Professor Hans Noodles

41 уровень

08.10.2019 4450 6

Какие задачи решает шаблон проектирования Адаптер

Часто разработку ПО усложняет несовместимость компонентов, работающих друг с другом. Например, если нужно интегрировать новую библиотеку со старой платформой, написанной еще на ранних версиях Java, можно столкнуться с несовместимостью объектов, а точнее — интерфейсов.

Что делать в таком случае? Переписать код? Но ведь это невозможно: анализ системы займет много времени или же будет нарушена внутренняя логика работы.

Для решения этой проблемы придумали паттерн Адаптер, который помогает объектам с несовместимыми интерфейсами работать вместе. Давай посмотрим, как его использовать!

Подробнее о проблеме

Для начала сымитируем поведение старой системы. Предположим, она генерирует причины опоздания на работу или учебу. Для этого у есть интерфейс `Excuse`, который содержит методы `generateExcuse()`, `likeExcuse()` и `dislikeExcuse()`.

```
1  public interface Excuse {
2     String generateExcuse();
3     void likeExcuse(String excuse);
4     void dislikeExcuse(String excuse);
```

Этот интерфейс реализует класс `WorkExcuse`:

```
public class WorkExcuse implements Excuse {
    private String[] reasonOptions = {"по невероятному стечению обстоятельств у нас в доме закончилась
    "искусственный интеллект в моем будильнике подвел меня и разбудил на час раньше обычного. Поскольк
    "предпраздничное настроение замедляет метаболические процессы в моем организме и приводит к подавл
    private String[] sorryOptions = {"Это, конечно, не повторится, мне очень жаль.", "Прошу меня извин

    @Override
    public String generateExcuse() { // Случайно выбираем отговорку из массива
        String result = "Я сегодня опоздал, потому что " + reasonOptions[(int) Math.round(Math.random(
                sorryOptions[(int) Math.round(Math.random() + 1)];
        return result;
    }

    @Override
    public void likeExcuse(String excuse) {
        // Дублируем элемент в массиве, чтобы шанс его выпадения был выше
    }

    @Override
    public void dislikeExcuse(String excuse) {
        // Удаляем элемент из массива
    }
}
```

Протестируем пример:

```
Excuse excuse = new WorkExcuse();
System.out.println(excuse.generateExcuse());
```

Вывод:

```
Я сегодня опоздал, потому что предпраздничное настроение замедляет метаболические процессы в моем организме

Прошу меня извинить за непрофессиональное поведение.
```

Теперь представим, что ты запустил сервис, собрал статистику и заметил, что большинство пользователей сервиса — студенты вузов. Чтобы улучшить его под нужды этой группы, ты заказал у другого разработчика систему генерации отговорок специально для нее.

Команда разработчика провела исследования, составила рейтинги, подключила искусственный интеллект, добавила интеграцию с пробками на дорогах, погодой и так далее. Теперь у тебя есть библиотека генерации отговорок для студентов, однако интерфейс взаимодействия с ней другой — `StudentExcuse`:

```
public interface StudentExcuse {
    String generateExcuse();
    void dislikeExcuse(String excuse);
```

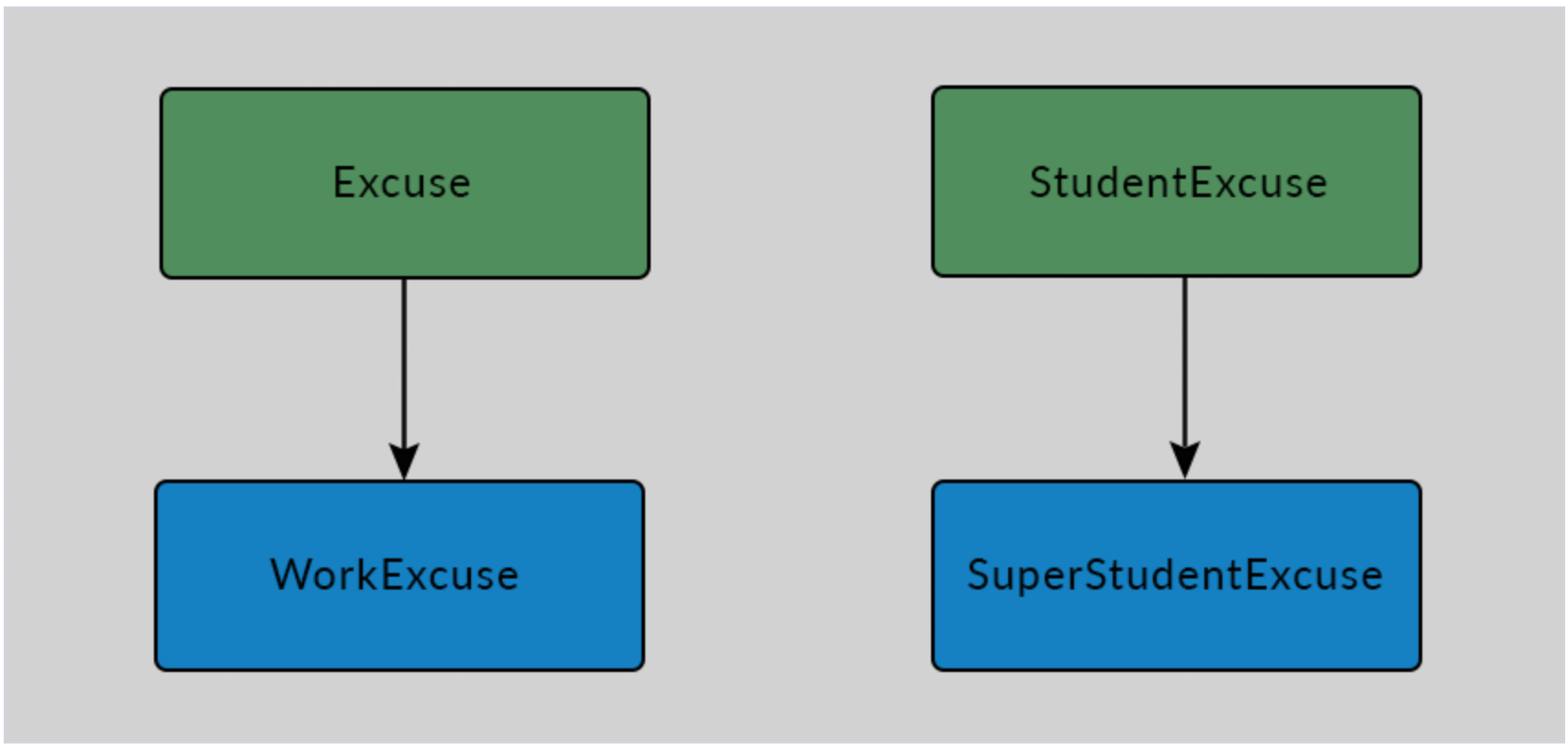
У интерфейса есть два метода: `generateExcuse`, который генерирует отговорку, и `dislikeExcuse`, который блокирует отговорку, чтобы она не появлялась в дальнейшем.

Библиотека стороннего разработчика закрыта для редактирования — ты не можешь изменять его исходный код.

В итоге в твоей системе есть два класса, реализующие интерфейс `Excuse`, и библиотека с классом `SuperStudentExcuse`, который реализует интерфейс `StudentExcuse`:

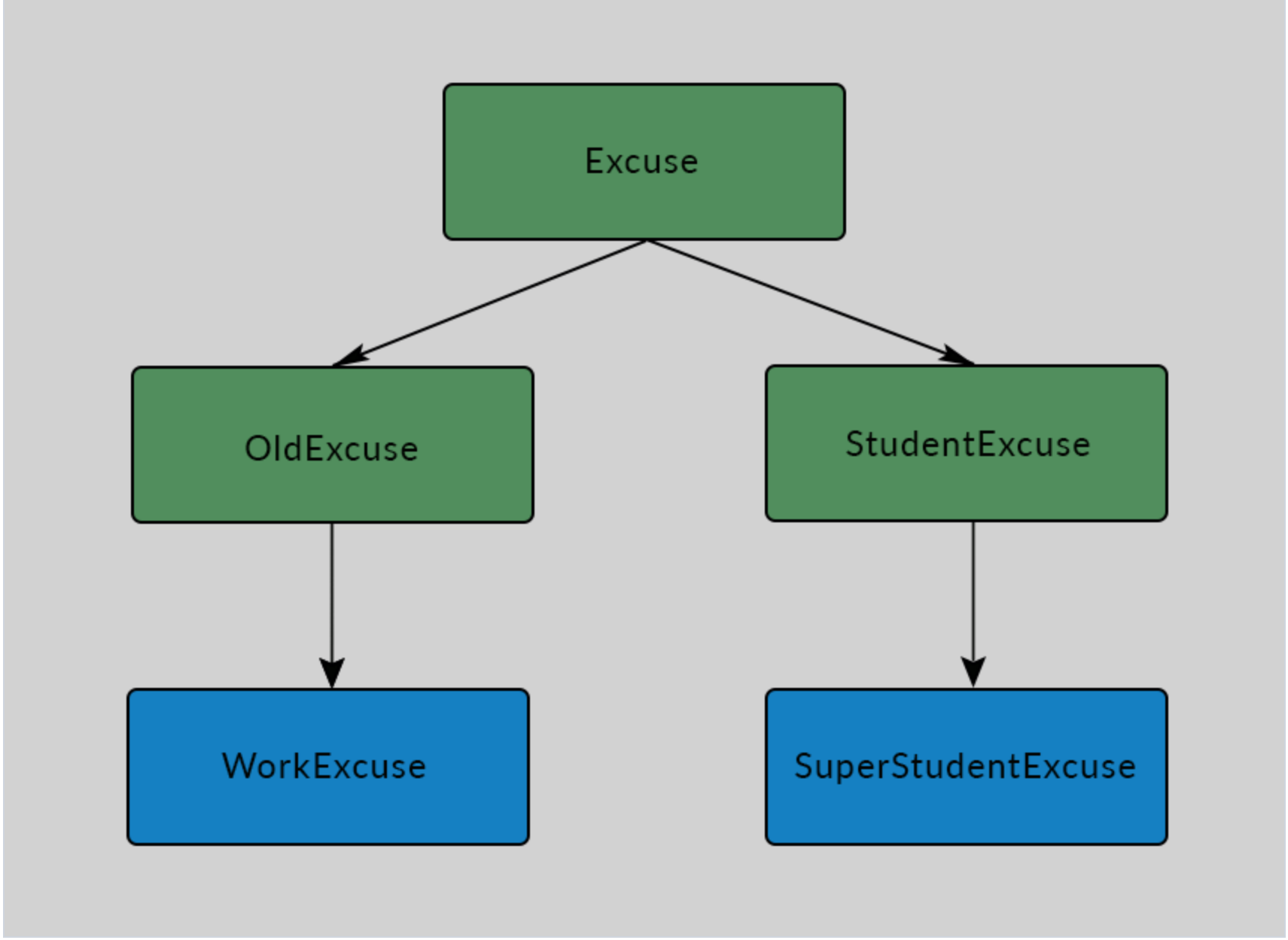
```
public class SuperStudentExcuse implements StudentExcuse {
    @Override
    public String generateExcuse() {
        // Логика нового функционала
        return "Невероятная отговорка, адаптированная под текущее состояние погоды, пробки или сбои в
    }

    @Override
    public void dislikeExcuse(String excuse) {
        // Добавляет причину в черный список
    }
}
```

Изменить код нельзя. Текущая схема будет выглядеть так:



Эта версия системы работает только с интерфейсом Excuse. Переписывать код нельзя: в большом приложении такие правки могут затянуться или нарушить логику приложения.

Можно предложить внедрение основного интерфейса и увеличение иерархии:



Для этого нужно переименовать интерфейс `Excuse`. Но дополнительная иерархия нежелательна в серьезных приложениях: внедрение общего корневого элемента нарушает архитектуру.

Следует реализовать промежуточный класс, который позволит использовать новый и старый функционал с минимальными потерями. Словом, **тебе нужен адаптер**.

Принцип работы паттерна Адаптер

Адаптер — это промежуточный объект, который делает вызовы методов одного объекта понятными другому.

Реализуем адаптер для нашего примера и назовем его `Middleware`.

Наш адаптер должен реализовывать интерфейс, совместимый с одним из объектов. Пусть это будет `Excuse`. Благодаря этому `Middleware` может вызывать методы первого объекта.

`Middleware` получает вызовы и передает их второму объекту в совместимом формате. Так выглядит реализация метода `Middleware` с методами `generateExcuse` и `dislikeExcuse`:

```
public class Middleware implements Excuse { // 1. Middleware становится совместимым с объектом WorkEx

    private StudentExcuse superStudentExcuse;

    public Middleware(StudentExcuse excuse) { // 2. Получаем ссылку на адаптируемый объект
        this.superStudentExcuse = excuse;
    }

    @Override
    public String generateExcuse() {
        return superStudentExcuse.generateExcuse(); // 3. Адаптер реализовывает метод интерфейса
    }

```

```
        // Метод предварительно помещает отговорку в черный список БД,
        // Затем передает ее в метод dislikeExcuse объекта superStudentExcuse.
    }
   // Метод likeExcuse появятся позже
}
```

Тестирование (в клиентском коде):

```
public class Test {
   public static void main(String[] args) {
       Excuse excuse = new WorkExcuse(); // Создаются объекты классов,
       StudentExcuse newExcuse = new SuperStudentExcuse(); // Которые должны быть совмещены.
       System.out.println("Обычная причина для работника:");
       System.out.println(excuse.generateExcuse());
       System.out.println("\n");
       Excuse adaptedStudentExcuse = new Middleware(newExcuse); // Оборачиваем новый функционал в об
       System.out.println("Использование нового функционала с помощью адаптера:");
       System.out.println(adaptedStudentExcuse.generateExcuse()); // Адаптер вызывает адаптированный
   }
}
```

Вывод:

```
Обычная причина для работника:

Я сегодня опоздал, потому что предпраздничное настроение замедляет метаболические процессы в моем организме

Нет оправдания моему поступку. Я недостоин этой должности.
Использование нового функционала с помощью адаптера
```

Невероятная отговорка, адаптированная под текущее состояние погоды, пробки или сбои в расписании общественного транспорта.

В методе `generateExcuse` выполнена простая передача вызова другому объекту, без дополнительных преобразований. Метод `dislikeExcuse` потребовал предварительного помещения отговорки в черный список базы данных. Дополнительная промежуточная обработка данных — причина, по которой любят паттерн Адаптер.

А как быть с методом `likeExcuse`, который есть в интерфейсе `Excuse`, но нет в `StudentExcuse`? Эта операция не поддерживается в новом функционале.

Для такого случая придумали исключение `UnsupportedOperationException`: оно выбрасывается, если запрашиваемая операция не поддерживается. Используем это.

Так выглядит новая реализация класса `Middleware`:

```
public class Middleware implements Excuse {

   private StudentExcuse superStudentExcuse;

```

```
    }

    @Override
    public String generateExcuse() {
        return superStudentExcuse.generateExcuse();
    }

    @Override
    public void likeExcuse(String excuse) {
        throw new UnsupportedOperationException("Метод likeExcuse не поддерживается в новом функционал
    }

    @Override
    public void dislikeExcuse(String excuse) {
        // Метод обращается за дополнительной информацией к БД,
        // Затем передает ее в метод dislikeExcuse объекта superStudentExcuse.
    }
}
```

На первый взгляд это решение не кажется удачным, но имитирование функционала может привести к более сложной ситуации. Если клиент будет внимателен, а адаптер — хорошо документирован, такое решение приемлемо.

Когда использовать Адаптер

1. Если нужно использовать сторонний класс, но его интерфейс не совместим с основным приложением. На примере выше видно, как создается объект-прокладка, который оборачивает вызовы в понятный для целевого объекта формат.

2. Когда у нескольких существующих подклассов должен быть общий функционал. Вместо дополнительных подклассов (их создание приведет к дублированию кода) лучше использовать адаптер.

Преимущества и недостатки

Преимущество: Адаптер скрывает от клиента подробности обработки запросов от одного объекта к другому. Клиентский код не думает о форматировании данных или обработке вызовов целевого метода. Это слишком сложно, а программисты ленивые :)

Недостаток: Кодовая база проекта усложняется дополнительными классами, а при большом количестве несовместимых точек их количество может вырасти до неконтролируемых размеров.

Не путать с Фасадом и Декоратором

При поверхностном изучении Адаптер можно перепутать с паттернами Фасад и Декоратор.

Отличие Адаптера от Фасада заключается в том, что Фасад внедряет новый интерфейс и оборачивает целую подсистему.

Ну а Декоратор, в отличие от Адаптера, меняет сам объект, а не интерфейс.

Пошаговый алгоритм реализации

2. Определи клиентский интерфейс, от имени которого будет использоваться другой класс.

3. Реализуй класс адаптера на базе интерфейса, определенного на предыдущем шаге.

4. В классе адаптера сделай поле, в котором хранится ссылка на объект. Эта ссылка передается в конструкторе.

5. Реализуй в адаптере все методы клиентского интерфейса. Метод может:

    a. Передавать вызов без изменения;

    b. Изменять данные, увеличивать/уменьшать количество вызовов целевого метода, дополнительно расширять состав данных и тд.

    c. В крайнем случае, при несовместимости конкретного метода, выбросить исключение UnsupportedOperationException, которое строго нужно задокументировать.

6. Если приложение будет использовать адаптер только через клиентский интерфейс (как в примере выше), это позволит безболезненно расширять адаптеры в будущем.

Само собой, паттерн проектирования — это не панацея от всех бед, но с его помощью можно элегантно решить задачу несовместимости объектов с разными интерфейсами.

Разработчик, знающий базовые паттерны, — на несколько ступенек выше тех, кто просто умеет писать алгоритмы, ведь они нужны для создания серьезных приложений. Повторно использовать код становится не так сложно, а поддерживать — одно удовольствие.

На сегодня все! Но мы скоро продолжим знакомство с разными шаблонами проектирования :)

```
23      @Override
24      public void connect(){
25          sd.insert();
26          sd.copy();
27      }
28  }
```

Ответить

+7

LuneFox инженер по сопровождению в **BIFIT** EXPERT — 5 марта, 21:22

Ещё бы скобки выровнять, и вообще красота)

Ответить

0

Andrey Уровень 41, Москва — 18 октября 2020, 14:22

эт конечно не имеет отгошения к теме лекции, но немного смущает:

```
1  sorryOptions[(int) Math.round(Math.random() + 1)];
```

вернет либо 1, либо 2, т.е. sorryOptions[0] не вернет никогда.

Ответить

+3

Валентин Кудинов Уровень 41, Самара, Россия — 3 апреля 2020, 08:34

Текст вылез за границы оформления кода.
public class Middleware implements Excuse { // 1. Middleware становится совместимым с объектом WorkExcuse через интерфейс Excuse

Ответить

0

Эллеонора Керри Уровень 41 — 3 апреля 2020, 08:45

Спасибо, поправили)

Ответить

0

Anonymous #2491313 Java Developer в **Росатом** — 24 апреля 2021, 09:46

```
    Невероятная отговорка, адаптированная под текущее состояние погоды, пробки ил
```

Вылезло за пределы блока кода.

Ответить

+1

Русский