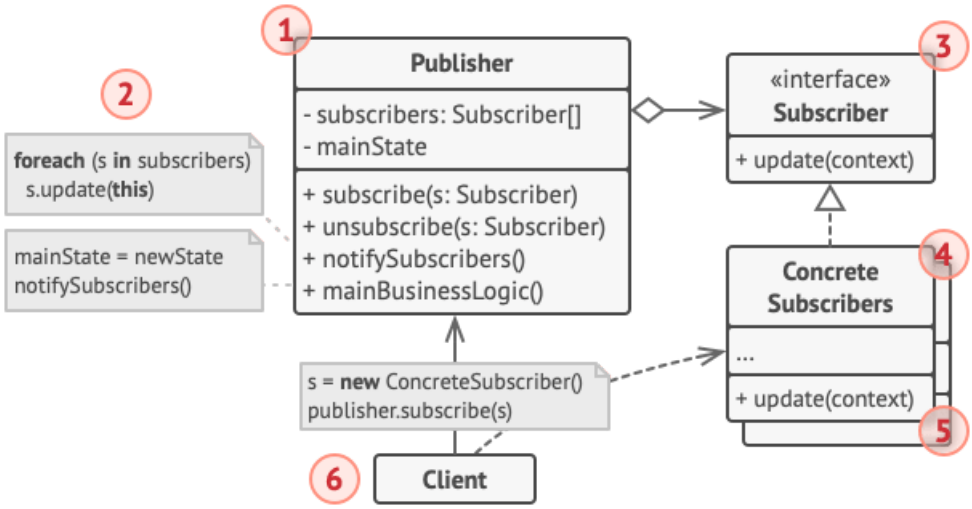
关系

- 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。
- 状态可被视为策略的扩展。 两者都基于组合机制： 它们都通过将部分工作委派给 “帮手” 对象来改变其在不同情景下的行为。 策略使得这些对象相互之间完全独立， 它们不知道其他对象的存在。 但状态模式没有限制具体状态之间的依赖， 且允许它们自行改变在不同情景下的状态。

Observer 观察者

观察者模式是一种行为设计模式， 允许你定义一种订阅机制， 可在对象事件发生时通知多个 “观察” 该对象的其他对象。

The Observer pattern is a behavior design pattern that allows you to define a subscription mechanism that notifies multiple other objects that "observe" an object when an event occurs.



1. **发布者** （Publisher） 会向其他对象发送值得关注的事件。 事件会在发布者自身状态改变或执行特定行为后发生。 发布者中包含一个允许新订阅者加入和当前订阅者离开列表的订阅构架。
2. 当新事件发生时， 发送者会遍历订阅列表并调用每个订阅者对象的通知方法。 该方法是在订阅者接口中声明的。
3. **订阅者** （Subscriber） 接口声明了通知接口。 在绝大多数情况下， 该接口仅包含一个 `update` 更新方法。 该方法可以拥有多个参数， 使发布者能在更新时传递事件的详细信息。
4. **具体订阅者** （Concrete Subscribers） 可以执行一些操作来回应发布者的通知。 所有具体订阅者类都实现了同样的接口， 因此发布者不需要与具体类相耦合。
5. 订阅者通常需要一些上下文信息来正确地处理更新。 因此， 发布者通常会将一些上下文数据作为通知方法的参数进行传递。 发布者也可将自身作为参数进行传递， 使订阅者直接获取所需的数据。
6. **客户端** （Client） 会分别创建发布者和订阅者对象， 然后为订阅者注册发布者更新。

场景

- 当一个对象状态的改变需要改变其他对象， 或实际对象是事先未知的或动态变化的时， 可使用观察者模式。
- 当应用中的一些对象必须观察其他对象时， 可使用该模式。 但仅能在有限时间内或特定情况下使用。

优缺点

- 开闭原则。 你无需修改发布者代码就能引入新的订阅者类 （如果是发布者接口则可轻松引入发布者类）。
- 你可以在运行时建立对象之间的联系。
- 订阅者的通知顺序是随机的。