Поведенческие шаблоны проектирования

Поведенческие паттерны проектирования являются важной частью объектно-ориентированного программирования. Они предоставляют общие решения для повторяющихся задач, связанных с поведением объектов и их взаимодействием.

Поведенческие паттерны нужны для:

1. Упрощения сложных взаимодействий между объектами: они позволяют разбить сложные поведенческие модели на более мелкие и понятные компоненты.
2. Увеличения гибкости и расширяемости кода: они позволяют изменять поведение объектов и их взаимодействие без изменения существующего кода.
3. Повышения уровня абстракции: они позволяют сфокусироваться на том, как объекты взаимодействуют друг с другом, а не на деталях реализации.

Разница между поведенческими, порождающими и структурными паттернами заключается в том, что они решают разные типы задач:

1. Порождающие паттерны предоставляют общие решения для создания объектов. Они помогают упростить код, связанный с созданием объектов, и увеличить гибкость и расширяемость кода.
2. Структурные паттерны предоставляют общие решения для составления объектов из других объектов. Они помогают упростить сложные структуры объектов и увеличить гибкость и расширяемость кода.
3. Поведенческие паттерны предоставляют общие решения для поведения объектов и их взаимодействия. Они помогают упростить сложные взаимодействия между объектами и увеличить гибкость и расширяемость кода.

Аналогии:

1. Порождающие паттерны можно сравнить с фабриками, которые производят различные типы продукции.
2. Структурные паттерны можно сравнить с конструкциями из лего, которые собираются из отдельных блоков.
3. Поведенческие паттерны можно сравнить с танцевальными шагами, которые определяют, как танцоры взаимодействуют друг с другом.

Цепочка обязанностей

Шаблон Цепочка обязанностей (Chain of Responsibility) предоставляет способ обработки запросов, позволяя передавать их между объектами, пока не будет найден объект, способный обработать запрос. Этот шаблон позволяет избежать связывания отправителя запроса с конкретным получателем, увеличивая гибкость и упрощая код.

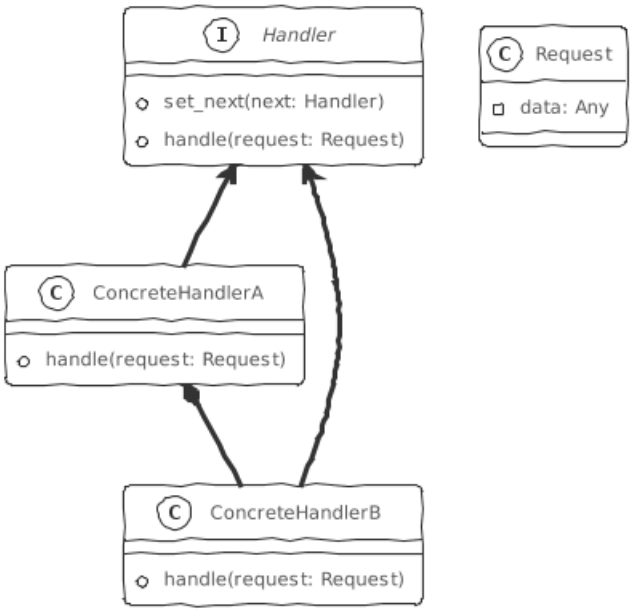
Аналогии из реальной жизни:

1. В компании есть несколько отделов, которые могут обработать заявку клиента. Заявка поступает в отдел продаж, который проверяет, может ли он обработать заявку. Если нет, то заявка передается в отдел технической поддержки, который проверяет, может ли он обработать заявку. Если нет, то заявка передается в отдел разработки, который обрабатывает заявку.
2. В школе есть несколько учителей, которые могут обработать жалобу родителей. Жалоба поступает к классному руководителю, который проверяет, может ли он обработать жалобу. Если нет, то жалоба передается к заместителю директора, который проверяет, может ли он обработать жалобу. Если нет, то жалоба передается к директору, который обрабатывает жалобу.
3. Почтовая служба: Представьте, что вы отправляете письмо по почте. Письмо проходит через несколько этапов обработки (сортировка, отправка на нужный транспорт, доставка), и на каждом этапе ответственный за него человек передает письмо дальше, пока оно не дойдет до адресата.
4. Обработка жалоб: В компании может существовать система обработки жалоб от клиентов. Жалоба поступает в отдел обработки жалоб, и сотрудник, который ее получил, может попытаться решить проблему самостоятельно. Если он не может этого сделать, он передает жалобу своему начальнику или другому сотруднику, который, возможно, сможет решить проблему. Это продолжается, пока жалоба не будет решена или не дойдет до высшего руководства компании.

Принципиальная схема шаблона Цепочка обязанностей в формате PlantUML:

```
@startuml

interface Handler {
  + set_next(next: Handler)
  + handle(request: Request)
}

class ConcreteHandlerA {
  + handle(request: Request)
}

class ConcreteHandlerB {
  + handle(request: Request)
}

class Request {
  - data: Any
}

Handler <-- ConcreteHandlerA
Handler <-- ConcreteHandlerB
ConcreteHandlerA *-- ConcreteHandlerB

@enduml
```

Принципиальная схема шаблона Цепочка обязанностей в формате UML:



Описание схемы:

- `Handler` (Обработчик) - интерфейс, определяющий методы для передачи запроса между объектами и обработки запроса.
- `ConcreteHandlerA` и `ConcreteHandlerB` (КонкретныйОбработчикА и КонкретныйОбработчикВ) - конкретные классы, реализующие интерфейс `Handler` и содержащие реализацию алгоритмов обработки запросов.
- `Request` (Запрос) - класс, содержащий данные, которые будут обрабатываться обработчиками.

Абстрактный код реализации шаблона Цепочка обязанностей на языке Python:

```
from abc import ABC, abstractmethod


class Handler(ABC):

    _next = None
```

```
    @abstractmethod
    def can_handle(self, request):
        pass

    @abstractmethod
    def handle(self, request):
        pass

    def set_next(self, handler):
        self._next = handler

    def _next_handler(self, request):
        if self._next is not None:
            self._next.handle(request)


class ConcreteHandler1(Handler):

    def can_handle(self, request):
        return request.type == 'type1'

    def handle(self, request):
        if self.can_handle(request):
            print('ConcreteHandler1 handled the request')
        else:
            self._next_handler(request)


class ConcreteHandler2(Handler):

    def can_handle(self, request):
        return request.type == 'type2'

    def handle(self, request):
        if self.can_handle(request):
            print('ConcreteHandler2 handled the request')
        else:
            self._next_handler(request)


class Client:

    def send_request(self, request):
        handler1 = ConcreteHandler1()
        handler2 = ConcreteHandler2()
        handler1.set_next(handler2)
        handler1.handle(request)


class Request:

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


if __name__ == '__main__':
    client = Client()
    request = Request('type1')
    client.send_request(request)
    request = Request('type2')
    client.send_request(request)
    request = Request('type3')
    client.send_request(request)
```

Описание кода:

- `Handler` - абстрактный класс, определяющий методы `can_handle`, `handle` и `set_next`.
- `ConcreteHandler1` и `ConcreteHandler2` - конкретные обработчики, реализующие абстрактный класс `Handler`.
- `Client` - отправитель запроса, который передает запрос первому обработчику в цепочке.
- `Request` - класс, определяющий тип запроса.
- В методе `handle` конкретного обработчика проверяется, может ли он обработать запрос. Если да, то запрос обрабатывается. Если нет, то запрос передается следующему обработчику в цепочке.
- В коде запуска создается экземпляр класса `Client`, создаются экземпляры класса `Request` с разными типами и отправляются запросы.

Пример кода с реализацией шаблона Цепочка обязанностей на Python, приближенный к реальности:

```
from abc import ABC, abstractmethod


class Approver(ABC):

    @abstractmethod
    def can_approve(self, request):
        pass

    @abstractmethod
    def approve(self, request):
        pass

    def set_next(self, approver):
        self._next = approver

    def _next_approver(self, request):
        if self._next is not None:
            self._next.approve(request)


class Director(Approver):

    def can_approve(self, request):
        return request.amount ≤ 10000

    def approve(self, request):
        if self.can_approve(request):
            print('Director approved the request')
        else:
            self._next_approver(request)


class VicePresident(Approver):

    def can_approve(self, request):
        return request.amount ≤ 50000

    def approve(self, request):
        if self.can_approve(request):
            print('Vice President approved the request')
        else:
            self._next_approver(request)


class President(Approver):

    def can_approve(self, request):
        return True

    def approve(self, request):
        print('President approved the request')
```

```
class Client:

    def send_request(self, request):
        director = Director()
        vice_president = VicePresident()
        president = President()
        director.set_next(vice_president)
        vice_president.set_next(president)
        director.approve(request)


class Request:

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


if __name__ == '__main__':
    client = Client()
    request = Request(5000)
    client.send_request(request)
    request = Request(25000)
    client.send_request(request)
    request = Request(100000)
    client.send_request(request)
```

Описание кода:

- `Approver` - абстрактный класс, определяющий методы `can_approve`, `approve` и `set_next`.
- `Director`, `VicePresident` и `President` - конкретные обработчики, реализующие абстрактный класс `Approver`.
- `Client` - отправитель запроса, который передает запрос первому обработчику в цепочке.
- `Request` - класс, определяющий сумму запроса.
- В методе `approve` конкретного обработчика проверяется, может ли он утвердить запрос. Если да, то запрос утверждается. Если нет, то запрос передается следующему обработчику в цепочке.
- В коде запуска создается экземпляр класса `Client`, создаются экземпляры класса `Request` с разными суммами и отправляются запросы.

Команда

Порождающий шаблон Команда (Command) позволяет инкапсулировать запросы к объектам в виде отдельных объектов, что обеспечивает гибкость и упрощает взаимодействие между объектами. Этот шаблон позволяет отделить объект, который инициирует запрос, от объекта, который его обрабатывает, и обеспечивает возможность отмены и повтора запросов.

Аналогии из реальной жизни:

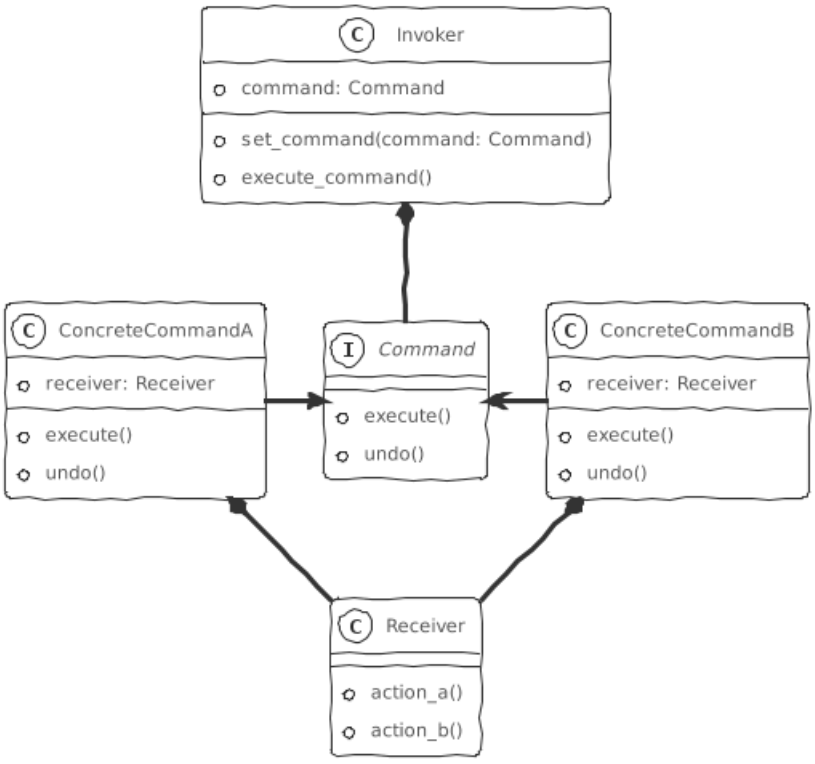
1. Ресторан: Представьте, что вы заходите в ресторан и делаете заказ. Заказ представляет собой инкапсулированный запрос к повару на приготовление блюда. Вы можете сделать несколько заказов, и каждый из них будет обрабатываться поваром. Кроме того, вы можете отменить свой заказ, если решите, что не хотите его больше.
2. Умный дом: В системе управления умным домом могут существовать различные команды для включения и выключения света, регулировки температуры и других действий. Каждая команда представляет собой инкапсулированный запрос к конкретному устройству, и вы можете отправлять эти команды из различных устройств (смартфона, планшета, голосового помощника), не завися от них.

Принципиальная схема шаблона Команда в формате PlantUML:

```
@startuml

interface Command {
  + execute()
  + undo()
```

```
}

class ConcreteCommandA {
  + receiver: Receiver
  + execute()
  + undo()
}

class ConcreteCommandB {
  + receiver: Receiver
  + execute()
  + undo()
}

class Receiver {
  + action_a()
  + action_b()
}

class Invoker {
  + command: Command
  + set_command(command: Command)
  + execute_command()
}

Command <—left- ConcreteCommandA
Command <—right- ConcreteCommandB
ConcreteCommandA *-- Receiver
ConcreteCommandB *-- Receiver
Invoker *-- Command

@enduml
```

Принципиальная схема шаблона Команда в формате UML:



Описание схемы:

- `Command` (Команда) - интерфейс, определяющий методы для выполнения и отмены запроса.
- `ConcreteCommandA` и `ConcreteCommandB` (КонкретнаяКомандаA и КонкретнаяКомандаB) - конкретные классы, реализующие интерфейс `Command` и содержащие реализацию алгоритмов выполнения и отмены запроса.
- `Receiver` (Получатель) - класс, содержащий методы, которые будут вызываться при выполнении запроса.
- `Invoker` (Инициатор) - класс, который инициирует запрос, вызывая метод `execute` у объекта `Command`.

Абстрактный код реализации шаблона Команда на языке Python:

```
from abc import ABC, abstractmethod


class Command(ABC):
    @abstractmethod
    def execute(self):
        pass

    @abstractmethod
    def undo(self):
        pass


class ConcreteCommandA(Command):
    def __init__(self, receiver):
        self._receiver = receiver

    def execute(self):
        self._receiver.action_a()

    def undo(self):
        self._receiver.undo_a()


class ConcreteCommandB(Command):
    def __init__(self, receiver):
        self._receiver = receiver

    def execute(self):
        self._receiver.action_b()

    def undo(self):
        self._receiver.undo_b()


class Receiver:
    def action_a(self):
        print("Action A")

    def undo_a(self):
        print("Undo Action A")

    def action_b(self):
        print("Action B")

    def undo_b(self):
        print("Undo Action B")


class Invoker:
    def __init__(self):
        self._commands = []

    def execute_command(self, command: Command):
        command.execute()
        self._commands.append(command)
```

```
    def undo_command(self):
        if len(self._commands) > 0:
            command = self._commands.pop()
            command.undo()


if __name__ == "__main__":
    receiver = Receiver()
    invoker = Invoker()

    commands = [
        ConcreteCommandA(receiver),
        ConcreteCommandB(receiver),
        ConcreteCommandA(receiver),
    ]

    for command in commands:
        invoker.execute_command(command)

    invoker.undo_command()
    invoker.undo_command()
```

Описание работы кода:

- Создаются абстрактные классы `Command` и конкретные классы `ConcreteCommandA`, `ConcreteCommandB`, реализующие интерфейс `Command` и содержащие реализацию алгоритмов выполнения и отмены запроса.
- Создаются классы `Receiver` и `Invoker`, которые соответствуют классам на схеме.
- В главном модуле создаются экземпляры классов `Receiver`, `Invoker` и конкретных команд.
- Выполняется цикл, в котором вызывается метод `execute_command` для инициатора, передавая ему конкретную команду.
- Инициатор вызывает метод `execute` у конкретной команды, которая, в свою очередь, вызывает метод `action_a` или `action_b` у получателя.
- После выполнения всех команд вызывается метод `undo_command` для инициатора, который отменяет последнюю выполненную команду.

Пример кода с реализацией шаблона Команда на Python, приближенный к реальности:

```
from abc import ABC, abstractmethod


class Light:
    def __init__(self, name):
        self._name = name
        self._state = False

    def turn_on(self):
        self._state = True
        print(f"{self._name} is turned on")

    def turn_off(self):
        self._state = False
        print(f"{self._name} is turned off")


class Command(ABC):
    @abstractmethod
    def execute(self):
        pass

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

```
class LightOnCommand(Command):
    def __init__(self, light: Light):
        self._light = light

    def execute(self):
        self._light.turn_on()

    def undo(self):
        self._light.turn_off()


class LightOffCommand(Command):
    def __init__(self, light: Light):
        self._light = light

    def execute(self):
        self._light.turn_off()

    def undo(self):
        self._light.turn_on()


class RemoteControl:
    def __init__(self):
        self._commands = []

    def execute_command(self, command: Command):
        command.execute()
        self._commands.append(command)

    def undo_command(self):
        if len(self._commands) > 0:
            command = self._commands.pop()
            command.undo()


if __name__ == "__main__":
    living_room_light = Light("Living room")
    kitchen_light = Light("Kitchen")

    remote_control = RemoteControl()

    commands = [
        LightOnCommand(living_room_light),
        LightOnCommand(kitchen_light),
        LightOffCommand(living_room_light),
    ]

    for command in commands:
        remote_control.execute_command(command)

    remote_control.undo_command()
    remote_control.undo_command()
```

Описание работы кода:

- Создается класс `Light`, представляющий умную лампу, которую можно включать и выключать.
- Создаются абстрактные классы `Command` и конкретные классы `LightOnCommand`, `LightOffCommand`, реализующие интерфейс `Command` и содержащие реализацию алгоритмов включения и выключения лампы.
- Создается класс `RemoteControl`, представляющий пульт дистанционного управления, который инициирует запросы на включение и выключение ламп.
- В главном модуле создаются экземпляры классов `Light`, `RemoteControl` и конкретных команд.
- Выполняется цикл, в котором вызывается метод `execute_command` для пульта, передавая ему конкретную команду.

- Пульт вызывает метод `execute` у конкретной команды, которая, в свою очередь, вызывает метод `turn_on` или `turn_off` у лампы.
- После выполнения всех команд вызывается метод `undo_command` для пульта, который отменяет последнюю выполненную команду.

Итератор

Порождающий шаблон Итератор (Iterator) позволяет последовательно обходить элементы составного объекта, не раскрывая его внутреннего представления. Этот шаблон обеспечивает гибкость и упрощает взаимодействие между объектами, а также позволяет использовать различные алгоритмы обхода элементов.

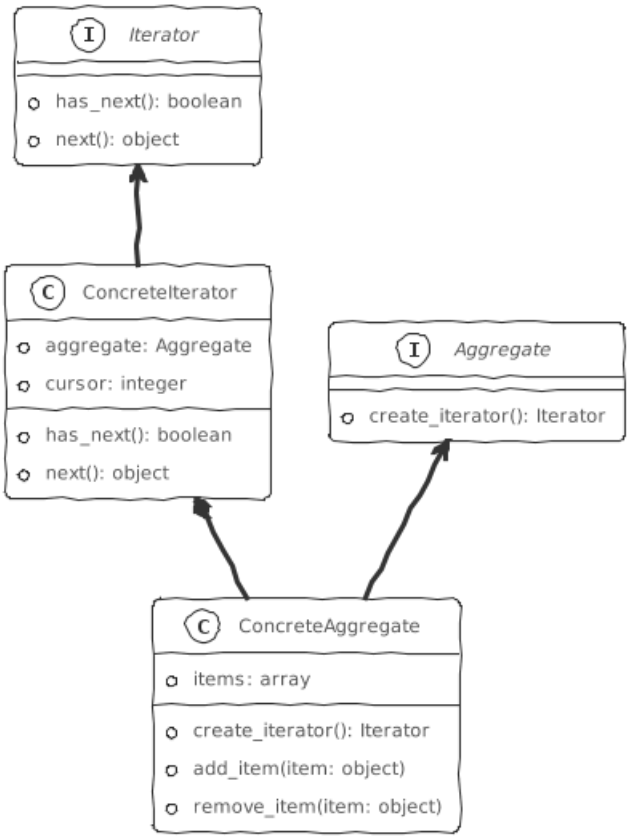
Аналогии из реальной жизни:

1. Библиотека: Представьте, что вы хотите прочитать все книги в библиотеке. Библиотекарь может предоставить вам специальный список, который позволяет вам последовательно переходить от одной книги к другой, не зная, как они хранятся и каталогизируются внутри библиотеки.
2. Музей: В музее могут существовать различные маршруты для осмотра экспонатов. Каждый маршрут представляет собой инкапсулированный алгоритм обхода экспонатов, и вы можете выбрать любой из них, не завися от музея и не зная, как экспонаты расположены внутри.

Принципиальная схема шаблона Итератор в формате PlantUML:

```
@startuml

interface Iterator {
  + has_next(): boolean
  + next(): object
}

class ConcreteIterator {
  + aggregate: Aggregate
  + cursor: integer
  + has_next(): boolean
  + next(): object
}

interface Aggregate {
  + create_iterator(): Iterator
}

class ConcreteAggregate {
  + items: array
  + create_iterator(): Iterator
  + add_item(item: object)
  + remove_item(item: object)
}

Iterator ⟵ ConcreteIterator
Aggregate ⟵ ConcreteAggregate
ConcreteIterator *-- ConcreteAggregate

@enduml
```

Принципиальная схема шаблона Итератор в формате UML:



Описание схемы:

- `Iterator` (Итератор) - интерфейс, определяющий методы для последовательного обхода элементов составного объекта (методы `has_next` и `next`).
- `ConcreteIterator` (КонкретныйИтератор) - конкретный класс, реализующий интерфейс `Iterator` и содержащий реализацию алгоритма обхода элементов.
- `Aggregate` (СоставнойОбъект) - интерфейс, определяющий метод для создания итератора (метод `create_iterator`).
- `ConcreteAggregate` (КонкретныйСоставнойОбъект) - конкретный класс, реализующий интерфейс `Aggregate` и содержащий методы для добавления и удаления элементов.

Абстрактный код реализации шаблона Итератор на языке Python:

```
from abc import ABC, abstractmethod


class Iterator(ABC):
    @abstractmethod
    def has_next(self):
        pass

    @abstractmethod
    def next(self):
        pass


class ConcreteIterator:
    def __init__(self, aggregate):
        self._aggregate = aggregate
        self._cursor = 0

    def has_next(self):
        return self._cursor < len(self._aggregate._items)

    def next(self):
        item = self._aggregate._items[self._cursor]
        self._cursor += 1
        return item
```

```
class Aggregate(ABC):
    @abstractmethod
    def create_iterator(self):
        pass


class ConcreteAggregate:
    def __init__(self):
        self._items = []

    def create_iterator(self):
        return ConcreteIterator(self)

    def add_item(self, item):
        self._items.append(item)

    def remove_item(self, item):
        self._items.remove(item)


if __name__ == "__main__":
    aggregate = ConcreteAggregate()

    aggregate.add_item("Item 1")
    aggregate.add_item("Item 2")
    aggregate.add_item("Item 3")

    iterator = aggregate.create_iterator()

    while iterator.has_next():
        print(iterator.next())
```

Описание работы кода:

- Создаются абстрактные классы `Iterator` и `Aggregate`, и конкретные классы `ConcreteIterator`, `ConcreteAggregate`, реализующие соответствующие интерфейсы.
- В главном модуле создается экземпляр класса `ConcreteAggregate` и добавляются в него несколько элементов.
- Создается экземпляр класса `ConcreteIterator`, передавая ему экземпляр класса `ConcreteAggregate`.
- Выполняется цикл, в котором вызываются методы `has_next` и `next` у итератора для последовательного обхода элементов составного объекта.

Пример кода с реализацией шаблона Итератор на Python, приближенный к реальности:

```
class MenuItem:
    def __init__(self, name, description, price):
        self.name = name
        self.description = description
        self.price = price

    def __str__(self):
        return f"{self.name} ({self.description}) - {self.price} $"


class Menu:
    def __init__(self):
        self._items = []

    def add_item(self, item: MenuItem):
        self._items.append(item)

    def remove_item(self, item: MenuItem):
        self._items.remove(item)
```

```
    def create_iterator(self):
        return MenuIterator(self)


class MenuIterator:
    def __init__(self, menu: Menu):
        self._menu = menu
        self._cursor = 0

    def has_next(self):
        return self._cursor < len(self._menu._items)

    def next(self):
        item = self._menu._items[self._cursor]
        self._cursor += 1
        return item

    def __iter__(self):
        return self

    def __next__(self):
        if not self.has_next():
            raise StopIteration
        return self.next()


class Waiter:
    def __init__(self, menu: Menu):
        self._menu = menu

    def print_menu(self):
        iterator = self._menu.create_iterator()
        while iterator.has_next():
            print(iterator.next())


if __name__ == "__main__":
    menu = Menu()

    menu.add_item(MenuItem("Burger", "with cheese", 5.5))
    menu.add_item(MenuItem("Pizza", "margherita", 7.0))
    menu.add_item(MenuItem("Steak", "medium rare", 12.0))

    waiter = Waiter(menu)
    waiter.print_menu()

    # Использование итератора с циклом for
    iterator = menu.create_iterator()
    for item in iterator:
        print(item)
```

Описание работы кода:

- Создаются классы `MenuItem`, `Menu`, `MenuIterator`, `Waiter`, представляющие меню ресторана, элементы меню, итератор для обхода элементов меню и официанта, который печатает меню.
- В главном модуле создаются экземпляры классов `Menu` и `Waiter`, добавляются в меню несколько элементов.
- Вызывается метод `print_menu` у экземпляра класса `Waiter`, который создает итератор для обхода элементов меню и выводит их на экран.
- Также показано, как можно использовать итератор с циклом `for`.

Посредник

Шаблон Посредник (Mediator) предоставляет способ взаимодействия между объектами, позволяя им общаться друг с другом не напрямую, а через посредника. Это позволяет снизить степень связанности между объектами и упростить их взаимодействие.

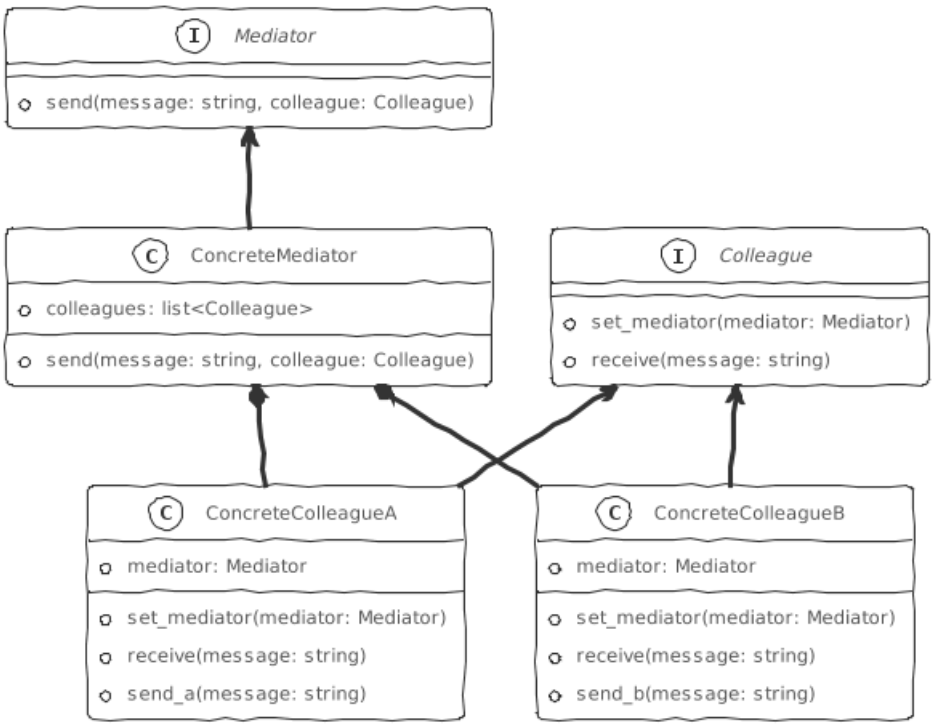
Аналогии из реальной жизни:

1. Авиадиспетчер в аэропорту. Авиадиспетчер выступает в роли посредника между пилотами самолетов, находящихся в воздухе, и персоналом аэропорта на земле. Пилоты и персонал аэропорта не общаются напрямую, а передают информацию через авиадиспетчера.
2. Умный дом. Контроллер умного дома выступает в роли посредника между различными устройствами в доме (кондиционер, освещение, телевизор и т.д.). Устройства не общаются напрямую друг с другом, а передают команды через контроллер.
3. Организация мероприятия: При организации мероприятия (свадьбы, конференции, корпоративного праздника) нужно координировать работу многих участников (поставщиков еды и напитков, музыкантов, фотографов, охраны). Вместо того чтобы каждый участник пытался самостоятельно согласовывать свои действия с другими, они общаются с организатором мероприятия, который координирует их работу и обеспечивает успешное проведение мероприятия.

Принципиальная схема шаблона Посредник в формате PlantUML:

```
@startuml

interface Mediator {
  + send(message: string, colleague: Colleague)
}

class ConcreteMediator {
  + colleagues: list<Colleague>
  + send(message: string, colleague: Colleague)
}

interface Colleague {
  + set_mediator(mediator: Mediator)
  + receive(message: string)
}

class ConcreteColleagueA {
  + mediator: Mediator
  + set_mediator(mediator: Mediator)
  + receive(message: string)
  + send_a(message: string)
}

class ConcreteColleagueB {
  + mediator: Mediator
  + set_mediator(mediator: Mediator)
  + receive(message: string)
  + send_b(message: string)
}

Mediator <-- ConcreteMediator
Colleague <-- ConcreteColleagueA
Colleague <-- ConcreteColleagueB
ConcreteMediator *-- ConcreteColleagueA
ConcreteMediator *-- ConcreteColleagueB

@enduml
```

Принципиальная схема шаблона Посредник в формате UML:



Описание схемы:

- `Mediator` (Посредник) - интерфейс, определяющий метод для отправки сообщений между объектами (метод `send`).
- `ConcreteMediator` (КонкретныйПосредник) - конкретный класс, реализующий интерфейс `Mediator` и содержащий реализацию алгоритма управления взаимодействием между объектами.
- `Colleague` (Коллега) - интерфейс, определяющий методы для установки связи с посредником и получения сообщений (методы `set_mediator` и `receive`).
- `ConcreteColleagueA` и `ConcreteColleagueB` (КонкретныйКоллегаA и КонкретныйКоллегаB) - конкретные классы, реализующие интерфейс `Colleague` и содержащие реализацию алгоритмов отправки и получения сообщений.

Абстрактный код реализации шаблона Посредник на языке Python:

```
from abc import ABC, abstractmethod


class Mediator(ABC):

    @abstractmethod
    def send(self, message, colleague):
        pass


class ConcreteMediator(Mediator):

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

    def add_colleague(self, colleague):
        self._colleagues.append(colleague)

    def send(self, message, colleague):
        for c in self._colleagues:
            if c != colleague:
                c.receive(message)


class Colleague(ABC):

    @abstractmethod
    def receive(self, message):
```

```
        pass


class ConcreteColleague1(Colleague):

    def __init__(self, mediator):
        self._mediator = mediator
        mediator.add_colleague(self)

    def send(self, message):
        self._mediator.send(message, self)

    def receive(self, message):
        print(f'ConcreteColleague1 received: {message}')


class ConcreteColleague2(Colleague):

    def __init__(self, mediator):
        self._mediator = mediator
        mediator.add_colleague(self)

    def send(self, message):
        self._mediator.send(message, self)

    def receive(self, message):
        print(f'ConcreteColleague2 received: {message}')


if __name__ == '__main__':
    mediator = ConcreteMediator()
    colleague1 = ConcreteColleague1(mediator)
    colleague2 = ConcreteColleague2(mediator)
    colleague1.send('Hello')
    colleague2.send('Hi')
```

Описание кода:

- `Mediator` - абстрактный класс, определяющий метод `send`.
- `ConcreteMediator` - конкретный посредник, реализующий абстрактный класс `Mediator`.
- `Colleague` - абстрактный класс, определяющий метод `receive`.
- `ConcreteColleague1` и `ConcreteColleague2` - конкретные коллеги, реализующие абстрактный класс `Colleague`.
- В методе `send` конкретного посредника передается сообщение и коллега, которому сообщение не предназначено. Метод `send` передает сообщение всем коллегам, кроме того, которому сообщение не предназначено.
- В коде запуска создается экземпляр класса `ConcreteMediator`, экземпляры классов `ConcreteColleague1` и `ConcreteColleague2`, и происходит обмен сообщениями между коллегами через посредника.

Пример кода с реализацией шаблона Посредник на Python, приближенный к реальности:

```
from abc import ABC, abstractmethod


class SmartHomeController:

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

    def add_device(self, device):
        self._devices.append(device)

    def send_command(self, command, device):
        for d in self._devices:
            if d != device:
                d.receive_command(command)
```

```
class Device(ABC):

    def __init__(self, controller):
        self._controller = controller
        controller.add_device(self)

    def send_command(self, command):
        self._controller.send_command(command, self)

    @abstractmethod
    def receive_command(self, command):
        pass


class AirConditioner(Device):

    def __init__(self, controller):
        super().__init__(controller)

    def receive_command(self, command):
        if command == 'on':
            print('Air conditioner is on')
        elif command == 'off':
            print('Air conditioner is off')


class Light(Device):

    def __init__(self, controller):
        super().__init__(controller)

    def receive_command(self, command):
        if command == 'on':
            print('Light is on')
        elif command == 'off':
            print('Light is off')


if __name__ == '__main__':
    controller = SmartHomeController()
    air_conditioner = AirConditioner(controller)
    light = Light(controller)
    air_conditioner.send_command('on')
    light.send_command('off')
```

Описание кода:

- `SmartHomeController` - конкретный посредник, реализующий взаимодействие между устройствами умного дома.
- `Device` - абстрактный класс, определяющий методы `send_command` и `receive_command`.
- `AirConditioner` и `Light` - конкретные устройства умного дома, реализующие абстрактный класс `Device`.
- В методе `send_command` конкретного посредника передается команда и устройство, которому команда не предназначена. Метод `send_command` передает команду всем устройствам, кроме того, которому команда не предназначена.
- В коде запуска создается экземпляр класса `SmartHomeController`, экземпляры классов `AirConditioner` и `Light`, и происходит обмен командами между устройствами через посредника.

Ещё один пример кода с реализацией шаблона Посредник на Python, приближенный к реальности:

```
from abc import ABC, abstractmethod


class User:
```

```
    def __init__(self, name: str):
        self.name = name
        self._mediator = None

    def set_mediator(self, mediator):
        self._mediator = mediator

    def send(self, message: str):
        self._mediator.send(message, self)

    def receive(self, message: str):
        print(f"{self.name}: {message}")


class Admin:
    def __init__(self, name: str):
        self.name = name
        self._mediator = None

    def set_mediator(self, mediator):
        self._mediator = mediator

    def send(self, message: str):
        self._mediator.send(message, self)

    def receive(self, message: str):
        print(f"{self.name} (admin): {message}")

    def ban_user(self, user: User):
        self._mediator.ban_user(user)


class Mediator(ABC):
    @abstractmethod
    def send(self, message: str, sender: User | Admin):
        pass

    @abstractmethod
    def add_user(self, user: User):
        pass

    @abstractmethod
    def add_admin(self, admin: Admin):
        pass

    @abstractmethod
    def ban_user(self, user: User):
        pass


class ChatMediator(Mediator):
    def __init__(self):
        self._users = []
        self._admins = []

    def send(self, message: str, sender: User | Admin):
        if isinstance(sender, User):
            for u in self._users:
                if u ≠ sender:
                    u.receive(message)
            for a in self._admins:
                a.receive(message)
        elif isinstance(sender, Admin):
            for u in self._users:
                u.receive(message)
            for a in self._admins:
```

```
                if a != sender:
                    a.receive(message)

    def add_user(self, user: User):
        self._users.append(user)

    def add_admin(self, admin: Admin):
        self._admins.append(admin)

    def ban_user(self, user: User):
        self._users.remove(user)


if __name__ == "__main__":
    mediator = ChatMediator()

    user_1 = User("Alice")
    user_2 = User("Bob")
    admin_1 = Admin("Charlie")

    mediator.add_user(user_1)
    mediator.add_user(user_2)
    mediator.add_admin(admin_1)

    user_1.set_mediator(mediator)
    user_2.set_mediator(mediator)
    admin_1.set_mediator(mediator)

    user_1.send("Hi, everyone!")
    user_2.send("Hello, Alice!")
    admin_1.send("Welcome to our chat!")
    admin_1.ban_user(user_2)
    user_1.send("Bye, everyone!")
```

Описание работы кода:

- Создаются классы `User`, `Admin`, `Mediator`, `ChatMediator`, представляющие пользователей, администраторов, посредника и чат-комнату.
- В главном модуле создается экземпляр класса `ChatMediator` и добавляются в него экземпляры классов `User` и `Admin`.
- Устанавливается связь между экземплярами классов `User` и `Admin` и экземпляром класса `ChatMediator`.
- Выполняется отправка сообщений между экземплярами классов `User` и `Admin` с помощью посредника, а также блокировка пользователя.

Снимок

Шаблон Снимок (Memento) предоставляет способ сохранения и восстановления внутреннего состояния объекта без нарушения инкапсуляции. Это позволяет избежать нежелательного изменения состояния объекта и упростить его использование. Этот шаблон обеспечивает гибкость и упрощает взаимодействие между объектами, а также позволяет создавать "точки сохранения" и "точки возврата" в программном обеспечении.

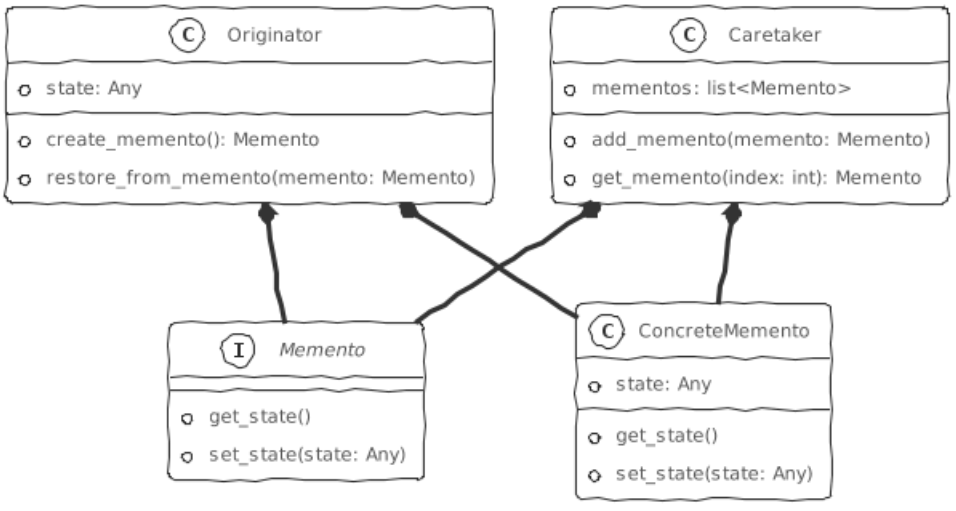
Аналогии из реальной жизни:

1. Сохранение в компьютерной игре. Игрок может сохранить текущее состояние игры в файл и позже восстановить его, не зная, как именно реализовано сохранение и восстановление внутри игры.
2. История версий в системе контроля версий (например, Git). Разработчик может сохранить текущее состояние проекта в системе контроля версий и позже восстановить его, не зная, как именно реализовано сохранение и восстановление внутри системы.
3. Текстовый редактор: В текстовом редакторе есть функции "отменить" и "вернуть", которые позволяют отменять и возвращать последние изменения в тексте. Вместо того чтобы пытаться самостоятельно сохранить и восстановить все данные о ваших изменениях, вы используете специальные инструменты, предоставляемые редактором, которые обеспечивают сохранность и целостность вашего текста.

Принципиальная схема шаблона Снимок в формате PlantUML:

```
@startuml

interface Memento {
  + get_state()
  + set_state(state: Any)
}

class ConcreteMemento {
  + state: Any
  + get_state()
  + set_state(state: Any)
}

class Originator {
  + state: Any
  + create_memento(): Memento
  + restore_from_memento(memento: Memento)
}

class Caretaker {
  + mementos: list<Memento>
  + add_memento(memento: Memento)
  + get_memento(index: int): Memento
}

Originator *-- Memento
Originator *-- ConcreteMemento
Caretaker *-- Memento
Caretaker *-- ConcreteMemento

@enduml
```

Принципиальная схема шаблона Снимок в формате UML:



Описание схемы:

- `Memento` (Снимок) - интерфейс, определяющий методы для сохранения и восстановления внутреннего состояния объекта (методы `get_state` и `set_state`).
- `ConcreteMemento` (КонкретныйСнимок) - конкретный класс, реализующий интерфейс `Memento` и содержащий реализацию алгоритма сохранения и восстановления внутреннего состояния объекта.
- `Originator` (Инициатор) - класс, содержащий внутреннее состояние, которое нужно сохранять и восстанавливать. Инициатор создает снимок своего состояния и предоставляет его хранителю.
- `Caretaker` (Хранитель) - класс, ответственный за хранение снимков и управление ими. Хранитель не может изменять внутреннее состояние снимков, он может только хранить их и предоставлять инициатору для восстановления.

Абстрактный код реализации шаблона Снимок на языке Python:

```python
from abc import ABC, abstractmethod
from typing import Any


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

    @abstractmethod
    def set_state(self, state: Any):
        pass


class ConcreteMemento:
    def __init__(self, state: Any):
        self._state = state

    def get_state(self):
        return self._state

    def set_state(self, state: Any):
        self._state = state


class Originator:
    def __init__(self, state: Any):
        self._state = state

    def create_memento(self):
        return ConcreteMemento(self._state)

    def restore_from_memento(self, memento: Memento):
        self._state = memento.get_state()

    def set_state(self, state: Any):
        self._state = state

    def get_state(self):
        return self._state


class Caretaker:
    def __init__(self):
        self._mementos = []

    def add_memento(self, memento: Memento):
        self._mementos.append(memento)

    def get_memento(self, index: int):
        return self._mementos[index]


if __name__ == "__main__":
    originator = Originator("Initial state")
    caretaker = Caretaker()

    print(f"Originator state: {originator.get_state()}")

    caretaker.add_memento(originator.create_memento())

    originator.set_state("State 1")
    print(f"Originator state: {originator.get_state()}")

    caretaker.add_memento(originator.create_memento())
```

```
    originator.set_state("State 2")
    print(f"Originator state: {originator.get_state()}")

    caretaker.add_memento(originator.create_memento())

    originator.restore_from_memento(caretaker.get_memento(1))
    print(f"Originator state: {originator.get_state()}")
```

Описание работы кода:

- Создаются абстрактные классы `Memento` и конкретный класс `ConcreteMemento`, реализующий интерфейс `Memento`.
- Создается класс `Originator`, содержащий внутреннее состояние, которое нужно сохранять и восстанавливать.
- Создается класс `Caretaker`, ответственный за хранение снимков и управление ими.
- В главном модуле создаются экземпляры классов `Originator`, `Caretaker` и выполняются операции сохранения и восстановления внутреннего состояния объекта с помощью снимков.

Пример кода с реализацией шаблона Снимок на Python, приближенный к реальности:

```
class TextEditor:

    def __init__(self):
        self._content = ''
        self._mementos = []

    def type(self, text):
        self._content += text
        self._mementos.append(TextEditorMemento(self._content))

    def undo(self):
        if len(self._mementos) > 0:
            self._content = self._mementos.pop().get_content()

    def get_content(self):
        return self._content


class TextEditorMemento:

    def __init__(self, content):
        self._content = content

    def get_content(self):
        return self._content


if __name__ == '__main__':
    editor = TextEditor()
    editor.type('Hello')
    print(f'Editor content: {editor.get_content()}')
    editor.type(' World')
    print(f'Editor content: {editor.get_content()}')
    editor.undo()
    print(f'Editor content: {editor.get_content()}')
```

Описание кода:

- `TextEditor` - объект, реализующий простой текстовый редактор с возможностью отмены действий.
- `TextEditorMemento` - конкретный снимок, сохраняющий состояние `TextEditor`.
- В коде запуска создается экземпляр класса `TextEditor`, и происходит ввод текста и отмена действий с помощью снимка.

Наблюдатель

Шаблон Наблюдатель (Observer) предоставляет способ оповещения подписчиков об изменениях в объекте, который они наблюдают. Это позволяет избежать жесткой связи между объектами и упростить их взаимодействие.

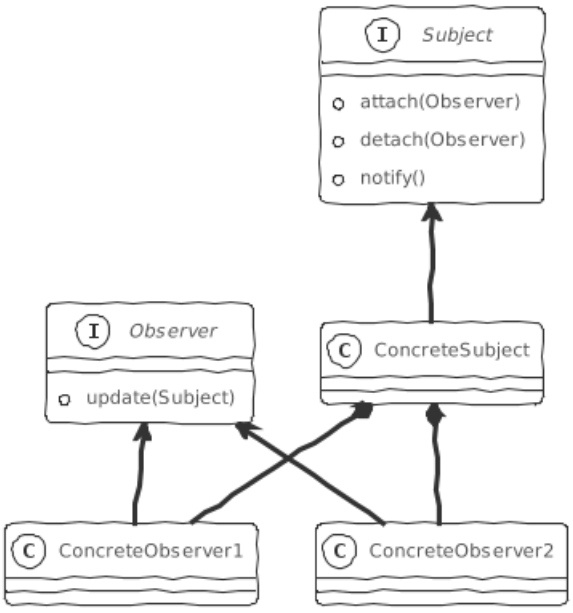
Аналогии из реальной жизни:

1. Подписка на новости в социальной сети. Пользователь подписывается на новости от определенного автора, и получает уведомления об новых публикациях.
2. Подписка на обновления курса валют. Пользователь подписывается на обновления курса валют, и получает уведомления об изменениях.

Принципиальная схема шаблона Наблюдатель в формате PlantUML:

```
@startuml

interface Subject {
  + attach(Observer)
  + detach(Observer)
  + notify()
}

interface Observer {
  + update(Subject)
}

Subject ⟵ ConcreteSubject
Observer ⟵ ConcreteObserver1
Observer ⟵ ConcreteObserver2
ConcreteSubject *-- ConcreteObserver1
ConcreteSubject *-- ConcreteObserver2

@enduml
```

Принципиальная схема шаблона Наблюдатель в формате UML:



Описание схемы:

- `Subject` (Издатель) - интерфейс, определяющий методы для подписки и отписки наблюдателей, а также метод для уведомления наблюдателей о событии.
- `ConcreteSubject` (КонкретныйИздатель) - конкретный класс, реализующий интерфейс `Subject` и содержащий данные, которые будут обновляться в ответ на события.
- `Observer` (Наблюдатель) - интерфейс, определяющий метод для обновления состояния наблюдателя в ответ на событие.
- `ConcreteObserver1` и `ConcreteObserver2` (КонкретныйНаблюдатель1 и КонкретныйНаблюдатель2) - конкретные классы, реализующие интерфейс `Observer` и содержащие реализацию алгоритмов обновления состояния.

Абстрактный код реализации шаблона Наблюдатель на языке Python:

```
from abc import ABC, abstractmethod


class Subject(ABC):

    @abstractmethod
    def attach(self, observer):
        pass

    @abstractmethod
    def detach(self, observer):
        pass

    @abstractmethod
    def notify(self):
        pass


class ConcreteSubject(Subject):

    def __init__(self):
        self._observers = []
        self._state = None

    def attach(self, observer):
        self._observers.append(observer)

    def detach(self, observer):
        self._observers.remove(observer)

    def notify(self):
        for observer in self._observers:
            observer.update(self)

    def set_state(self, state):
        self._state = state
        self.notify()

    def get_state(self):
        return self._state


class Observer(ABC):

    @abstractmethod
    def update(self, subject):
        pass


class ConcreteObserver(Observer):

    def __init__(self, subject):
        self._subject = subject
        self._subject.attach(self)

    def update(self, subject):
        if subject == self._subject:
            sbj_state = self._subject.get_state()
            msg = f'ConcreteObserver: Subject state changed to {sbj_state}'
            print(msg)


if __name__ == '__main__':
    subject = ConcreteSubject()
```

```
    observer1 = ConcreteObserver(subject)
    observer2 = ConcreteObserver(subject)
    subject.set_state('State 1')
    subject.set_state('State 2')
    subject.detach(observer1)
    subject.set_state('State 3')
```

Описание кода:

- `Subject` - абстрактный класс, определяющий методы `attach`, `detach` и `notify`.
- `ConcreteSubject` - объект, наблюдаемый подписчиками.
- `Observer` - абстрактный класс, определяющий метод `update`.
- `ConcreteObserver` - конкретный подписчик, реализующий абстрактный класс `Observer`.
- В коде запуска создается экземпляр класса `ConcreteSubject`, экземпляры класса `ConcreteObserver`, и происходит оповещение подписчиков об изменениях в объекте, который они наблюдают.

Пример кода с реализацией шаблона Наблюдатель на Python, приближенный к реальности:

```
from abc import ABC, abstractmethod


class WeatherStation:

    def __init__(self):
        self._observers = []
        self._temperature = None
        self._humidity = None

    def attach(self, observer):
        self._observers.append(observer)

    def detach(self, observer):
        self._observers.remove(observer)

    def notify(self):
        for observer in self._observers:
            observer.update(self)

    def set_measurements(self, temperature, humidity):
        self._temperature = temperature
        self._humidity = humidity
        self.notify()

    def get_temperature(self):
        return self._temperature

    def get_humidity(self):
        return self._humidity


class Observer(ABC):

    @abstractmethod
    def update(self, weather_station):
        pass


class Display(Observer):

    def __init__(self, weather_station):
        self._weather_station = weather_station
        self._weather_station.attach(self)

    def update(self, weather_station):
        if weather_station == self._weather_station:
```

```
            t = self._weather_station.get_temperature()
            h = self._weather_station.get_humidity()
            print(f'Display: Temperature: {t}, Humidity: {h}')


class Statistician(Observer):

    def __init__(self, weather_station):
        self._weather_station = weather_station
        self._temperature_sum = 0
        self._temperature_count = 0
        self._weather_station.attach(self)

    def update(self, weather_station):
        if weather_station == self._weather_station:
            self._temperature_sum += self._weather_station.get_temperature()
            self._temperature_count += 1
            avg_t = self._temperature_sum / self._temperature_count
            print(f'Statistician: Average temperature: {avg_t}')


if __name__ == '__main__':
    weather_station = WeatherStation()
    display = Display(weather_station)
    statistician = Statistician(weather_station)
    weather_station.set_measurements(20, 60)
    weather_station.set_measurements(22, 55)
    weather_station.set_measurements(25, 45)
```

Описание кода:

- `WeatherStation` - объект, наблюдаемый подписчиками.
- `Observer` - абстрактный класс, определяющий метод `update`.
- `Display` - конкретный подписчик, реализующий абстрактный класс `Observer`.
- `Statistician` - конкретный подписчик, реализующий абстрактный класс `Observer`.
- В коде запуска создается экземпляр класса `WeatherStation`, экземпляры классов `Display` и `Statistician`, и происходит оповещение подписчиков об изменениях в объекте, который они наблюдают.

Состояние

Порождающий шаблон Состояние (State) позволяет объекту менять своё поведение в зависимости от внутреннего состояния. Этот шаблон предоставляет способ разделить алгоритм от представления объектов, что повышает гибкость и упрощает поддержку программного обеспечения. Кроме того, шаблон Состояние позволяет избежать больших конструкций с множеством условий, которые могут быть сложны для понимания и поддержки.

Аналогии из реальной жизни:

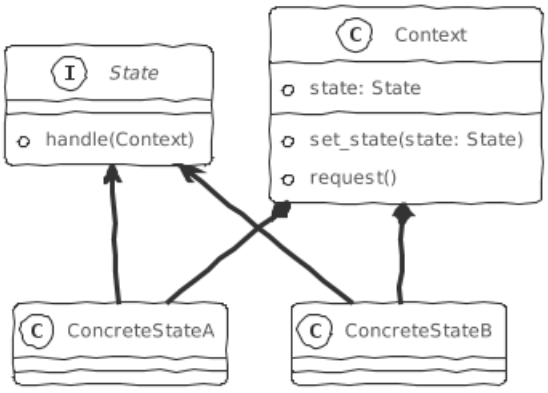
1. Автомат: Представьте, что у вас есть автомат, продающий напитки. В зависимости от того, сколько монет внесено, автомат может находиться в разных состояниях (ожидание монет, ожидание выбора напитка, выдача напитка). Каждое состояние определяет, какие действия автомат может выполнять в данный момент и как он будет реагировать на внешние события (внесение монет, нажатие кнопки выбора напитка).
2. Термостат: Термостат используется для регулирования температуры в помещении. В зависимости от текущей температуры и установленного значения термостат может находиться в двух состояниях (включен, выключен). В состоянии "включен" термостат подает ток на нагревательный элемент, чтобы повысить температуру, а в состоянии "выключен" прекращает подачу тока, чтобы температура не превышала установленного значения.

Принципиальная схема шаблона Состояние в формате PlantUML:

```
@startuml

interface State {
  + handle(Context)
}
```

```
class Context {
  + state: State
  + set_state(state: State)
  + request()
}

State ⟵ ConcreteStateA
State ⟵ ConcreteStateB
Context *-- ConcreteStateA
Context *-- ConcreteStateB

@enduml
```

Принципиальная схема шаблона Состояние в формате UML:



Описание схемы:

- `State` (Состояние) - интерфейс, определяющий методы для обработки событий в контексте.
- `ConcreteStateA` и `ConcreteStateB` (КонкретноеСостояниеА и КонкретноеСостояниеВ) - конкретные классы, реализующие интерфейс `State` и содержащие реализацию алгоритмов обработки событий для конкретных состояний.
- `Context` (Контекст) - класс, содержащий текущее состояние и предоставляющий интерфейс для взаимодействия с внешним миром. Контекст также может изменять текущее состояние, вызывая метод `set_state`.

Абстрактный код реализации шаблона Состояние на языке Python:

```python
from abc import ABC, abstractmethod

class State(ABC):
    @abstractmethod
    def handle(self, context):
        pass

class ConcreteStateA(State):
    def handle(self, context):
        print("ConcreteStateA handling")
        context.set_state(ConcreteStateB())

class ConcreteStateB(State):
    def handle(self, context):
        print("ConcreteStateB handling")
        context.set_state(ConcreteStateA())

class Context:
    def __init__(self, state: State = None):
        self._state = state or ConcreteStateA()

    def set_state(self, state: State):
        self._state = state
```

```
    def request(self):
        self._state.handle(self)

if __name__ == "__main__":
    context = Context()

    for _ in range(5):
        context.request()
```

Описание работы кода:

- Создаются абстрактные классы `State` и конкретные классы `ConcreteStateA`, `ConcreteStateB`, реализующие интерфейс `State` и содержащие реализацию алгоритмов обработки событий для конкретных состояний.
- Создается класс `Context`, содержащий текущее состояние и предоставляющий интерфейс для взаимодействия с внешним миром.
- В главном модуле создается экземпляр класса `Context` и выполняется цикл, в котором вызывается метод `request` для обработки событий.
- Текущее состояние обрабатывает событие, вызывая метод `handle`, и может изменить текущее состояние, вызывая метод `set_state`.

Пример кода с реализацией шаблона Состояние на Python, приближенный к реальности:

```
from abc import ABC, abstractmethod


class VendingMachineState(ABC):
    @abstractmethod
    def insert_coin(self, context):
        pass

    @abstractmethod
    def select_drink(self, context):
        pass

    @abstractmethod
    def dispense_drink(self, context):
        pass


class IdleState(VendingMachineState):
    def insert_coin(self, context):
        print("Coin inserted")
        context.set_state(SelectionState())

    def select_drink(self, context):
        pass

    def dispense_drink(self, context):
        pass


class SelectionState(VendingMachineState):
    def insert_coin(self, context):
        pass

    def select_drink(self, context, drink_name):
        print(f"{drink_name} selected")
        context.set_state(DispenseState(drink_name))

    def dispense_drink(self, context):
        pass
```

```
class DispenseState(VendingMachineState):
    def __init__(self, drink_name):
        self._drink_name = drink_name

    def insert_coin(self, context):
        pass

    def select_drink(self, context):
        pass

    def dispense_drink(self, context):
        print(f"Dispensing {self._drink_name}")
        context.set_state(IdleState())


class VendingMachine:
    def __init__(self, state: VendingMachineState = None):
        self._state = state or IdleState()

    def set_state(self, state: VendingMachineState):
        self._state = state

    def insert_coin(self):
        self._state.insert_coin(self)

    def select_drink(self, drink_name):
        self._state.select_drink(self, drink_name)

    def dispense_drink(self):
        self._state.dispense_drink(self)


if __name__ == "__main__":
    machine = VendingMachine()

    machine.insert_coin()
    machine.select_drink("Coke")
    machine.dispense_drink()

    machine.insert_coin()
    machine.select_drink("Sprite")
    machine.dispense_drink()
```

Описание работы кода:

- Создаются абстрактные классы `VendingMachineState` и конкретные классы `IdleState`, `SelectionState`, `DispenseState`, реализующие интерфейс `VendingMachineState` и содержащие реализацию алгоритмов обработки событий для конкретных состояний автомата.
- Создается класс `VendingMachine`, содержащий текущее состояние и предоставляющий интерфейс для взаимодействия с пользователем (вставка монеты, выбор напитка, выдача напитка).
- В главном модуле создается экземпляр класса `VendingMachine` и выполняются действия по вставке монеты, выбору напитка и выдаче напитка.
- Текущее состояние автомата обрабатывает событие, вызывая соответствующий метод, и может изменить текущее состояние, вызывая метод `set_state`.

Стратегия

Шаблон Стратегия (Strategy) предоставляет способ выбора алгоритма для решения задачи во время выполнения программы. Это позволяет избежать жесткой привязки к конкретному алгоритму и упростить код.

Аналогии из реальной жизни:

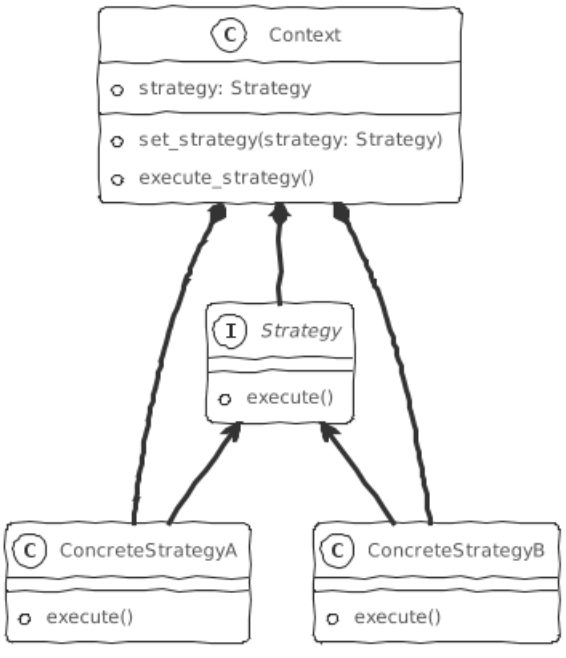
1. Способы сортировки. В зависимости от размера и типа данных, а также от требуемой скорости и памяти, можно выбрать различные алгоритмы сортировки (пузырьковая, быстрая, слиянием и т.д.).

2. Способы передвижения: Представьте, что вы хотите добраться до определенного места. Вы можете выбрать из нескольких способов передвижения, таких как пешком, на велосипеде, на автобусе или на машине. Каждый способ передвижения имеет свои преимущества и недостатки, и вы выбираете тот, который лучше всего подходит для вашей ситуации.
3. Выбор игровой стратегии: В стратегических играх, таких как шахматы или риск, игроки выбирают различные стратегии и тактики, чтобы одержать победу над своими противниками. Выбор стратегии зависит от многих факторов, таких как текущая ситуация на игровом поле, ресурсы, доступные игроку, и стиль игры противника.
4. Автомобильные маршруты: Представьте, что вы едете на автомобиле из одного города в другой. Вы можете выбрать различные маршруты, например, самый короткий, самый быстрый или самый экономичный. Каждый маршрут представляет собой определенную стратегию, которую вы можете выбрать в зависимости от ваших потребностей и предпочтений.

Принципиальная схема шаблона Стратегия в формате PlantUML:

```
@startuml

interface Strategy {
  + execute()
}

class ConcreteStrategyA {
  + execute()
}

class ConcreteStrategyB {
  + execute()
}

class Context {
  + strategy: Strategy
  + set_strategy(strategy: Strategy)
  + execute_strategy()
}

Strategy <|-- ConcreteStrategyA
Strategy <|-- ConcreteStrategyB
Context *-- Strategy
Context *-- ConcreteStrategyA
Context *-- ConcreteStrategyB

@enduml
```

Принципиальная схема шаблона Стратегия в формате UML:



Описание схемы:

- `Strategy` (Стратегия) - интерфейс, определяющий методы для выполнения определенной операции.
- `ConcreteStrategyA` и `ConcreteStrategyB` (КонкретнаяСтратегияА и КонкретнаяСтратегияВ) - конкретные классы, реализующие интерфейс `Strategy` и содержащие реализацию алгоритмов для выполнения операции.
- `Context` (Контекст) - класс, который использует стратегию для выполнения операции. Контекст предоставляет интерфейс для клиентов, которые хотят использовать различные стратегии.

Абстрактный код реализации шаблона Стратегия на языке Python:

```
from abc import ABC, abstractmethod


class Strategy(ABC):

    @abstractmethod
    def algorithm(self):
        pass


class ConcreteStrategyA(Strategy):

    def algorithm(self):
        print('ConcreteStrategyA algorithm.')


class ConcreteStrategyB(Strategy):

    def algorithm(self):
        print('ConcreteStrategyB algorithm.')


class Context:

    def __init__(self):
        self._strategy = None

    def set_strategy(self, strategy):
        self._strategy = strategy

    def execute_strategy(self):
        self._strategy.algorithm()


if __name__ == '__main__':
    context = Context()
    context.set_strategy(ConcreteStrategyA())
    context.execute_strategy()
    context.set_strategy(ConcreteStrategyB())
    context.execute_strategy()
```

Описание кода:

- `Strategy` - абстрактный класс, определяющий метод `algorithm`.
- `ConcreteStrategyA` и `ConcreteStrategyB` - конкретные стратегии, реализующие абстрактный класс `Strategy`.
- `Context` - объект, выполняющий выбранную стратегию.
- В коде запуска создается экземпляр класса `Context`, устанавливается начальная стратегия `ConcreteStrategyA` и вызывается метод `execute_strategy`. Затем устанавливается стратегия `ConcreteStrategyB` и снова вызывается метод `execute_strategy`. В зависимости от текущей стратегии `Context` выполняет соответствующий алгоритм.

Пример кода с реализацией шаблона Стратегия на Python, приближенный к реальности:

```python
from abc import ABC, abstractmethod


class SortingStrategy(ABC):
    @abstractmethod
    def sort(self, data: list):
        pass


class BubbleSort(SortingStrategy):
    def sort(self, data: list) -> list:
        n = len(data)
        for i in range(n - 1):
            for j in range(0, n - i - 1):
                if data[j] > data[j + 1]:
                    data[j], data[j + 1] = data[j + 1], data[j]
        return data


class MergeSort(SortingStrategy):
    def merge(self, left: list, right: list) -> list:
        result = []
        i = j = 0
        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                result.append(left[i])
                i += 1
            else:
                result.append(right[j])
                j += 1
        result.extend(left[i:])
        result.extend(right[j:])
        return result

    def sort(self, data: list) -> list:
        if len(data) ≤ 1:
            return data
        mid = len(data) // 2
        left = self.sort(data[:mid])
        right = self.sort(data[mid:])
        return self.merge(left, right)


class Sorter:
    def __init__(self, strategy: SortingStrategy):
        self._strategy = strategy

    def set_strategy(self, strategy: SortingStrategy):
        self._strategy = strategy

    def sort_data(self, data: list):
        return self._strategy.sort(data)


if __name__ == "__main__":
    sorter = Sorter(BubbleSort())
    data = [5, 3, 8, 4, 9, 1, 7, 2, 6]
    print("Bubble sort:", sorter.sort_data(data))

    sorter.set_strategy(MergeSort())
    print("Merge sort:", sorter.sort_data(data))
```

Описание кода:

- `Sorter` - объект, выполняющий сортировку данных.

- `SortingStrategy` - абстрактный класс, определяющий метод `sort`.
- `BubbleSort` и `MergeSort` - конкретные стратегии, реализующие абстрактный класс `Strategy`.
- В коде запуска создается экземпляр класса `Sorter`, устанавливается начальная стратегия `BubbleSort` и вызывается метод `sort` для списка данных. Затем устанавливается стратегия `MergeSort` и снова вызывается метод `sort` для того же списка данных. В зависимости от текущей стратегии `Sorter` выполняет сортировку данных различными алгоритмами.

Шаблонный метод

Шаблон Шаблонный метод (Template Method) предоставляет способ определения общих шагов алгоритма в базовом классе и их конкретной реализации в производных классах. Это позволяет избежать дублирования кода и упростить его поддержку.

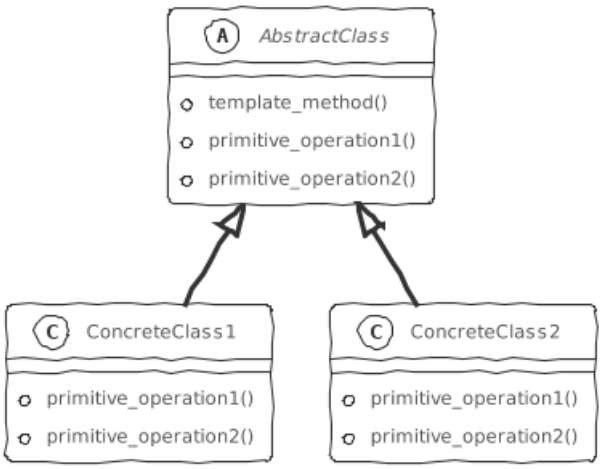
Аналогии из реальной жизни:

1. Приготовление еды. Общие шаги приготовления еды (подготовка ингредиентов, приготовление, подача) могут быть одинаковыми для разных блюд, но конкретная реализация этих шагов (какие ингредиенты использовать, как приготовить, как подать) будет различаться.
2. Построение дома. Общие шаги построения дома (проектирование, подготовка строительной площадки, возведение стен, крыши, установка коммуникаций) могут быть одинаковыми для разных домов, но конкретная реализация этих шагов (какой проект использовать, какие материалы, как возводить стены, как устанавливать коммуникации) будет различаться.

Принципиальная схема шаблона Шаблонный метод в формате PlantUML:

```
@startuml

abstract class AbstractClass {
  + template_method()
  + primitive_operation1()
  + primitive_operation2()
}

class ConcreteClass1 {
  + primitive_operation1()
  + primitive_operation2()
}

class ConcreteClass2 {
  + primitive_operation1()
  + primitive_operation2()
}

AbstractClass <|-- ConcreteClass1
AbstractClass <|-- ConcreteClass2

@enduml
```

Принципиальная схема шаблона Шаблонный метод в формате UML:



Описание схемы:

- `AbstractClass` - абстрактный класс, определяющий общие шаги алгоритма в виде шаблонного метода `template_method` и конкретные реализации этих шагов в виде примитивных операций `primitive_operation1` и `primitive_operation2`.
- `ConcreteClass1` и `ConcreteClass2` - производные классы, реализующие конкретные реализации примитивных операций `primitive_operation1` и `primitive_operation2`.
- Стрелка между `AbstractClass` и `ConcreteClass1`, `ConcreteClass2` означает, что `ConcreteClass1` и `ConcreteClass2` являются производными классами от `AbstractClass`.

Абстрактный код реализации шаблона Шаблонный метод на языке Python:

```
from abc import ABC, abstractmethod


class AbstractClass(ABC):

    @abstractmethod
    def primitive_operation1(self):
        pass

    @abstractmethod
    def primitive_operation2(self):
        pass

    def template_method(self):
        self.primitive_operation1()
        self.primitive_operation2()


class ConcreteClass1(AbstractClass):

    def primitive_operation1(self):
        print('ConcreteClass1 primitive operation 1')

    def primitive_operation2(self):
        print('ConcreteClass1 primitive operation 2')


class ConcreteClass2(AbstractClass):

    def primitive_operation1(self):
        print('ConcreteClass2 primitive operation 1')

    def primitive_operation2(self):
        print('ConcreteClass2 primitive operation 2')
```

```
if __name__ == '__main__':
    cc1 = ConcreteClass1()
    cc1.template_method()
    cc2 = ConcreteClass2()
    cc2.template_method()
```

Описание кода:

- `AbstractClass` - абстрактный класс, определяющий общие шаги алгоритма в виде шаблонного метода `template_method` и конкретные реализации этих шагов в виде примитивных операций `primitive_operation1` и `primitive_operation2`.
- `ConcreteClass1` и `ConcreteClass2` - производные классы, реализующие конкретные реализации примитивных операций `primitive_operation1` и `primitive_operation2`.
- В коде запуска создаются экземпляры классов `ConcreteClass1` и `ConcreteClass2` и вызывается шаблонный метод `template_method`. В зависимости от типа объекта, вызывающего метод, будут выполняться разные конкретные реализации примитивных операций.

Пример кода с реализацией шаблона Шаблонный метод на Python, приближенный к реальности:

```
from abc import ABC, abstractmethod


class Game(ABC):

    @abstractmethod
    def initialize(self):
        pass

    @abstractmethod
    def start_play(self):
        pass

    @abstractmethod
    def end_play(self):
        pass

    def play(self):
        self.initialize()
        self.start_play()
        self.end_play()


class Cricket(Game):

    def initialize(self):
        print('Cricket Game Initialized! Start playing ... ')

    def start_play(self):
        print('Cricket Game Started. Enjoy the game ... ')

    def end_play(self):
        print('Cricket Game Finished!')


class Football(Game):

    def initialize(self):
        print('Football Game Initialized! Start playing ... ')

    def start_play(self):
        print('Football Game Started. Enjoy the game ... ')

    def end_play(self):
        print('Football Game Finished!')
```

```
if __name__ == '__main__':
    game = Cricket()
    game.play()
    game = Football()
    game.play()
```

Описание кода:

- `Game` - абстрактный класс, определяющий общие шаги алгоритма игры (инициализация, начало игры, конец игры) в виде шаблонного метода `play` и конкретные реализации этих шагов в виде примитивных операций `initialize`, `start_play`, `end_play`.
- `Cricket` и `Football` - производные классы, реализующие конкретные реализации примитивных операций `initialize`, `start_play`, `end_play` для игр в крикет и футбол соответственно.
- В коде запуска создаются экземпляры классов `Cricket` и `Football` и вызывается шаблонный метод `play`. В зависимости от типа объекта, вызывающего метод, будут выполняться разные конкретные реализации примитивных операций.

Посетитель

Порождающий шаблон Посетитель (Visitor) позволяет добавлять новые операции для объектов без изменения их классов. Он предоставляет способ разделить алгоритм от представления объектов, что повышает гибкость и упрощает поддержку программного обеспечения.

Аналогии из реальной жизни:

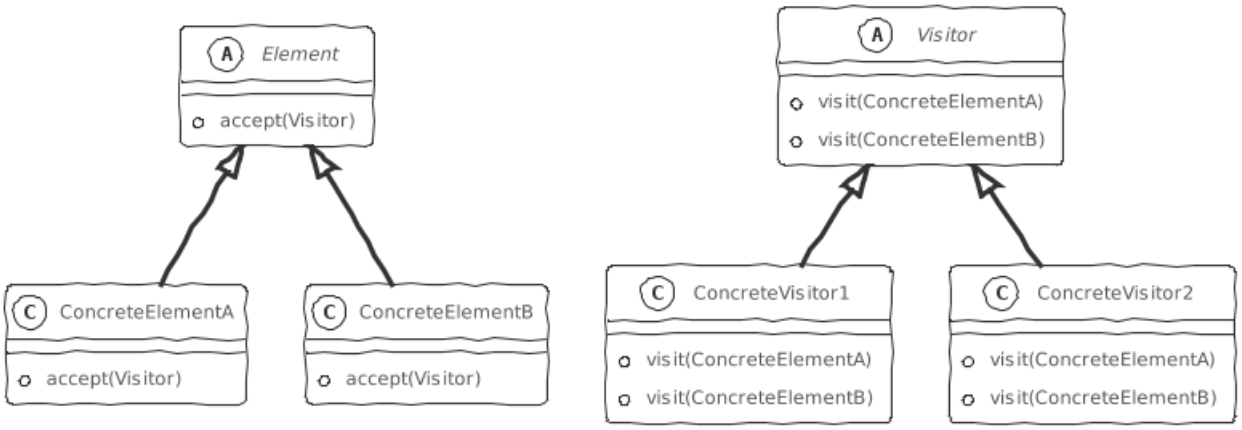
1. Команда инспекторов: Представьте, что у вас есть несколько разных типов объектов (заводы, офисы, магазины), и вам нужно проверить их соответствие различным стандартам (пожарная безопасность, санитарно-эпидемиологические нормы). Вместо того чтобы создавать для каждого типа объекта методы проверки, вы нанимаете команду инспекторов (посетителей). Каждый инспектор знает, как проверить соответствие конкретного стандарта для любого объекта.
2. Туристическое агентство: Пусть у вас есть несколько туристических маршрутов (города, природа, экстремальный отдых), и вам нужно предоставить туристам различные услуги (проживание, питание, трансферы). Вместо того чтобы для каждого маршрута создавать методы предоставления услуг, вы сотрудничаете с туристическим агентством (посетителем). Агентство знает, как предоставить конкретную услугу для любого маршрута.

Принципиальная схема шаблона Посетитель в формате PlantUML:

```
@startuml

abstract class Element {
  + accept(Visitor)
}

class ConcreteElementA {
  + accept(Visitor)
}

class ConcreteElementB {
  + accept(Visitor)
}

abstract class Visitor {
  + visit(ConcreteElementA)
  + visit(ConcreteElementB)
}

class ConcreteVisitor1 {
  + visit(ConcreteElementA)
  + visit(ConcreteElementB)
}

class ConcreteVisitor2 {
```

```
  + visit(ConcreteElementA)
  + visit(ConcreteElementB)
}

Element <|-- ConcreteElementA
Element <|-- ConcreteElementB
Visitor <|-- ConcreteVisitor1
Visitor <|-- ConcreteVisitor2

@enduml
```

Принципиальная схема шаблона Посетитель в формате UML:



Описание схемы:

- `Element` (Элемент) - абстрактный класс, определяющий интерфейс для принятия посетителя (метод `accept`).
- `ConcreteElementA` и `ConcreteElementB` (КонкретныйЭлементА и КонкретныйЭлементВ) - конкретные классы, реализующие интерфейс `Element` и содержащие данные, которые будут обрабатываться посетителем.
- `Visitor` (Посетитель) - абстрактный класс, определяющий интерфейс для обработки конкретных элементов (методы `visit`).
- `ConcreteVisitor1` и `ConcreteVisitor2` (КонкретныйПосетитель1 и КонкретныйПосетитель2) - конкретные классы, реализующие интерфейс `Visitor` и содержащие реализацию алгоритмов обработки данных элементов.

Абстрактный код реализации шаблона Посетитель на языке Python:

```python
from abc import ABC, abstractmethod


class Element(ABC):
    @abstractmethod
    def accept(self, visitor):
        pass


class ConcreteElementA(Element):
    def __init__(self, value):
        self._value = value

    def accept(self, visitor):
        visitor.visit_concrete_element_a(self)


class ConcreteElementB(Element):
    def __init__(self, value):
        self._value = value

    def accept(self, visitor):
        visitor.visit_concrete_element_b(self)


class Visitor(ABC):
```

```
    @abstractmethod
    def visit_concrete_element_a(self, element):
        pass

    @abstractmethod
    def visit_concrete_element_b(self, element):
        pass


class ConcreteVisitor1(Visitor):
    def visit_concrete_element_a(self, element):
        print(f"ConcreteVisitor1: {element._value}")

    def visit_concrete_element_b(self, element):
        print(f"ConcreteVisitor1: {element._value}")


class ConcreteVisitor2(Visitor):
    def visit_concrete_element_a(self, element):
        print(f"ConcreteVisitor2: {element._value * 2}")

    def visit_concrete_element_b(self, element):
        print(f"ConcreteVisitor2: {element._value * 2}")


if __name__ == "__main__":
    elements = [
        ConcreteElementA(1),
        ConcreteElementB(2),
        ConcreteElementA(3),
        ConcreteElementB(4)
    ]

    visitor1 = ConcreteVisitor1()
    for element in elements:
        element.accept(visitor1)

    visitor2 = ConcreteVisitor2()
    for element in elements:
        element.accept(visitor2)
```

Описание работы кода:

- Создаются абстрактные классы `Element` и `Visitor`, определяющие интерфейсы для элементов и посетителей.
- Создаются конкретные классы `ConcreteElementA`, `ConcreteElementB`, `ConcreteVisitor1`, `ConcreteVisitor2`, реализующие интерфейсы и содержащие данные и алгоритмы обработки.
- В главном модуле создаются экземпляры конкретных элементов и посетителей.
- Выполняется цикл, в котором каждый элемент принимает посетителя и вызывает метод `visit` для своего типа.
- Посетитель выполняет алгоритм обработки данных элемента.

Пример кода с реализацией шаблона Посетитель на Python, приближенный к реальности:

```
from abc import ABC, abstractmethod


class Document(ABC):
    @abstractmethod
    def accept(self, visitor):
        pass


class PDFDocument(Document):
    def __init__(self, file_name):
        self._file_name = file_name
```

```
    def accept(self, visitor):
        visitor.visit_pdf_document(self)


class WordDocument(Document):
    def __init__(self, file_name):
        self._file_name = file_name

    def accept(self, visitor):
        visitor.visit_word_document(self)


class DocumentVisitor(ABC):
    @abstractmethod
    def visit_pdf_document(self, document):
        pass

    @abstractmethod
    def visit_word_document(self, document):
        pass


class DocumentConverter(DocumentVisitor):
    def visit_pdf_document(self, document):
        print(f"Converting PDF document: {document._file_name}")

    def visit_word_document(self, document):
        print(f"Converting Word document: {document._file_name}")


class DocumentValidator(DocumentVisitor):
    def visit_pdf_document(self, document):
        print(f"Validating PDF document: {document._file_name}")

    def visit_word_document(self, document):
        print(f"Validating Word document: {document._file_name}")


if __name__ == "__main__":
    documents = [
        PDFDocument("file1.pdf"),
        WordDocument("file2.docx"),
        PDFDocument("file3.pdf"),
        WordDocument("file4.docx")
    ]

    converter = DocumentConverter()
    for document in documents:
        document.accept(converter)

    validator = DocumentValidator()
    for document in documents:
        document.accept(validator)
```

Описание работы кода:

- Создаются абстрактные классы `Document` и `DocumentVisitor`, определяющие интерфейсы для документов и посетителей документов.
- Создаются конкретные классы `PDFDocument`, `WordDocument`, `DocumentConverter`, `DocumentValidator`, реализующие интерфейсы и содержащие данные и алгоритмы обработки.
- В главном модуле создаются экземпляры конкретных документов и посетителей.
- Выполняется цикл, в котором каждый документ принимает посетителя и вызывает метод `visit` для своего типа.

- Посетитель выполняет алгоритм обработки данных документа (конвертация, валидация)