



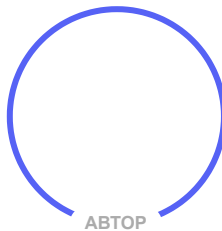
Реальная скидка -50%

ПОДРОБНЕЕ

[Статьи](#) [Авторы](#) [Все группы](#) [Все статьи](#) [Мои группы](#)

[JavaRush](#) > [Java блог](#) > [Java Developer](#) > Паттерн проектирования Мост (Bridge Pattern)

[Управление](#)



АВТОР

Артём Divertitto

Senior Android-
разработчик в **United
Tech**



02.03.2020



13303



6

Паттерн проектирования Мост (Bridge Pattern)

Статья из группы **Java Developer**

48662 участника

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:

- Shape.java

```
1 public abstract class Shape {
2     public abstract void draw();
3 }
```

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

- Rectangle.java:

```
1 public class Rectangle extends Shape {
2     @Override
3     public void draw() {
4         System.out.println("Drawing rectangle");
5     }
6 }
```

- Triangle.java:

```
1 public class Triangle extends Shape {  
2     @Override  
3     public void draw() {  
4         System.out.println("Drawing triangle");  
5     }  
6 }
```

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`:

```
1 public interface Color {  
2     void fillColor();  
3 }
```

- `BlackColor.java`:

```
1 public class BlackColor implements Color {  
2     @Override  
3     public void fillColor() {  
4         System.out.println("Filling in black color");  
5     }  
6 }
```

- GreenColor.java

```
1 public class GreenColor implements Color {
2     @Override
3     public void fillColor() {
4         System.out.println("Filling in green color");
5     }
6 }
```

- RedColor.java

```
1 public class RedColor implements Color {
2     @Override
3     public void fillColor() {
4         System.out.println("Filling in red color");
5     }
6 }
```

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

- Shape.java:

```
1 public abstract class Shape {
2     protected Color color;
3
4     public Shape(Color color) {
5         this.color = color;
6     }
7
8     public abstract void draw();
9 }
```

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

- Rectangle.java

```
1 public class Rectangle extends Shape {
2
3     public Rectangle(Color color) {
```

```

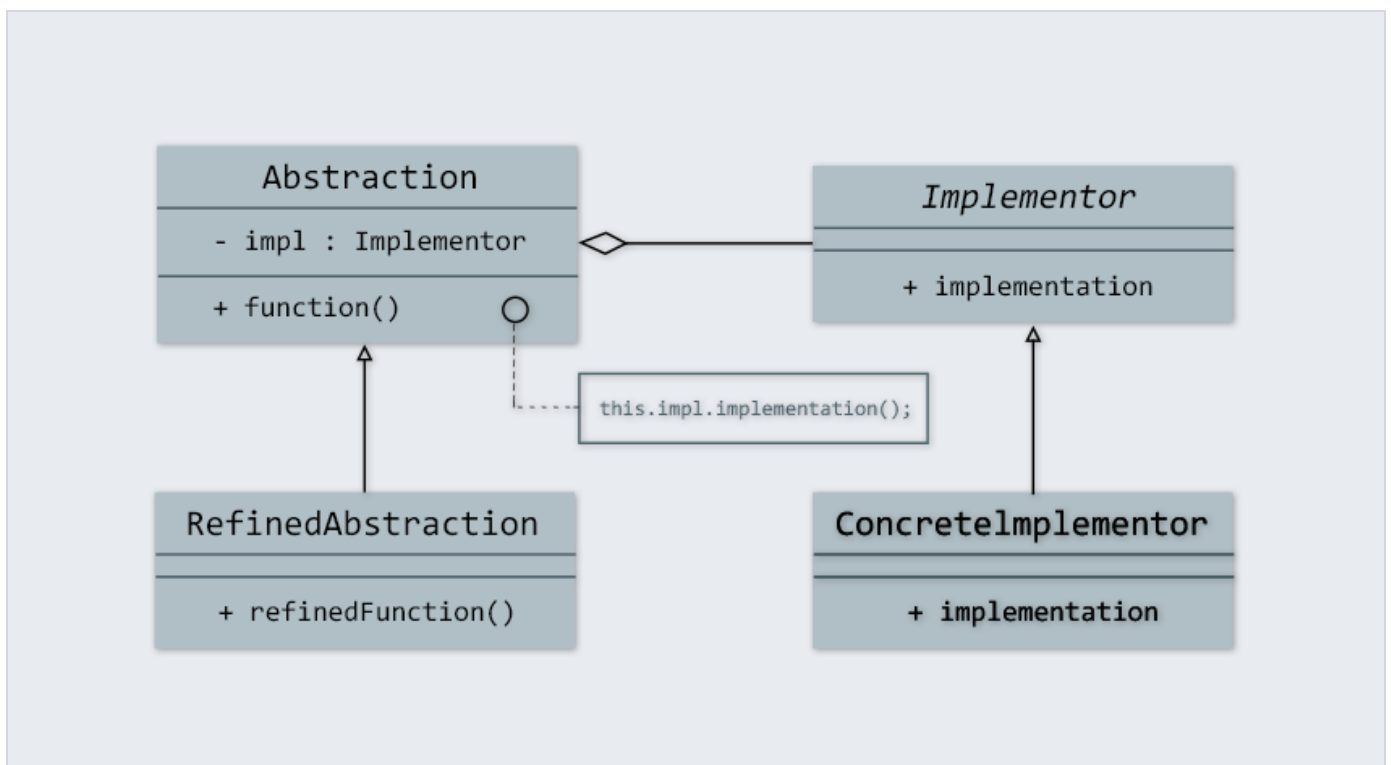
4         super(color);
5     }
6
7     @Override
8     public void draw() {
9         System.out.println("Drawing rectangle");
10        color.fillColor();
11    }
12 }

```

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 связывает две отдельные иерархии классов.

Устройство Bridge: что такое абстракция и реализация

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



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`;
- ConcretImplementor — 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).

При использовании паттерна каждый раз нужно помнить, что он добавляет дополнительные сущности в код — не совсем логично применять его в проекте, где всего одна геометрическая фигура и один-два возможных ее цвета.

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.

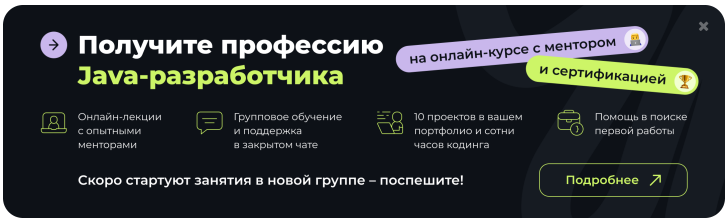
Отличие от паттерна Strategy

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](#)



Артем Divertitto

Senior Android-разработчик в United Tech

Артем — программист-свитчер, его первая профессия — реабилитолог. Раньше он помогал людям восстанавливать здоровье, даже получил м ...

[\[Читать полную биографию\]](#)



Комментарии (6)

популярные новые старые

sqr

Введите текст комментария

Maks Panteleev Java Developer в **Bell Integrator**

26 июля 2021, 15:02

Как нахрен разобраться в десятках почти одинаковых паттернов?)

Ответить

👍 +13 🗨

Alexey Prilessky Уровень 40

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 ?

Ответить

👍 0 🗨

Александр Уровень 40

16 декабря 2020, 20:52 ⋮

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 завести, допустим RGB

Ответить

👍 +2 🗨

Pavlo Plynko Java-разработчик в **CodeGym** 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

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

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 🗨

[Курсы программирования](#)

[Курс Java](#)

[Помощь по задачам](#)

[Подписки](#)

[Задачи-игры](#)

СООБЩЕСТВО

[Пользователи](#)

[Статьи](#)

[Форум](#)

[Чат](#)

[Истории успеха](#)

[Активности](#)

КОМПАНИЯ

[О нас](#)

[Контакты](#)

[Отзывы](#)

[FAQ](#)

[Поддержка](#)



RUSH

JavaRush — это интерактивный онлайн-курс по изучению Java-программирования с нуля. Он содержит 1200 практических задач с проверкой решения в один клик, необходимый минимум теории по основам Java и мотивирующие фишки, которые помогут пройти курс до конца: игры, опросы, интересные проекты и статьи об эффективном обучении и карьере Java-девелопера.

ПОДПИСЫВАЙТЕСЬ

ЯЗЫК ИНТЕРФЕЙСА

Русский



"Программистами не рождаются" © 2023 JavaRush