

# Шаблоны проектирования. Порождающие шаблоны



Филипп  
Воронов




**Филипп Воронов**

Teamlead, Поиск Mail.ru



# План занятия

1. [Что такое шаблоны проектирования?](#)
2. [Порождающий шаблон Builder](#)
3. [Порождающий шаблон Singleton](#)
4. [Порождающий шаблон Factory Method](#)
5. [Порождающий шаблон Abstract Factory](#)
6. [Порождающий шаблон Prototype](#)
7. [Итоги](#)
8. [Домашнее задание](#)



# **Что такое шаблоны проектирования?**

# Что такое шаблоны проектирования?

Знанию *языка программирования* можно сопоставить знание *естественного языка* (например, русского):

Пример	Русский	Java
Слова	<i>мама, папа</i>	<code>break, Exception</code>
Предложения	<i>Скажи маме!</i>	<code>you.sayTo(mom)</code>
Грамматика	<i>красив<u>ое</u> солнце</i>	<code>foo((int) myVar)</code>

Достаточно ли знать *слова и грамматические правила*, чтобы написать хорошую книгу, популярную статью или влиятельную речь?

Достаточно ли знать *синтаксис языка программирования*, чтобы написать хорошую программу, эффективно работающую, лёгкую в поддержке и добавлении новых функций?



\_\_\_\_\_

```

83 <link href={url} >name </link>
84 })
85 </div>
86 <-preview code={code} eval={eval} onChange={onChange} />
87 </div>
88 {showCode ? (
89   <div>
90     <-editor code={code} onChange={onChange} />
91     <-button type="button" className={classes.hideCode} /> Hide code
92   </button>
93   </div>
94 ) : (
95   <-button type="button" className={classes.showCode} /> Show code
96   <-button>
97 )
98 </div>

```




Для написания качественного программного решения необходимо понимать принципы правильных подходов к решению поставленных проблем. Стандартные подходы к решению стандартных проблем называются **шаблонами проектирования**.

# Что такое шаблоны проектирования?

Шаблоны проектирования затрагивают множество различных тем: от вопросов создания новых объектов до особенностей их взаимодействия между собой.

Они являются не только набором готовых рецептов для часто встречающихся ситуаций, стандартность этих подходов позволяет программистам понимать архитектуру решений друг друга просто ссылкой на название нужного шаблона. Например: *“Для перебора элементов коллекции у неё есть метод, возвращающий итератор”*.

Качество подбора нужного шаблона для проектирования решения поставленной задачи отчасти является творческой задачей искусства программирования и *признаком опытности программиста*. Поэтому вопросы про них можно *часто встретить на собеседованиях*.

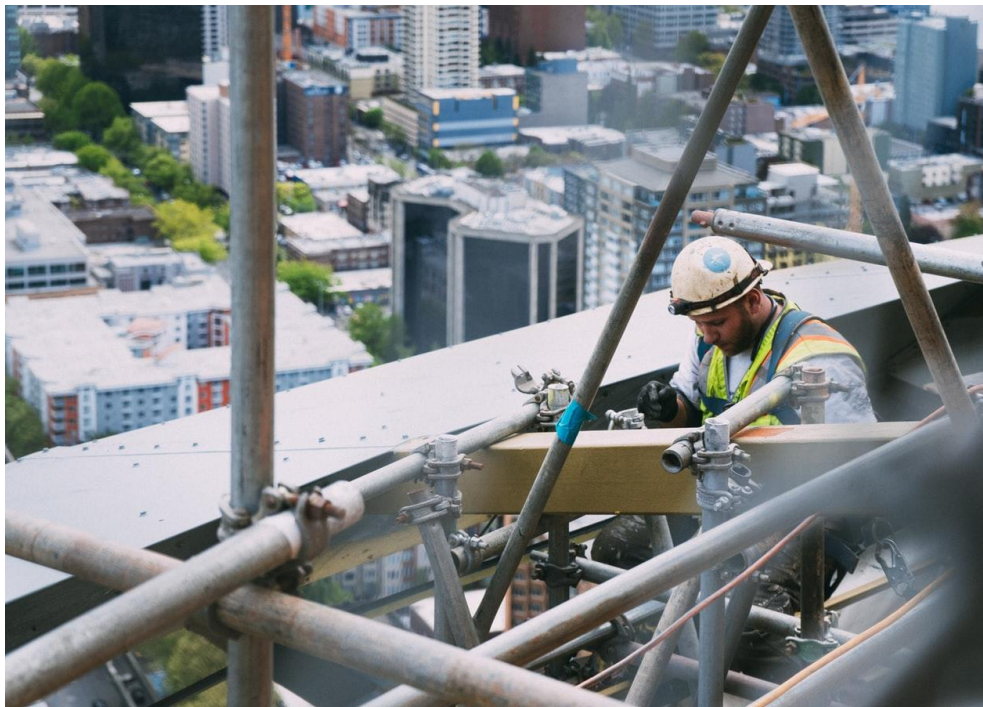


# Порождающий шаблон Builder



# Порождающий шаблон Builder

Нужно ли для эксплуатации дома знать всё то что нужно для его строительства? Или же все знания про все особенности строительства дома можно вынести за пределы нашего понятия “дом” и предоставить отдельной сущности - строителю.



Дом слишком сложный объект для создания чтобы заключать эту логику в знаниях о нём самом. Мы используем **шаблон Builder** (с англ. строитель), вынося весь процесс строительства в отдельную от использования готового дома тему.

---

# Порождающий шаблон Builder

Как спроектировать класс, обладающий следующими свойствами:

- **Имеет очень много параметров.** Гигантский конструктор?
- **Может инициализироваться поэтапно.** Часть параметров заполняется в одном месте программы, часть в другом.
- **Полуинициализированный объект невалиден.** Есть набор обязательных параметров, без которых объект невалиден.
- **Эффективное конструирование и эффективное хранение готового объекта взаимоисключены.** Например, создаём массив или строку.
- **Неизменяемость готовых объектов.** Параметры нельзя изменять.




# Порождающий шаблон Builder

ОТВЕТ: это можно сделать через **шаблон проектирования Builder** (с англ. строитель).

- Мы хотим сконструировать объект класса `A`
- Создаём вспомогательный класс `ABuilder`, он и будет нашим строителем
- Весь процесс конструирования мы взаимодействуем с объектом класса `ABuilder`, передавая ему все нужные параметры
- Когда хотим получить готовый объект класса `A`, просим дать его нам у объекта вспомогательного класса `ABuilder`

```
public class Main {  
    public static class A {  
        // Наш целевой класс  
    }  
  
    // Интерфейс для строителей класса A  
    public interface IBuilder {  
        A build();  
    }  
  
    // Конкретный класс строителя для A  
    public static class ABuilder implements IBuilder {  
        //...  
    }  
  
    public static void main(String[] args) {  
        ABuilder builder = new ABuilder();  
        //builder.setName(...)  
        //...  
        A a = builder.build();  
    }  
}
```



# Порождающий шаблон Singleton

---

# Порождающий шаблон Singleton

Самолётов, как и наших объектов в программе, может быть много. Но все они при полётах взаимодействуют с небом нашей планеты. Сколько объектов неба правильнее создать? Только одно - общее для всех.



**Самолёты** - *много штук.*

**Небо** - *одно, общее.*

Небо подходит под **шаблон Singleton** (с англ. одиночка), в отличие от самолётов.

# Порождающий шаблон Singleton

Как спроектировать класс, обладающий следующими свойствами:

- **Имеет всего ОДИН объект на всю программу.**  
Статичные поля и методы?
- **Мы хотим всю мощь ООП.** Полиморфизм и всё то, для чего статичность членов класса не подойдёт.

Такое часто нужно для представления какого-то ресурса, общего для всей программы (база данных, система оплаты), а также когда нужны глобальные объекты, (учёт глобальной квоты для каких-то действий).



# Порождающий шаблон Singleton

ОТВЕТ: это можно сделать через **шаблон проектирования Singleton** (с англ. одиночка).

- Мы хотим сконструировать объект класса `A`, общий для всей программы
- *Скрываем все конструкторы* нашего класса, чтобы программист не смог создать объект через `new`
- Вместо конструктора даём программисту *статичный метод* для получения объекта, в нём контролируем чтобы объект был всегда один и тот же
- Этот единственный объект храним в статичном поле нашего класса

```
public class Main {  
    public static class A {  
        // Храним ссылку на единственный объект класса  
        private static A instance = null;  
  
        private long start;  
  
        // Скрыли конструктор  
        private A() {  
            this.start = System.currentTimeMillis();  
        }  
  
        // Метод для получения объекта  
        public static A get() {  
            if (instance == null) instance = new A();  
            return instance;  
        }  
  
        public String toString() {  
            return "Стартовал в " + start;  
        }  
    }  
  
    public static void main(String[] args) {  
        A a = A.get();  
        System.out.println(a);  
    }  
}
```


# Порождающий шаблон Singleton

Лайфхак от джавы: *енам*!

- Джава сама следит, что объектов у енама столько, сколько перечислено у него самого
- Также автоматически получаем *потокобезопасность* (на случай многопоточного использования шаблона, что очень удобно)
- *Меньше гибкости*, тк енам делает нашу работу по реализации синглтона за нас

```
public class Main {  
    public enum A {  
        // Единственное значение енама =  
        // ссылка на его единственный объект  
        INSTANCE;  
  
        private long start;  
  
        // Конструктор енама скрыт  
        // по определению  
        A() {  
            start = System.currentTimeMillis();  
        }  
  
        public String toString() {  
            return "Стартовал в " + start;  
        }  
    }  
  
    public static void main(String[] args) {  
        A a = A.INSTANCE;  
        System.out.println(a);  
    }  
}
```

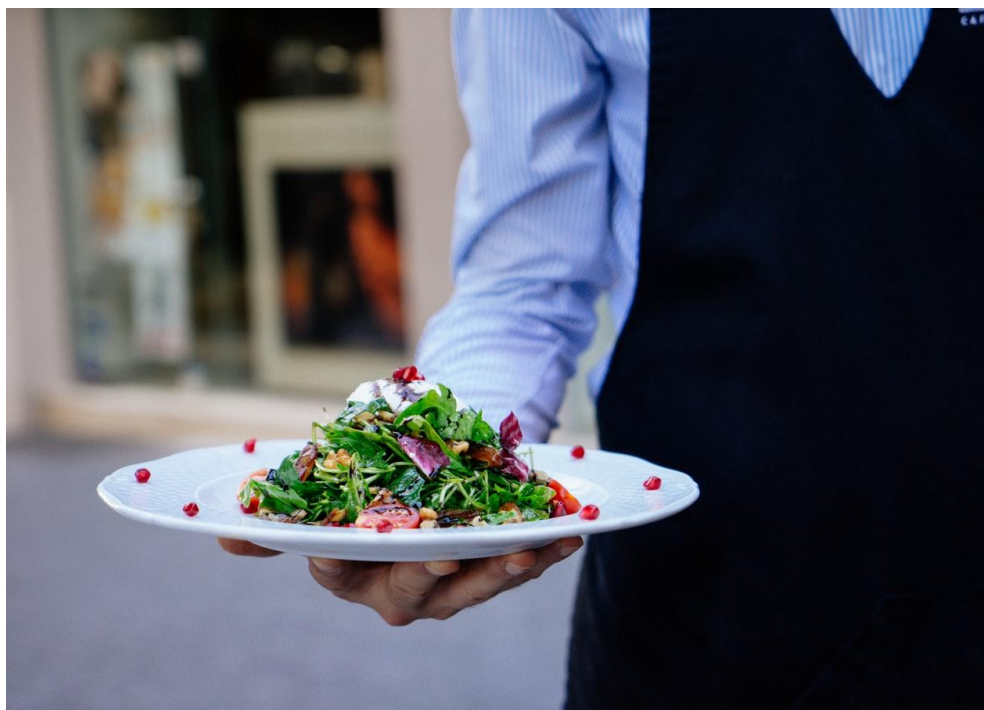




# **Порождающий шаблон Factory Method**

# Порождающий шаблон Factory method

Официант взаимодействует с готовыми блюдами, разнося их на нужные столики, однако ему не важно, как конкретно оно было приготовлено. Для получения блюда официант использует свой **фабричный метод** - подойти на кухню и попросить блюдо для нужного столика, весь процесс его создания будет на кухне, а не на официанте.



Официант использует “подойти на кухню, назвав номер столика” как **шаблон Factory method** (с англ. фабричный метод), получая блюдо не приготавливая его самим.

---

# Порождающий шаблон Factory method

Как спроектировать класс, объекты которого:

- **Использует другой объект определённого интерфейса.** Но не получает его в конструкторе. Создаёт сам?
- **Не зависит от конкретных реализаций этого объекта.** Т.е. также не создаёт его, но разрешает заняться его созданием своим потомкам




# Порождающий шаблон Factory method

ОТВЕТ: это можно сделать через **шаблон проектирования Factory method** (с англ. фабричный метод).

- Добавляем в наш класс метод для получения нужного нам объекта определённого интерфейса
- Даём потомкам свободу конструирования этого объекта, скрывая конкретную реализацию от нас

Ранее в шаблоне Singleton мы использовали фабричный метод получения экземпляра класса, чтобы контролировать количество созданных экземпляров

```
public class Main {  
    public static abstract class AbstractServer {  
        // Фабричный метод  
        protected abstract ILog getLog();  
  
        public void doWork() {  
            ILog logger = getLog();  
            logger.log("Hello! Давай поработаем!");  
        }  
    }  
  
    public interface ILog {  
        void log(String msg);  
    }  
  
    public static class TelegramLog implements ILog {  
        @Override  
        public void log(String msg) { /* ... */ }  
    }  
  
    public static class ModernServer extends AbstractServer {  
        @Override  
        protected ILog getLog() {  
            return new TelegramLog();  
        }  
    }  
}
```



# **Порождающий шаблон Abstract Factory**

# Порождающий шаблон Abstract Factory

Если нам нужны детали от одного и того же устройства, но не хотим задумываться о том, как они создаются, мы можем купить фабрику их производства и не задумываться, что одна деталь к другой не подойдёт - об этом будет думать фабрика.



Купив фабрику для телефонов определённой модели, мы использовали **шаблон Abstract Factory** (с англ. абстрактная фабрика), получая всё что нужно для нашего телефона не задумываясь о совместимости комплектующих между собой.

---

# Порождающий шаблон Abstract Factory

Как спроектировать класс в котором:

- **Используется набор объектов других интерфейсов.** Но не получает его в конструкторе. Создаёт сам?
- **Не зависит от конкретных реализаций этого объекта.** Т.е. также не создаёт его, но разрешает заняться его созданием кому-то другому
- **Объекты в наборе совместимы между собой** по своей внутренней логике, детали которой неизвестны для нашего класса



# Порождающий шаблон Abstract Factory

ОТВЕТ: это можно сделать через **шаблон проектирования Abstract Factory** (с англ. абстрактная фабрика).

- Создаём интерфейсы для целевых объектов, создаём интерфейс для фабрики их производящих
- Реализуем конкретную имплементацию фабрики, внутри которой содержится логика создания целевых объектов и их согласованности между собой

В отличие от шаблона фабричного метода, здесь у нас *отдельный объект* фабрики с *несколькими методами*, производящими разные объекты

```
public class Main {  
    public interface ErrorMessage {  
        void print(int errCode);  
    }  
  
    public interface ByeMessage {  
        void bye();  
    }  
  
    public interface MsgFactory {  
        ErrorMessage createErrorMessage(String user);  
        ByeMessage createByeMessage();  
    }  
  
    public static class RuMsgFactory implements MsgFactory { ... }  
  
    public static class EnMsgFactory implements MsgFactory { ... }  
  
    public static void main(String[] args) {  
        boolean isEng = false;  
        MsgFactory factory = isEng ? new EnMsgFactory() : new  
RuMsgFactory();  
        ErrorMessage err = factory.createErrorMessage("Лёшка");  
        ByeMessage bye = factory.createByeMessage();  
  
        err.print(403);  
        bye.bye();  
    }  
}
```





# Порождающий шаблон Prototype

---

# Порождающий шаблон Prototype

Банановые растения размножают клонированием, так как размножившиеся половым путём бананы почти несъедобны. Копируют ли фермеры бананы вручну копируя биохимию оригинала? Или доверяют банановому растению самому создать себе копию?



Фермер использует **шаблон Prototype** (с англ. прототип) для создания новых банановых растений, ведь фермер не разбирает внутреннюю биохимическую структуру банана, а доверяет ему самому вырастить себе копию.

---

# Порождающий шаблон Prototype

Как спроектировать класс, объекты которого:

- **Мы хотим при желании копировать в разных местах программы.** Заполнять в таких местах поля нового объекта полями старого?
- **Может иметь приватную внутреннюю структуру,** которая не будет доступна для внешних классов в местах копирования
- **Может менять свою внутреннюю структуру,** не меняя код, который использует копирование наших объектов у себя в логике



# Порождающий шаблон Prototype

ОТВЕТ: это можно сделать через **шаблон проектирования Prototype** (с англ. прототип).

- Добавляем всю логику копирования внутрь нашего класса в виде метода `clone`
- Все кто хочет скопировать объект нашего класса делают это через наш метод, а не вручную копируя поля

```
public class Main {  
    public static class Point {  
        protected int x, y;  
        protected double fromOrigin;  
  
        private Point() {}  
  
        public Point(int x, int y) {  
            this.x = x;  
            this.y = y;  
  
            // Может быть тяжеловесное вычисление  
            this.fromOrigin = Math.sqrt(x * x + y * y);  
        }  
  
        public Point clone() {  
            Point ans = new Point();  
            ans.x = x;  
            ans.y = y;  
  
            // Копируем вместо вычисления заново  
            ans.fromOrigin = fromOrigin;  
            return ans;  
        }  
  
        public int getX() { return x; }  
        public void setX(int x) { this.x = x; }  
  
        public int getY() { return y; }  
        public void setY(int y) { this.y = y; }  
    }  
}
```



# Итоги

---

# Итоги

- Мы поняли что скрывается за термином “Шаблоны проектирования”
- Какую роль знание шаблонов проектирования играет в разработке
- Познакомились с тремя порождающими шаблонами:
  - **Builder** (строитель) — вынос сложного процесса создания в отдельный класс-строитель наших объектов
  - **Singleton** (одиночка) — ограничиваем количество созданных объектов одним
  - **Factory method** (фабричный метод) — логику создания вспомогательных объектов через переопределение метода оставляем потомкам
  - **Abstract factory** (абстрактная фабрика) — логику создания вспомогательных объектов даём отдельному классу
  - **Prototype** (прототип) — логику копирования реализуем внутри класса, экспортируем в виде отдельного метода



## 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**