

Паттерны проектирования

Приемы написания поддерживаемого кода

Валерий Алексеевич Овчинников
valery.ovchinnikov@phystech.edu

Мотивация

Мотивация: Зачем изучать паттерны?

- Перенимать опыт

Мотивация: Зачем изучать паттерны?

- Перенимать опыт
- Читая чужой код, можно понять, как улучшить свой

Мотивация: Зачем изучать паттерны?

- Перенимать опыт
- Читая чужой код, можно понять, как улучшить свой
- Заранее узнать о подводных камнях

Мотивация: Зачем изучать паттерны?

- Перенимать опыт
- Читая чужой код, можно понять, как улучшить свой
- Заранее узнать о подводных камнях
- Не изобретать велосипед

Мотивация: Чего мы хотим?



Мотивация: Чего мы хотим?

- SOLID код
- Понятный другим программистам код
- Поддерживаемый код

Мотивация: Как в этом помогают паттерны?

- Соответствуют принципам SOLID, даже помогают писать SOLID код
- Общеизвестны, а значит, понятны другим программистам
- Были разработаны для сложных ситуаций (в плане поддержки кода)

Мотивация: Когда использовать паттерны?

Только тогда, когда это необходимо



Command. Команда



Command. Команда: Что это?

```
interface Command {  
    void execute();  
}  
  
class ShoutingCommand implements Command {  
    private final String text;  
    ShoutingCommand(String text) {  
        this.text = text;  
    }  
    @Override  
    public void execute() {  
        System.out.println(text.toUpperCase());  
    }  
}
```

Command. Команда: Зачем это?

- Позволяет задавать различное поведение через единый интерфейс

Command. Команда: Зачем это?

- Позволяет задавать различное поведение через единый интерфейс
- Хранит данные и действия над ними рядом

Command. Команда: Зачем это?

- Позволяет задавать различное поведение через единый интерфейс
- Хранит данные и действия над ними рядом
- Разрывает связь между инициатором операции (знает когда и что) и ее исполнителем (знает как)

Command. Команда: Зачем это?

- Позволяет задавать различное поведение через единый интерфейс
- Хранит данные и действия над ними рядом
- Разрывает связь между инициатором операции (знает когда и что) и ее исполнителем (знает как)
- *Является альтернативой call-back'ам (функциям обратного вызова)

Command. Команда: Примеры использования

- Executors Framework (Runnable/Callable)
- Функторы в C++
- Транзакции
- В реактивном программировании

Decorator. Декоратор



Decorator. Декоратор: Что это?

```
interface Worker {  
    void work();  
}  
  
class GardenWorker implements Worker {  
    @Override  
    public void work() {  
        System.out.println("Cutting grass");  
    }  
}
```

Decorator. Декоратор: Что это?

```
class ProfilingWorkerDecorator implements Worker {  
    private final Worker worker;  
    ProfilingWorker(Worker worker) {  
        this.worker = worker;  
    }  
    @Override  
    public void work() {  
        long start = System.currentTimeMillis();  
        worker.work();  
        long duration = System.currentTimeMillis() - start;  
        System.out.println("Took " + duration + "ms");  
    }  
}
```

Decorator. Декоратор: Зачем это?

- Позволяет расширять функциональность классов не нарушая OCP, LSP

Decorator. Декоратор: Зачем это?

- Позволяет расширять функциональность классов не нарушая OCP, LSP
- Спасает от комбинаторного взрыва

Decorator. Декоратор: Зачем это?

- Позволяет расширять функциональность классов не нарушая OCP, LSP
- Спасает от комбинаторного взрыва
- Удобен для профилирования (при этом сохраняет SRP)

Decorator. Декоратор: Зачем это?

- Позволяет расширять функциональность классов не нарушая OCP, LSP
- Спасает от комбинаторного взрыва
- Удобен для профилирования (при этом сохраняет SRP)
- *Позволяет писать в стиле Aspect Oriented Programming

Decorator. Декоратор: Примеры использования

- Input/Output Streams в Java
- Добавление графических оформлений виджетам (темы)
- Добавление отладочной информации, проверки прав
- Proху класс в Java
- Встроенный механизм в язык Python (@)
- AspectJ

Весёлая задачка: (на самом деле нет)

Прокаченный ExecutorService

1. Добавить возможность

- 1.1 Измерять время выполнения задачи

Весёлая задачка: (на самом деле нет)

Прокаченный ExecutorService

1. Добавить возможность
 - 1.1 Измерять время выполнения задачи
 - 1.2 Ловить все ошибки задачи и выводить их на экран

Весёлая задачка: (на самом деле нет)

Прокаченный ExecutorService

1. Добавить возможность
 - 1.1 Измерять время выполнения задачи
 - 1.2 Ловить все ошибки задачи и выводить их на экран
 - 1.3 Изменять имя потока на имя класса выполняемой задачи

Весёлая задачка: (на самом деле нет)

Прокаченный ExecutorService

1. Добавить возможность
 - 1.1 Измерять время выполнения задачи
 - 1.2 Ловить все ошибки задачи и выводить их на экран
 - 1.3 Изменять имя потока на имя класса выполняемой задачи
2. Соответствовать принципам SOLID

Весёлая задачка: (на самом деле нет)

Прокаченный ExecutorService

1. Добавить возможность
 - 1.1 Измерять время выполнения задачи
 - 1.2 Ловить все ошибки задачи и выводить их на экран
 - 1.3 Изменять имя потока на имя класса выполняемой задачи
2. Соответствовать принципам SOLID
3. Использовать паттерны Builder, Fabric (method) + Decorator

Весёлая задача: (на самом деле нет)

Прокаченный ExecutorService

1. Добавить возможность
 - 1.1 Измерять время выполнения задачи
 - 1.2 Ловить все ошибки задачи и выводить их на экран
 - 1.3 Изменять имя потока на имя класса выполняемой задачи
2. Соответствовать принципам SOLID
3. Использовать паттерны Builder, Fabric (method) + Decorator
4. Конфигурировать (включать/выключать) новые функции

Builder. Строитель



Builder. Строитель: Что это?

```
class Person {  
    final String firstName;  
    final String middleName;  
    final String surname;  
    final Date dateOfBirth;  
    Person(String firstName, String middleName,  
           String surname, Date dateOfBirth) {  
        this.firstName = firstName;  
        this.middleName = middleName;  
        this.surname = surname;  
        this.dateOfBirth = dateOfBirth;  
    }  
}
```

Builder. Строитель: Что это?

```
class PersonBuilder {  
    String firstName;  
    ...  
    Person build() {  
        check();  
        return new Person(firstName, middleName, ...);  
    }  
  
    void withFirstName(String name) {  
        this.firstName = name;  
    }  
    ...  
}
```

Builder. Строитель: Что это?

```
PersonBuilder builder = new PersonBuilder();  
builder.withFirstName("Oleg");  
builder.withMiddleName("Olegovich");  
builder.withSurname("Olegov");  
builder.bornAt(dateOfBirth);  
Person oleg = builder.build();
```

Builder. Строитель: Зачем это?

- Сделать код более читаемым, если в конструкторе много (однотипных) параметров

Builder. Строитель: Зачем это?

- Сделать код более читаемым, если в конструкторе много (однотипных) параметров
- Если нужно создавать неизменяемый объект, но не все его параметры доступны сразу

Builder. Строитель: Зачем это?

- Сделать код более читаемым, если в конструкторе много (однотипных) параметров
- Если нужно создавать неизменяемый объект, но не все его параметры доступны сразу
- Если нужно делать сложные проверки аргументов

Builder. Строитель: Примеры использования

- StringBuilder
- Различные парсеры и конструкторы JSON/XML

Builder. Строитель: Fluent interface

Часто используется совместно с паттерном Строитель

```
Person oleg = builder.withFirstName("Oleg")  
                      .withMiddleName("Olegovich")  
                      .withSurname("Olegov")  
                      .bornAt(dateOfBirth)  
                      .build();
```

Builder. Строитель: Fluent interface

```
class PersonFluentBuilder {  
    String firstName;  
    ...  
    Person build() {  
        check();  
        return new Person(firstName, middleName, ...);  
    }  
  
    PersonFluentBuilder withFirstName(String name) {  
        this.firstName = name;  
        return this;  
    }  
    ...  
}
```

Factory. Фабрика



Factory. Фабрика: Что это?

Фабрика

```
class ElephantFactory {  
    static Elephant pinkElephant() {  
        return new Elephant("#FFC0CB");  
    }  
  
    static Elephant africanElephant() {  
        return new Elephant(ElephantType.African, "#D3D3D3",  
            getPhysicalParams(ElephantType.African));  
    }  
}
```

Factory. Фабрика: Что это?

Фабричный метод

```
public static LocalDate now();  
public static LocalDate ofEpochDay(long epochDay);  
public static LocalDate ofYearDay(int year, int dayOfYear);  
public static LocalDate parse(CharSequence text);
```


Factory. Фабрика: Зачем это?

- Сделать код более читаемым, если существует несколько разных конструкторов

Factory. Фабрика: Зачем это?

- Сделать код более читаемым, если существует несколько разных конструкторов
- Скрывает реализации, оставляя открытым только интерфейс

Factory. Фабрика: Зачем это?

- Сделать код более читаемым, если существует несколько разных конструкторов
- Скрывает реализации, оставляя открытым только интерфейс
- Позволяет компоновать семейства из совместимых объектов

Factory. Фабрика: Зачем это?

- Сделать код более читаемым, если существует несколько разных конструкторов
- Скрывает реализации, оставляя открытым только интерфейс
- Позволяет компоновать семейства из совместимых объектов
- *Один из способов организовать Inversion of Control

Factory. Фабрика: Примеры использования

- Executors Framework
- LocalDate
- Spring ServiceLocator

Observer. Наблюдатель



Observer. Наблюдатель: Что это?

```
interface Observer<T> {  
    void update(T observed);  
}  
  
class Publisher {  
    Collection<Observer<Publisher>> observers;  
    void subscribe(Observer<Publisher> observer) {  
        observers.add(observer);  
    }  
    void publish() {  
        for (Observer<Publisher> observer : observers) {  
            observer.update(this);  
        }  
    }  
}
```


Observer. Наблюдатель: Зачем это?

- Позволяет оповещать заинтересованные объекты о событиях

Observer. Наблюдатель: Зачем это?

- Позволяет оповещать заинтересованные объекты о событиях
- Механизм обеспечивается через единый интерфейс, избегается жесткая связанность

Observer. Наблюдатель: Примеры использования

- В подходе к проектированию Model-View-Controller
- В различных обработчиках событий (например, Button.onClick)
- В реализации паттерна-подхода Proactor

Весёлая задачка: (на самом деле нет)

Прокаченный ExecutorService

1. Добавить возможность
 - 1.1 Измерять время выполнения задачи
 - 1.2 Ловить все ошибки задачи и выводить их на экран
 - 1.3 Изменять имя потока на имя класса выполняемой задачи
 - 1.4 Подписываться на отправку задач в Executor
2. Соответствовать принципам SOLID
3. Использовать паттерны Builder, Fabric (method) + Decorator
4. Конфигурировать (включать/выключать) новые функции

Singleton. Уникум/Одиночка. Антипаттерн*



Singleton. Уникум/Одиночка. Антипаттерн*: Что это?

Объект класса Singleton существует в единственном экземпляре

Singleton. Уникум/Одиночка. Антипаттерн*: Что это?

```
class Singleton {  
    public static final Singleton INSTANCE = new Singleton();  
  
    private Singleton() {}  
}
```


Singleton. Уникум/Одиночка. Антипаттерн*: Что это?

```
class Singleton {  
    private static Singleton instance;  
  
    private Singleton() {}  
  
    public static Singleton instance() {  
        if (instance == null) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
}
```

Singleton. Уникум/Одиночка. Антипаттерн*: Зачем это?

- Не дает по ошибке создать несколько экземпляров объекта, который всегда должен быть уникальным

Singleton. Уникум/Одиночка. Антипаттерн*: Зачем это?

- Не дает по ошибке создать несколько экземпляров объекта, который всегда должен быть уникальным
- Часто реализован неправильно, еще чаще не к месту

Singleton. Уникум/Одиночка. Антипаттерн*: Примеры использования

- Объект, представляющий фаловую систему
- Редко – объект, представляющие соединение с базой

