

# Шаблоны проектирования. Структурные шаблоны



Филипп  
Воронов



**Филипп Воронов**

Teamlead, Поиск VK



# План занятия

1. [Вспоминаем прошлое занятие](#)
2. [Структурные шаблоны. Место в иерархии](#)
3. [Структурный шаблон Adapter](#)
4. [Структурный шаблон Proxy](#)
5. [Структурный шаблон Decorator](#)
6. [Структурный шаблон Facade](#)
7. [Структурный шаблон Flyweight](#)
8. [Итоги](#)
9. [Домашнее задание](#)



# **Вспоминаем прошлое занятие**



## Вопрос 1

### Что такое шаблон проектирования?

Выберите один правильный ответ:

1. Специальный синтаксис джавы, дающий новые возможности
2. Стандартный подход к решению распространённых проблем
3. Скопированный в вашу программу с сайта `stackoverflow` кусок кода

---

## Вопрос 1

### Что такое шаблон проектирования?

Выберите один правильный ответ:

1. Специальный синтаксис джавы, дающий новые возможности
2. **Стандартный подход к решению распространённых проблем**
3. Скопированный в вашу программу с сайта stackoverflow кусок кода



## Вопрос 2

**Какой шаблон лучше подойдёт для выноса сложной поэтапной логики создания объекта?**

Выберите один правильный ответ:

1. Абстрактная фабрика
2. Строитель
3. Прототип

---

## Вопрос 2

**Какой шаблон лучше подойдёт для выноса сложной поэтапной логики создания объекта?**

Выберите один правильный ответ:

1. Абстрактная фабрика
- 2. Строитель**
3. Прототип



---

## Вопрос 3

**Какие утверждения о Фабричном методе и Абстрактной фабрики верны?**

Выберите два правильных ответа:

1. Абстрактная фабрика и Фабричный метод - это одно и то же
2. Фабричный метод - это метод абстрактной фабрики
3. Абстрактная фабрика - это объект для создания объектов
4. Фабричный метод - это метод для создания объектов


---

## Вопрос 3

**Какие утверждения о Фабричном методе и Абстрактной фабрики верны?**

Выберите два правильных ответа:

1. Абстрактная фабрика и Фабричный метод - это одно и то же
2. Фабричный метод - это метод абстрактной фабрики
3. **Абстрактная фабрика - это объект для создания объектов**
4. **Фабричный метод - это метод для создания объектов**



# **Структурные шаблоны.**

## **Место в иерархии**

# Структурные шаблоны. Место в иерархии

**Порождающие шаблоны.** О том как правильнее подходить к созданию объектов под разные обстоятельства их дальнейшего использования.



Было

**Структурные шаблоны.** О том как правильнее продумывать и совмещать структуры разных объектов, проектировать иерархию классов и интерфейсов.



**ВЫ ЗДЕСЬ**

**Поведенческие шаблоны.** О том как правильнее подбирать возможности ваших объектов для удобного взаимодействия с ними.



Будет



# Структурный шаблон Adapter

# Структурный шаблон Adapter

Окружающие нас объекты, как и объекты в программировании, взаимодействуют между собой. Например, розетка и устройство, которое мы хотим зарядить.

Но что делать, когда объекты имеют несовместимые интерфейсы? Как зарядить телефон, его же не воткнуть в розетку напрямую?



Для решения этой проблемы мы используем **шаблон Adapter** (с англ. адаптер), совмещающая несовместимые интерфейсы взаимодействия с помощью специального переходника.

# Структурный шаблон Adapter

Как разрешить ситуацию, в которой:

- **Есть два класса с разными интерфейсами взаимодействия.**  
Менять эти исходные классы нельзя (например, они библиотечные).
- **Один из этих классов хочет использовать функциональность другого.** Но обращаться первый класс ко второму хочет не в том формате, который поддерживает последний.

# Структурный шаблон Adapter

ОТВЕТ: это можно сделать через **шаблон проектирования Adapter** (с англ. адаптер).

- Мы хотим чтобы объект класса **A** использовал объект класса **B** через интерфейс, отличный от интерфейса класса **B**
- Создаём вспомогательный класс **ABAdapter**, он и будет нашим переходником
- Он будет принимать запросы, которые хочет слать класс **A**, переформулировать их и отдавать классу **B** в нужной для последнего форме.

```
public class Main {  
    public interface IStorage {  
        void append(String s);  
    }  
  
    public static class Log {  
        protected final IStorage storage;  
        public Log(IStorage storage) { this.storage = storage; }  
  
        public void log(String line) {  
            storage.append(line);  
        }  
    }  
  
    public static class ListStorageAdapter implements IStorage {  
        protected List<String> list;  
        public ListStorageAdapter(List<String> list) { this.list = list; }  
  
        @Override  
        public void append(String s) {  
            list.add(s);  
        }  
    }  
  
    public static void main(String[] args) {  
        List<String> list = new ArrayList<>();  
        Log logger = new Log(new ListStorageAdapter(list));  
        logger.log("Hello!");  
    }  
}
```





# Структурный шаблон Proxy

# Структурный шаблон Proxy

Не все взаимодействия в нашей жизни мы совершаем напрямую - иногда мы пользуемся посредниками. Например, в случае обхода заблокированных ресурсов, мы можем пользоваться доступными прокси-серверами - отправляя запросы и получая ответы через сервера-посредники, а не общаясь напрямую с адресатом.



Мы воспользовались **шаблоном Proxy** (с англ. заместитель), используя посредника как если бы он был целевым адресатом, а посредник же просто передаёт соответствующий запрос адресату или ответ от адресата нам.

---

# Структурный шаблон Proxy

Как разрешить ситуацию, в которой:

- **Есть два взаимодействующих класса.** Их интерфейсы совместимы, с этим всё в порядке.
- **Мы хотим контролировать их взаимодействие.** Например, совершать какие-то другие действия при каждом взаимодействии этих двух классов.

# Структурный шаблон Proxy

ОТВЕТ: это можно сделать через **шаблон проектирования Proxy** (с англ. заместитель).

- Мы хотим чтобы объект класса **A** использовал объект класса **B** и мы могли выполнять какую-то свою логику при каждом таком взаимодействии
- Создаём вспомогательный класс **BProxy** с таким же как у **B** интерфейсом, все взаимодействия **A** будет производить с **BProxy**, а **BProxy** когда надо обращаться к **B**

```
public class MainProxyPres {  
    public interface IStorage {  
        void append(String line);  
    }  
  
    public static class Log {  
        // то же, что и в предыдущем примере  
    }  
  
    public static class StringBuilderStorage implements IStorage {  
        protected StringBuilder s = new StringBuilder();  
  
        @Override  
        public void append(String line) { s.append(line + "\n"); }  
    }  
  
    public static class ConsoleStorageProxy implements IStorage {  
        protected IStorage storage;  
  
        public ConsoleStorageProxy(IStorage storage) { this.storage = storage; }  
  
        @Override  
        public void append(String line) {  
            System.out.println("Эта строка будет залогирована: " + line  
+ "");  
            storage.append(line);  
        }  
    }  
}
```



# Структурный шаблон Decorator

---

# Структурный шаблон Decorator

Мы делим вещи вокруг нас на разные типы (как объекты в программировании на классы). Однако существует свойства, из-за существования или отсутствия которого мы не создаём у себя в мировоззрении нового типа объектов: к примеру, “токсичность” предмета. Мы не придумываем для каждого типа окружающего объекта, например, “ручки”, новый тип “токсичная ручка” - мы выносим токсичность в отдельную характеристику, которой при случае наделяем конкретные предметы.



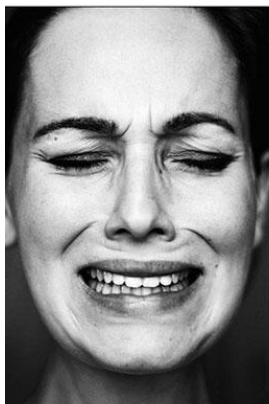
Для решения этой проблемы мы используем **шаблон Decorator** (с англ. декоратор), считая, что некоторые функции независимо друг от друга *дополняют* уже существующие типы, *не создавая новых*, но наделяя уже существующие объекты конкретными характеристиками.

---

# Структурный шаблон Decorator

Люди привыкли делить вещи вокруг себя на типы и свойства. Например, есть такие типы как экстраверт и интроверт. У них есть свойства - радостный и печальный. Но свойства радостный и печальный не принадлежат только типу экстраверт и интроверт: с тем же успехом их можно применить к типам мужчина и женщина (радостный мужчина, радостная женщина).

То есть, эти свойства независимы от типа. Такие независимые свойства позволяют нам не создавать кучу ненужных отдельных типов: печальный интроверт, печальный мужчина, печальная женщина. Мы просто выносим свойство “печальный” как отдельную характеристику, которой можно дополнить любой другой тип. Таким образом, мы можем наделить тип “мужчина” различными независимыми свойствами - печальный, строгий, высокий, ловкий.



По такому же принципу мы работаем с **шаблоном Decorator** (с англ. декоратор), считая, что некоторые функции независимо друг от друга *дополняют* уже существующие типы, при этом *не создавая новых типов*, а наделяя уже существующие объекты конкретными характеристиками.



# Структурный шаблон Decorator

Как разрешить ситуацию, в которой:

- Есть один или несколько классов, **имплементирующих общий интерфейс.**
- **Есть набор новых функций**, которым мы хотим научить наши классы.
- **Новых функций может быть очень много**, создавать классов-наследников на каждую комбинацию было бы утомительно.



# Структурный шаблон Decorator

ОТВЕТ: это можно сделать через **шаблон проектирования Decorator** (с англ. декоратор).

- Мы хотим чтобы объект класса **A** получил новую функциональность, но не хотим писать под неё новый класс-наследник
- Создаём вспомогательный класс **ADecorator**, он не будет наследовать **A**, но будет принимать его объекты и возвращать расширенный интерфейс - с новой функциональностью
- Чем отличается от наследования? Динамически расширяем любой класс этого интерфейса, а не заранее фиксированный.

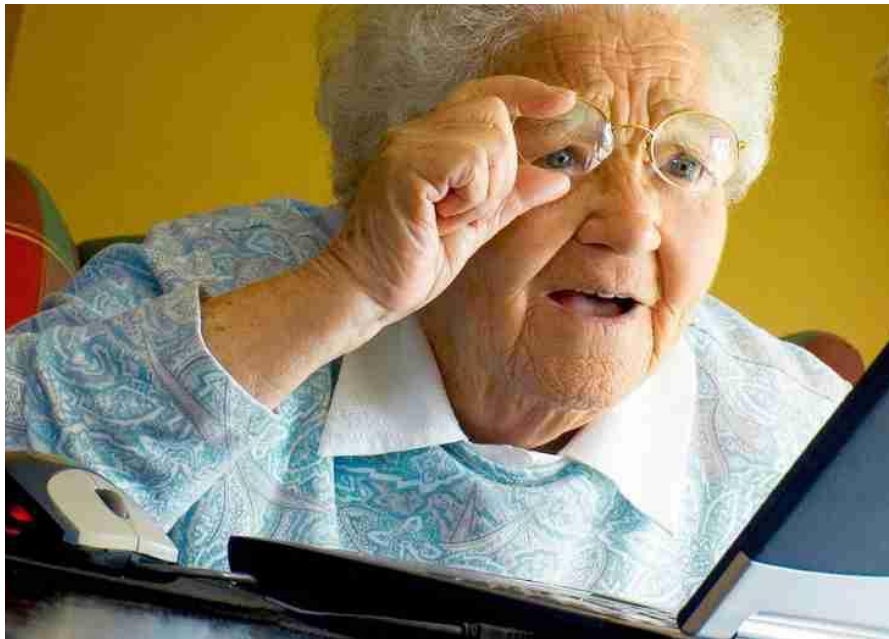
```
public class Main {  
    public interface IStorage {  
        void append(String line);  
    }  
  
    public static class StringBuilderStorage implements IStorage {  
        // то же что и в предыдущем примере  
    }  
  
    public static class EmotionalStorage implements IStorage {  
        protected final IStorage storage;  
        public EmotionalStorage(IStorage storage) { this.storage = storage; }  
  
        @Override  
        public void append(String line) {  
            storage.append(line);  
        }  
  
        public void shout(String line) {  
            append(line.toUpperCase() + "!!!");  
        }  
    }  
  
    public static void main(String[] args) {  
        IStorage storage = ...  
        EmotionalStorage withEmotions = new EmotionalStorage(storage);  
    }  
}
```



# Структурный шаблон Facade

# Структурный шаблон Facade

Некоторым пользователям нужна только базовая функциональность компьютера. Но они пользуются теми же компьютерами, на которых можно писать java-программы, shell-скрипты, запускать виртуальные машины и создавать свои службы и демонов. Но нужно ли таким пользователем всё это знать?



Для решения этой проблемы мы используем **шаблон Facade** (с англ. фасад), представляя для пользователя сложный многофункциональный компьютер гораздо более упрощённым фасадом, но удовлетворяющим все запрошенные нужды.

---

# Структурный шаблон Facade

Как разрешить ситуацию, в которой:


- **Есть многофункциональная гибкая библиотека.** Менять её исходные классы нельзя.
- **Библиотека нетривиальна в использовании,** что является необходимым следствием её многофункциональности
- **Нам нужна только урезанная часть её функциональности.**

# Структурный шаблон Facade

ОТВЕТ: это можно сделать через **шаблон проектирования Facade** (с англ. фасад).

- У нас есть класс **A** (или целая библиотека) со сложным интерфейсом но богатой функциональностью
- Создаём вспомогательный класс **AFacade**, у него будет меньше функций, но проще интерфейс
- Свою урезанную функциональность **AFacade** будет внутри себя реализовывать через взаимодействие с исходным классом

```
public class Main {  
    public static class Logger {  
        public enum Type { FILE, CONSOLE, EMAIL, TELEGRAM }  
        public enum LogLevel { DEBUG, INFO, WARN, ERROR }  
  
        protected final String user, encoding;  
        protected final Type reciever;  
  
        public Logger(String user, Type reciever, String encoding) {  
            this.user = user;  
            this.reciever = reciever;  
            this.encoding = encoding;  
        }  
  
        public long log(String msg, byte[] binData, LogLevel logLevel) {...}  
    }  
  
    public static class SimpleLogger {  
        protected static final byte[] NONE = new byte[0];  
        protected final Logger logger;  
  
        public SimpleLogger() {  
            logger = new Logger("tmp", Logger.Type.CONSOLE, "UTF-8");  
        }  
  
        public void log(String msg) {  
            logger.log(msg, NONE, Logger.LogLevel.INFO);  
        }  
    }  
}
```

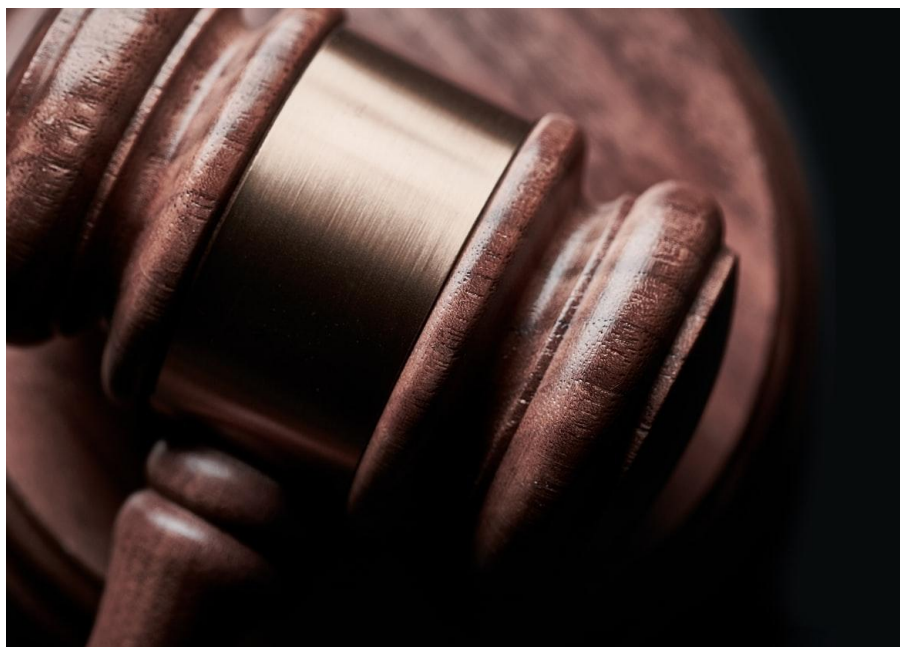


# Структурный шаблон Flyweight

---

# Структурный шаблон Flyweight

Представьте, что вы адвокат. К вам приходят с консультациями по УК и КоАП. Каждый клиент купил себе экземпляр кодекса. Правильно ли будет запоминать содержимое каждого экземпляра каждого клиента *по отдельности* или же лучше *выучить только два своих экземпляра*, ведь хоть у каждого клиента свои книжки, они всего лишь копии ваших?



Для решения этой проблемы мы используем **шаблон Flyweight** (с англ. легковес), держа в памяти часто повторяющиеся объёмные данные всего один раз, а при обращении клиента к номеру статьи из *своего* экземпляра кодекса смотрим только в *свой* экземпляр.



# Структурный шаблон Flyweight

Как разрешить ситуацию, в которой:

- **Есть много объектов одного класса.** Много объектов = много занимаемой памяти.
- **Часть данных этих объектов повторяется.** Эти данные нужны внутри наших объектов, но они не изменяются и часто повторяются от объекта к объекту.



# Структурный шаблон Flyweight

ОТВЕТ: это можно сделать через **шаблон проектирования Flyweight** (с англ. легковес).

- Мы хотим чтобы объект класса **A** имел внутри себя данные, но из-за частого их повторения они не хранились бы повторно
- Передаём контроль над созданием объектов *данных для A* другому классу, с одним и тем же объектом на повторяющиеся экземпляры
- Такое внешнее одноразовое хранение для повторяющихся данных внутри объектов будет *шаблоном Flyweight*

```
public class MainFlyweight {
    public static class Pic {
        protected byte[] picture;
        protected Pic(byte[] picture) { this.picture = picture; }
        void draw() { ... }
    }

    public static class MyChar {
        protected char c;
        protected Pic pic;

        public MyChar(char c, Pic pic) {
            this.c = c;
            this.pic = pic;
        }
    }

    public static class Text {
        protected static Map<Character, Pic> pics = new HashMap<>();

        protected List<MyChar> chars = new ArrayList<>();

        public void append(char c) {
            if (!pics.containsKey(c)) {
                Pic pic = ...
                pics.put(c, pic);
            }
            chars.add(new MyChar(c, pics.get(c)));
        }
    }
}
```



# Итоги

# Итоги

- Какую роль среди шаблонов проектирования играют структурные шаблоны
- Познакомились с пятью структурными шаблонами:
  - **Adapter** (адаптер) — “переходник” для взаимодействия двух классов с *разными* интерфейсами взаимодействия
  - **Proxy** (прокси) — для взаимодействия с адресатом через посредника
  - **Decorator** (декоратор) — добавление новой функциональности к объекту *определённого интерфейса* без наследования
  - **Facade** (фасад) — урезаем, но упрощаем взаимодействие с адресатом
  - **Flyweight** (легковес) — храним часто повторяющиеся внутренние данные отдельно для компактности



## Домашнее задание

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задавайте **в чате** мессенджера .
- Задачи можно сдавать **по частям**.
- Зачёт по домашней работе проставляется после того, как **приняты все задачи**.

⌘ нетология

**Задавайте вопросы и  
пишите отзыв о лекции!**

**Филипп Воронов**