

# Объектно-ориентированное проектирование. Шаблоны проектирования

Алексей Островский

Физико-технический учебно-научный центр НАН Украины

21 ноября 2014 г.

# Объектно-ориентированное проектирование

## Определение

**Объектно-ориентированное проектирование** (англ. *object-oriented design*) — решение задачи проектирования программной системы с использованием объектов и взаимодействий между ними.

## Определение

**Объект** — сильная связь между структурами данных и методами ( $\simeq$  функциями), обрабатывающими эти данные.

### Составляющие объекта:

- ▶ идентификатор;
- ▶ свойства;
- ▶ методы.

## Концепции ООП

- ▶ **Объекты;**
- ▶ **инкапсуляция** — скрывание информации от внешних (по отношению к системе/объекту) сущностей;
- ▶ **наследование** — повторное использование методов работы с данными в различных условиях; дополнение функциональности объектов;
- ▶ **интерфейсы** — формальное описание методов и данных, используемое для взаимодействия между объектами; разделение описания и имплементации;
- ▶ **полиморфизм** — возможность использования наследованных от объекта потомков в том же контексте, что и сам объект.

# Процесс ООП

## Входные данные для ООП:

- ▶ концептуальная модель (диаграмма классов UML с основными понятиями предметной области, независимая от реализации);
- ▶ варианты применения;
- ▶ диаграммы последовательности для вариантов применения;
- ▶ реляционная модель данных (может разрабатываться параллельно с объектами).

## Процесс ООП (продолжение)

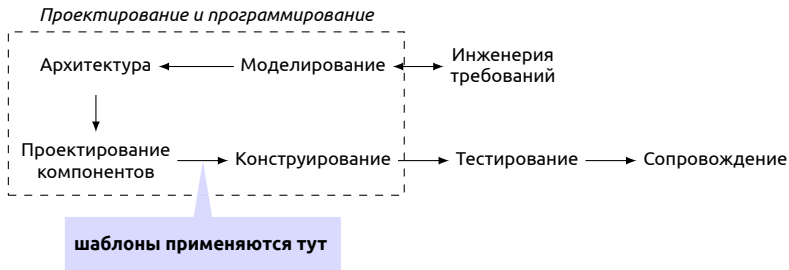
### Процесс ООП:

- ▶ разграничение объектов, составление диаграммы классов (сущности  $\Rightarrow$  объекты);
- ▶ конкретизация диаграмм последовательности;
- ▶ определение контекста: используемых библиотек, КПИ, ...;
- ▶ подбор и реализация шаблонов проектирования;
- ▶ определение взаимодействия объектов с источниками данных (базы данных, удаленные объекты).

# Шаблоны проектирования

## Определение

**Шаблон проектирования** (англ. *design pattern*) — типовой конструктивный элемент программной системы, задающий взаимодействие нескольких компонентов системы, а также роли и сферы ответственности исполнителей.



Роль шаблонов в разработке ПО

## Составляющие шаблонов

### Составляющие:

- ▶ название;
- ▶ область применения, описание проблемы, которую решает шаблон проектирования;
- ▶ обобщенная структура шаблона: основные компоненты, их взаимоотношения и выполняемые функции (на естественном языке или диаграмма классов UML);
- ▶ результат применения шаблона, возможные отрицательные последствия.

### Чем не являются шаблоны:

- ▶ **Шаблон  $\neq$  архитектура:** архитектура системы более абстрактна, шаблон подразумевает конкретную реализацию;
- ▶ **Шаблон  $\neq$  КПИ:** шаблон требует имплементации, КПИ — это готовый код.

# Классификация шаблонов проектирования

## Область применения:

- ▶ общего назначения;
- ▶ для конкретной предметной области (пользовательский интерфейс, защита информации, веб-дизайн, ...)

## Уровень проектирования:

- ▶ уровень архитектуры;
- ▶ уровень отдельных компонентов.

## Цель применения:

- ▶ порождающие шаблоны (англ. *creational patterns*);
- ▶ структурные шаблоны (англ. *structural patterns*);
- ▶ поведенческие шаблоны (англ. *behavioral patterns*).



## Примеры шаблонов

**Порождающие:** фабрика, [строитель](#), [одиночка](#), прототип, ...

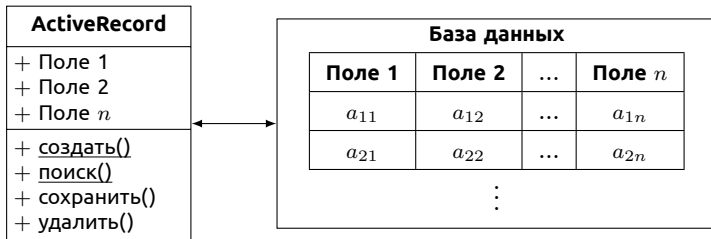
**Структурные:** адаптер, [мост](#), [декоратор](#), фасад, компоновщик, Ргоху, ...

**Поведенческие:** [итератор](#), [наблюдатель](#), команда, состояние, хранитель, посредник, цепочка обязанностей, ...

**Архитектурные:** [ActiveRecord](#), Data Mapper, ленивая загрузка, ...

**Параллелизация:** блокировка, семафоры, монитор, пул нитей исполнения, ...

## Архитектурный шаблон: ActiveRecord



Шаблон доступа к БД ActiveRecord. Черта снизу в UML обозначает статические поля и методы. «+» обозначает общедоступные поля/методы.

## ActiveRecord — описание

<b>Название:</b>	ActiveRecord
<b>Проблема:</b>	обеспечение доступа к реляционным базам данных в объектно-ориентированных приложениях.
<b>Решение:</b>	Каждой таблице (представлению) в БД соответствует свой класс; каждой строке таблицы — экземпляр класса; столбцам таблицы — поля объекта. В классе определены методы для сохранения объекта в БД, удаления и поиска. Ключи (foreign key) определяют отношения между классами AR.
<b>Недостатки:</b>	избыточное количество и/или непрозрачность запросов к СУБД (ср. с Data Mapper); проблема идентичности структуры класса и таблицы БД.
<b>Примеры:</b>	в составе MVC в веб-фреймворках (напр., CakePHP, Propel, Yii, Ruby on Rails).

## Порождающий шаблон: Singleton

Singleton
– <u>instance</u> : Singleton
+ <u>getInstance()</u> : Singleton
– Singleton() : void

UML-диаграмма классов для шаблона Singleton

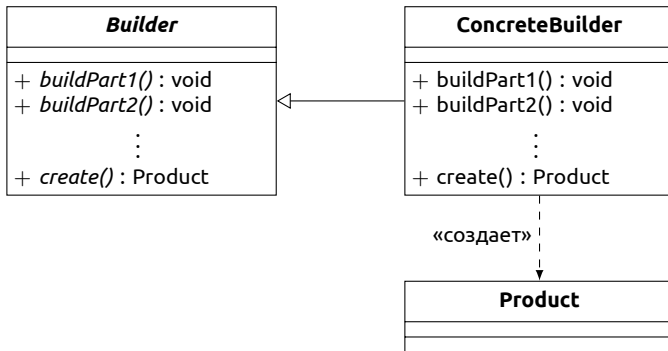
## Singleton — описание

<b>Название:</b>	Singleton (одиночка)
<b>Проблема:</b>	необходимость в строго одном объекте определенного класса (напр., для координации действий в системе; из соображений производительности).
<b>Решение:</b>	публичный статический метод для доступа к объекту, создающий при необходимости экземпляр класса и сохраняющий его в скрытой статической переменной.
<b>Недостатки:</b>	усложнение тестирования; введение скрытых зависимостей ( <a href="#">детальнее</a> ).
<b>Примеры:</b>	Системы ведения логов.
<b>Замена:</b>	<a href="#">внедрение зависимости</a> .

## Singleton — реализация

```
1  public class Singleton {
2      private static Singleton instance;
3
4      public static synchronized Singleton getInstance() {
5          if (instance == null) {
6              instance = new Singleton();
7          }
8          return instance;
9      }
10
11     private Singleton() { /* код инициализации */ }
12
13     public void run() { /* ... */ }
14 }
15
16 /* использование */
17 Singleton.getInstance().run();
```

## Порождающий шаблон: Builder



UML-диаграмма классов для шаблона Builder. *Курсивом* в UML обозначаются интерфейсы и абстрактные методы.

## Builder — описание

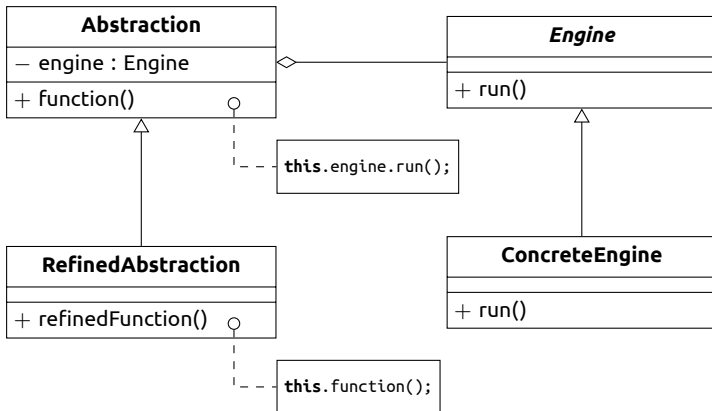
<b>Название:</b>	Builder (строитель)
<b>Проблема:</b>	создание объектов с заданным набором свойств без имплементации большого количества конструкторов.
<b>Решение:</b>	использование служебного объекта для пошагового задания свойств и создания результирующего объекта.
<b>Преимущества:</b>	тип объекта может варьироваться в зависимости от заданных параметров.
<b>Примеры:</b>	создание документов с жесткой структурой (SQL-запросов, XML/HTML-документов).



## Builder — реализация

```
1  /* Пример использования — Android API */
2  AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
3  builder.setMessage("Message")
4      .setTitle("text")
5      .setIcon(...)
6      .setPositiveButton(...)
7      .setNegativeButton(...);
8  AlertDialog dialog = builder.create();
```

## Структурный шаблон: Bridge



UML-диаграмма классов для шаблона Bridge

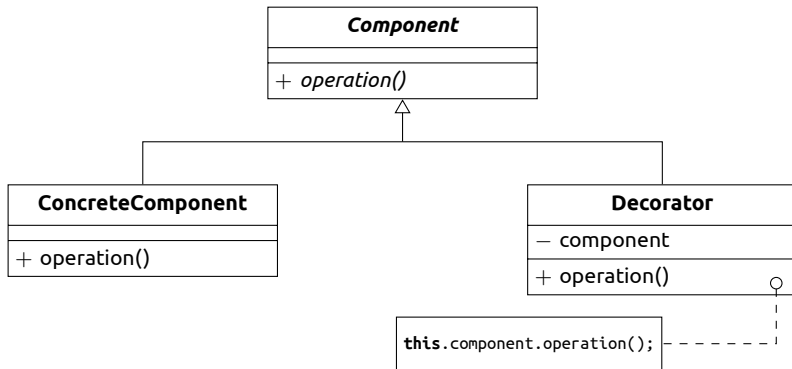
## Bridge — описание

- Название:** Bridge (мост)
- Проблема:** отделение функциональности, предоставляемой интерфейсом, от конкретной имплементации, чтобы они могли меняться независимо.
- Решение:** выделение методов с несколькими реализациями в отдельные классы; создание интерфейса, общего для всех реализаций.
- Недостатки:** возможно излишнее усложнение кода при наличии одной имплементации.
- Примеры:** мультиплатформенные графические интерфейсы (Java AWT, Qt).

## Bridge — реализация

```
1  public abstract class Shape {
2      public abstract void draw();
3  }
4
5  public class Circle extends Shape {
6      public Circle(api, center, radius) { /*...*/ }
7      public void draw() { api.drawCircle(center, radius); }
8  }
9
10 public interface DrawingAPI {
11     void drawCircle(Point c, double radius);
12     /* другие методы */
13 }
14
15 public class WindowsAPI implements DrawingAPI {
16     public void drawCircle(Point c, double radius) { /* ... */ }
17 }
18
19 public class LinuxAPI implements DrawingAPI {
20     public void drawCircle(Point c, double radius) { /* ... */ }
21 }
```

## Структурный шаблон: Decorator



UML-диаграмма классов для шаблона Decorator

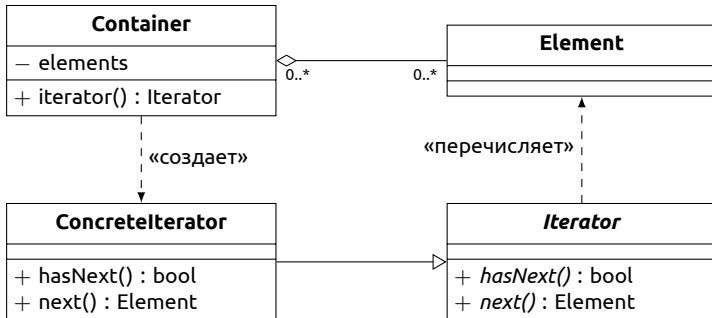
## Decorator — описание

<b>Название:</b>	Decorator (декоратор)
<b>Проблема:</b>	Изменение поведения конкретного объекта (при сохранении интерфейса), а не его класса в целом.
<b>Решение:</b>	Создание класса с интерфейсом исходного класса, который направляет вызовы методов исходному объекту после определенной обработки.
<b>Недостатки:</b>	усложнение читаемости кода; некорректное использование вместо создания подклассов.
<b>Примеры:</b>	ввод/вывод в Java; по аналогичному принципу работают декораторы в Python.

## Decorator — реализация

```
1  public interface Component {
2      public int operation(int x);
3  }
4
5  public class ComponentA implements Component {
6      public int operation(int x) { /* ... */ }
7  }
8
9  public class LoggingComponent implements Component {
10     public LoggingComponent(Component base) { /* ... */ }
11
12     public int operation(int x) {
13         int result = this.base.operation(x);
14         Log.info("Operation(" + x + ") = " + result);
15         return result;
16     }
17 }
```

## Поведенческий шаблон: Iterator



UML-диаграмма классов для шаблона Iterator



## Iterator — описание

<b>Название:</b>	Iterator (итератор)
<b>Проблема:</b>	Отделение функциональности последовательного доступа к элементам контейнера от внутренней структуры контейнера.
<b>Решение:</b>	Создание объекта-итератора, возвращаемого контейнером и содержащего в себе необходимую функциональность.
<b>Недостатки:</b>	некоторые алгоритмы не могут использовать итераторы, так как зависят от внутренней структуры контейнера.
<b>Примеры:</b>	коллекции в большинстве языков программирования.

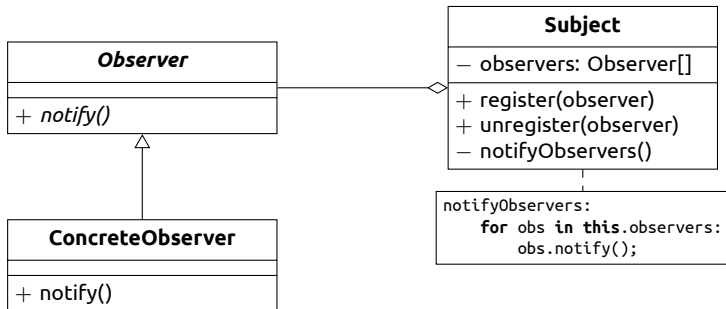
## Iterator — реализация

```

1  /* Пример использования в Java */
2  public class MyContainer<T> implements Collection<T> {
3      public Iterator<T> iterator() { /* ... */ }
4  }
5
6  Container<String> container = ...;
7  for (String str : container) {
8      System.out.println(str);
9  }
10
11  /* эквивалентный способ */
12  for (Iterator<String> it = container.iterator(); it.hasNext(); ) {
13      String str = it.next();
14      System.out.println(str);
15  }

```

## Поведенческий шаблон: Observer



UML-диаграмма классов для шаблона Observer

## Observer — описание

<b>Название:</b>	Observer (наблюдатель)
<b>Проблема:</b>	своевременное обновление состояния для зависимых друг от друга объектов.
<b>Решение:</b>	Хранение списка зависимых объектов и уведомление их об изменении состояния.
<b>Недостатки:</b>	Утечки памяти (зависимые объекты хранятся в памяти до явного удаления зависимости с помощью метода <code>unregister</code> ).
<b>Примеры:</b>	системы графического пользовательского интерфейса.

## Observer — реализация

```
1  public interface ClickListener {
2      void onClick(Object sender);
3  }
4
5  public class Button {
6      private ClickListener listener;
7      public void setListener(ClickListener l) { this.listener = l; }
8
9      protected void doClick() {
10         /* ... */
11         if (this.listener != null) this.listener.onClick(this);
12     }
13 }
14
15 Button button = new Button();
16 button.setListener(new ClickListener() {
17     public void onClick(sender) {
18         System.out.println(sender + " was clicked!");
19     }
20 });
```

## Выводы

1. Объектно-ориентированное проектирование — один из основных подходов к проектированию ПО. В его рамках предметная область разбивается на объекты, взаимодействующие между собой.
2. Ключевые понятия ООП — наследование, полиморфизм, инкапсуляция, интерфейсы.
3. В рамках ООП часто используются стандартные элементы (шаблоны) программных систем. Выделяют архитектурные, порождающие, структурные и поведенческие шаблоны.

# Материалы



**Gamma, Erich et al.**

**Design Patterns.**

Addison-Wesley, 1995.



**Fowler, Martin et al.**

**Patterns of Enterprise Application Architecture.**

Addison-Wesley, 2002.



**Ward Cunningham et al.**

**Portland Pattern Repository.**

<http://c2.com/cgi/wiki?DesignPatterns>

(Вики по шаблонам проектирования. По совместительству — первая вики в мире.)

Спасибо за внимание!