

# Шаблоны проектирования. Поведенческие шаблоны



Филипп  
Воронов



**Филипп Воронов**

Teamlead, Поиск VK



# План занятия

1. [Вспоминаем прошлое занятие](#)
2. [Поведенческие. Место в иерархии](#)
3. [Поведенческий шаблон Command](#)
4. [Поведенческий шаблон CoR: Chain of responsibility](#)
5. [Поведенческий шаблон Observer](#)
6. [Поведенческий шаблон Iterator](#)
7. [Итоги](#)
8. [Домашнее задание](#)



# **Вспоминаем прошлое занятие**



## Вопрос 1

### Для чего нужен шаблон проектирования Facade?

Выберите один правильный ответ:

1. Чтобы обеспечить упрощённый интерфейс к сложной системе
2. Чтобы перехватить и контролировать любое взаимодействие с системой
3. Чтобы создать единственный экземпляр класса, общий для всех клиентов

---

## Вопрос 1

**Для чего нужен шаблон проектирования Facade?**

Выберите один правильный ответ:

- 1. Чтобы обеспечить упрощённый интерфейс к сложной системе**
2. Чтобы перехватить и контролировать любое взаимодействие с системой
3. Чтобы создать единственный экземпляр класса, общий для всех клиентов

---

## Вопрос 2

**Какие из следующих шаблонов являются порождающими?**

Выберите два правильных ответа:

1. Абстрактная фабрика
2. Прокси
3. Декоратор
4. Одиночка

---

## Вопрос 2

Какие из следующих шаблонов являются порождающими?

Выберите два правильных ответа:

- 1. Абстрактная фабрика
- 2. Прокси
- 3. Декоратор
- 4. Одиночка





## Вопрос 3

**Какой из следующих шаблонов является структурным?**

Выберите один правильный ответ:

1. Прототип
2. Фабричный метод
3. Строитель
4. Адаптер


---

## Вопрос 3

**Какой из следующих шаблонов является структурным?**

Выберите один правильный ответ:

1. Прототип
2. Фабричный метод
3. Строитель
- 4. Адаптер**



# **Поведенческие. Место в иерархии**

---

# Структурные шаблоны. Место в иерархии

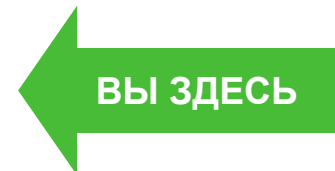
**Порождающие шаблоны.** О том как правильнее подходить к созданию объектов под разные обстоятельства их дальнейшего использования.



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



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





# Поведенческий шаблон Command

# Поведенческий шаблон Command

С помощью методов класса мы научили джаву выполнять различные задания. Но в своей реальной жизни мы задания не только *выполняем*, но любим также проводить над ними *другие действия*: планировать, отменять, откладывать, перепоручать.



Обращение с заданием как с “вещью” представляет собой **шаблон Command** (с англ. команда), совершая действия с заданием помимо непосредственного его выполнения.

---

# Поведенческий шаблон Command

Как разрешить ситуацию, в которой:

- **Есть действия, которым мы научили нашу программу. Действия?**  
Это мы знаем - методы классов!
- **Мы хотим не только моментально начинать их выполнять, но обращаться с ними как с вещами.** Например, положить в список и выполнить только последние 10.

# Поведенческий шаблон Command

ОТВЕТ: это можно сделать через **шаблон проектирования Command** (с англ. команда).

- Мы хотим действия не только выполнять, но и обращаться с ними как с объектами
- Просто создать метод с java-кодом действия теперь недостаточно
- Создадим для действия не метод, а *объект* этого действия, у которого будет метод при вызове которого это действие выполняется.
- Теперь мы можем обращаться с действиями так, как обращаемся с объектами. Например, создать список из них и выполнить в обратном порядке (пример на слайде)

```
public class Main {  
    public interface Command {  
        void execute();  
    }  
  
    public static void main(String[] args) {  
        LinkedList<Command> commands = new LinkedList<>();  
        commands.add(() -> System.out.println("Action 1"));  
        commands.add(() -> System.out.println("Action 2"));  
        commands.add(() -> System.out.println("Action 3"));  
  
        System.out.println("Executing in reverse!");  
        commands.descendingIterator()  
            .forEachRemaining(Command::execute);  
    }  
}
```



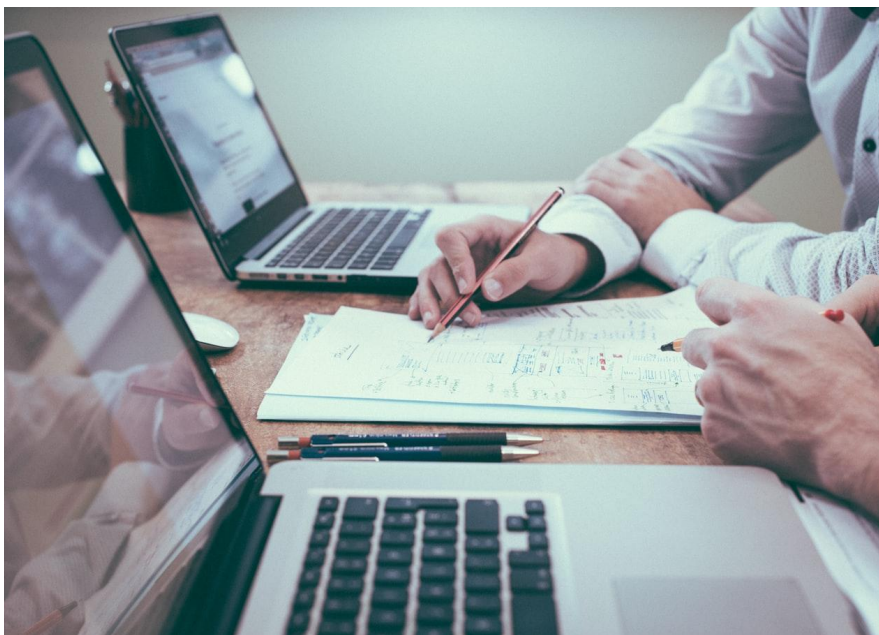


# **Поведенческий шаблон CoR: Chain of responsibility**

---

# Поведенческий шаблон CoR

Представим себе, что мы администрируем сайт вопросов и ответов в области права. Пользователь пишет вопрос, а мы ему - ответ. Однако областей права много и держать знания в одной голове будет проблематично. Например, даже определить, к какой области права относится вопрос, может быть нетривиальной задачей.



Что с этим сделаем? Давайте выстроим всех специалистов по праву в очередь и на каждый новый вопрос будем их по порядку спрашивать “твоя область?”. Первый, кто ответит “да”, и займётся вопросом. Так мы применили **шаблон Chain of responsibility** (с англ. цепочка ответственности).

---

# Поведенческий шаблон CoR

Как разрешить ситуацию, в которой:

- **Надо обработать новую информацию.** Написать метод-обработчик, верно?
- **Логика обработки огромна, но логически разделяема.** Для одного метода слишком много.

# Поведенческий шаблон CoR

ОТВЕТ: это можно сделать через **шаблон проектирования Chain of responsibility** (с англ. цепочка ответственности).

- Мы хотим сделать слишком большую обработку
- Разделим её на части, логику каждой части поместим в отдельный объект-обработчик
- Пройдёмся по нашим обработчикам и последовательно попытаемся их применить
- Могут быть вариации: с прерыванием после первого успеха (как в примере) или же с обязательным применением всех обработчиков

```
public class Main {  
    public interface Processor { boolean process(String msg); }  
  
    static final List<Processor> PROCESSORS = Arrays.asList(  
        (msg) -> {  
            if (msg.startsWith("Hello")) {  
                System.out.println("Hi, customer!");  
                return true;  
            }  
            return false;  
        },  
        (msg) -> {  
            if (msg.startsWith("Привет")) {  
                System.out.println("Здравствуй, клиент!");  
                return true;  
            }  
            return true;  
        },  
        (msg) -> {  
            System.out.println("Админ, мы этого языка не знаем!");  
            return true;  
        }  
    );  
  
    public static void main(String[] args) {  
        String msg = "Привет, дружок";  
        for (Processor processor: PROCESSORS)  
            if (processor.process(msg))  
                break;  
    }  
}
```



# Поведенческий шаблон Observer

# Поведенческий шаблон Observer

Представим себе ситуацию, когда один объект хочет реагировать на *некоторые* события, происходящие с *некоторыми* другими объектами. Например, прослушивать каждую песню, которую выкладывают у себя на страничке ваши любимые музыканты. Как мы решаем такую ситуацию? Регулярно проверяем эти страницы на обновления? Было бы неэффективно. *Каждый* музыкант шлёт уведомление *каждому* человеку? Было бы ужасно.



Для решения этой проблемы мы используем **шаблон Observer** (с англ. наблюдатель), просто нажимая на кнопку “Подписаться на обновления”. Тогда *каждый* исполнитель при добавлении песни будет слать уведомления *только подписчикам*.

---

# Поведенческий шаблон Observer

Как разрешить ситуацию, в которой:

- **Есть объекты, в которых могут происходить события.** Например, пользователь опубликовал пост.
- **Есть объекты, ожидающие некоторые события от некоторых объектов.** Например, пользователь, ждущий посты от *некоторых* других пользователей для собственной “обработки”.

# Поведенческий шаблон Observer

ОТВЕТ: это можно сделать через **шаблон проектирования Observer** (с англ. наблюдатель).

- Мы хотим эффективно доносить события (например, `String msg`) от объекта `a` до объектов `b` и `c` и только до них.
- Делаем возможность у объекта `b` через метод “подписаться” на события. Теперь можно указать код, который надо выполнить у `a` или `c` если событие произойдёт. `a` запоминает своих подписчиков.
- Когда событие происходит, `a` пробегается по своим подписчикам и вызывает указанный код обработчиков.

```
public class MainCommandPres {
    public static class Emitter {
        private List<Consumer<String>> subscribers = new ArrayList<>();

        public void subscribe(Consumer<String> s) {
            subscribers.add(s);
        }

        public void say(String msg) {
            System.out.println("I say " + msg);
            subscribers.forEach(s -> s.accept(msg));
        }
    }

    public static class R {
        public void refute(String msg) { System.out.println("No! " + msg +
            " is false!"); }
    }

    public static void main(String[] args) {
        R r = new R();

        Emitter emitter = new Emitter();
        emitter.subscribe(r::refute);
        emitter.subscribe(msg -> System.out.println(msg.toUpperCase() +
            "!!!!"));

        emitter.say("Earth is round");
    }
}
```





# Поведенческий шаблон Iterator

# Поведенческий шаблон Iterator

Представим себе магазин, который обслуживает покупателей *по порядку*. Говорит ли это о том, что покупатели должны выстроиться и стоять в одном ряду? Нет! Магазин ожидает от очереди лишь определённого её поведения, по одному выдавать следующего человека для обслуживания пока все не кончатся, без требований к структуре очереди (кто-то может отойти, занять место, купить место и тп).



В этой ситуации мы использовали **шаблон Iterator** (с англ. итератор), выдавая пользователю возможность перебрать всё содержимое по порядку, при этом внутренние структурные решения оставляя за нами.



# Поведенческий шаблон Iterator

Как разрешить ситуацию, в которой:

- **Есть объект, являющийся коллекцией элементов.**
- **Мы хотим перебрать все элементы этой коллекции, но не хотим указывать ей как эти элементы хранить**

# Поведенческий шаблон Iterator

ОТВЕТ: это можно сделать через **шаблон проектирования Iterator** (с англ. итератор).

- У нас есть класс `C`, который по смыслу является коллекцией элементов
- Если кто-то хочет перебрать элементы нашей коллекции, мы выдаём ему “бегунок” - объект, у которого можно просить “дай следующий элемент в коллекции”. Это и будет **итератором**.
- Получив итератор, пользователя больше не волнует то как мы внутри нашей коллекции храним элементы. Всё что ему надо - подстроить свой код под интерфейс взаимодействия с итератором.

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
}  
  
public interface Iterable<E> {  
    Iterator<E> iterator();  
}
```

Java уже имеет стандартный интерфейс `Iterator`, основная часть которого приведена выше. Он активно используется в коллекциях и многих других местах, а цикл `foreach` под капотом чаще всего сводится именно к нему.

Получив от коллекции объект-итератор, вы через `next()` получаете следующий элемент в переборе, а через `hasNext` проверяете остались ли ещё неперебранные элементы.

`Iterable` - интерфейс для коллекций, по которым можно пройти итератором.



# Итоги

---

# Итоги

- Какую роль среди шаблонов проектирования играют поведенческие шаблоны
- Познакомились с четырьмя структурными шаблонами:
  - **Command** (команда) — для работы с действиями как с объектами
  - **CoR** (цепочка ответственности) — для разделения сложной логики обработки на цепочку обработчиков
  - **Observer** (наблюдатель) — для эффективной доставки событий до заинтересованных адресатов
  - **Iterator** (итератор) — для перебора элементов коллекции без требований к её внутренней структуре их хранения



## 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**