Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт цифрового развития
Кафедра инфокоммуникаций

РЕФЕРАТ
На тему: «Паттерн Декоратор vs. Паттерн Адаптер: сравнение их использования для расширения функциональности»

Выполнил:
Шайдеров Дмитрий Викторович
3 курс, группа ИВТ-б-о-21-1,
09.03.01 «Информатика и вычислительная техника»,
направленность (профиль) «Программное обеспечение средств вычислительной техники и автоматизированных систем», очная форма обучения

_____
(подпись)

Руководитель практики:
Воронкин Р.А., канд. тех. наук, доцент,
доцент кафедры инфокоммуникаций

_____
(подпись)

Отчет защищен с оценкой _____ Дата защиты_____

Ставрополь, 2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Шаблоны, или паттерны программирования - это повторяющиеся решения типовых задач, возникающих в процессе разработки программного обеспечения. Паттерны проектирования были созданы, чтобы обеспечить более легкую переносимость, легкость поддержки, улучшение расширяемости и многоразовое использование кода.

Паттерны можно разделить на три группы:

1. Порождающие – предоставляют возможность создания контролируемым образом, инициализации и конфигурации объектов, классов и типов данных на основе требуемых критериев.
2. Структурные – помогают организовать структуры связанных объектов и классов, предоставляя новые функциональные возможности.
3. Поведенческие – направлены на выявление общих моделей взаимодействия между объектами.

Паттерны Декоратор и Адаптер относятся к группе структурных паттернов, они позволяют определять отношения между различными элементами системы для создания более сложных структур.

Актуальность: Использование паттернов проектирования делает систему более устойчивой к смене требований. При этом дальнейшая доработка системы становится гораздо проще. Применение паттернов очень полезно и при интеграции информационных систем организации. Структурные паттерны определяют всевозможные структуры высокого уровня сложности, которые вносят изменения в интерфейс уже созданных объектов или его реализацию. Это позволяет упростить процесс разработки и сделать программу более оптимизированной.

Цель: изучить паттерны Адаптер и Декоратор, их сущности и механизмы действия, сравнить паттерны и выявить сильные и слабые качества каждого, их сходства и различия.

1. ПАТТЕРН АДАПТЕР

1.1 Общее описание паттерна Адаптер

Адаптер — это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.

Это объект-переводчик, который трансформирует интерфейс или данные одного объекта в такой вид, чтобы он стал понятен другому объекту.

При этом адаптер оборачивает один из объектов, так что другой объект даже не знает о наличии первого. Например, вы можете обернуть объект, работающий в метрах, адаптером, который бы конвертировал данные в футы.

Адаптеры могут не только переводить данные из одного формата в другой, но и помогать объектам с разными интерфейсами работать сообща. Это работает так:

1. Адаптер имеет интерфейс, который совместим с одним из объектов.
2. Поэтому этот объект может свободно вызывать методы адаптера.
3. Адаптер получает эти вызовы и перенаправляет их второму объекту, но уже в том формате и последовательности, которые понятны второму объекту.

Иногда возможно создать даже двухсторонний адаптер, который работал бы в обе стороны.

Шаблон Адаптер позволяет в процессе проектирования не принимать во внимание возможные различия в интерфейсах уже существующих классов. Если есть класс, обладающий требуемыми методами и свойствами (по крайней мере, концептуально), то при необходимости всегда можно воспользоваться шаблоном Адаптер для приведения его интерфейса к нужному виду.

1.2 Структура паттерна Адаптер

1.2.1 Адаптер объектов

Эта реализация использует агрегацию: объект адаптера «оборачивает», то есть содержит ссылку на служебный объект. Такой подход работает во всех языках программирования.

1. Клиент — это класс, который содержит существующую бизнес-логику программы.
2. Клиентский интерфейс описывает протокол, через который клиент может работать с другими классами.
3. Сервис — это какой-то полезный класс, обычно сторонний. Клиент не может использовать этот класс напрямую, так как сервис имеет непонятный ему интерфейс.
4. Адаптер — это класс, который может одновременно работать и с клиентом, и с сервисом. Он реализует клиентский интерфейс и содержит ссылку на объект сервиса. Адаптер получает вызовы от клиента через методы клиентского интерфейса, а затем переводит их в вызовы методов обёрнутого объекта в правильном формате.
5. Работая с адаптером через интерфейс, клиент не привязывается к конкретному классу адаптера. Благодаря этому, вы можете добавлять в программу новые виды адаптеров, независимо от клиентского кода. Это может пригодиться, если интерфейс сервиса вдруг изменится, например, после выхода новой версии сторонней библиотеки.

1.2.2 Адаптер классов

Эта реализация базируется на наследовании: адаптер наследует оба интерфейса одновременно. Такой подход возможен только в языках, поддерживающих множественное наследование, например, C++.

Адаптер классов не нуждается во вложенном объекте, так как он может одновременно наследовать и часть существующего класса, и часть сервиса.

1.3 Решаемые задачи паттерна Адаптер

- Разделение ответственностей

Согласно принципу единственной ответственности (Single Responsibility) один класс должен иметь одну область ответственности. Когда существует сущность на которую возложены 2 или более ответственностей, они разделяются посредством вынесения каждой в адаптер. Ответственности делаются независимыми друг от друга.

- Подмена одного интерфейса на другой

Позволяет преобразовывать интерфейс одного класса в интерфейс другого, который ожидают клиенты.

Интерфейс является абстракцией, которая определяет, как объект может быть использован, независимо от его конкретной реализации. Он представляет из себя набор методов и свойств класса.

- Использование сторонних библиотек

Использовать класс из нестандартной библиотеки следует через адаптер. Делается это для решения проблем совместимости интерфейсов библиотеки и рабочей среды, адаптации кода к новым версиям библиотеки, устранения сильной зависимости кода от библиотеки. Возможность контролировать и модифицировать адаптер делает код более гибким и легким в сопровождении.

- Расширение базового интерфейса

Интерфейс базового класса в общем случае плохо расширять, так как нарушаются принципы инкапсуляции и единственной ответственности (Single Responsibility). Код становится тяжелее поддерживать и модифицировать. Появляется потребность в дополнительных проверках и приведениях типов при работе с указателями на базовый класс.

При использовании адаптера расширение может относиться не к одному классу, а к целой иерархии классов.

1.4 Реализация паттерна Адаптер

1. Убедитесь, что у вас есть два класса с несовместимыми интерфейсами:

- полезный сервис — служебный класс, который вы не можете изменять (он либо сторонний, либо от него зависит другой код);
- один или несколько клиентов — существующих классов приложения, несовместимых с сервисом из-за неудобного или несовпадающего интерфейса.

2. Опишите клиентский интерфейс, через который классы приложения смогли бы использовать класс сервиса.

3. Создайте класс адаптера, реализовав этот интерфейс.

4. Поместите в адаптер поле, которое будет хранить ссылку на объект сервиса. Обычно это поле заполняют объектом, переданным в конструктор адаптера. В случае простой адаптации этот объект можно передавать через параметры методов адаптера.

5. Реализуйте все методы клиентского интерфейса в адаптере. Адаптер должен делегировать основную работу сервису.

6. Приложение должно использовать адаптер только через клиентский интерфейс. Это позволит легко изменять и добавлять адаптеры в будущем.

1.5 Пример реализация паттерна Адаптер на языке Python

Этот пример показывает структуру паттерна Адаптер, а именно — из каких классов он состоит, какие роли эти классы выполняют и как они взаимодействуют друг с другом.

Листинг программы:

```
class Target:
    """
    Целевой класс объявляет интерфейс, с которым может работать клиентский код.
    """

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


class Adaptee:
    """
    Адаптируемый класс содержит некоторое полезное поведение, но его интерфейс
    несовместим с существующим клиентским кодом. Адаптируемый класс нуждается в
    некоторой доработке, прежде чем клиентский код сможет его использовать.
    """

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


class Adapter(Target, Adaptee):
    """
    Адаптер делает интерфейс Адаптируемого класса совместимым с целевым
    интерфейсом благодаря множественному наследованию.
    """
```

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


    def client_code(target: "Target") -> None:
        """
        Клиентский код поддерживает все классы, использующие интерфейс
Target.
        """

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


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

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

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

1.6 Преимущества и недостатки паттерна Адаптер

Преимущества:

- отделяет и скрывает от клиента подробности преобразования различных интерфейсов
- позволяет адаптировать интерфейс к требуемому
- позволяет разделить роли сущности
- дает возможность независимо развивать различные ответственности сущности
- расширение интерфейса

Недостатки:

- необходимость плодить много классов приводит к увеличению количества времени и памяти,необходимых для исполнения программы
- дублирование кода (в различных конкретных адаптерах может требоваться одна и та же реализация методов)
- часто адаптер должен иметь доступ к реализации класса

2. ПАТТЕРН ДЕКОРАТОР

2.1 Общее описание паттерна Декоратор

Декоратор — это структурный паттерн проектирования, который позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».

Наследование — это первое, что приходит в голову многим программистам, когда нужно расширить какое-то существующее поведение. Но механизм наследования имеет несколько досадных проблем.

- Он статичен. Вы не можете изменить поведение существующего объекта. Для этого вам надо создать новый объект, выбрав другой подкласс.
- Он не разрешает наследовать поведение нескольких классов одновременно. Из-за этого вам приходится создавать множество подклассов-комбинаций для получения совмещённого поведения.

Одним из способов обойти эти проблемы является замена наследования агрегацией либо композицией . Это когда один объект содержит ссылку на другой и делегирует ему работу, вместо того чтобы самому наследовать его поведение. Как раз на этом принципе построен паттерн Декоратор.

Декоратор имеет альтернативное название — обёртка. Оно более точно описывает суть паттерна: вы помещаете целевой объект в другой объект-обёртку, который запускает базовое поведение объекта, а затем добавляет к результату что-то своё.

Оба объекта имеют общий интерфейс, поэтому для пользователя нет никакой разницы, с каким объектом работать — чистым или обёрнутым. Вы можете использовать несколько разных обёрток одновременно — результат будет иметь объединённое поведение всех обёрток сразу.

2.2 Структура паттерна Декоратор

1. Компонент задаёт общий интерфейс обёрток и оборачиваемых объектов.

2. Конкретный компонент определяет класс оборачиваемых объектов. Он содержит какое-то базовое поведение, которое потом изменяют декораторы.

3. Базовый декоратор хранит ссылку на вложенный объект-компонент. Им может быть как конкретный компонент, так и один из конкретных декораторов. Базовый декоратор делегирует все свои операции вложенному объекту. Дополнительное поведение будет жить в конкретных декораторах.

4. Конкретные декораторы — это различные вариации декораторов, которые содержат добавочное поведение. Оно выполняется до или после вызова аналогичного поведения обёрнутого объекта.

5. Клиент может оборачивать простые компоненты и декораторы в другие декораторы, работая со всеми объектами через общий интерфейс компонентов.

2.3 Решаемые задачи паттерна Декоратор

- динамическое, прозрачное для клиентов добавление обязанностей объектам
- реализация обязанностей, которые могут быть сняты с объекта
- применяется, когда расширение путем наследования по каким-то причинам неудобно или невозможно

Наследование от некоторых классов может быть запрещено.

2.4 Реализация паттерна Декоратор

1. Убедитесь, что в вашей задаче есть один основной компонент и несколько опциональных дополнений или надстроек над ним.

2. Создайте интерфейс компонента, который описывал бы общие методы как для основного компонента, так и для его дополнений.

3. Создайте класс конкретного компонента и поместите в него основную бизнес-логику.

4. Создайте базовый класс декораторов. Он должен иметь поле для хранения ссылки на вложенный объект-компонент. Все методы базового декоратора должны делегировать действие вложенному объекту.

5. И конкретный компонент, и базовый декоратор должны следовать одному и тому же интерфейсу компонента.

6. Теперь создайте классы конкретных декораторов, наследуя их от базового декоратора. Конкретный декоратор должен выполнять свою добавочную функцию, а затем (или перед этим) вызывать эту же операцию обёрнутого объекта.

7. Клиент берёт на себя ответственность за конфигурацию и порядок обёртывания объектов.

2.5 Пример реализации паттерна Декоратор на языке Python

Этот пример показывает структуру паттерна Декоратор, а именно — из каких классов он состоит, какие роли эти классы выполняют и как они взаимодействуют друг с другом.

Листинг программы:

```
class Component():
    """
    Базовый интерфейс Компонента определяет поведение, которое изменяется
    декораторами.
    """
```

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


class ConcreteComponent(Component):
    """
    Конкретные Компоненты предоставляют реализации поведения по умолчанию. Может
    быть несколько вариаций этих классов.
    """

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


class Decorator(Component):
    """
    Базовый класс Декоратора следует тому же интерфейсу, что и другие
    компоненты. Основная цель этого класса - определить интерфейс обёртки для
    всех конкретных декораторов. Реализация кода обёртки по умолчанию может
    включать в себя поле для хранения завёрнутого компонента и средства его
    инициализации.
    """

    _component: Component = None

    def __init__(self, component: Component) -> None:
```

```
        self._component = component

    @property
    def component(self) -> Component:
        """
        Декоратор делегирует всю работу обёрнутому компоненту.
        """

        return self._component

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


class ConcreteDecoratorA(Decorator):
    """
    Конкретные Декораторы вызывают обёрнутый объект и изменяют его результат
    некоторым образом.
    """

    def operation(self) -> str:
        """
        Декораторы могут вызывать родительскую реализацию операции, вместо того,
        чтобы вызвать обёрнутый объект напрямую. Такой подход упрощает
        расширение классов декораторов.
        """
        return f"ConcreteDecoratorA({self.component.operation()})"
```

```
class ConcreteDecoratorB(Decorator):
    """
    Декораторы могут выполнять своё поведение до или после вызова обёрнутого
    объекта.
    """

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


def client_code(component: Component) -> None:
    """
    Клиентский код работает со всеми объектами, используя интерфейс Компонента.
    Таким образом, он остаётся независимым от конкретных классов компонентов, с
    которыми работает.
    """

    # ...

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

    # ...


if __name__ == "__main__":
```

```
    # Таким образом, клиентский код может поддерживать как простые компоненты...
    simple = ConcreteComponent()
    print("Client: I've got a simple component:")
    client_code(simple)
    print("\n")

    # ...так и декорированные.
    #
    # Обратите внимание, что декораторы могут обёртывать не только простые
    # компоненты, но и другие декораторы.
    decorator1 = ConcreteDecoratorA(simple)
    decorator2 = ConcreteDecoratorB(decorator1)
    print("Client: Now I've got a decorated component:")
    client_code(decorator2)
```

2.6 Преимущества и недостатки паттерна Декоратор

Преимущества:

- Гибкость: возможность добавлять желаемую реализацию к любому классу. Появлется возможность "декорировать" декораторы.
- Отсутсвие разрастания иерархии.
- Позволяет избежать перегруженных функциями классов на верхних уровнях иерархии.
- Возможность обращаться с декорированным объектом так же как и с исходной сущностью.
- Отсутствие дублирования кода, данный код просто уходит в конкретный декоратор.

Недостатки:

- Снижение производительности программы.

Не используя декораторы можно напрямую пользоваться методом класса. Декоратор же может долго полиморфно совершать цепочку вызовов, что сказывается на времени выполнения программы.

- Вся ответственность за декорирование "ложится на плечи" программиста

Нет сущности, которая бы полностью отвечала за последовательность декорирования: за порядок вызова и оборачивания декораторов

- В случае если в цепочке вызовов декоратора появляется необходимость изменить какую-либо обертку или удалить ее, то приходится заново оборачивать исходный объект.
- Необходимость в создании сущности, отвечающей за декорирование.

3. СРАВНЕНИЕ ПАТТЕРНОВ АДАПТЕР И ДЕКОРАТОР

Адаптер предоставляет совершенно другой интерфейс для доступа к существующему объекту. С другой стороны, при использовании паттерна Декоратор интерфейс либо остается прежним, либо расширяется. Причём Декоратор поддерживает рекурсивную вложенность, чего не скажешь об Адаптере.

С Адаптером вы получаете доступ к существующему объекту через другой интерфейс. Используя Декоратор, вы получаете доступ к объекту через расширенный интерфейс.

Когда применяется паттерн Адаптер?

- Когда вы хотите использовать сторонний класс, но его интерфейс не соответствует остальному коду приложения.
- Когда нужно использовать несколько существующих подклассов, но в них не хватает какой-то общей функциональности, причём расширить суперкласс вы не можете.

Когда применяется паттерн Декоратор?

- Когда вам нужно добавлять обязанности объектам на лету, незаметно для кода, который их использует.
- Когда нельзя расширить обязанности объекта с помощью наследования.

ЗАКЛЮЧЕНИЕ

Таким образом, можно заключить, что знание шаблонов программирования в Python является крайне полезным для разработчиков, позволяя им повышать эффективность и оптимизировать свой код.

Паттерн Адаптер предназначен для преобразования интерфейса одного класса в интерфейс другого. Благодаря реализации данного паттерна мы можем использовать вместе классы с несовместимыми интерфейсами.

Декоратор представляет структурный шаблон проектирования, который позволяет динамически подключать к объекту дополнительную функциональность.

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

1. А. Швец. Погружение в паттерны проектирования. URL - https://refactoring.guru/files/design-patterns-ru-demo.pdf
2. Г. Персиваль, Б. Грегори. Паттерны разработки на Python: TDD, DDD и событийно-ориентированная архитектура. URL - https://habr.com/ru/companies/piter/articles/588060/
3. Паттерны проектирования в C# и .NET. URL - https://metanit.com/sharp/patterns/
4. Структурные паттерны. URL - https://makarov-ivan.gitbook.io/patterns/patterns/structural-patterns
5. Структурные паттерны проектирования: для каких задач нужны, виды и примеры реализации. URL - https://academy.mediasoft.team/article/strukturnye-patterny-proektirovaniya-dlya-kakikh-zadach-nuzhny-vidy-i-primery-realizacii/
6. Шаблоны проектирования простым языком. Часть вторая. Структурные шаблоны. URL - https://tproger.ru/translations/design-patterns-simple-words-2
7. Что такое паттерны проектирования и как их использовать в Python. URL - https://sky.pro/media/chto-takoe-patterny-proektirovaniya-i-kak-ih-ispolzovat-v-python/