GeekBrains

Архитектура и шаблоны проектирования на Python

Поведенческие паттерны

На этом уроке

Обзор поведенческих шаблонов.

Оглавление

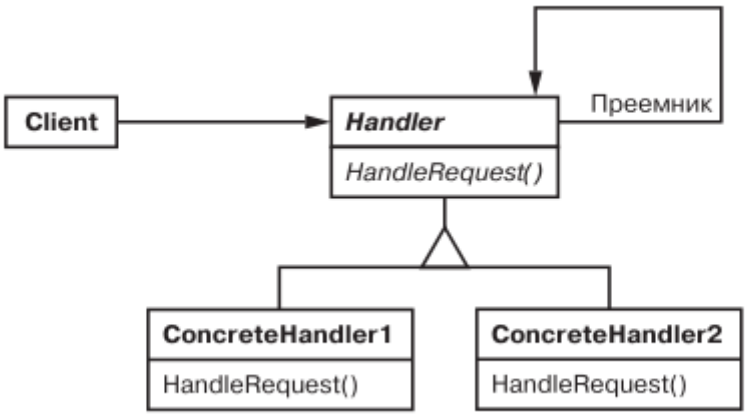
Поведенческие паттерны взаимодействия

Поведенческие паттерны идентифицируют различные подходы к реализации взаимодействия и коммуникации между объектами. Они описывают варианты управления состоянием объектов, использование одних объектов другими, не приводящее к образованию жёстких связей между ними.

Chain of responsibility // Цепочка ответственности

Связывает объекты-получатели в цепочку и передаёт запрос вдоль этой цепочки, пока его не обработают. Позволяет избежать жёсткой связи между отправителем и получателем запроса (вызова). Цепочки обработки могут быть построены динамически.

Один объект инициирует запрос (вызов метода), другой конкретный объект на него отвечает — уже нехорошо, это жёсткая связь. В таком случае инициирующий объект не удастся использовать повторно. Кроме того, инициирующий объект может не владеть информацией о конкретном исполнителе. Для решения подобных задач может использоваться паттерн *Цепочка ответственности*.



Определяем интерфейс обработчика, описываем операции, которые может выполнить один из обработчиков в цепочке, и метод (или поле в абстрактном классе) для получения следующего обработчика в цепочке. При невозможности обработать запрос конкретный обработчик, используя ссылку на следующее звено, передает запрос (вызов) ему.

Часто этот паттерн используется совместно с *Компоновщиком*, передавая необработанный запрос вверх по иерархии *Компоновщика*. Возможно выделение отдельного конкретного класса, реализующего интерфейс обработчика, который агрегирует набор возможных исполнителей и перебирает их, например, последовательно, для исполнения запроса.

Пример

Механизм выбора свободного оператора для ответа в online-чате. При поступлении запроса на соединение его следует маршрутизировать свободному оператору. Если прямо сейчас все операторы заняты, клиента оповещают об этом, пока попытки найти свободного продолжаются.

Определяем интерфейс обработчика в абстрактном классе Handler (Обработчик). Там же определяем общие для всех конкретных обработчиков поле связи next (следующий) и метод для создания цепочки.

```
class Handler(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def handle(self, request):
        if self.next is not None:
            self.next.handle(request)

    def link(self, next):
        self.next = next
        return self.next
```

Конкретный обработчик (*Оператор*), который в случае незанятости может обработать запрос.

```
class Operator(Handler):
    // вероятность занятости оператора
    probability = 0.75

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

    def handle(self, request):
        if self.is_busy():
            print(f'Оператор {self.name} занят')
            super().handle(request)
        else:
            print(f'Оператор {self.name} обрабатывает: "{request.get_data()}"')

    def is_busy(self):
        return random.random() < __class__.probability
```

Задача конкретного обработчика — понять, что все операторы заняты. Фактической обработки он не выполняет — лишь проверяет, ходит ли по кругу необработанный запрос.

```
class BusyHandler(Handler):
    def __init__(self):
        self.request = None

    def handle(self, request):
        if (self.request == request):
            print('Все операторы заняты, пожалуйста подождите')
        else:
            self.request = request
```

```
        super().handle(request)
```

Собираем цепочку и закольцовываем обработку:

```
handler = BusyHandler()

handler.link(Operator("#1")).\
    link(Operator("#2")).\
    link(Operator("#3")).\
    link(handler)
```

Тест цепочки обработчиков. Клиент взаимодействует с интерфейсом, а не с конкретной реализацией:

```
// генерируем поток из 3 запросов
for _ in range(3):
    handler.handle(Request())
```

Пример вывода:

```
Оператор #1 занят
Оператор #2 обрабатывает: "вопрос по новинке"
Оператор #1 занят
Оператор #2 занят
Оператор #3 занят
Все операторы заняты, пожалуйста подождите
Оператор #1 занят
Оператор #2 занят
Оператор #3 занят
Все операторы заняты, пожалуйста подождите
Оператор #1 обрабатывает: "вопрос по новинке"
Оператор #1 обрабатывает: "вопрос по дефекту"
```

Плюсы паттерна:

1. Уменьшение зависимости между инициатором и обработчиком.
2. Возможность динамически изменять цепочку обработчиков.
3. Соблюдение принципа единственной обязанности для классов *Обработчиков*.
4. Простота реализации.

Минус: запрос может быть не обработан.

Command // Команда

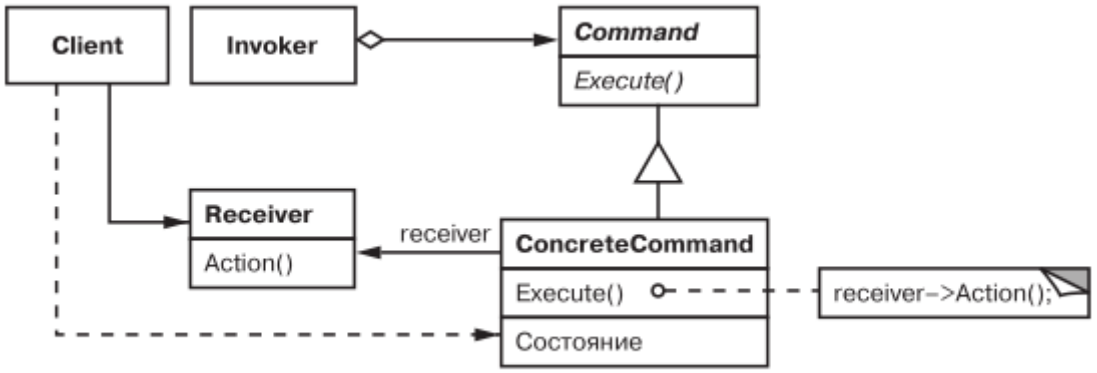
Команда инкапсулирует запрос в объект (тем самым позволяя задавать параметры клиентов для обработки соответствующих запросов), ставит запросы в очередь, логирует их, а также поддерживает откат операций.

Операция — это действие над объектом, вызов его метода — передача управления внутри.
Мы не можем напрямую передать действие на расстояние (между двумя разнесёнными узлами). Для

этого мы формируем команду, описывающую, что именно необходимо выполнить, и передаем непосредственно её.

Пример: выполнение вызова с задержкой или в заранее установленное время. Потенциально можно использовать вариации time.sleep(). Однако при большом количестве вызовов задача перестаёт быть тривиальной. Значительно проще описать вызов полями некоторого объекта.

«Жизнь не multiedit, undo не сделаешь», — старая шутка, которая показывает, как сильно мы привыкли к возможности отмены операций. Вы пробовали откатить **вызов** после выхода из метода? Благодаря представлению действия как объекта мы можем составить и сохранить последовательность операций, на каждом шаге определить обратную операцию или сохранить состояние. Таким образом, благодаря использованию паттерна, *Команда* легко реализует откат операций.



Command — интерфейс выполнения операции. Необходимо описать как минимум метод *Выполнить()*.

ConcreteCommand — конкретная команда, которая содержит в себе ссылку на конкретный объект *Исполнитель* и реализует абстрактный метод *Выполнить()* путём вызова операций в конкретном исполнителе.

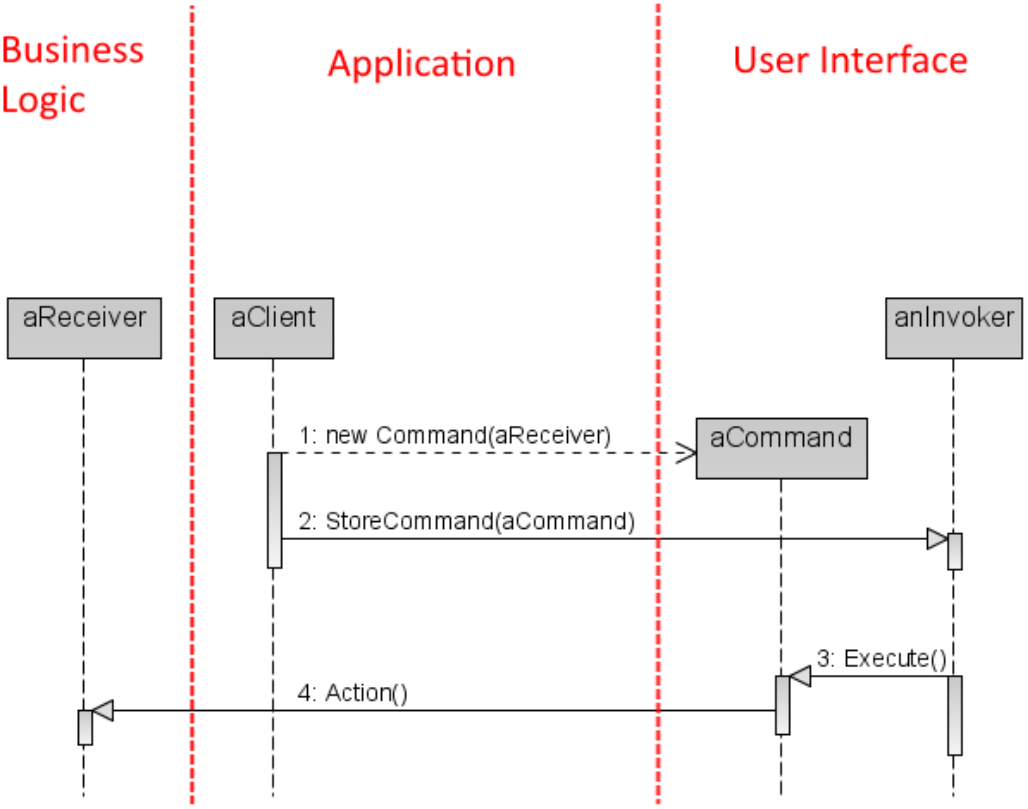
Receiver — получатель команды, исполнитель. Обычно это объект бизнес-логики. Выполняет нужные приложению действия.

Invoker — отправитель команды, который хранит готовые сформированные объекты ***ConcreteCommand,*** созданные приложением. ***Invoker*** не знает о конечном типе команды, а оперирует лишь ссылками на объекты, реализующие интерфейс ***Command***, вызывает метод *Выполнить()*.

Client — приложение, которое создаёт объекты команд ***ConcreteCommand*** и связывает их с конкретными получателями. Готовые команды хранит в отправителе (Invoker).

Пользователь, взаимодействуя с интерфейсом, выбирает в отправителе заранее подготовленные команды к исполнению.

Часто паттерн *Команда* используется для разнесения бизнес-логики и пользовательского интерфейса. Рассмотрим диаграмму взаимодействия.



Receiver только выполняет действие — бизнес-логика.

Client — приложение, знающее, кто (Receiver) отвечает за исполнение какого-либо действия. Для этого Client создаёт реализующие интерфейс *Command* объекты, в которые включает ссылку на исполняющий компонент бизнес-логики, и сохраняет эти готовые объекты-команды в отправителе.

Invoker — объект пользовательского интерфейса. При взаимодействии с пользователем исполняет известный ему *абстрактный* метод *Выполнить(),* реализация которого в конкретной команде выполняет конкретную операцию в бизнес-логике.

Пример

Нам необходимо запускать или приостанавливать некоторое действие объекта.

Определяем класс отправителя команд:

```
class CommandsInvoker:
    def __init__(self):
        self._commands_list = []

    def store_command(self, command):
        self._commands_list.append(command)

    def execute_commands(self):
        for command in self._commands_list:
            command.execute()
```

Абстрактный класс команды:

```
class Command(metaclass=abc.ABCMeta):
    def __init__(self, receiver):
        self._receiver = receiver

    @abc.abstractmethod
    def execute(self):
        pass
```

Классы конкретных команд только вызывают методы, но не знают ничего об их реализации:

```
class ActionCommand(Command):
    def execute(self):
        self._receiver.action()


class PauseCommand(Command):
    def execute(self):
        self._receiver.pause()
```

Класс исполнителя, в котором должны быть реализации **ВСЕХ** конкретных команд:

```
class CommandsReceiver:
    def action(self):
        print('action in receiver')

    def pause(self):
        print('pause in receiver')
```

Клиентский код:

```
commands_receiver = CommandsReceiver()

action_command = ActionCommand(commands_receiver)
pause_command = PauseCommand(commands_receiver)

commands_invoker = CommandsInvoker()

commands_invoker.store_command(action_command)
commands_invoker.store_command(pause_command)

commands_invoker.execute_commands()
```

Создаём команды и добавляем их в список, а затем выполняем.

Mediator // Посредник

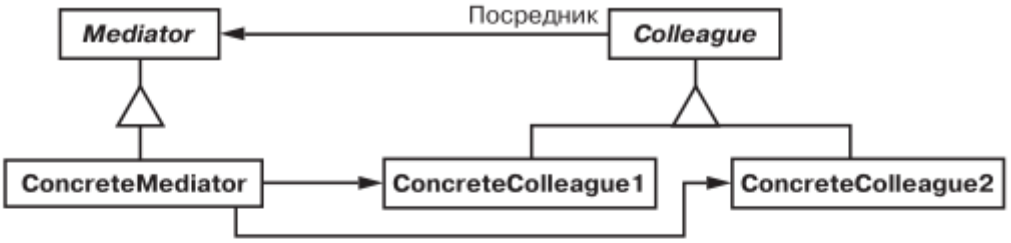
Определяет объект, инкапсулирующий **способ** взаимодействия множества объектов. *Посредник* обеспечивает слабую связанность системы, избавляя объекты от необходимости явно ссылаться друг на друга и позволяя независимо изменять взаимодействия между ними.

Объектно-ориентированное проектирование разбивает сложную задачу на взаимодействующие объекты. Взаимодействие объектов напрямую (один объект вызывает метод другого) приводит к образованию жёстких связей. Число связей (зависимостей) между классами, взаимодействующими таким образом, будет расти быстрее количества классов. Это сделает систему жёстко связанной, монолитной. При этом развитие какого-либо класса будет осложнено большой связанностью с другими компонентами системы. Одно из решений проблемы — использование *Посредник*а. Его ответственность — **определение и координация** взаимодействия различных объектов. *Посредник* замыкает взаимодействие на себя и координирует передачу вызовов и распространение информации.

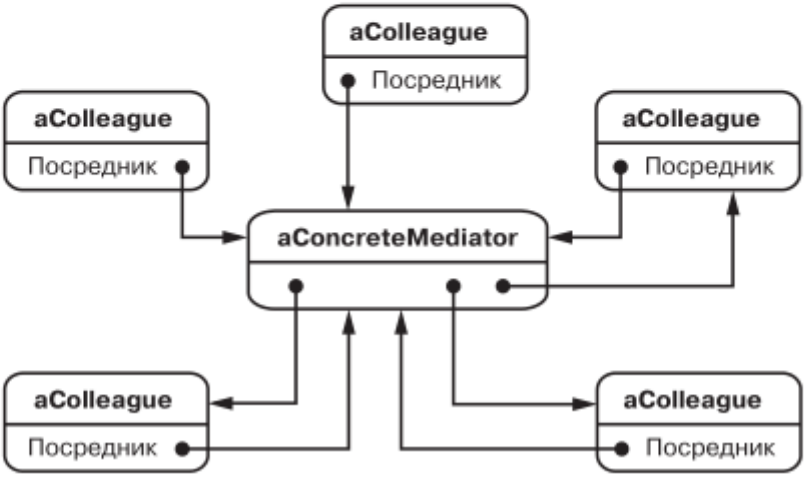
Пример из реального мира: организация авиационного движения на подходах к ВПП аэропорта. Теоретически пилоты могли бы договориться о порядке взлёта / посадки между собой, однако на практике существует выделенный диспетчер, замыкающий на себя координацию их действий.

Пример из UI. Форма регистрации, два поля ввода, электронная почта, пароль, кнопка Ok. Мы требуем ввода валидного адреса электронной почты и сложного пароля, и только при выполнении обоих условий кнопка Ok становится активной. Без использования *Посредника* модификация одного из полей и его анализ недостаточны для принятия решения о разблокировке кнопки Ok, поскольку требуется выполнение обоих условий. Разумное решение — валидатор формы, *Посредник*. При изменении одного из полей валидатор собирает (или, возможно, хранит) информацию о состоянии других элементов управления, принимает решение относительно доступности третьего элемента.

.



Типичная структура объектов.



Mediator — интерфейс абстрактного *Посредника*.

ConcreteMediator — конкретный *Посредник*, координирует действия объектов.

ConcreteColleague — классы одного уровня абстракции, взаимодействуют косвенным образом через *Посредника*.

При такой форме организации взаимодействия падает связанность (зависимость конкретной реализации друг от друга) классов системы. Это позволяет развивать и тестировать их независимо друг от друга.

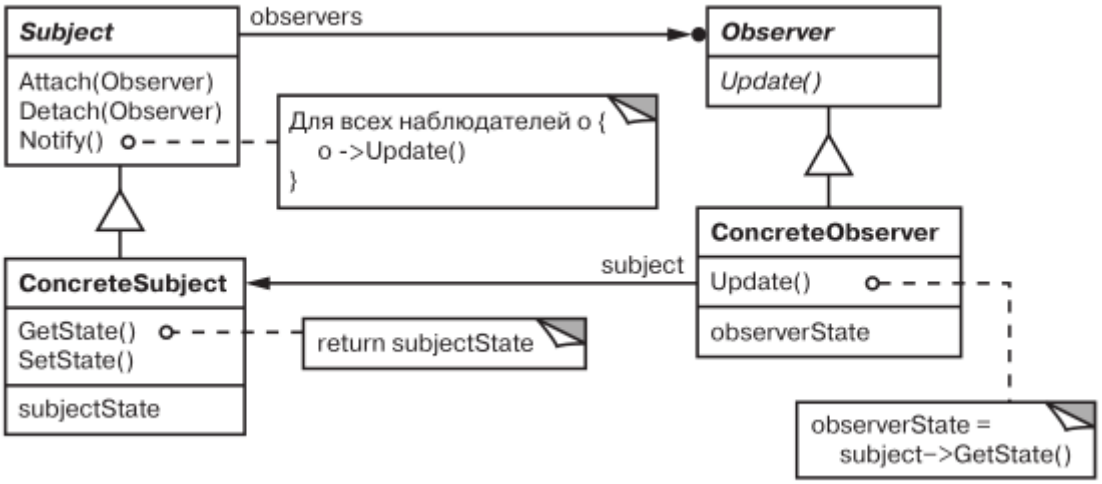
Observer // Наблюдатель

Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом и автоматически обновляются.

Паттерн *Наблюдатель* позволяет избавиться от жёсткой связи между объектами: владеющим информацией (например, модель) и другими объектами, использующими эту информацию (например, объекты пользовательского интерфейса). Для реализации этой цели вводятся понятия:

- *Подписчик* (Observer — *Наблюдатель*, Listener — *Слушатель*) — подписывается на получение уведомлений об изменении состояния владельца информации — *Издателя*. Подписчики — обычно объекты пользовательского интерфейса.
- *Издатель* (Subject, Observable) — реализует механизм подписки наблюдателей на сообщения (в данном случае — вызовы метода) об изменении состояния владельца информации.

Таким образом, подписчики зависят от издателей, но эта связь слабая, поскольку и те, и другие зависят от взаимодействующих абстракций — описывающих поведение интерфейсов. Рассмотрим диаграмму классов.



Subject — абстрактный субъект, *Издатель*. Описывает интерфейс подписки (отписки) абстрактных *Наблюдателей*.

Observer — абстрактный *Наблюдатель*. Описывает интерфейс вызываемого *Издателем* обновления при необходимости известить *Подписчиков* об изменениях.

ConcreteSubject — конкретный субъект, владелец информации. При изменении состояния посредством вызова метода notify() *Издателя* оповещает подписчиков о произошедших изменениях или просто об их факте.

ConcreteObserver — конкретный *Наблюдатель*. Реализует интерфейс обновления, поддерживает согласованность с владельцем информации — субъектом.

Важной особенностью конкретной реализации паттерна будет вид модели распространения информации: PUSH- или PULL-модель. У обеих моделей есть достоинства и недостатки. Истина, как обычно, где-то посередине. При использовании PUSH-модели владелец информации указывает максимум данных в сообщении *Подписчикам* (вызове метода update). С одной стороны, это влечёт за собой накладные расходы на передачу этой информации между объектами, с другой — освобождает *Наблюдателей* от необходимости дополнительно опрашивать владельца информации (вызов метода GetState). Один из минусов такого метода — избыток информации на стороне *Наблюдателей*, поскольку вся полнота отправляемой *Издателем* картины требуется им не всегда. На другом полюсе находится PULL-модель, когда *Издатель* в крайнем случае уведомляет *Подписчиков* лишь о факте изменений, а необходимую информацию они запрашивают у конкретного владельца. При этом, как видно из диаграммы классов, возникает жёсткая связь между конкретным *Подписчиком* и конкретным *Издателем*, что в некоторых случаях нежелательно. Оптимальное решение заключается в поиске минимально устраивающего всех *Подписчиков* набора информации, распространяющегося PUSH-методом. Требующие более тесного взаимодействия *Подписчики* используют PULL-модель для получения всей необходимой информации.

	PUSH	PULL
Активная сторона	Субъект	Наблюдатель
Сообщение	Много информации	Факт изменения
Обратные вызовы	Нет	Да
Использование ресурсов	Хуже	Лучше
Эффективность, итого	Лучше	Хуже

Помимо термина «Наблюдатель» (Observer, Subscriber) используется термин Listener, EventListener — это *Наблюдатель*, ориентированный на приём PUSH-сообщений и не ориентированный на модификацию состояния владельца информации.

Пример

Нагреватель с термостатом. Датчик температуры, класс Sensor — владелец информации о температуре. Кроме того, в системе участвуют: DisplayObserver (дисплей отображения температуры) и HeaterObserver («нагреватель»). Задача класса — держать температуру в указанном диапазоне путём нагрева (повышения температуры) датчика. Система состоит из двух логических частей:

1. Ядро системы — Sensor (владелец информации, *Издатель*).
2. Зависимые части — *Подписчики*.

Сначала опишем интерфейс *Наблюдателя*. Подходит как для PUSH-, так и для PULL-модели, подробности будут в разных его имплементациях.

```
class Observer(metaclass=abc.ABCMeta):
    def __init__(self):
        self._subject = None
        self._observer_state = None

    @abc.abstractmethod
    def update(self, arg):
        pass
```

Абстрактный издатель, ответственность класса — подписка/отписка и нотификация *Наблюдателей*, представленных абстрактными интерфейсами.

```
class Subject:
    def __init__(self):
        self._observers = set()
        self._subject_state = None

    def attach(self, observer):
        observer._subject = self
        self._observers.add(observer)

    def detach(self, observer):
        observer._subject = None
        self._observers.discard(observer)

    def _notify(self):
        for observer in self._observers:
            observer.update(self._subject_state)
```

Владелец информации наследует поведение абстрактного *Издателя* (при этом возможно использование композиции) и нотифицирует *Подписчиков* при возникновении изменений в его состоянии.

```
class Sensor(Subject):
    @property
    def t(self):
        return self._subject_state

    @t.setter
    def t(self, t):
        self._subject_state = t
        self._notify()
```

Конкретные подписчики.

Индикатор температуры отображает температуру при получении сообщения — типичный PUSH-клиент, получает температуру как payload вызова update().

```
class DisplayObserver(Observer):
    def update(self, arg):
        print(f'{self.__class__.__name__} temperature {arg}')
```

Наблюдатель-нагреватель — типичный PULL-клиент, не обращает внимания на payload, состояние наблюдаемого объекта получает и модифицирует вызовом его методов.

```
class HeaterObserver(Observer):
    def __init__(self, low_threshold, step):
        super().__init__()
        self.low_threshold = low_threshold
        self.step = step

    def update(self, arg):
        if isinstance(self._subject, Sensor):
            sensor = self._subject

            t = sensor.t
            delta_low = t - self.low_threshold

            if delta_low < 0:
                t += self.step
                print(f'{self.__class__.__name__} heat impulse +{self.step}')
                sensor.t = t
```

Клиентский код:

```
// демо
sensor = Sensor()

// подключаем наблюдателей за сенсором
sensor.attach(DisplayObserver())
sensor.attach(HeaterObserver(40, 20))

// начальное значение
sensor.t = 20

// цикл энтропии - естественное охлаждение сенсора
for _ in range(5):
    random_t = random.random() * 10
    sensor.t = sensor.t - random_t

    time.sleep(0.5)
```

Итог

Основная цель паттернов (*Цепочка обязанностей*, *Команда*, *Посредник и Наблюдатель*) — разделение отправителей и получателей запросов, при этом каждый из них решает задачу по-своему.

Цепочка обязанностей передаёт запрос отправителя через цепочку потенциальных обработчиков, ожидая, что какой-то из них обработает запрос, но не гарантирует его исполнение.

Команда соединяет отправителя и получателя посредством передачи объекта команды.

Посредник замыкает на себя коммуникации отправителя и получателя.

Наблюдатель позволяет динамически подключать и отключать инициаторов и разработчиков взаимодействия. При этом он гарантирует доставку сообщения всем участникам.

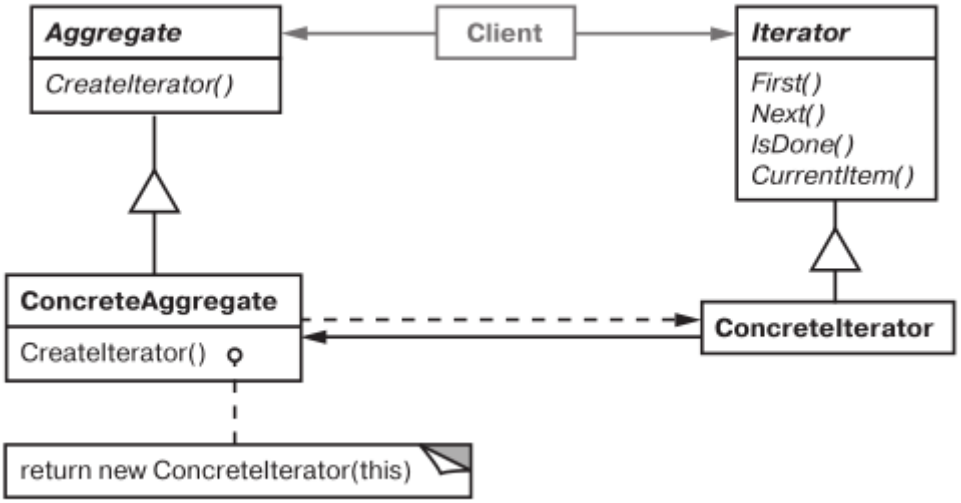
Отличие *Посредника* от *Наблюдателя* состоит в том, что *Посредник*, делая компоненты независимыми друг от друга, замыкает их зависимость на себе, в то время как *Наблюдатель* сконцентрирован на динамическом изменении круга потенциально взаимодействующих объектов.

Другие поведенческие паттерны

Iterator // Итератор

Предоставляет способ последовательного доступа ко всем элементам составного объекта, не раскрывая его внутреннего представления.

В Python реализован в классе iter().



Iterator — *Итератор*, определяет интерфейс для доступа и обхода элементов.

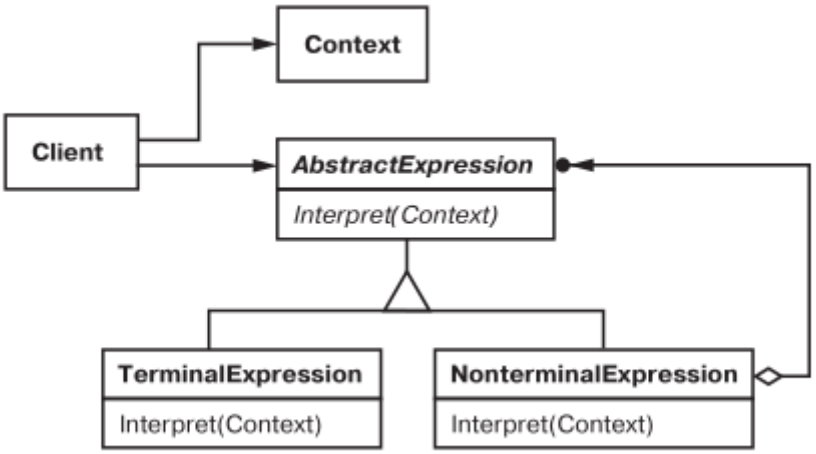
ConcreteIterator — конкретный *Итератор*, реализует интерфейс Iterator, следит за текущей позицией.

Aggregate — интерфейс составного объекта, описывает операцию создания *Итератора*.

ConcreteAggregate — конкретный составной объект, реализует интерфейс создания *Итератора* и возвращает экземпляр конкретного класса **ConcreteIterator**.

Interpreter // Интерпретатор

Этот шаблон задаёт язык, определяет представление его грамматики, а также интерпретатор предложений этого языка.



Широко известное применение этого паттерна — регулярные выражения. Есть язык регулярных выражений. Используя его, мы описываем действия, которые необходимо выполнить над строкой (Context). Конкретная реализация языка регулярных выражений определяет классы ConcreteExpression.

Пример

Параметрический расчёт спецификации сложного изделия на основе модели. В производстве продукция описывается некоторой моделью, у неё законченный или расширяемый список свойств. На основе свойств модели необходимо составить подробную спецификацию для производства изделия. Модели меняются достаточно часто, а список свойств — значительно реже. Например, для построения забора высотой в три и длиной 10 метров потребуется 100 вертикальных стоек частокола и четыре горизонтальных балки по пять метров.

Жёсткое кодирование этого расчёта неоправданно, так как заборы и дизайн частокола меняются часто, а суть — редко. Поэтому в долгосрочной перспективе проще описать язык (в данном случае иерархический), проходя по узлам которого, *Интерпретатор* собирает необходимую спецификацию на основе данных модели.

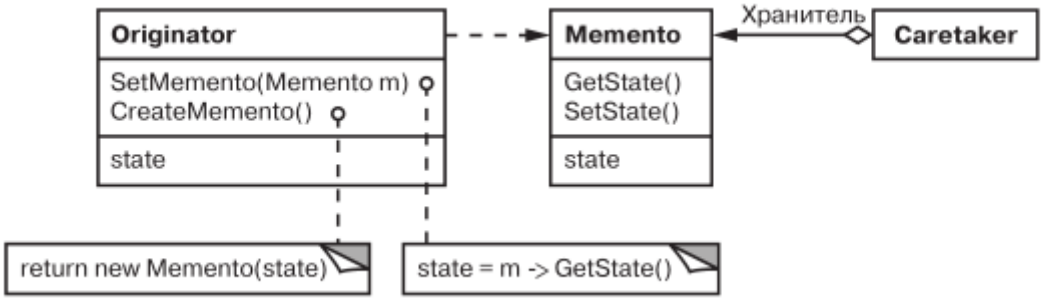
Memento // Хранитель

Не нарушая инкапсуляции, фиксирует и выносит за пределы объекта его внутреннее состояние так, чтобы позднее можно было восстановить в нём объект.

Объекты, находящиеся в памяти компьютера, не вечны. Необходим способ сохранять состояние объекта для последующего его восстановления на этом же компьютере или для передачи между узлами сети. Но как реализовать такое поведение, не нарушая принцип открытости/закрытости?

Python предлагает «из коробки» готовые механизмы преобразования объектов в двоичный код, реализованные в модулях pickle и marshal. Использование pickle предпочтительнее.

В отличие от «бездушного» сериализатора, Memento добавляет настраиваемую логику (программный код) в операции сохранения и восстановления состояния. Для этого приходится реализовывать два метода на стороне заинтересованного в сохранении состояния объекта: CreateMemento и SetMemento() (соответственно для сохранения и восстановления состояния).

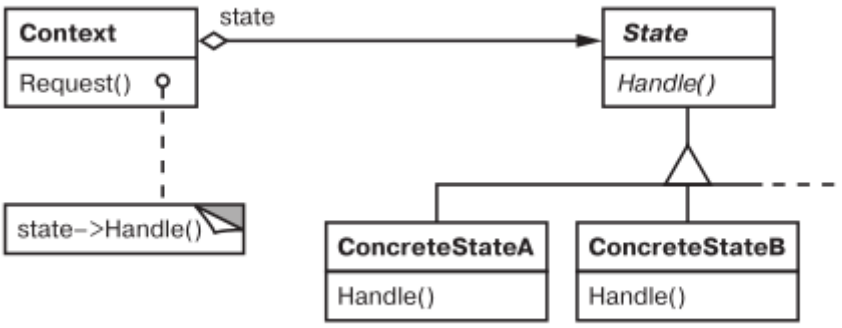


Memento часто используется совместно с паттерном *Команда* для реализации механизма отката выполненных операций.

State // Состояние

Позволяет объекту варьировать свое поведение в зависимости от внутреннего состояния. Извне создается впечатление, что изменился класс объекта.

Паттерн *Состояние* предполагает выделение базового класса или интерфейса для всех допустимых операций и наследника для каждого возможного состояния.



Context — контекст состояния, определяет интерфейс для клиентов и хранит экземпляр **state** подкласса **ConcreteState**, которым определяется текущее состояние.

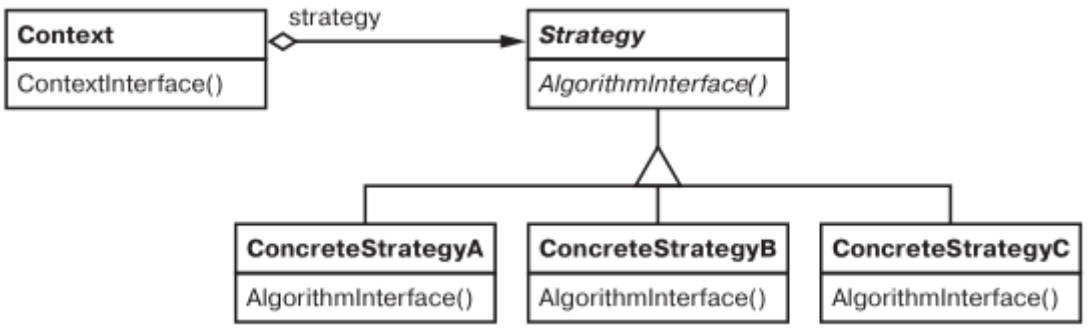
State — состояние, интерфейс состояния, определяет методы конкретных состояний.

ConcreteState — конкретное состояние, каждый подкласс реализует поведение, ассоциированное с некоторым конкретным состоянием контекста Context.

Смена состояния в контексте паттерна — это изменение ссылки хранимого полем state. При изменении состояния меняется конкретный реализующий его объект, но не его интерфейс.

Strategy // Стратегия

Определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.



Интерфейс Strategy (Стратегия) реализует его конкретные реализации в классах ConcreteStrategy. Клиентский код создаёт объект конкретной *Стратегии* и передает его коду, который использует методы конкретной реализации *Стратегии* по её абстрактному интерфейсу. С одной стороны, есть код, который готов работать с любой *Стратегией*, и он не знает, с какой именно. С другой — клиентский код может динамически выбирать конкретный вариант реализации поведения *Стратегии*.

Ответственность за выбор поведения (*Стратегии*) лежит на клиентском коде — в отличие от паттерна *Состояние*, где ответственность за выбор поведения лежит на абстрактном классе State, скрывающем различные реализации состояния.

Python позволяет заменить lambda-выражениями многие случаи применения *Стратегии*.

Пример

Различные варианты оплаты заказа.

Интерфейс *Стратегии*, общая задача любой реализации *Стратегии* — провести платёж на конкретную сумму:

```
class PaymentStrategy(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def pay(self, amount):
        pass
```

Разные варианты *Стратегии* требуют разной учётной информации о средстве платежа, особенности авторизации и т. д., чувствительные методы могут быть сконцентрированы в конкретных реализациях *Стратегии* оплаты.

Платеж через PayPal:

```
class PayPalPaymentStrategy(PaymentStrategy):
    // требуем учетку от PayPal
    def __init__(self, email, token):
        self.email = email
        self.token = token

    def pay(self, amount):
        print(f'processing {amount} via PayPal account {self.email}')
```

Оплата кредитной картой:

```
class CreditCard:
    def __init__(self, number):
        self._number = number

    def get_number(self):
        return self._number


class CreditCardPaymentStrategy(PaymentStrategy):
    // требуется кредитка
    def __init__(self, card):
        self.card = card

    def pay(self, amount):
        print(f'processing {amount} via credit card {self.card.get_number()}')
```

Метод Заказа, отвечающий за оплату заказа, принимает как аргумент объект *Стратегии* по интерфейсу.

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


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

    def pay(self, strategy):
        total = self.get_total()
        strategy.pay(total)

    def get_total(self):
        total = 0
        for _item in self._items:
            total += _item.price

        return total

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

Клиентский код:

```
// товары
item1 = Item("Book", 515)
item2 = Item("Magazine", 298)
```

```
// создаём и наполняем заказ
order = Order()
order.add_item(item1)
order.add_item(item2)

// выбор конкретной Стратегии и оплата заказа
paypal_payment_strategy = PayPalPaymentStrategy("patterns@geekbrains.com",
"token")
order.pay(paypal_payment_strategy)

// выбор конкретной Стратегии и оплата заказа
credit_card = CreditCard("1234 5678 9101 2131 4156")
credit_card_payment_strategy = CreditCardPaymentStrategy(credit_card)
order.pay(credit_card_payment_strategy)
```

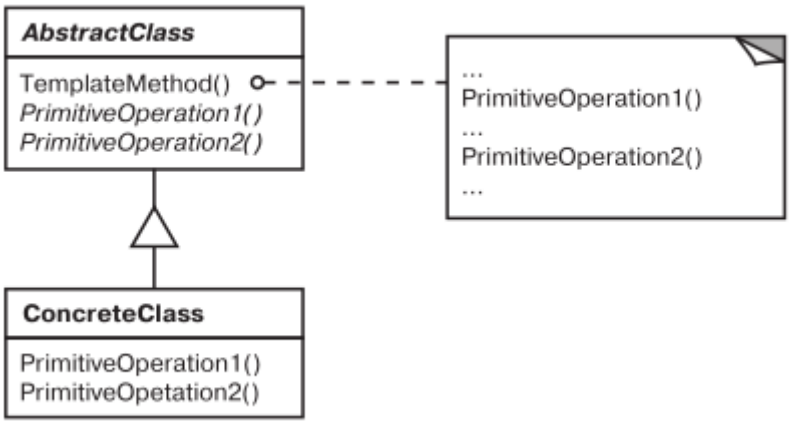
Вывод:

```
processing 813 via PayPal account patterns@geekbrains.com
processing 813 via credit card 1234 5678 9101 2131 4156
```

Template Method // Шаблонный метод

Определяет основу алгоритма и позволяет подклассам переопределить некоторые его шаги, не изменяя структуру в целом.

Паттерн отделяет основу алгоритма или шаги реализации, общие для всех потомков, от деталей, которые каждый из них выполняет по-своему, при этом меняя суть поведения основного класса.



AbstractClass — абстрактный класс, определяет конкретный шаблонный метод, вызывающий, в свою очередь, абстрактные операции.

ConcreteClass — реализует определенные выше абстрактные операции.

Применяется, чтобы:

- Однократно определить инвариантные части алгоритма, оставляя реализацию изменяющегося поведения на усмотрение подклассов.
- Локализовать в одном, базовом, классе поведение, общее для всех подклассов (final method), что позволяет избежать дублирования кода.

- Управлять расширением подклассов. Некоторые операции шаблонного метода суперкласса определяются как возможные к расширению (protected) с указанием конкретной базовой реализации по умолчанию. Операции, которые подкласс обязан реализовать, определяются как абстрактные (abstract), явно указывая участок ответственности для клиентского кода.

Пример

Разные пользователи выбирают разные методы доставки уведомлений. Реализация доставки уведомления на электронную почту отличается от доставки через Facebook, Twitter или по SMS, но в то же время имеется и сходство. Процесс состоит из:

1. Входа в систему доставки сообщений.
2. Собственно отправки.
3. Выхода.

Абстрактный класс нотификатора описывает последовательность действий:

```
class Notifier(metaclass=abc.ABCMeta):
    def __init__(self):
        self._log_list = []

    def notify(self, address, subject, message):
        self._login()
        self._send(address, subject, message)
        self._logout()
        self._log(address, subject, message)

    // войти в систему доставки сообщений
    @abc.abstractmethod
    def _login(self):
        pass

    // отправка сообщения
    @abc.abstractmethod
    def _send(self, address, subject, message):
        pass

    // выход
    @abc.abstractmethod
    def _logout(self):
        pass

    // внутреннее логирование, задаем поведение по умолчанию
    def _log(self, address, subject, message):
        self._log_list.append([address, subject, message])
```

Реализации конкретных нотификаторов осуществляют только подзадачи основного алгоритма, не влияя на последовательность действий.

```
class EmailNotifier(Notifier):
    def __init__(self):
        super().__init__()
```

```
        self.mail_from = ''

    def _login(self):
        // no need to login
        pass

    def _send(self, mail_to, subject, message):
        // send_mail(self.mail_from, mail_to, subject, message)
        print(f'send_mail: {mail_to}, {subject}, {message}')

    def _logout(self):
        // no need to logout
        pass
```

```
class FacebookNotifier(Notifier):
    def _login(self):
        // login to facebook
        print('login to facebook')

    def _send(self, address, subject, message):
        // send facebook message
        print(f'send facebook message: {address}, {subject}, {message}')

    def _logout(self):
        // logout from facebook
        print('logout from facebook')

    // переопределяем поведение шага внутреннего логирования
    def _log(self, address, subject, message):
        // не будем внутренне логировать нотификацию по FB, это избыточно
        pass
```

Использование:

```
class NotifierFabric:
    @staticmethod
    def get_notifier(communication_type):
        if communication_type == 'EMAIL':
            return EmailNotifier()
        elif communication_type == 'FACEBOOK':
            return FacebookNotifier()
        // ...


// инстанциируем объект конкретного нотификатора, используя Фабричный метод
notifier_1 = NotifierFabric.get_notifier('EMAIL')
notifier_1.notify('patterns@geekbrains.ru', 'notify_1', 'hello world')

// инстанциируем объект конкретного нотификатора
notifier_2 = NotifierFabric.get_notifier('FACEBOOK')
notifier_2.notify('patterns_facebook', 'notify_2', 'hi')
```

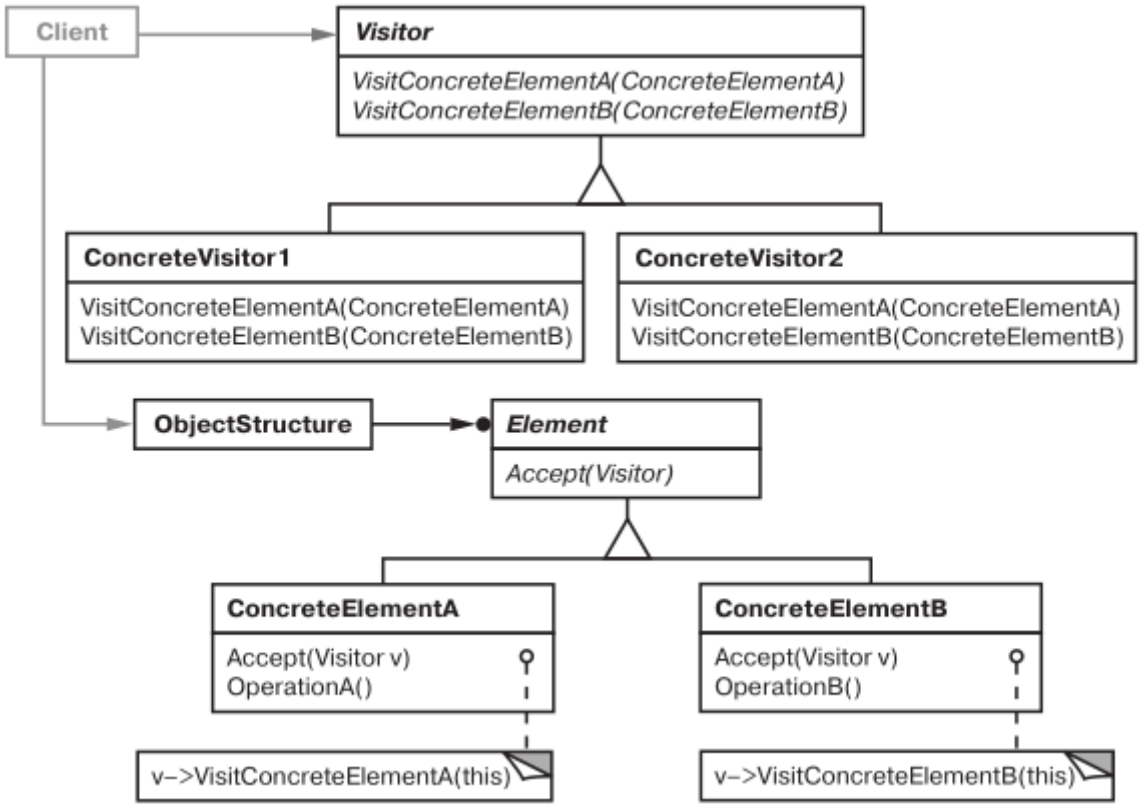
Метод шаблона **notify(...)** состоит из последовательности шагов, порядок которых фиксирован.

Идеология паттерна укладывается в голливудский принцип: «Не звоните нам, мы сами вам позвоним». Задача подклассов конкретных реализаций — имплементация конкретных шагов, суть алгоритма определяется в базовом классе.

Visitor // Посетитель

Описывает операцию, выполняемую с каждым объектом из некоторой структуры. Паттерн *Посетитель* позволяет определить новую операцию, не изменяя классы этих объектов.

Класс с точки зрения ООП — это и есть описание данных и методов работы с ними. Класс предоставляет методы доступа, скрывая сами данные от внешнего мира. В то же время встречаются ситуации, когда структуры данных достаточно сложны, а производимые над ними операции или настолько разнообразны, что объединять данные и операции неразумно (это размывает ответственность), или ещё не известны на момент проектирования структур данных. Встречаются и обратные ситуации: для стабильной иерархии объектов требуется выполнять разноплановые операции без возможности модифицировать код, который описывает иерархию объектов, представляющую некоторую предметную область.



ObjectStructure — структура объектов, обладает знанием (данными) внешних операций, которые мы пытаемся реализовать, используя паттерн.

Element — элемент, базовый интерфейс иерархии элементов структуры, для которой добавляется новая функциональность. Описывает операцию Accept, принимающую аргументом объект *Посетителя*, реализующего интерфейс **Visitor**.

Visitor — *Посетитель*, интерфейс, описывающий метод Visit...(ConcreteElement*)* для каждого класса ConcreteElement. С одной стороны, это позволяет обращаться к конкретному элементу, учитывая его

семантику, с другой — жёстко сцепляет описание этого интерфейса с составом структуры объектов, т. к. при модификации структуры изменения придётся отражать в описании этого интерфейса. Поэтому паттерн рекомендуется применять только для стабильных структур объектов.

ConcreteVisitor — конкретный *Посетитель*, имплементирует операции интерфейса **Visitor**.

ConcreteElement — конкретный элемент, реализует операцию Accept путём вызова метода, соответствующего классу **ConcreteElement**.

Здесь мы не можем использовать общий, описанный в интерфейсе, метод, готовый принять объект базового класса Element. При конкретной реализации **ConcreteVisitor** требуется объект конкретного класса, раскрывающего особенности своей семантики, а не ссылка на объект обобществленного интерфейса.

Клиентский код создает объект конкретного *Итератора* и, используя внешний или внутренний *Итератор* структуры объектов, вызывает для каждого элемента метод Accept(). *Итератор* отдает объекты по интерфейсу Element, а не конкретного типа.

Что мы имеем в итоге: достаточно стабильная структура данных взаимодействует с внешним по отношению к структуре кодом (иерархия наследников Visitor) через интерфейс, зависит только от этого интерфейса (вызов метода *Accept*) и, собственно, от самой себя, налагая на интерфейс требования по описанию многообразия иерархии наследников Element (описание вызовов Visit…()). С этой точки зрения интерфейс Visitor — скорее часть кода структуры данных, нежели клиентского приложения. Благодаря этому структура объектов и интерфейс Visitor представляют собой обособленную библиотеку.

Имплементируя интерфейс Visitor, предоставленный структурой объектов, внешний код получает возможность реализовывать добавочный функционал, не меняя кода структуры.

Стоит отметить, что объект **ConcreteVisitor,** получая вызовы VisitConcreteElement(), может агрегировать информацию, проходя по элементам структуры.

Пример

Имеем модель предметной области — например, модель конструкции. Абстрактный класс её элемента:

```
class ConstructionElement(metaclass=abc.ABCMeta):
   @abc.abstractmethod
   def accept(self, visitor):
       pass
```

Конкретные элементы — рычаг и передача:

```
class LeverElement(ConstructionElement):
    def accept(self, visitor):
        visitor.visit_lever_element(self)


class GearElement(ConstructionElement):
    def accept(self, visitor):
        visitor.visit_gear_element(self)
```

Абстрактный *Посетитель*:

```
class ConstructionElementVisitor(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def visit_lever_element(self, lever_element):
        pass

    @abc.abstractmethod
    def visit_gear_element(self, gear_element):
        pass
```

Реализация действий над элементом конструкции (прочностной расчёт конструкции, калькуляция спецификации для её производства) — конкретные *Посетители*:

```
class StrengthCalculatorVisitor(ConstructionElementVisitor):
    def visit_lever_element(self, lever_element):
        print(f'do Strength Calculation for {lever_element}')

    def visit_gear_element(self, gear_element):
        print(f'do Strength Calculation for {gear_element}')


class SpecificationCalculatorVisitor(ConstructionElementVisitor):
    def visit_lever_element(self, lever_element):
        print(f'do Specification Calculation for {lever_element}')

    def visit_gear_element(self, gear_element):
        print(f'do Specification Calculation for {gear_element}')
```

Клиентский код:

```
strength_calculator_visitor = StrengthCalculatorVisitor()
specification_calculator_visitor = SpecificationCalculatorVisitor()

lever_1 = LeverElement()
gear_1 = GearElement()


lever_1.accept(strength_calculator_visitor)
lever_1.accept(specification_calculator_visitor)

gear_1.accept(strength_calculator_visitor)
gear_1.accept(specification_calculator_visitor)
```

Пример вывода:

```
do Strength Calculation for <__main__.LeverElement object at 0x00000000026A7240>
do Specification Calculation for <__main__.LeverElement object at 0x00000000026A7240>
do Strength Calculation for <__main__.GearElement object at 0x00000000026A7278>
do Specification Calculation for <__main__.GearElement object at 0x00000000026A7278>
```

Практическое задание

В этой самостоятельной работе тренируем умения:

1. Выбирать подходящий поведенческий шаблон.
2. Применять поведенческие шаблоны в своём коде.

Зачем:

Для использования поведенческих шаблонов в своём коде.

Последовательность действий:

1. Реализовать создание профиля студента (регистрация). Список студентов. Механику записи студента на курс.
2. Далее можно сделать всё или одно на выбор, применив при этом один из структурных паттернов, либо аргументировать, почему эти паттерны не были использованы:
    - Создать страницу для изменения курса. После изменения отправлять уведомления всем студентам на курсе по SMS, email (для имитации можно просто выводить сообщения в консоль). Также известно, что скоро способов уведомления будет больше.
    - Добавить возможность применять цикл for к объекту категории курса (в каждой итерации получаем курс) и объекта курса (в каждой итерации получаем студента). Например: for student in course: … for course in group.
    - Создать API для курсов. По определённому адресу выводить не веб-страницу, а отдавать пользователю данные о списке курсов в формате json.
    - Улучшить логгер (или добавить, если его нет). Добавить в логгер возможность писать в файл, в консоль. Также известно, что в будущем вариантов сохранения может стать ещё больше.

- Реализовать CBV (Class Based VIews). Возможность создавать view в виде класса (по аналогии с Django). И убрать таким образом часть дублирования во view

Дополнительные материалы

1. Поведенческие паттерны проектирования.
2. Behavioral patterns.
3. Pickle.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Поведенческие шаблоны проектирования.
2. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2015.