

Шаблоны проектирования

План

1. Что такое шаблон проектирования?
2. История появления
3. Преимущества и критика
4. Классификация шаблонов проектирования
5. Примеры некоторых шаблонов
6. Выводы

Что такое шаблон проектирования?

- В процессе написания программ (как и в любой другой области) часто возникают похожие задачи
- Мы люди умные, и не хотим каждый раз “изобретать велосипед”
- Мы можем выработать готовые решения для этих похожих задач и использовать их при необходимости
- Такие решения называются шаблонами проектирования

Что такое шаблон проектирования?

- Шаблон проектирования (англ. design pattern) – многократно используемое решение распространённых задач при разработке программного обеспечения
- По своей сути шаблоны – абстрактные решения для абстрактных ситуаций

История появления

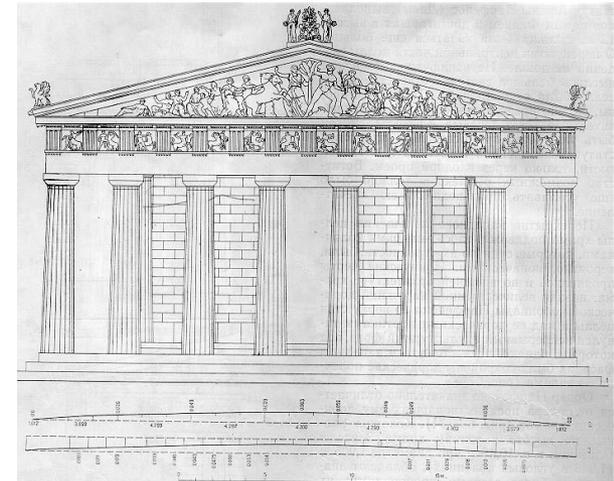
Шаблоны проектирования
впервые появились в
строительной архитектуре



Человечество на
сегодняшний день:

5000 лет опыта в
строительстве

Менее 40 лет опыта в
разработке и проектировании



История появления

Кристофер Александер – архитектор и философ.

Две книги:

- “A Pattern Language: Towns, Buildings, Constructions” (Oxford University Press, 1977)
- “The Timeless Way of Building” (Oxford University Press, 1979)

Идеи нашли применение в других областях, в том числе и в разработке ПО

Эрик Гамма, Ричард Хелм, Джон Влиссидес, Ральф Джонсон – “Design patterns” или “Gang of Four” (GoF – банда четырёх), середина 1990-х

Преимущества

- Стандартный шаблон можно хорошо проработать
 - В программах, где он используется, будет меньше кода, потенциально содержащего ошибки
- **Упрощается коммуникации между программистами**
 - Если два программиста говорят об одной и той же ситуации, но разными словами – они могут друг друга не понять. Когда они говорят о стандартном шаблоне – обоим понятно, о чём идёт речь.
- Легче разобраться в новой системе – в ней есть знакомые части
- Всё придумано до нас
 - Существует множество готовых и проверенных временем стандартных шаблонов. Мы можем использовать их в своих программах.

Критика

- Зацикливание на шаблонах порождает привычку пользоваться стандартными вещами и отучает нас придумывать нестандартные решения
- Шаблоны часто применяют слепо, поэтому использование оборачивается только ограничениями:
 - “У нас есть задача - этот шаблон лучше всего её решает - выбираем его” – правильно
 - “Я знаю клёвый шаблон - давайте подгоним под него задачу” – не правильно

Классификация шаблонов

1. Основные
2. Порождающие
3. Структурные
4. Поведенческие
5. Разделяющие
6. Архитектурные
7. Анти-паттерны

И другие

Классификация шаблонов

- Основные
 - Фундаментальные, используются другими шаблонами
 - Delegation, interface, abstract superclass, interface and abstract class, immutable, marker-interface, proxy
- Порождающие
 - Делают систему независимой от процесса создания объектов
 - Factory method, abstract factory, builder, prototype, singleton, object pool

Классификация шаблонов

- Структурные
 - Описывают, как их классов и объектов собираются более крупные структуры
 - Adapter, iterator, bridge, facade, flyweight, dynamic linkage, virtual proxy, decorator, cache management
- Поведенческие
 - Определяют взаимодействие между компонентами. Увеличивают его гибкость
 - Chain of responsibility, command, little language, mediator, snapshot, observer, state, null object, strategy, template method, visitor

Классификация шаблонов

- Разделяющие
 - Описывают, как их классов и объектов собираются более крупные структуры
 - Filter, composite, read only interface
- Архитектурные
 - Определяют взаимодействие между компонентами. Увеличивают его гибкость
 - Model-view-controller, layer, client-server...
- Анти-паттерны
 - Как не следует поступать при разработке программ

Singleton (одиначка)

- Гарантирует, что у класса есть только один экземпляр и предоставляет к нему глобальную точку доступа
- Сам класс запрещает создание дополнительных экземпляров

Singleton (одиночка)

```
public class Singleton {  
    private static Singleton instanse;  
    private String message = "Hello, I'm Singleton!";  
  
    private Singleton() {  
    }  
  
    public static synchronized Singleton getInstance() {  
        if (instanse == null) {  
            instanse = new Singleton();  
        }  
        return instanse;  
    }  
  
    public void say() {  
        System.out.println(message);  
    }  
}
```

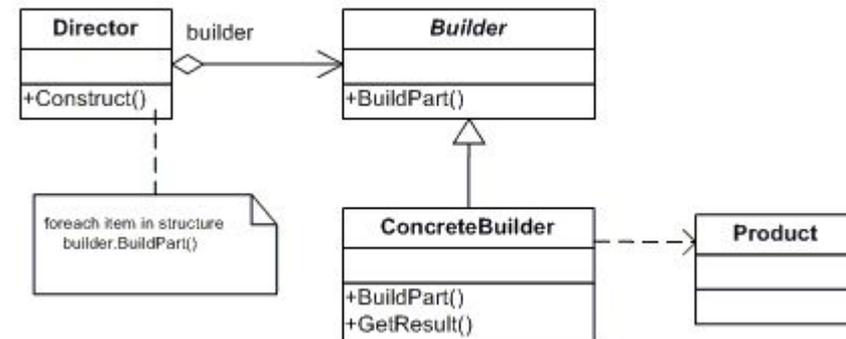
Singleton
- <u>singleton</u> : Singleton
- Singleton()
+ <u>getInstance()</u> : Singleton

Singleton (одиночка)

- Цели использования
 - Централизованное управление ресурсом
 - Доступ из любой точки программы
 - Создание объекта требует больших ресурсов памяти или процессора
- Примеры из жизни
 - Текущий проигрываемый аудиоклип в плеере
 - Текущий пользователь в системе
 - Класс для локализации
 - Spring application context (требует много ресурсов)
 - Hibernate session factory (при создании считывает конфигурационные файлы)

Builder (Строитель)

- Делает процесс создания сложного объекта независимым от того, как конструируются сами части и как они взаимодействуют между собой
- Состоит из нескольких ролей:
 - Builder - определяет, как создаются части
 - Director - определяет, как собирается объект из этих частей
 - Product – то, что получается в результате построения



Builder (Строитель)

```
class Pizza {
    private String dough = "";
    private String sauce = "";
    private String topping = "";
    public void setDough(String dough) {
        this.dough = dough;
    }

    public void setSauce(String sauce) {
        this.sauce = sauce;
    }

    public void setTopping(String topping) {
        this.topping = topping;
    }
}

abstract class PizzaBuilder {
    protected Pizza pizza;

    public Pizza getPizza() {
        return pizza;
    }

    public void createNewPizzaProduct() {
        pizza = new Pizza();
    }

    public abstract void buildDough();
    public abstract void buildSauce();
    public abstract void buildTopping();
}
```

Builder (Строитель)

```
class HawaiianPizzaBuilder extends PizzaBuilder {  
    public void buildDough() {  
        pizza.setDough("cross");  
    }  
    public void buildSauce() {  
        pizza.setSauce("mild");  
    }  
    public void buildTopping() {  
        pizza.setTopping("ham+pineapple");  
    }  
}
```

```
class SpicyPizzaBuilder extends PizzaBuilder {  
    public void buildDough() {  
        pizza.setDough("pan baked");  
    }  
    public void buildSauce() {  
        pizza.setSauce("hot");  
    }  
    public void buildTopping() {  
        pizza.setTopping("pepperoni+salami");  
    }  
}
```

Builder (Строитель)

```
class Cook {
    private PizzaBuilder pizzaBuilder;
    public void setPizzaBuilder(PizzaBuilder pb) {
        pizzaBuilder = pb;
    }
    public Pizza getPizza() {
        return pizzaBuilder.getPizza();
    }
    public void constructPizza() {
        pizzaBuilder.createNewPizzaProduct();
        pizzaBuilder.buildDough();
        pizzaBuilder.buildSauce();
        pizzaBuilder.buildTopping();
    }
}

public class BuilderExample {
    public static void main(String[] args) {
        Cook cook = new Cook();
        PizzaBuilder hawaiianPizzaBuilder = new HawaiianPizzaBuilder();
        PizzaBuilder spicyPizzaBuilder = new SpicyPizzaBuilder();

        cook.setPizzaBuilder(hawaiianPizzaBuilder);
        cook.constructPizza();
        Pizza hawaiian = cook.getPizza();

        cook.setPizzaBuilder(spicyPizzaBuilder);
        cook.constructPizza();
        Pizza spicy = cook.getPizza();
    }
}
```

Builder (Строитель)

- Пример – выборка товара из каталога по сложным критериям
- При выборе очередного критерия могут открываться новые контролы
 - Builder: различный для каждого типа товаров: фотоаппараты, телефоны, плееры
 - Director: форма выбора товара
 - Product: объект с критериями

Abstract Factory (Абстрактная фабрика)

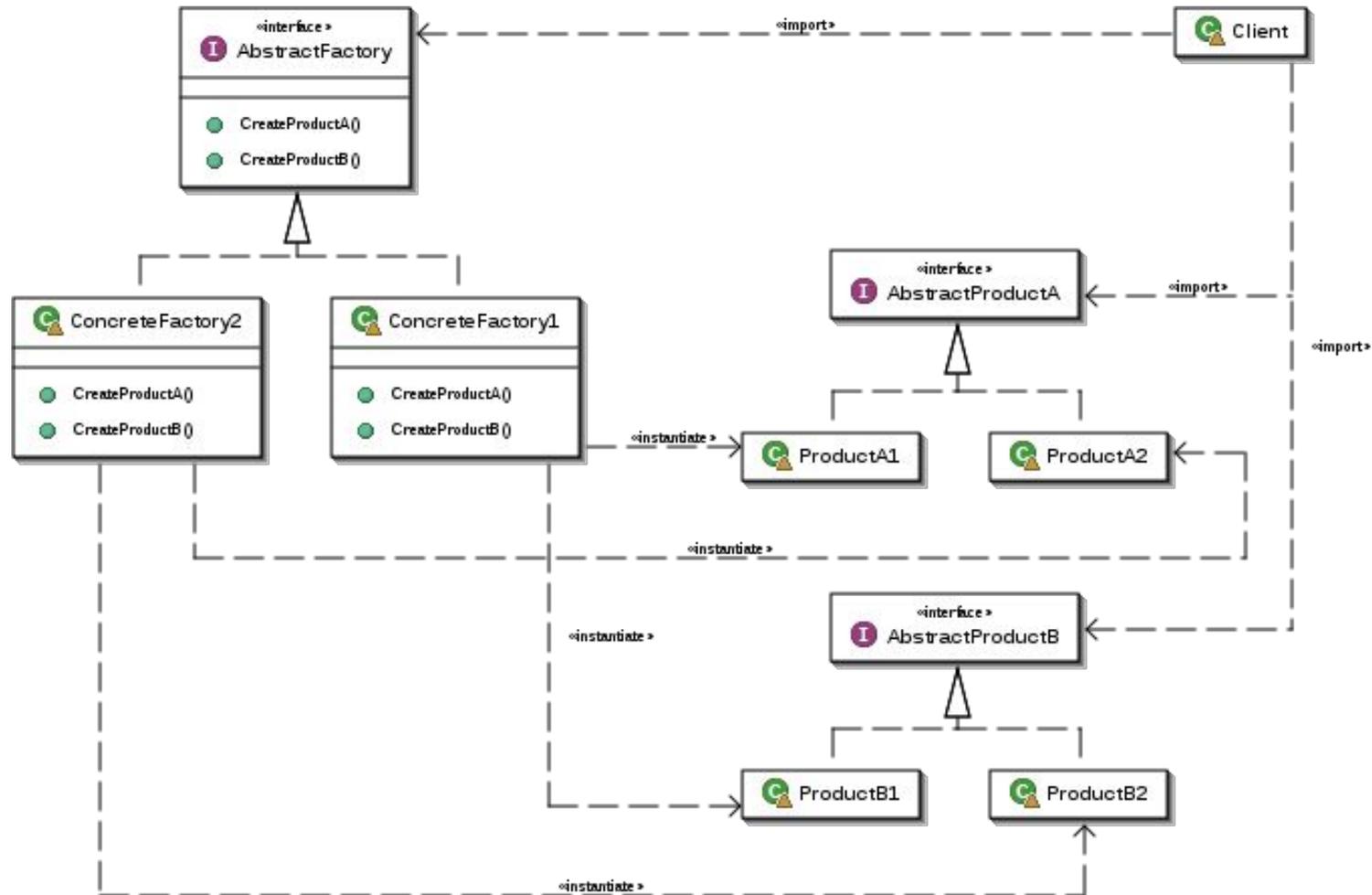
- Интерфейс для создания группы объектов, принадлежащих одному набору классов и используемых совместно
- Конкретные классы созданных объектов неизвестны
- Система не зависит от того, как создаются объекты
- Логика создания объектов спрятана в фабрике
- Абстрактная фабрика переопределяется в конкретных классах-фабриках

Abstract Factory (Абстрактная фабрика)

Пример – различные стили пользовательского интерфейса

- Мы хотим, чтобы в программе можно было поменять тему оформления
- Описываем интерфейс для фабрики, которая создаёт компоненты пользовательского интерфейса: кнопки, текстовые поля, списки, и т.д.
- Для каждой темы делаем свою реализацию фабрики
- В любой момент подменяем одну фабрику на другую и внешний вид пользовательского интерфейса меняется

Abstract Factory (Абстрактная фабрика)



Abstract Factory (Абстрактная фабрика)

```
interface GUIFactory {  
    public Button createButton();  
}  
  
class WinFactory implements GUIFactory {  
    public Button createButton() {  
        return new WinButton();  
    }  
}  
  
class LinuxFactory implements GUIFactory {  
    public Button createButton() {  
        return new LinuxButton();  
    }  
}  
  
interface Button {  
    public void paint();  
}
```

Abstract Factory (Абстрактная фабрика)

```
class WinButton implements Button {
    public void paint() {
        System.out.println("I'm a WinButton");
    }
}

class LinuxButton implements Button {
    public void paint() {
        System.out.println("I'm an LinuxButton");
    }
}

class Application {
    public Application(GUIFactory factory) {
        Button button = factory.createButton();
        button.paint();
    }
}

public class Main{
    public static void main(String[] args) {
        new Application(new LinuxFactory());
    }
}
```

Observer (Наблюдатель)

- Второе название – “издатель-подписчик”
- Один объект генерирует события, другие объекты эти события получает и на них реагируют
- Первый объект – наблюдаемый (издатель), остальные – наблюдатели (подписчики)

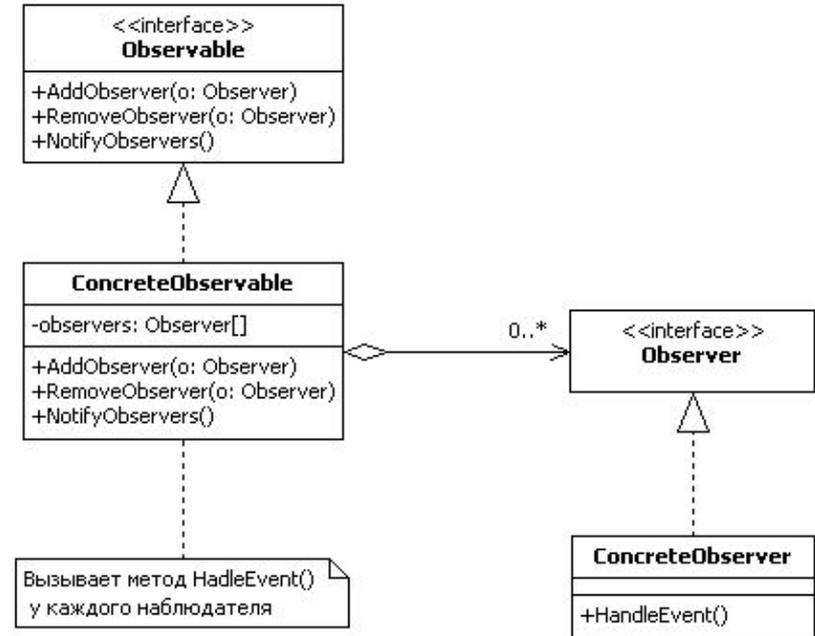
Observer (Наблюдатель)

```
import java.util.Observable;  
import java.util.Observer;
```

```
public class MyObservable extends Observable {  
    public void doSomething() {  
        setChanged();  
        notifyObservers();  
    }  
}
```

```
public class MyObserver implements Observer {  
    public void update (Observable obj, Object arg) {  
        System.out.println("updated");  
    }  
}
```

```
public class MyApp {  
    public static void main(String args[]) {  
        MyObservable a = new MyObservable();  
        MyObserver b = new MyObserver();  
        a.addObserver(b);  
        a.doSomething();  
    }  
}
```



Observer (Наблюдатель)

- Пример – диаграмма в Excel, построенная по таблице
- При изменении данных в таблице – меняется диаграмма
- Объект “диаграмма” является подписчиком всех объектов “ячейка”

Strategy (Стратегия)

- Общий интерфейс для абстрактного алгоритма
- Разные реализации конкретных алгоритмов - стратегий
- Если нужно поменять алгоритм – подменяем одну стратегию другой

Strategy (Стратегия)

```
class StrategyExample {  
    public static void main(String[] args) {  
        IStrategy myStrategy;  
        myStrategy = new AddStrategy();  
        int resultA = myStrategy.count (3,4);  
        myStrategy = new MultStrategy();  
        int resultB = myStrategy .count(3,4);  
    }  
}
```

```
interface Strategy {  
    int count(int a, int b);  
}
```

```
class AddStrategy implements IStrategy {  
    public int count(int a, int b) {  
        return a + b;  
    }  
}
```

```
class MultStrategy implements IStrategy {  
    public int count(int a, int b) {  
        return a * b;  
    }  
}
```

Strategy (Стратегия)

- Пример 1 – фильтры в Photoshop, на вход подаётся изображение, к нему применяется стратегия-фильтр
- Пример 2 – поведение ботов в играх
- Пример 3 – Forex, каждый участник может реализовать свой алгоритм, который будет вести торги

Facade (Фасад)

- Единая обёртка для группы классов
- Мы пользуемся фасадом и не думаем о классах, которые лежат за ним
- Уменьшается связанность между частями системы



Facade (Фасад)

- Пример – любые библиотеки и модули, у которых описан интерфейс

Facade (Фасад)

```
public class MessageHeader {  
    private String from;  
    private String to;  
    public String getFrom() {  
        return from;  
    }  
    public void setFrom(String from) {  
        this.from = from;  
    }  
    public String getTo() {  
        return to;  
    }  
    public void setTo(String to) {  
        this.to = to;  
    }  
}
```

```
public class MessageBody {  
    private String text;  
    public String getText() {  
        return text;  
    }  
    public void setText(String text) {  
        this.text = text;  
    }  
}
```

Facade (Фасад)

```
public class Message {  
    private MessageBody body;  
    private MessageHeader header;  
  
    public MessageBody getBody() {  
        return body;  
    }  
  
    public void setBody(MessageBody body) {  
        this.body = body;  
    }  
  
    public MessageHeader getHeader() {  
        return header;  
    }  
  
    public void setHeader(MessageHeader header) {  
        this.header = header;  
    }  
}
```

Facade (Фасад)

```
public class MessageCreatorFacade {  
    private Message message = new Message();  
    private MessageHeader mh = new MessageHeader();  
    private MessageBody mb = new MessageBody();  
  
    public void setFrom(String from) {  
        mh.setFrom(from);  
    }  
  
    public void setTo(String to) {  
        mh.setTo(to);  
    }  
  
    public void setText(String text) {  
        mb.setText(text);  
    }  
  
    public Message createMessage() {  
        message.setBody(mb);  
        message.setHeader(mh);  
        return message;  
    }  
}
```

Выводы

- Использование паттернов избавляет от необходимости постоянно придумывать новые решения
 - В этом как положительный, так и отрицательный момент
- Использование паттернов делает код менее подверженным ошибкам
- Знание паттернов упрощает коммуникации между разработчиками
- Зная паттерны - мы имеем множество готовых решений наших задач