Professor Hans Noodles

41 уровень

02.03.2020 9095 6

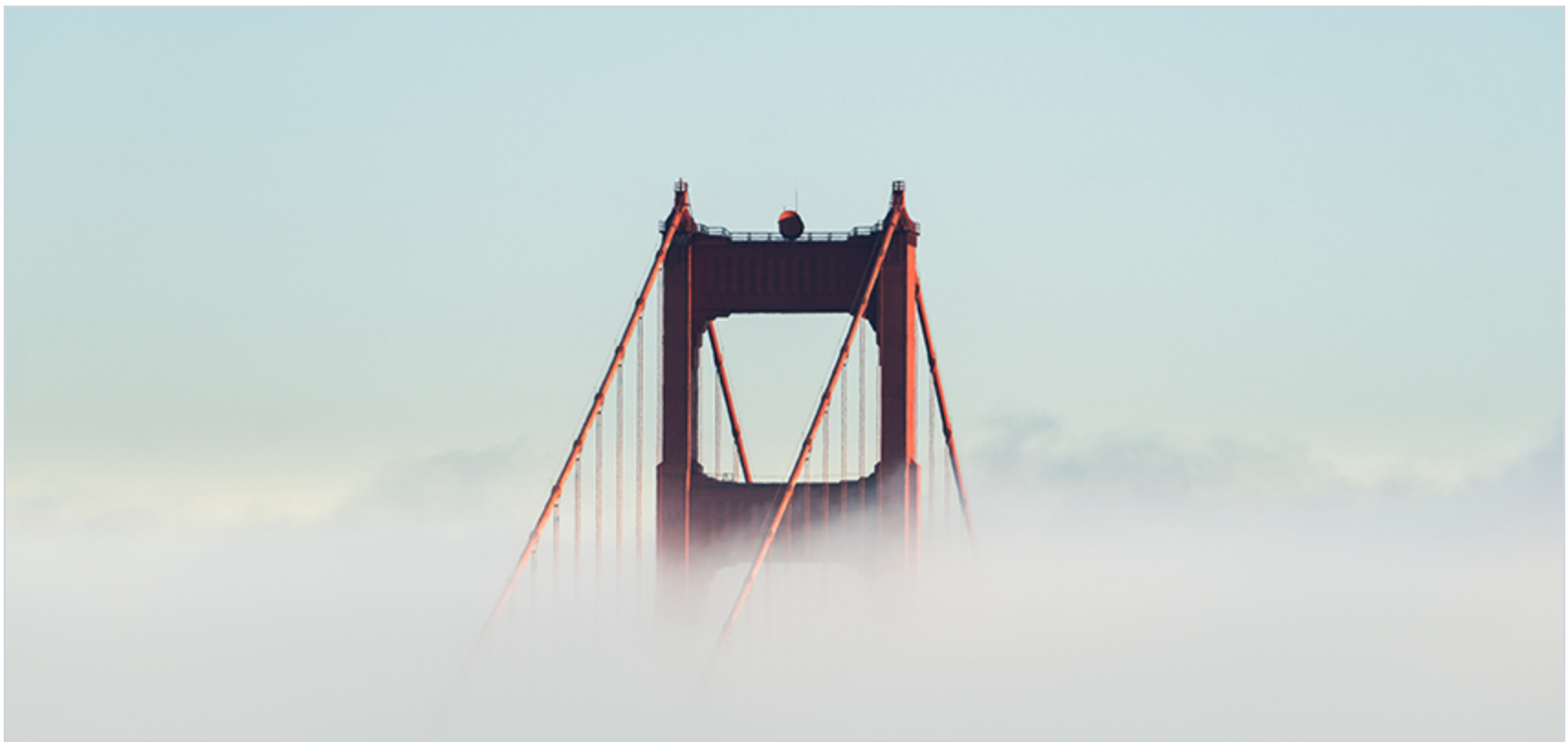
Знакомство с паттерном проектирования Bridge

Статья из группы Java Developer

43658 участников

Вы в группе

Привет! Продолжаем разбираться в обширной и очень полезной теме — паттерны проектирования.



Сегодня поговорим о Bridge. Как и другие паттерны, Bridge служит для решения типичных проблем, с которыми сталкивается разработчик при проектировании архитектуры программного обеспечения. Давай сегодня изучим его особенности и узнаем, как его стоит использовать.

Что представляет собой паттерн Bridge?

Паттерн Bridge (Мост) — структурный шаблон проектирования. То есть, его основная задача — создание полноценной структуры из классов и объектов. Bridge решает эту задачу путем разделения одного или нескольких классов на отдельные иерархии — *абстракцию* и *реализацию*. Изменение функционала в одной иерархии не влечет за собой изменения в другой.

Вроде все понятно, но по факту это определение звучит очень широко и не дает ответ на главный вопрос: “Что представляет собой паттерн Bridge?”.

Думаю, с этим тебе будет проще разобраться на практике. Давай сразу смоделируем классический пример для паттерна Bridge.

У нас есть абстрактный класс `Shape`, который обобщенно описывает геометрическую фигуру:

НАЧАТЬ ОБУЧЕНИЕ

```java
public abstract class Shape {
    public abstract void draw();
}
```

Когда мы решим добавить фигуры треугольника и прямоугольника, мы унаследуемся от класса `Shape`:

- Rectangle.java:

```java
public class Rectangle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing rectangle");
    }
}
```

- Triangle.java:

```java
public class Triangle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing triangle");
    }
}
```

Выглядит все просто до того момента, пока мы не вводим понятие “цвета”. То есть, у каждой фигуры будет свой цвет, от которого будет зависеть функционал метода `draw()`. Чтобы иметь различные реализации метода `draw()`, нам необходимо создать класс для каждой фигуры, соответствующий цвету. Если три цвета, то шесть классов: `TriangleBlack`, `TriangleGreen`, `TriangleRed`, `RectangleBlack`, `RectangleGreen` и `RectangleRed`.

Шесть классов — не такая уж и большая проблема. Но! Если нам нужно будет добавить новую фигуру или цвет, количество классов будет расти в геометрической прогрессии.

Как выйти из сложившейся ситуации? Хранение цвета в поле и перебор вариантов через условные конструкции — не лучший выход. Хорошее решение — **вывести цвет в отдельный интерфейс**.

Сказано — сделано: давай создадим интерфейс `Color` и три его имплементации — `BlackColor`, `GreenColor` и `RedColor`:

- Color.java:

```java
public interface Color {
    void fillColor();
}
```

- BlackColor.java:

```java
public class BlackColor implements Color {
    @Override
    public void fillColor() {
        System.out.println("Filling in black color");
    }
}
```

- GreenColor.java

```java
public class GreenColor implements Color {
```

НАЧАТЬ ОБУЧЕНИЕ

```
        System.out.println("Filling in green color");
    }
}
```

- RedColor.java

```
public class RedColor implements Color {
    @Override
    public void fillColor() {
        System.out.println("Filling in red color");
    }
}
```

Теперь добавим поле типа `Color` в класс `Shape` — его значение будем получать в конструкторе.

- Shape.java:

```
public abstract class Shape {
    protected Color color;

    public Shape(Color color) {
        this.color = color;
    }

    public abstract void draw();
}
```

Переменную `color` мы будем использовать в реализациях `Shape`. А это значит, что фигуры теперь могут использовать функционал интерфейса `Color`.

- Rectangle.java

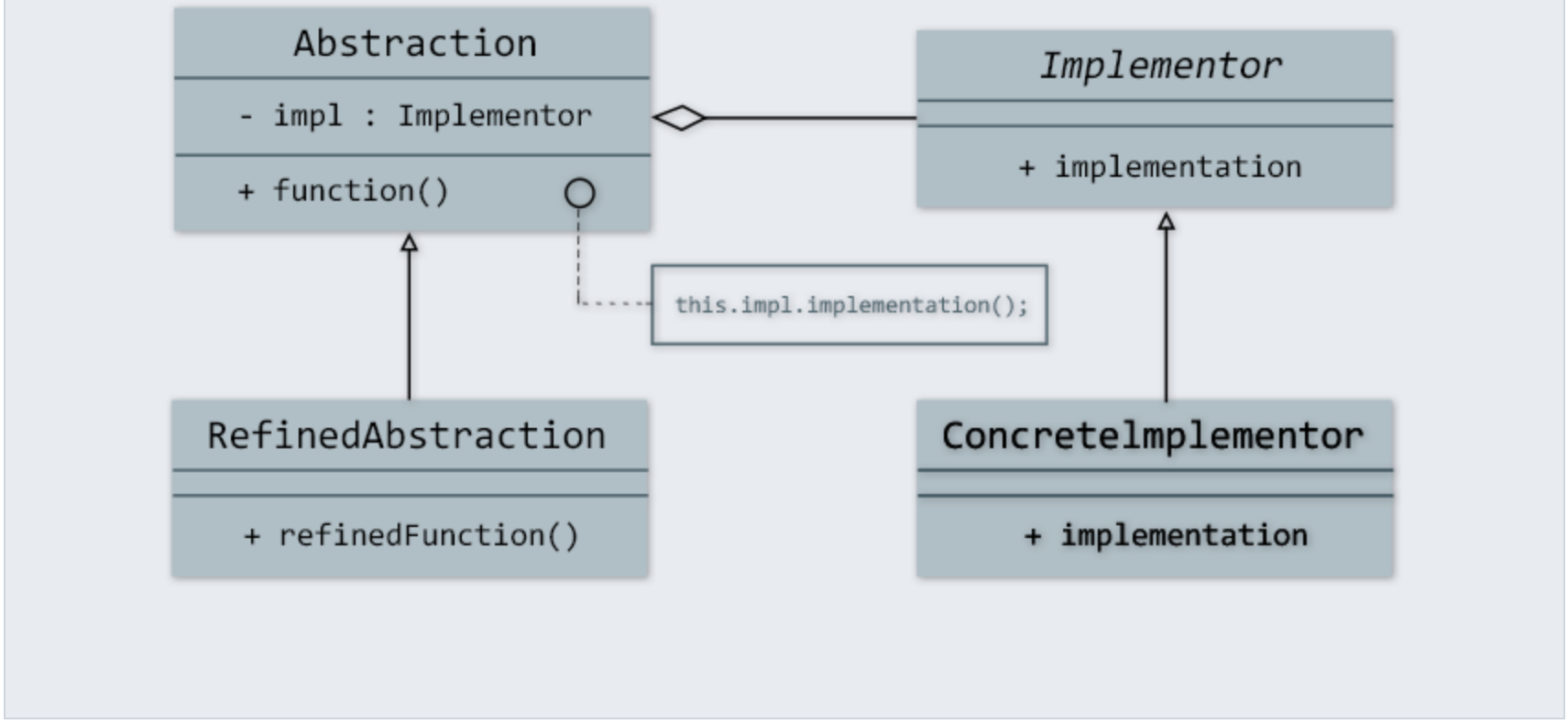
```
public class Rectangle extends Shape {

    public Rectangle(Color color) {
        super(color);
    }

    @Override
    public void draw() {
        System.out.println("Drawing rectangle");
        color.fillColor();
    }
}
```

Ну вот! Теперь мы можем плодить различные цвета и геометрические фигуры хоть до бесконечности, увеличивая количество классов в арифметической прогрессии. Поле `Color color` и является мостом (bridge), который взаимосвязывает две отдельные иерархии классов.

Устройство Bridge: что такое абстракция и реализация

Давай рассмотрим с тобой диаграмму классов, которая описывает паттерн Bridge:

НАЧАТЬ ОБУЧЕНИЕ



Здесь можно увидеть две независимые структуры, которые могут модифицироваться, не затрагивая функционал друг друга.

В нашем случае это:

- Abstraction — класс `Shape`;
- RefinedAbstraction — классы `Triangle`, `Rectangle`;
- Implementor — интерфейс `Color`;
- ConcreteImplementor — классы `BlackColor`, `GreenColor` и `RedColor`.

Класс `Shape` представляет собой Абстракцию — механизм управления раскраской фигур в различные цвета, который делегирует Реализацию интерфейсу `Color`.

Классы `Triangle`, `Rectangle` являются реальными объектами, которые используют механизм, предложенный классом `Shape`.

`BlackColor`, `GreenColor` и `RedColor` — конкретные имплементации в ветке Реализация. Их часто называют платформой.

Научитесь программировать с нуля с JavaRush:
1200 задач, автопроверка решения и стиля кода
НАЧАТЬ ОБУЧЕНИЕ

Где используют паттерн Bridge

Огромный плюс использования этого паттерна заключается в том, что можно вносить изменения в функционал классов одной ветки, не ломая при этом логику другой. Также такой подход помогает уменьшить связанность классов программы.

Главное условие применения паттернов — “следовать инструкции”: не совать их куда попало! Собственно, давай разберемся, в каких случаях точно нужно использовать Bridge:

1. Если необходимо расширить количество сущностей в две стороны (геометрические фигуры, цвета).
2. Если есть желание разделить большой класс, который не отвечает принципу Single responsibility, на более маленькие классы с узкопрофильным функционалом.
3. При возможной необходимости вносить изменения в логику работы неких сущностей во время работы программы.
4. При необходимости спрятать реализацию от клиентов класса (библиотеки).

НАЧАТЬ ОБУЧЕНИЕ

Плюсы и минусы паттерна

Как и другие паттерны, у Моста есть и преимущества, и недостатки.

Преимущества Bridge:

1. Улучшает масштабируемость кода — можно добавлять функционал, не боясь сломать что-то в другой части программы.
2. Уменьшает количество подклассов — работает при необходимости расширения количества сущностей в две стороны (например, количество фигур и количество цветов).
3. Дает возможность отдельно работать над двумя самостоятельными ветками Абстракции и Реализации — это могут делать два разных разработчика, не вникая в детали кода друг друга.
4. Уменьшение связанности классов — единственное место связки двух классов — это мост (поле `Color color`).

Недостатки Bridge:

1. В зависимости от конкретной ситуации и структуры проекта в целом, возможно негативное влияние на продуктивность программы (например, если нужно инициализировать большее количество объектов).
2. Усложняет читаемость кода из-за необходимости навигации между классами.

Отличие от паттерна Strategy

Паттерн Bridge часто путают с другим шаблоном проектирования — Strategy. Они оба используют композицию (в примере с фигурами и цветами мы использовали агрегацию, но паттерн Bridge может использовать и композицию), делегируя работу другим объектам. Но разница между ними есть, и она огромная. Паттерн Strategy является поведенческим паттерном: он решает совсем другие задачи. Strategy обеспечивает взаимозаменяемость алгоритмов, в то время как Bridge отделяет абстракцию от реализации, чтобы обеспечить возможность выбора между различными имплементациями. То есть, Bridge, в отличие от Strategy, применяется к целым конструкциям или иерархическим структурам.

Паттерн Bridge может стать хорошим оружием в арсенале разработчика, главное нащупать те ситуации где стоит применить его, или воспользоваться каким-то другим шаблоном. Если ты еще не знаком с другими шаблонами, почитай вот эти материалы:

- Singleton
- Factory
- Abstract Factory
- FactoryMethod
- Proxy
- Adapter

+33

Комментарии (6) популярные новые старые

JavaCoder

Введите текст комментария

Maks Panteleev Java Developer в Bell Integrator 26 июля 2021, 15:02

Как нахрен разобраться в десятках почти одинаковых паттернов?)

Ответить +10

Alexey Prilessky Уровень 40, Минск 2 ноября 2020, 19:02

Объясните: Создавать тысячу классов для фигур разных цветов это, значит, затратно, а создавать тысячу классов для цветов- это уже не затратно ?

НАЧАТЬ ОБУЧЕНИЕ

Если у тебя десять фигур и тысяча цветов, то будет десять тысяч вариантов разных фигур и цветов, красный круг, красный треугольник и это все будут разные классы. Поэтому цвет вынесли в отдельный интерфейс. Естественно ты можешь в дальнейшем не плодить цвета, а туда логику по определению цвета завезти, допустим RGB

Ответить +2

Pavlo Plynko Java-разработчик в **JavaRush** EXPERT 10 марта 2021, 16:59

Если у нас две фигуры, и одна тысяча цветов, то это либо 2000 классов фигур разных цветов, либо 2 класса фигур и 1000 классов цветов. Уже видна разница. А если фигур будет 20, то разница будет еще заметнее.

Ответить +1

Burakov Vladimir Уровень 41, Харьков, Украина 14 июня 2020, 11:28

В этой строке ошибка - "Отличие от паттерна Strategy Паттерн Bridge часто путают с другим шаблоном проектирования — Strategy. Они оба используют **композицию**, делегируя работу другим объектам."

Здесь показана **агрегация**, так как объект Color инициализируется в конструкторе с параметром, то есть этот Color можно вставить в несколько Shape, и при удалении Shape объект Color будет жить. Плюс на UML диаграмме показано правильно, как незакрашенный ромбик.

Ответить 0

Soros Уровень 39, Харьков, Украина 15 мая 2020, 16:43

"BlackColor, GreenColor и RedColor — конкретные имплементации в ветке **Абстракция**"
Или правильно - "в ветке Реализация"?
Ведь "Класс Shape представляет собой Абстракцию — механизм управления раскраской фигур в различные цвета, который делегирует **Реализацию интерфейсу Color**."
Классы BlackColor, GreenColor и RedColor имплементируют интерфейс Color. Возможно правильно будет сказать, что "BlackColor, GreenColor и RedColor — конкретные имплементации в ветке Реализация"?

Ответить 0

"Программистами не рождаются" © 2022 JavaRush