Статья

Управление

АВТОР

Pavlo Plynko

Java-разработчик в **CodeGym**

22.02.2023 61465 62 + 5

Паттерн проектирования Адаптер

Статья из группы Java Developer

48666 участников

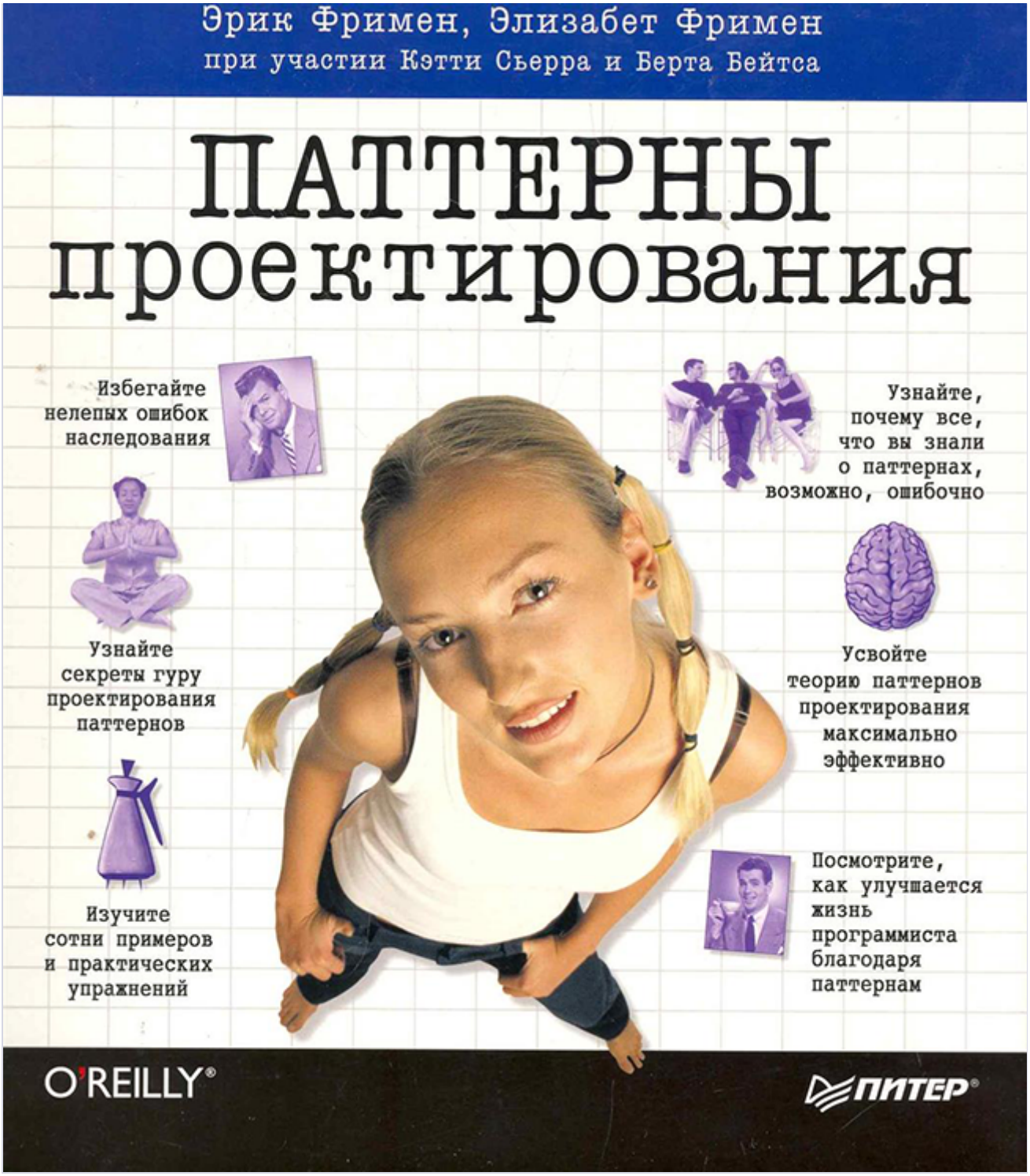
Вы в группе

Привет! Сегодня мы затронем важную новую тему — **паттерны, или по-другому — шаблоны проектирования**. Что же такое паттерны?

Думаю, тебе известно выражение «не надо изобретать велосипед». В программировании, как и во многих других сферах, есть большое количество типовых ситуаций. Для каждой из них в процессе развития программирования создавались готовые работающие решения. Это и есть шаблоны проектирования.

Условно говоря, паттерн — это некий пример, который предлагает решение ситуации вида: «если в вашей программе нужно сделать то-то, как это лучше всего сделать».

Паттернов очень много, им посвящена отличная книга «Изучаем шаблоны проектирования», с которой обязательно нужно ознакомиться.



Если говорить максимально кратко, паттерн состоит из распространенной проблемы и ее решения, которое уже можно считать неким стандартом.

В сегодняшней лекции мы познакомимся с одним из таких паттернов под названием «Адаптер».

Название у него говорящее, и ты не раз встречался с адаптерами в реальной жизни. Один из самых распространенных адаптеров — кардридеры, которыми снабжены множество компьютеров и ноутбуков.

Представь, что у нас есть какая-то карта памяти. В чем состоит проблема?

В том, что она не умеет взаимодействовать с компьютером. У них нет общего интерфейса.

У компьютера есть разъем USB, но карту памяти в него не вставить.

Карту невозможно вставить в компьютер, из-за чего мы не сможем сохранить наши фотографии, видео и другие данные.

Кардридер является адаптером, решающим данную проблему. Ведь у него есть USB-кабель! В отличие от самой карты, кардридер можно вставить в компьютер. У них с компьютером есть общий интерфейс — USB.

Давай посмотрим, как это будет выглядеть на примере:

```
1  public interface USB {
2
3      void connectWithUsbCable();
4  }
```

Это наш интерфейс USB с единственным методом — вставить USB-кабель:

```
public class MemoryCard {

    public void insert() {
        System.out.println("Карта памяти успешно вставлена!");
    }

    public void copyData() {
        System.out.println("Данные скопированы на компьютер!");
    }
}
```

Это наш класс, реализующий карту памяти. В нем уже есть 2 нужных нам метода, но вот беда: интерфейс USB он не реализует. Карту нельзя вставить в USB-разъем.

```
public class CardReader implements USB {

    private MemoryCard memoryCard;

    public CardReader(MemoryCard memoryCard) {
        this.memoryCard = memoryCard;
    }

    @Override
    public void connectWithUsbCable() {
        this.memoryCard.insert();
        this.memoryCard.copyData();
    }
}
```

А вот и наш адаптер!

Что же делает **класс** `CardReader` и почему, собственно, он является адаптером?

Все просто. Адаптируемый класс (карта памяти) становится одним из полей адаптера. Это логично, ведь в реальной жизни мы тоже вставляем карту внутрь кардридера, и она тоже становится его частью.

В отличие от карты памяти, у адаптера есть общий интерфейс с компьютером. У него есть USB-кабель, то есть он может соединяться с другими устройствами по USB.

Поэтому в программе наш класс `CardReader` реализует интерфейс USB. Но что же происходит внутри этого метода?

А там происходит ровно то, что нам нужно! Адаптер делегирует выполнение работы нашей карте памяти. Ведь сам-то адаптер ничего не делает, какого-то самостоятельного функционала у кардридера нет. Его задача — только связать компьютер и карту памяти, чтобы карта могла сделать свою работу и скопировать файлы!

Наш адаптер позволяет ей сделать это, предоставив свой интерфейс (метод `connectWithUsbCable()`) для «нужд» карты памяти.

Давай создадим какую-то программу-клиент, которая будет имитировать человека, желающего скопировать данные с карты памяти:

```
1  public class Main {
2
3     public static void main(String[] args) {
4
5         USB cardReader = new CardReader(new MemoryCard());
6         cardReader.connectWithUsbCable();
7
8     }
9  }
```

Что же у нас в результате получилось?

Вывод в консоль:

```
1  Карта памяти успешно вставлена!
2  Данные скопированы на компьютер!
```

Отлично, наша задача успешно выполнена!

Вот несколько дополнительных ссылок с информацией о паттерне Адаптер:

- Видео Adapter Pattern – Design Patterns;
- Паттерн проектирования «Адаптер» / «Adapter»;
- Шаблоны проектирования простым языком.

Абстрактные классы Reader и Writer

Теперь мы вернемся к нашему любимому занятию: выучим парочку новых классов для работы со вводом и выводом :) Сколько мы их уже выучили, интересно?

Сегодня речь пойдет о классах `Reader` и `Writer`.

Почему именно о них? Потому что это будет в тему нашему предыдущему разделу — адаптерам.

Давай рассмотрим их подробнее. Начнем с `Reader`'а.

`Reader` — это абстрактный класс, поэтому явно создавать его объекты у нас не получится.

Но на самом деле ты с ним уже знаком! Ведь хорошо знакомые тебе классы `BufferedReader` и `InputStreamReader` являются его наследниками :)

```
1  public class BufferedReader extends Reader {
2  …
3  }
4
5  public class InputStreamReader extends Reader {
6  …
7  }
```

Так вот, **класс `InputStreamReader` — это классический адаптер**.

Как ты, наверное, помнишь, мы можем передать в его конструктор объект `InputStream`. Чаще всего мы для этого используем переменную `System.in`:

```
1  public static void main(String[] args) {
2
3      InputStreamReader inputStreamReader = new InputStreamReader(System.
4  }
```

Что же делает `InputStreamReader`? **Как и всякий адаптер, он преобразует один интерфейс к другому.** В данном случае — интерфейс `InputStream`'а к интерфейсу `Reader`'а.

Изначально у нас был класс `InputStream`. Он неплохо работает, но с его помощью можно читать только отдельные байты.

Кроме того, у нас есть абстрактный класс `Reader`. У него есть отличный и очень нужный нам функционал — он умеет читать символы! Нам такая возможность, конечно, очень нужна.

Но здесь мы сталкиваемся с классической проблемой, которую обычно решают адаптеры — несовместимость интерфейсов. В чем же она проявляется?

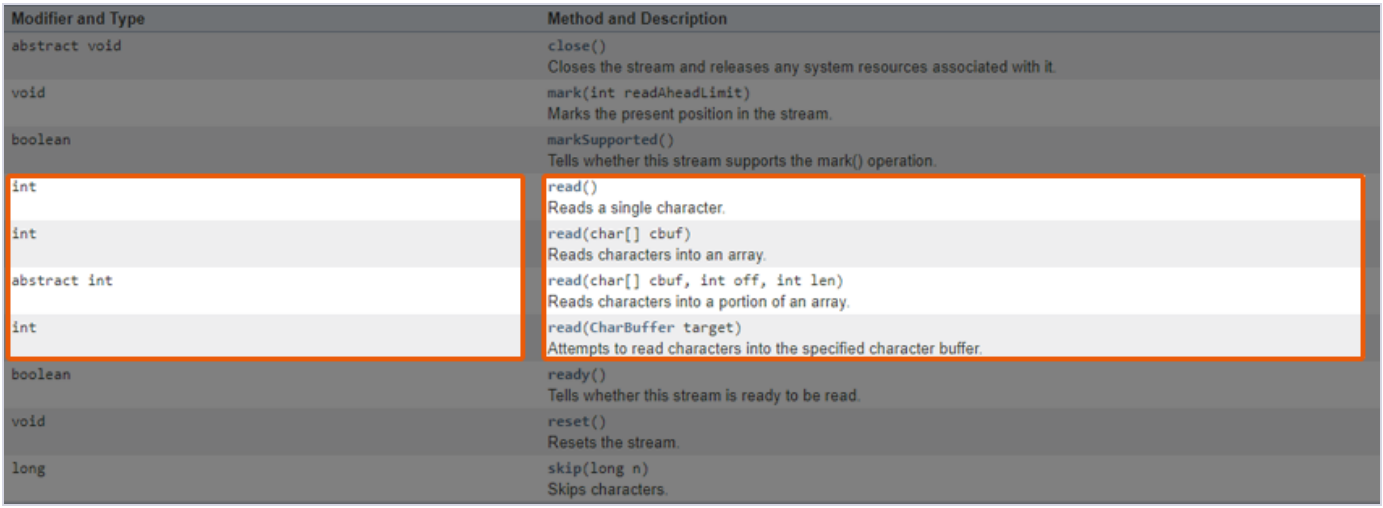
Давай заглянем прямо в документацию Oracle. Вот методы класса `InputStream`.

Modifier and Type	Method and Description
int	available() Returns an estimate of the number of bytes that can be read (or skipped over) from this input stream with
void	close() Closes this input stream and releases any system resources associated with the stream.
void	mark(int readlimit) Marks the current position in this input stream.
boolean	markSupported() Tests if this input stream supports the mark and reset methods.
abstract int	read() Reads the next byte of data from the input stream.
int	read(byte[] b) Reads some number of bytes from the input stream and stores them into the buffer array b.
int	read(byte[] b, int off, int len) Reads up to len bytes of data from the input stream into an array of bytes.
void	reset() Repositions this stream to the position at the time the mark method was last called on this input stream.
long	skip(long n) Skips over and discards n bytes of data from this input stream.

Совокупность методов — это и есть интерфейс.

Как видишь, метод **`read()`** у этого класса есть (даже в нескольких вариантах), но читать он может только байты: или отдельные байты, или несколько байт с использованием буфера. Нам такой вариант не подходит — мы хотим читать символы.

Нужный нам функционал **уже реализован в абстрактном классе** `Reader`. Это тоже можно увидеть в документации.



Modifier and Type	Method and Description
abstract void	close() Closes the stream and releases any system resources associated with it.
void	mark(int readAheadLimit) Marks the present position in the stream.
boolean	markSupported() Tells whether this stream supports the mark() operation.
int	read() Reads a single character.
int	read(char[] cbuf) Reads characters into an array.
abstract int	read(char[] cbuf, int off, int len) Reads characters into a portion of an array.
int	read(CharBuffer target) Attempts to read characters into the specified character buffer.
boolean	ready() Tells whether this stream is ready to be read.
void	reset() Resets the stream.
long	skip(long n) Skips characters.

Однако интерфейсы `InputStream`'а и `Reader`'а несовместимы! Как видишь, во всех реализациях метода `read()` у них отличаются и передаваемые параметры, и возвращаемые значения.

И именно здесь нам понадобится `InputStreamReader`! Он выступит **Адаптером** между нашими классами.

Как и в примере с кардридером, который мы рассмотрели выше, мы передаем объект «адаптируемого» класса «внутрь», то есть в конструктор класса-адаптера.

В прошлом примере мы передавали объект `MemoryCard` внутрь `CardReader`. А теперь передаем объект `InputStream` в конструктор `InputStreamReader`!

В качестве `InputStream` мы используем уже ставшую привычной переменную `System.in`:

```
1  public static void main(String[] args) {
2
3     InputStreamReader inputStreamReader = new InputStreamReader(System.
4  }
```

И действительно: заглянув в документацию `InputStreamReader`'а мы увидим, что «адаптация» прошла успешно :) Теперь в нашем распоряжении есть методы, которые позволяют нам читать символы.

И хотя изначально наш объект `System.in` (поток, привязанный к клавиатуре) не позволял этого делать, создав паттерн **Адаптер** создатели языка решили эту проблему.

У абстрактного класса `Reader`, как и у большинства I/O-классов, есть брат-близнец — `Writer`. Он имеет тот же большой плюс, что и `Reader` — предоставляет удобный интерфейс для работы с символами.

С выходными потоками проблема и ее решение выглядят так же, как и в случае со входными.

Есть класс `OutputStream`, который умеет записывать только байты; есть абстрактный класс `Writer`, который умеет работать с символами, и есть два несовместимых интерфейса.

Эту проблему вновь успешно решает паттерн Адаптер. При помощи класса `OutputStreamWriter` мы легко «адаптируем» два интерфейса классов `Writer` и `OutputStream` друг другу. И, получив байтовый поток `OutputStream` в конструктор, с помощью `OutputStreamWriter` мы, тем не менее, можем записывать символы, а не байты!

```
1  import java.io.*;
2
3  public class Main {
4
5     public static void main(String[] args) throws IOException {
6
7         OutputStreamWriter streamWriter = new OutputStreamWriter(new Fi
8         streamWriter.write(32144);
9         streamWriter.close();
10    }
11 }
```

Мы записали в наш файл символ с кодом 32144 — 綰, таким образом избавившись от необходимости работать с байтами :)

На этом на сегодня все, до встречи на следующих лекциях! :)

Pavlo Plynko

Java-разработчик в CodeGym

До того, как стать разработчиком, Павел 15 лет посвятил системному администрированию, но понимал, что не хочет заниматься этим всю ...
[Читать полную биографию]

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