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** — скрытие 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)

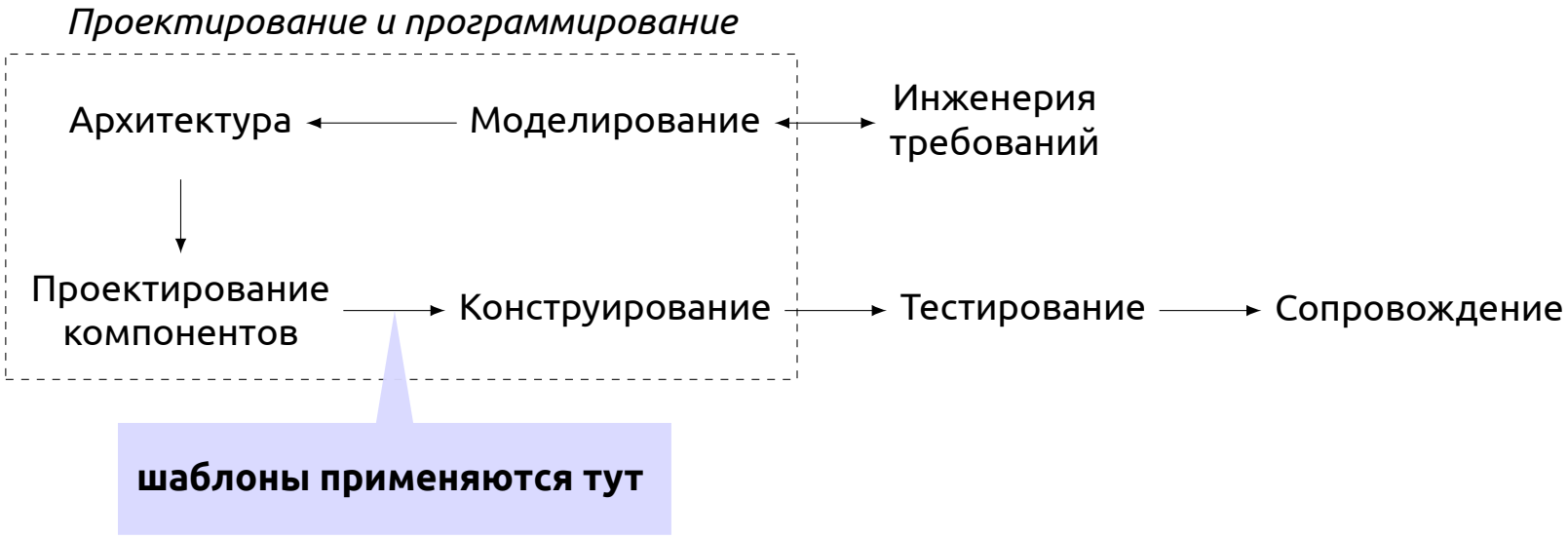
Процесс ООП:

- 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-дизнайн, …)

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, …

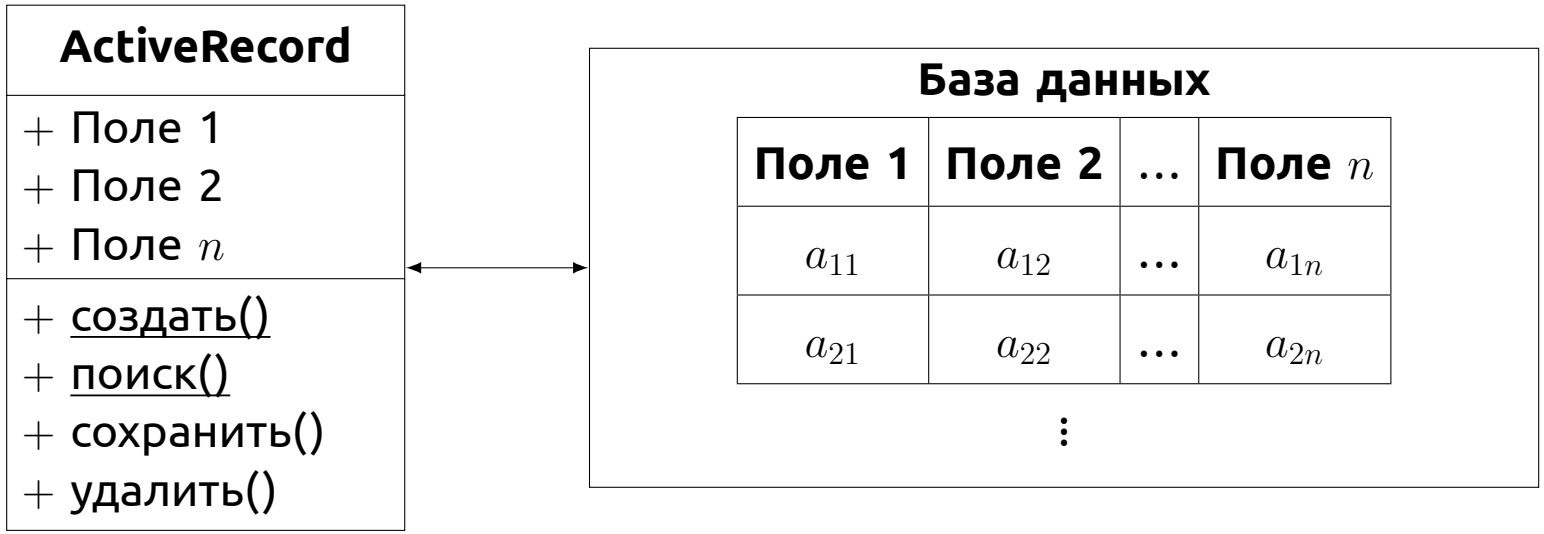
Структурные: адаптер, мост, декоратор, фасад, компоновщик, Proxy, …

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

Название: ActiveRecord

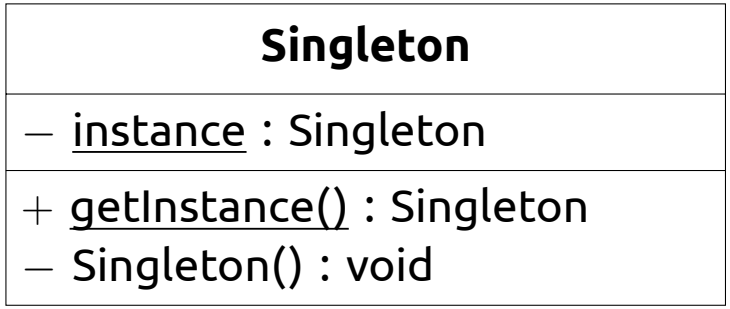
Проблема: обеспечение доступа к реляционным базам данных в объектно-ориентированных приложениях.

Решение: 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.

Недостатки: избыточное количество и/или непрозрачность запросов к СУБД (ср. с Data Mapper); проблема идентичности структуры класса и таблицы БД.

Примеры: в составе MVC в веб-фреймворках (напр., CakePHP, Propel, Yii, Ruby on Rails).

Порождающий шаблон: Singleton



UML-диаграмма классов для шаблона Singleton

Singleton — описание

Название: Singleton (одиночка)

Проблема: необходимость в строго одном объекте определенного класса (напр., для координации действий в системе; из соображений производительности).

Решение: публичный статический метод для доступа к объекту, создающий при необходимости экземпляр класса и сохраняющий его в скрытой статической переменной.

Недостатки: усложнение тестирования; введение скрытых зависимостей (детальнее).

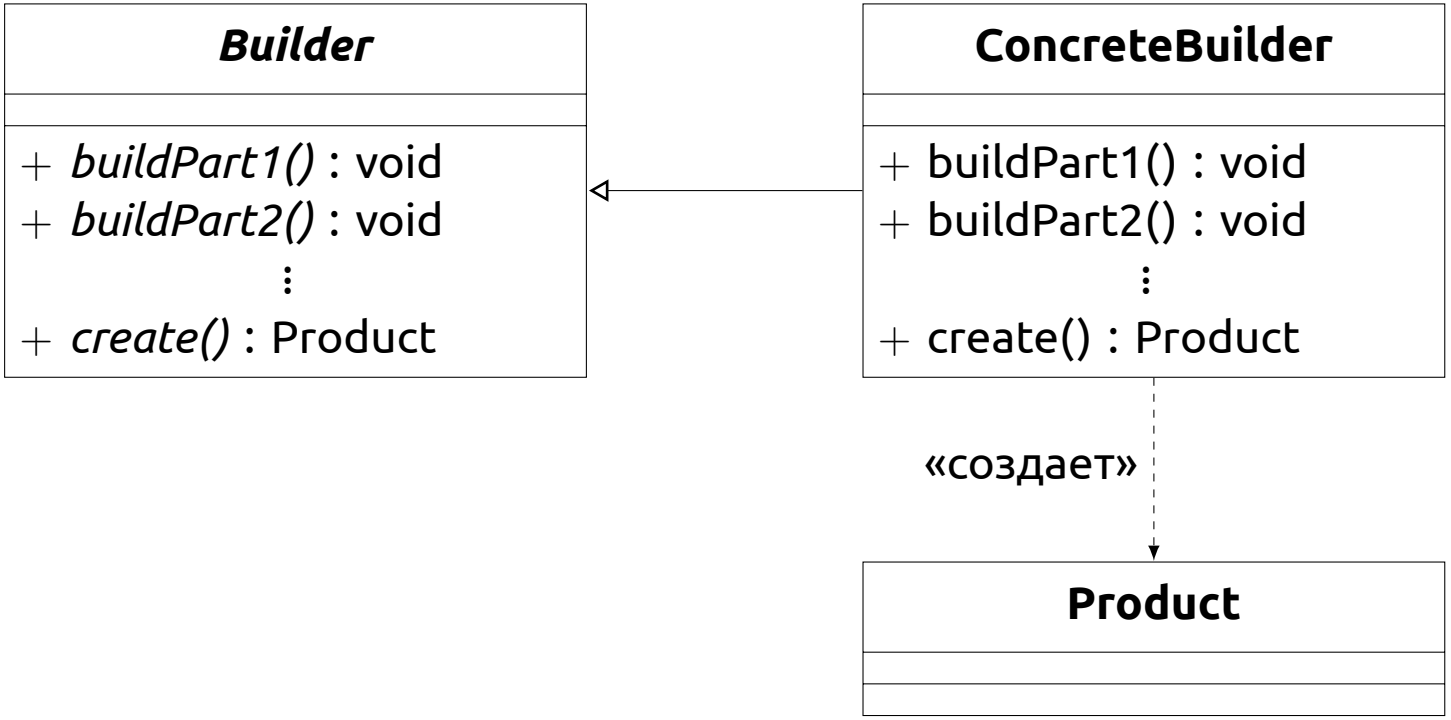
Примеры: Системы ведения логов.

Замена: внедрение зависимости.

Singleton — реализация

```java
public class Singleton {
    private static Singleton instance;

    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }

    private Singleton() { /* код инициализации */ }

    public void run() { /* ... */ }
}

/* использование */
Singleton.getInstance().run();
```

Порождающий шаблон: Builder



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

Builder — описание

Название: Builder (строитель)

Проблема: создание объектов с заданным набором свойств без имплементации большого количества конструкторов.

Решение: использование служебного объекта для пошагового задания свойств и создания результирующего объекта.

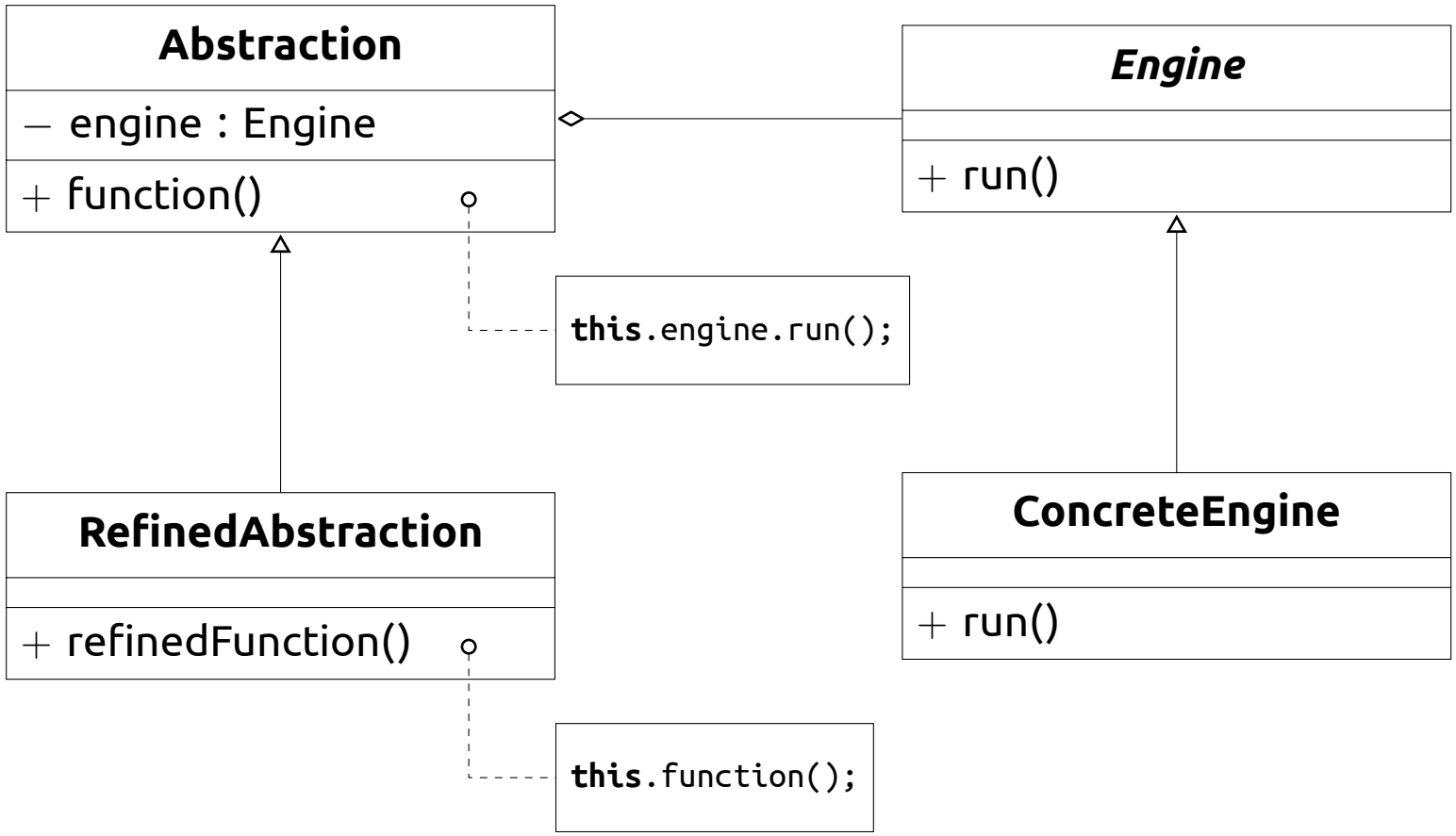
Преимущества: тип объекта может варьироваться в засимости от заданных параметров.

Примеры: 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.

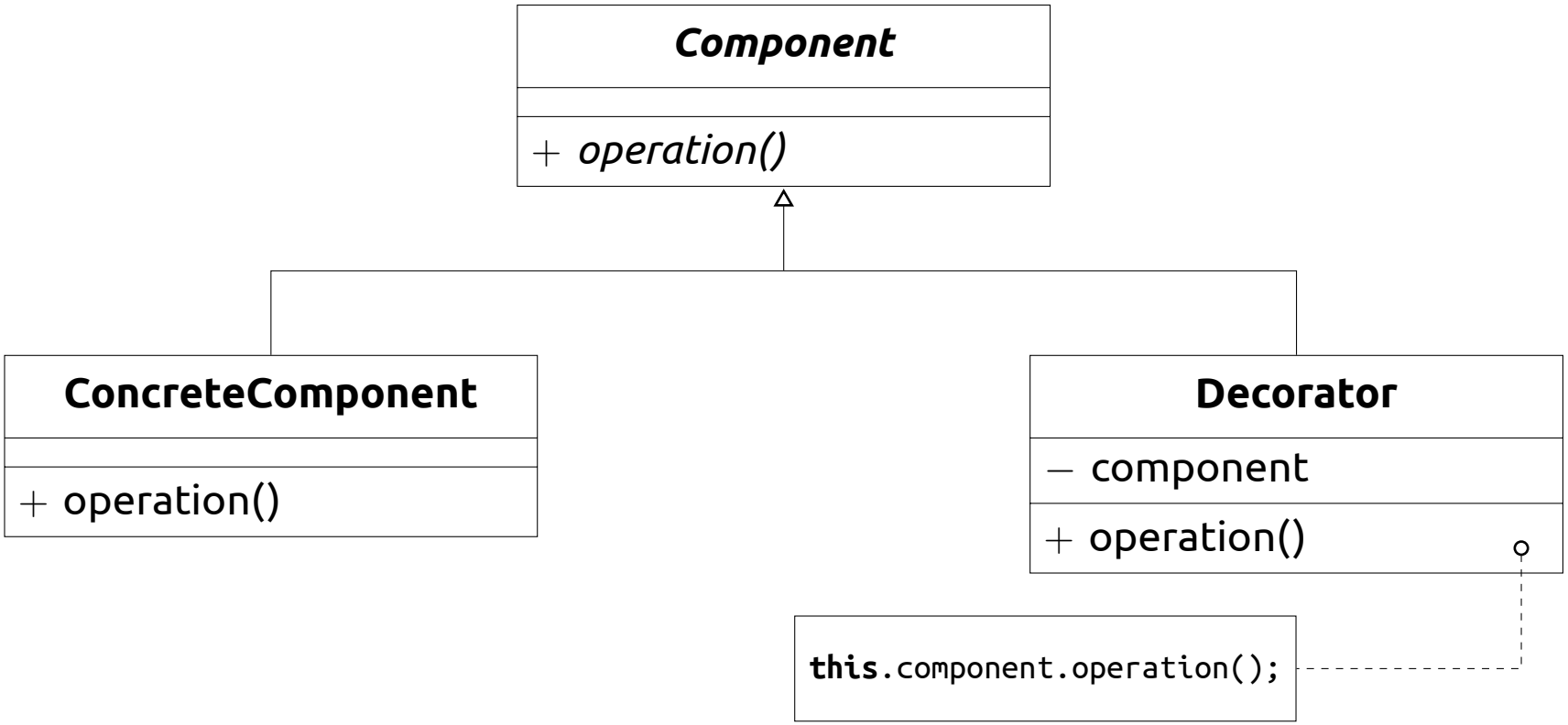
Недостатки: возможно излишнее усложнение кода при наличии одной имплементации.

Примеры: мультиплатформенные графические интерфейсы (Java AWT, Qt).

Bridge — реализация

```java
public abstract class Shape {
    public abstract void draw();
}

public class Circle extends Shape {
    public Circle(api, center, radius) { /*...*/ }
    public void draw() { api.drawCircle(center, radius); }
}

public interface DrawingAPI {
    void drawCircle(Point c, double radius);
    /* другие методы */
}

public class WindowsAPI implements DrawingAPI {
    public void drawCircle(Point c, double radius) { /* ... */ }
}

public class LinuxAPI implements DrawingAPI {
    public void drawCircle(Point c, double radius) { /* ... */ }
}
```

Структурный шаблон: Decorator



UML-диаграмма классов для шаблона Decorator

Decorator — описание

Название: Decorator (декоратор)

Проблема: Изменение поведения конкретного объекта (при сохранении интерфейса), а не его класса в целом.

Решение: Создание класса с интерфейсом исходного класса, который направляет вызовы методов исходному объекту после определенной обработки.

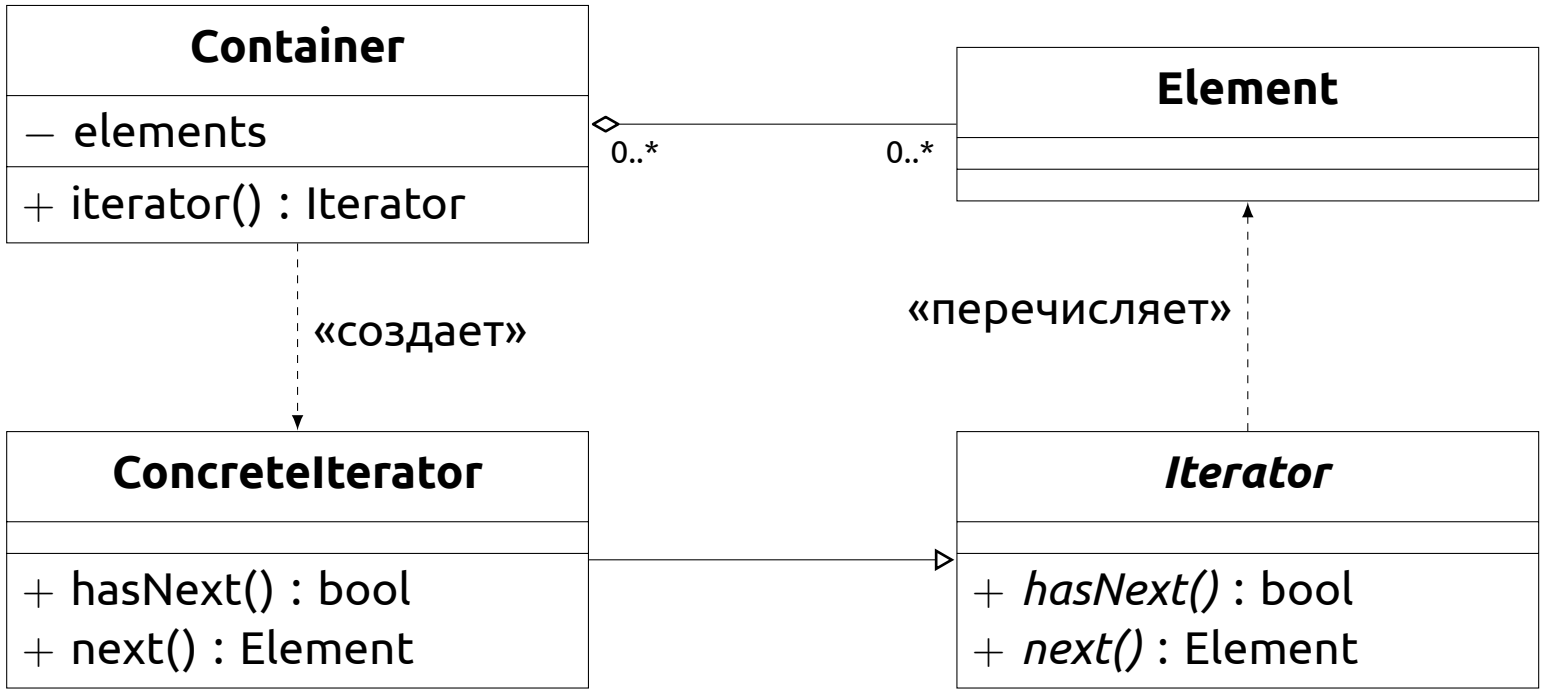
Недостатки: усложнение читаемости кода; некорректное использование вместо создания подклассов.

Примеры: 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

Название: Iterator (итератор)

Проблема: Отделение функциональности последовательного доступа к элементам контейнера от внутренней структуры контейнера.

Решение: Создание объекта-итератора, возвращаемого контейнером и содержащего в себе необходимую функциональность.

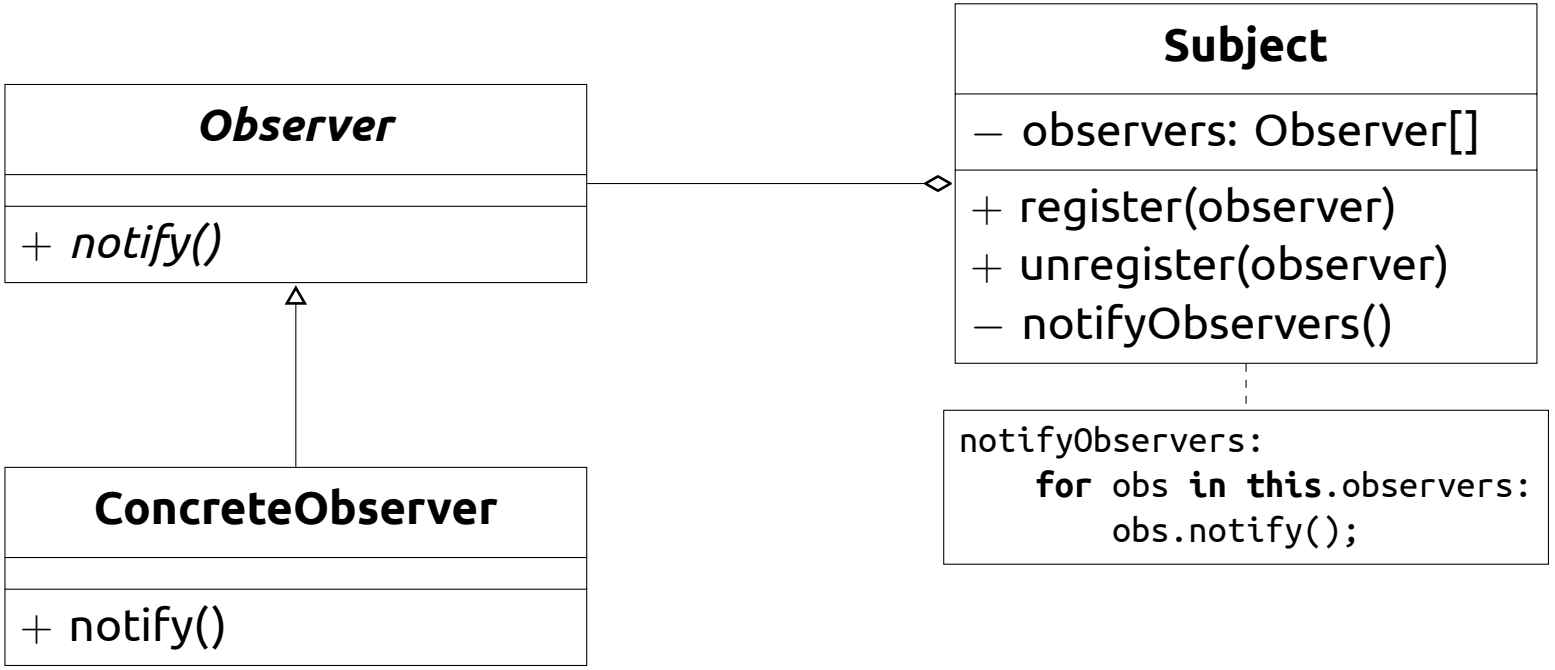
Недостатки: некоторые алгоритмы не могут использовать итераторы, так как зависят от внутренней структуры контейнера.

Примеры: коллекции в большинстве языков программирования.

Iterator — реализация

```java
/* Пример использования в Java */
public class MyContainer<T> implements Collection<T> {
    public Iterator<T> iterator() { /* ... */ }
}

Container<String> container = ...;
for (String str : container) {
    System.out.println(str);
}

/* эквивалентный способ */
for (Iterator<String> it = container.iterator(); it.hasNext(); ) {
    String str = it.next();
    System.out.println(str);
}
```

Поведенческий шаблон: Observer



UML-диаграмма классов для шаблона Observer

Observer — описание

Название: Observer (наблюдатель)

Проблема: своевременное обновление состояния для зависимых друг от друга объектов.

Решение: Хранение списка зависимых объектов и уведомление их об изменении состояния.

Недостатки: Утечки памяти (зависимые объекты хранятся в памяти до явного удаления зависимости с помощью метода `unregister`).

Примеры: 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!