Оглавление

Неделя 3

Паттерны проектирования (часть 1)

3.1 Достоинства паттернов проектирования и особенности их применения в Python

3.1.1 Введение в паттерны проектирования

В ходе работы программистам часто приходится решать похожие задачи. В создании игр это может быть программирование шаблонов поведения противников; в сетевых приложениях — ведение логов или отслеживание состояний клиентских приложений. Использование паттернов (шаблонов) проектирования позволяет использовать качественные решения, проверенные временем, а не изобретать велосипед.

Применяя их, вы избежите многих ошибок, ведь паттерны отлажены поколениями программистов.

Паттернов проектирования очень много. Некоторые из них, например Adapter, используются постоянно. В различных областях, таких как: трёхмерная графика, обработка данных и сетевое взаимодействие, существуют специальные шаблоны, позволяющие эффективно решать задачи, специфичные для данных областей.

Паттерн проектирования (Design Pattern) — повторяемая архитектурная конструкция, применяемая для решения часто встречающихся задач.

Шаблоны проектирования по своей сути очень похожи на кулинарные рецепты: они также описывают некоторый рекомендуемый способ решения какой-то стандартной задачи. Знание паттернов, умение их видеть и применять для решения конкретной задачи - это признак опытного программиста.

Литература:

1. К.Александер — «A Pattern Language: Towns, Buildings, Constructions», 1977 год. Это первое упоминание о паттернах проектирования.
2. Э.Гамм, Р.Хелм, Р.Джонсон, Д.Влиссидес — «Design Patterns: Elements of Reusable Object-Oriented Software», 1995 год. Это первая книга,

где паттерны были применены к ООП. Было описано 23 паттерна проектирования.

Далее в курсе будут рассмотрены самые часто встречающиеся паттерны проектирования.

3.1.2 Виды паттернов проектирования и задачи

Классификация паттернов по уровню абстракции:

- **Низкоуровневые паттерны (идиомы) —** паттерны уровня языка программирования. Например, реализация тернарного оператора или генерация списков на лету (list comprehension);
- **Паттерны проектирования —** более абстрактные и менее привязанные к конкретному языку паттерны, использующиеся для решения больших задач. Они будут рассмотрены далее;
- **Архитектурные шаблоны —** паттерны, описывающие архитектуру программной системы полностью и не привязанные к языку программирования или решению частных задач. Например, клиент-серверная архитектура или архитектура Model-View-Controller (MVC), позволяющая разделить бизнес логику и её графическое представление.

Примеры архитектурных шаблонов:

- Model-View-Controller (MVC);
- Model-View-Presenter;
- Model-View-View Model;
- Presentation-Abstraction-Control;
- Naked Objects;
- Hierarchial Model-View-Controller;
- View-Interactor-Presenter-Entity-Routing (VIPER).

Виды паттернов проектирования и их назначение:
Структурные шаблоны модифицируют структуру объектов. Могут быть использованы для получения более сложных структур из классов или для реализации альтернативного доступа к объектам.
Основные представители:

- Адаптер (adapter) — взаимодействие несовместимых объектов;
- Мост (bridge) — разделение абстракции и реализации;

- Компоновщик (composite) — агрегирование нескольких объектов в одну структуру;
- Декоратор (decorator) — динамическое создание дополнительного поведения объекта;
- Фасад (facade) — сокрытие сложной структуры за одним объектом, являющимся общей точкой доступа;
- Приспособленец (flyweight) — общий объект, имеющий различные свойства в разных местах программы;
- Заместитель (proxy) — контроль доступа к некоторому объекту.

Порождающие паттерны используются при создании различных объектов. Они призваны разделить процесс создания объекта и использования их системой. Например, для реализации способа создания объектов независимо от типов создаваемых объектов, или для сокрытия процесса создания объекта от системы, которая его использует.

- Абстрактная фабрика (abstract factory) — создание семейств взаимосвязанных объектов;
- Строитель (builder) — сокрытие инициализации для сложного объекта;
- Фабричный метод (factory method) — общий интерфейс создания экземпляров подклассов некоторого класса;
- Отложенная инициализация (lazy initialization) — создание объекта только при доступе к нему;
- Пул одиночек (multiton) — повторное использование сложных объектов вместе повторного создания;
- Прототип (object pool) — упрощение создания объекта за счёт клонирования уже имеющегося;
- Одиночка (singleton) — объект, присутствующий в системе в единственном экземпляре.

Поведенческие паттерны описываю способы реализации взаимодействия между объектами различных типов. Самые основные примеры:

- Цепочка обязанностей (chain of responsibility) — обработка данных несколькими объектами;
- Интерпретатор (interpreter) — решение частой незначительно изменяющейся задачи;

- Итератор (iterator) — последовательный доступ к объекту-коллекции;
- Хранитель (memento) — сохранение и восстановление объекта;
- Наблюдатель (observer) — оповещение об изменении некоторого объекта;
- Состояние (state) — изменение поведения в зависимости от состояния;
- Стратегия (strategy) — выбор из нескольких вариантов поведения объекта;
- Посетитель (visitor) — выполнение некоторой операции над группой различных объектов.

Конкурентные паттерны — особые шаблоны, реализующие взаимодействия различных процессов и потоков. Применяются в параллельном программировании.

- Блокировка (lock) — позволяет потоку захватывать общие ресурсы на период выполнения;
- Монитор (monitor) — механизм синхронизации и взаимодействия процессов, обеспечивающий доступ к общим неразделяемым ресурсам;
- Планировщик (scheduler) — позволяет планировать порядок выполнения параллельных процессов с учетом приоритетов и ограничений;
- Активный объект (actrive object) — позволяет отделять поток выполнения некоторого метода от потока, в котором данный метод был вызван.

Рис. 3.1: Виды паттернов проектирования

3.2 Паттерн Decorator

3.2.1 Задача паттерна Decorator

Декоратор (decorator) — структурный паттерн, который используется для динамического создания дополнительного поведения объекта.

Пусть для некоторого множества задач определен класс, объекты которого эту задачу могут решать. И пусть для некоторых из объектов требуется дополнительная функциональность, реализованная в оригинальном классе. Это большая проблема. Классическим решением этой проблемы было бы создание подкласса, в котором реализована необходимая функциональность. Но что если заранее неизвестно, какая функциональность или какие свойства нужны объекту? Можно попытаться реализовать классы-потомки со всеми возможными комбинациями нужных свойств. Но число комбинаций растёт экспоненциально и становится очень большим.

Декоратор позволяет красиво решить проблему реализации дополнительной функциональности. Разберёмся, как он работает на примере класса «пицца». Свойства: размер, набор ингредиентов и стоимость. Определим абстрактный базовый класс Pizza с методами посчитать стоимость (get_cost()), узнать ингредиенты (get_undependience()) и посмотреть размер (get_size()). Чем больше пицца и её набор ингредиентов, тем больше её стоимость.

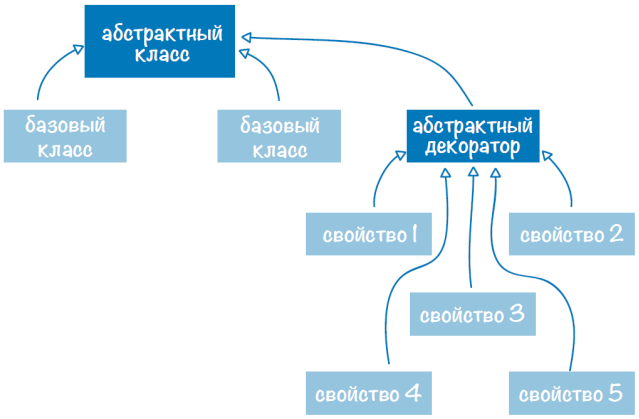

Рис. 3.2: Схема абстрактного декоратора

Определим реализацию этого класса — margarita, состоящая из теста, томатной пасты и сыра. Все остальные пиццы являются лишь её улучшениями.
Объявим абстрактный класс декоратор, в котором объявим конструктор, принимающий на вход пиццу. Именно наследники этого абстрактного декоратора и будут изменять нашу Маргариту, делая её вкуснее. Добавляемые ингредиенты: мясо, помидоры, перец и оливки. Пицца может быть трёх размеров: большой, средней и маленькой.

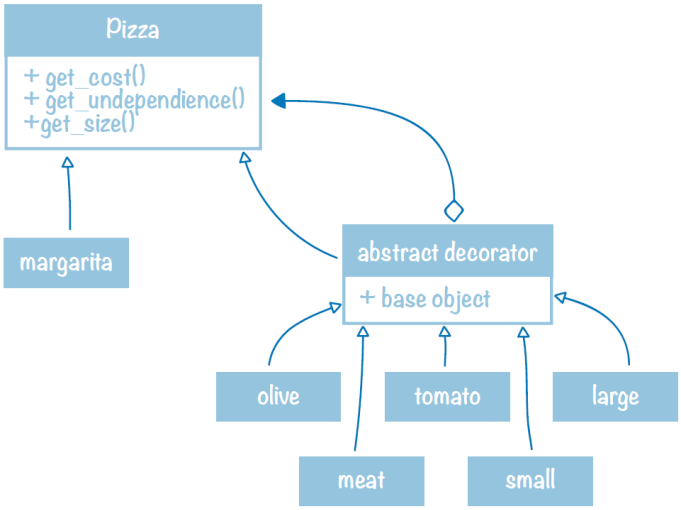

Рис. 3.3: Декоратор на примере пиццы

Пусть пицца уже частично готова, и на данном шаге нужно добавить в неё мясо. Тогда список ингредиентов будет пополнен, и стоимость пиццы возрастёт на стоимость мяса, а размер не изменится. Если же изменить размер пиццы, то изменится соответствующее свойство пиццы и мультипликатор стоимости.

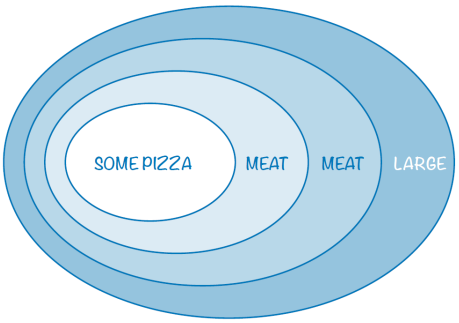

Рис. 3.4: Вложенность декораторов

Использование паттерна декоратор позволяет динамически добавлять объекту функциональность, которой у него до этого не было. Реализуется же он путём создания абстрактного базового класса, абстрактного декоратора для этого класса и их наследников — базовых классов и базовых декорато-

ров. Применение декоратора к объекту заключается в оборачивании нашего объекта в некоторый декоратор с дополнительной функциональностью. К одному объекту может быть применено произвольное число декораторов.

3.2.2 Реализация декоратора класса

Рассмотрим реализацию конкретного примера паттерна «Декоратор». Пусть мы занимаемся разработкой игры. В нашей игре есть различные животные: некоторые живут на суше, некоторые в воде. Есть хищники и травоядные. Есть быстрые и медленные. Определим основной объект животное (animal) и декораторы, которые могут к нему применяться. Напишем абстрактный базовый класс Существо (creature) с методами: кормиться, двигаться и мычать.

```
from abc import ABC, abstractmethod
class Creature(ABC):
    @abstractmethod
    def feed(self):
        pass

    @abstractmethod
    def move(self):
        pass

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

Теперь объявим базовое животное (наследник класса «Существо»), которое будет травоядным (т.е. писать нам "I eat grass"), способным ходить (т.е. выдавать нам "I walk forward" и что-то кричать:

```
class Animal(Creature):
    def feed(self):
        print("I eat grass")

    def move(self):
        print("I walk forward")

    def make_noise(self):
        print("WOOO!")
```

Для создания иерархии декораторов сначала опишем абстрактный декоратор (он так же наследуется от базового класса) , от которого уже будут наследоваться конкретные реализации, добавляющие нашему объекту допол-

нительную функциональность. Конструктор (__init__) должен принимать обязательный аргумент self и в качестве дополнительного аргумента наш декорируемый объект. Объявим у декоратора необходимые методы, взятые из декорируемого объекта (пока без изменений): feed, move, make_noise. В этих методах мы будем вызывать соответствующий метод базового объекта.

```
class AbstractDecorator(Creature):
    def __init__(self, obj):
        self.obj = obj

    def feed(self):
        self.obj.feed()

    def move(self):
        self.obj.move()

    def make_noise(self):
        self.obj.make_noise()
```

Теперь создадим несколько декораторов. Первый декоратор — водоплавающее животное. Водоплавающие вместо того, чтобы ходить, будут плавать. И не будут уметь говорить. Переопределим у них методы move и make_noise.

```
class Swimming(AbstractDecorator):
    def move(self):
        print("I swim")

    def make_noise(self):
        print("...")
```

Следующее свойство, которое мы хотим добавить — хищник, который будет поедать других животных. Создадим класс Хищник и переопределим метод feed так, что он будет есть каких-то других животных.

```
class Predator(AbstractDecorator):
    def feed(self):
        print("I eat other animals")
```

Определим третий класс свойств — скорость движения. Класс Fast будет классом быстрых животных. Переопределим у него только метод move, который кроме обычного перемещения ещё и будет делать это быстро.

```
class Fast(AbstractDecorator):
    def move(self):
        self.obj.move()
        print("Fast!")
```

Создадим базовое животное и выполним все его методы:

```
animal = Animal()
animal.feed()
animal.move()
animal.make_noise()

print()
```

```
I eat grass
I walk forward
WOOO!
```

Теперь сделаем из нашего животного водоплавающее. Для этого определим, что оно является водоплавающим и в качестве декорируемого объекта укажем наше животное.

```
swimming = Swimming(animal)
swimming.feed()
swimming.move()
swimming.make_noise()
```

```
I eat grass
I swim
...
```

Оно ест траву, плавает и не издаёт звуков. Сделаем из нашего животного хищника:

```
predator = Predator(animal)
predator.feed()
predator.move()
predator.make_noise()
```

```
I eat other animals
I swim
...
```

Теперь наше водоплавающее ест других животных и не издаёт звуков. Сделаем нашего водоплавающего хищника быстрым:

```
fast = Fast(animal)
fast.feed()
fast.move()
fast.make_noise()
```

```
I eat other animals
I swim
Fast!
...
```

Получили быстрого водоплавающего хищника. Рассмотрим такое свойство,

что можно применять несколько одинаковых декораторов к одному объекту. Мы можем сделать наше животное ещё быстрее:

```
faster = Fast(animal)
faster.feed()
faster.move()
faster.make_noise()
```

```
I eat other animals I swim Fast! Fast!
```

Таким образом, обложив наше животное декораторами, мы получили очень быстрого водоплавающего хищника.

Теперь поговорим о снятии декораторов. У каждого декоратора есть базовый объект, на котором он строится. Доступ к нему можно получить обращением к полю base. Например:

```
faster.base
```

```
<__main__.Fast at ...
```

```
faster.base.base
```

```
<__main__.Predator at ...
```

Для снятия эффекта "Хищник" с животного присвоим базовому(для быстрого) объекта животное, которое было раньше по иерархии декораторов.

```
faster.base.base = faster.base.base.base
faster.feed()
faster.move()
faster.make_noise()
```

```
I eat grass I swim Fast! Fast!
```

Видим, что животное так же плавает и быстро бегает, но теперь питается травой. Исходный код decorator.ipynb

3.3 Паттерн Adapter

3.3.1 Задача паттерна Adapter

Адаптер (adapter, "обёртка") — структурный паттерн, который используется для реализации взаимодействия несовместимых объектов.

Пусть есть объект и система, с которой этот объект должен взаимодействовать. При этом, его интерфейс не может быть напрямую встроен в систему. В таких случаях помогает адаптер. Он позволяет создать объект, который может обеспечить взаимодействие нашего исходного объекта с системой. Пример адаптера в реальной жизни: переходник для розеток разных типов.

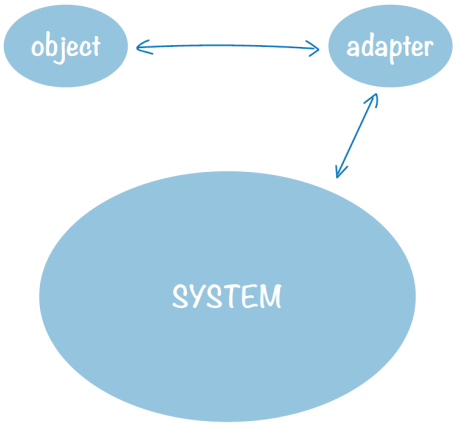

Рис. 3.5: Взаимодействия адаптера

В программировании же может быть, например, такая ситуация: имеется консольная утилита, которая читает данные из файла, обрабатывает их и сохраняет результат в другой файл. И мы пользуемся системой, которая может создавать данные для обработки, но обрабатывать их не умеет. В качестве обработчика ей нужен объект, принимающий данные в виде списка и возвращать результат обработки также в виде списка.

Как видно, интерфейсы консольной утилиты и системы совершенно несовместимы. В данном случае адаптер будет представлять из себя объект с интерфейсом ввода-вывода, который нужен системе, при этом внутри адаптера данные преобразуются в файл, который передаётся на обработку консольной утилите. Результат считывается из выходного файла и передаётся на вход системе. Проблема решена.

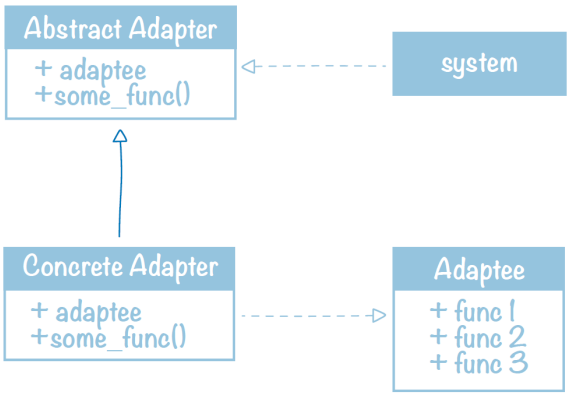

Рис. 3.6: UML-диаграмма адаптера

Подобным образом работают обёртки над некоторыми библиотеками, такими как fastText или Vowpal Wabbit. В машинном обучении же используется библиотека Scikit-Learn, в которой многие объекты по сути являются обёртками над классами из других библиотек. При этом объекты из этой библиотеки имеют понятные названия и стандартизованный интерфейс доступа, что позволяет гораздо удобнее работать с ними, а не с изначальными классами из библиотек.

3.3.2 Реализация адаптера класса

Пусть у нас есть некоторая система, которая берет текст, делает его предварительную обработку, а дальше хочет вывести слова в порядке убывания их частоты. Но собственного обработчика у системы нет. Она принимает в качестве обработчика некоторый объект TextProcessor с данным интерфейсом:

```
from System import *
import re
from abc import ABC, abstractmethod

class System:
    def __init__(self, text):
        tmp = re.sub(r'\W', ' ', text.lower())
        tmp = re.sub(r' +', ' ', tmp).strip()
        self.text = tmp

    def get_processed_text(self, processor):
        result = processor.process_text(self.text)
        print(*result, sep = '\n')

class TextProcessor:
    @abstractmethod
    def process_text(self, text):
        pass
```

В качестве обработчика есть некоторый счётчик слов, который по заданному тексту может посчитать в нём слова(count_words()), может сказать, сколько раз встретилось конкретное слово (get_count()) и может вывести частотный словарь всех встреченных слов (get_all_words()).

```
class WordCounter:
    def count_words(self, text):
        self.__words = dict()
        for word in text.split():
            self.__words[word] = self.__words.get(word, 0)
              + 1

    def get_count(self, word):
        return self.__words.get(word, 0)

    def get_all_words(self):
        return self.__words.copy()
```

Создадим объект системы и передадим в него некоторый текст.

```
system = System(text)
system.text
```

```
Design Patterns: Elements of Reusable Object-Oriented Software is a
software engineering book describing software design patterns...
```

Это отрывок из статьи в Википедии, про книгу Elements of reusable object-oriented software, о которой было рассказано ранее. Создадим наш обработчик и передадим в него текст.

```
counter = WordCounter()
system.get_processed_text(counter)
```

```
AttributeError: 'WordCounter' object has no attribute 'process_text'
```

Эта ошибка появляется потому, что интерфейсы обработчика и системы совершенно несовместимы. Напишем адаптер, который позволит системе использовать наш обработчик. Объявим класс WordCounterAdapter, который будет наследоваться от нашего абстрактного обработчика TextProcessor. Объявим конструктор __init__, принимающий в качестве дополнительного аргумента объект, который мы хотим адаптировать, и сохраняющий этот объект в некоторую переменную класса. Метод process_text() будет принимать на вход текст и возвращать слова в порядке убывания частоты их вхождений. Чтобы вывести список слов используем метод get_all_words(), нашего счётчика. Для получения списка из словаря всех слов воспользуемся методом keys(). И вернём наш отсортированный массив слов с помощью return sorted() сортируем список слов по ключу. В данном случае ключ — это количество встреч конкретного слова в частотном словаре. Для этого воспользуемся лямбда-функцией, которая выдаёт количество встреч слова в нашем счётчике (используем метод get_count() от адаптируемого объекта). reverse=True для сортировки в порядке убывания.

```
class WordCounterAdapter(TextProcessor):
    def __init__(self, adaptee):
        self.adaptee = adaptee

    def process_text(self, text):
        self.adaptee.count_words(text)
        words = self.adaptee.get_all_words().keys()
        return sorted(words, key=lambda
                              x:self.adaptee.get_count(x),
                      reverse=True)
```

Объявим адаптер, который будет экземпляром класса WordCounterAdapter и принимать в качестве адаптируемого объекта объявляенный ранее счётчик. Обработаем текст с ипоользованием адаптера и системы, а в качестве обработчика для get_processed_text передадим наш адаптер:

```
adapter = WordCounterAdapter(counter)
system.get_processed_text(adapter)
```

```
the
and
software
design
of ...
```

Видим список слов, причём их порядок примерно совпадает с частотой встречи в английском языке, так что результат похож на правду.

Исходный код адаптера

3.4 Паттерн Observer

3.4.1 Задача паттерна Observer

Наблюдатель (observer) часто применяется в самых разных задачах: от веб программирования и gamedevelopment-а до обработки сложных физических экспериментов. Это поведенческий шаблон, оповещающий об изменении некоторого объекта.

Пусть в некоторой системе есть наблюдаемый объект, который со временем изменяет своё состояние, и объект-наблюдатель, который отслеживает состояние наблюдаемого объекта и который хочет своевременно узнавать об изменениях в наблюдаемом объекте. Самый простой способ это сделать — напрямую спрашивать у наблюдаемого объекта, произошли ли с ним какие-то изменения. Но тогда с какой частотой наблюдатель должен запрашивать эти изменения? 1, 100, 1000 раз в секунду? И каждый раз при запросе наблюдаемый объект вместо того, чтобы продолжать делать свою задачу, должен отвечать наблюдателю на вопрос о своём состоянии. Такое можно использовать, если наблюдатель один. Но если за обновлениями следят тысячи объектов, нужен иной подход: научить наблюдаемый объект самостоятельно ставить в известность наблюдателей при возникновении изменений.

Именно для этого и применяется данный паттерн. Он позволяет от pull-системы отслеживания изменений перейти к push-системе. На примере социальной сети Вконтакте разберёмся в устройстве паттерна. В ней есть сообщества, в которых периодически публикуются новости, и подписчики, которые при появлении новых записей хотели бы выидеть их в разделе новостей. Таким образом, мы имеем отношение «один — много» (или single to multiple). Реализуем абстрактные интерфейсы группы и её подписчика.

Группа может рассылать подписчикам новости. У каждой группы для этого существуют методы: подписать пользователя на обновления (subscribe()), отписать его (unsubscribe()) и рассылка подписчикам сообщений об изменении состояния (notify()). У абстрактного интерфейса подписчика определяем метод update(), описывающий действия, которые будут выполнены при получении сообщений.

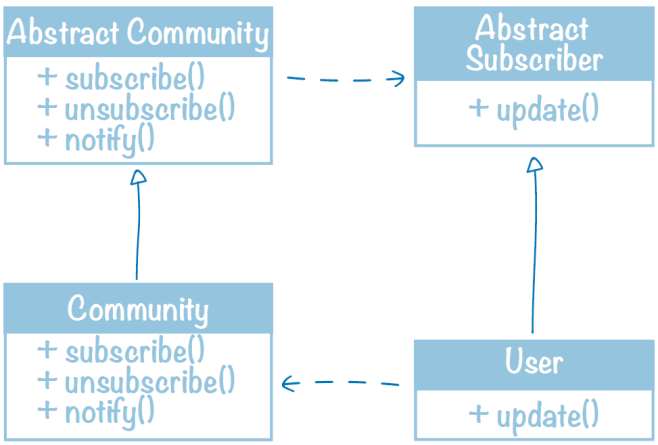

Рис. 3.7: UML-диаграмма наблюдателя

В конкретных реализациях подписчиков эти действия определяются поразному. Метод «подписать пользователя» добавляет пользователя во множество возможных подписчиков, а метод «отписать» удаляет его из этого множества. Метод Request(), который оповещает пользователей об изменениях, вызывает у конкретных пользователей метод update.

Использование такой структуры позволяет минимизировать нагрузку на наблюдаемый объект за счёт того, что вместо постоянных запросов об изменениях от пользователей, он сам сообщает о том, что с ним происходит, когда изменения появляются.

Паттерн наблюдатель используется, когда существуют два объекта: наблюдающий и наблюдаемый. Он предназначен для организации системы оповещений наблюдающего объекта, об изменении состояния наблюдаемого объекта. Вместо pull-системы (когда наблюдатель самостоятельно запрашивает наблюдаемый объект об изменениях), используется push-система(наблюдаемый объект оповезает наблюдателя об изменениях). Оповещение происходит путём вызова метода update() у всех подписчиков.

3.4.2 Реализация паттерна Наблюдатель

Пусть у нас есть менеджер уведомлений, который может рассылать уведомления его подписчикам. И есть сами подписчики двух типов: printer, который печатает сообщение менеджера и notifier, который уведомляет о том, что сообщение пришло.

Сначала создадим класс notifier manager. В инициализаторе объявим пустой список подписчиков __subscribers. Для того, чтобы подписчик мог подписаться на уведомления менеджера, добавим методы subscribe (добавляет подписчика в список) и unsubscribe (удаляет подписчика из списка). Важной частью наблюдателя является отправка уведомлений, и для этого объявим у менеджера метод notify(), который отправит уведомление всем подписчикам (т.е. вызовет у них метод update() с сообщением в качестве параметра).

```
from abc import ABC, abstractmethod

class NotificationManager:
    def __init__(self):
        self.__subscribers = set()

    def subscribe(self, subscriber):
        self.__subscribers.add(subscriber)

    def unsubcribe(self, subscriber):
        self.__subscribers.remove(subscriber)

    def notify(self, message):
        for subscriber in self.__subscribers:
            subscriber.update(message)
```

Теперь объявим абстрактного наблюдателя AbstractObserver, который наследуется от абстрактного базового класса. Дадим ему имя в инициализаторе и объявим у него метод update (пока абстрактный и пустой)

```
class AbstractObserver(ABC):
    @abstractmethod
    def update(self, message):
        pass
```

Теперь объявим две конкретные реализации: notifier (просто печатает, что сообщение пришло) и printer (печатает и сам текст сообщения). В них нам нужно по-разному реализовать метод update.

```
class MessageNotifier(AbstractObserver):
    def __init__(self, name):
        self.__name = name

    def update(self, message):
        print(f'{self.__name} recieved!')

class MessagePrinter(AbstractObserver):
    def __init__(self, name):
        self.__name = name

    def update(self, message):
        print(f'{self.__name} recieved: {message}')
```

Создадим один notifier и два printer-а. Далее объявим менеджер уведомлений и подпишем на него наших наблюдателей. И отправим им уведомление методом notify().

```
notifier1 = MessageNotifier("Notifier1")
printer1 = MessagePrinter("Printer1")
printer2 = MessagePrinter("Printer2")

manager = NotificationManager()
manager.subscribe(notifier1)
manager.subscribe(printer1)
manager.subscribe(printer2)

manager.notify("Hi!")
```

```
Notifier1 recieved message!
Printer1 recieved message: Hi!
Printer2 recieved message: Hi!
```

Всё работает, как мы и хотели: notifier получил сообщение, а принтеры его ещё и распечатали.

Исходный код наблюдателя