Оглавление

- Адаптер (adapter)
- Декоратор (decorator)
- Заместитель (proxy)
- Адаптер, декоратор и прокси (обобщение)
- Мост (bridge)
- Компоновщик (composite)
- Фасад (facade)
- Приспособленец (flyweight)

Примеры поведенческих шаблонов проектирования

- Цепочка обязанностей (chain of responsibility)
- Итератор (iterator)
- Шаблонный метод (template method)
- Команда (command)
- Интерпретатор (interpreter)
- Посредник (mediator)
- Хранитель (memento)
- Наблюдатель (observer)
- Состояние (state)
- Стратегия (strategy)
- Посетитель (visitor)

Антипаттерн (anti-pattern)

Dependency Injection (внедрение зависимости)

Паттерны в Spring Framework

Паттерны в Hibernate

Шаблон (паттерн) проектирования (design pattern)

Проверенное и готовое к использованию решение. Никак не связан с конкретным языком программирования.

Плюсы использования:

· снижает сложность разработки (готовое решение некоторых проблем);

· облегчает коммуникацию разработчиками (ссылаться на конкретный шаблон);

· унификация деталей решений: модулей и элементов проекта;

· возможность отыскать универсальное решение нескольких проблем;

· помощь в выборе наиболее подходящего варианта проектирования.

Минусы использования:

· усложнению программы при слепом следовании выбранному шаблону;

· желание попробовать шаблон без особых на то оснований.

Основные характеристики шаблонов

· имя (уникальное для идентификации);

· назначение;

· задача (которую он решает);

· способ решения (способ, предлагаемый в шаблоне для решения задачи в том контексте, где этот шаблон был найден);

· участники (сущности);

· следствия (последствия от использования шаблона как результат действий);

· реализация (возможный вариант реализации шаблона).

Типы шаблонов проектирования

· ***Основные (fundamental)*** — для построения других типов шаблонов.

· ***Порождающие шаблоны (creational)*** — шаблоны проектирования, которые абстрагируют процесс создание экземпляра. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов. Шаблон, порождающий классы, использует наследование, чтобы изменять созданный объект, а шаблон, порождающий объекты, делегирует создание объектов другому объекту.

· ***Структурные шаблоны (structural)*** определяют различные сложные структуры, которые изменяют интерфейс уже существующих объектов или его реализацию, позволяя

облегчить разработку и оптимизировать программу.

· ***Поведенческие шаблоны (behavioral)*** определяют взаимодействие между объектами, увеличивая таким образом его гибкость.

Примеры основных шаблонов проектирования

Делегирование (delegation pattern) — передача ответственности за выполнение поведения связанному объекту.

Функциональный дизайн (functional design) — гарантия, что сущность имеет только одну задачу (обязанность).

Неизменяемый интерфейс (immutable interface) — создание неизменяемого объекта.

Интерфейс (interface) — общий метод структурирования сущностей, облегчающий их понимание.

Интерфейс-маркер (marker interface) — использование аннотаций для пометки объектной сущности.

Контейнер свойств (property container) — позволяет добавлять дополнительные свойства сущности в контейнер внутри себя, вместо расширения новыми свойствами.

Канал событий (event channel) — создает централизованный канал для событий. Использует сущность-представитель для подписки и сущность-представитель для публикации события в канале. Представитель существует отдельно от реального издателя или подписчика. Подписчик может получать опубликованные события от более чем одной сущности, даже если он зарегистрирован только на одном канале.

Примеры порождающих шаблонов проектирования

Абстрактная фабрика (abstract factory) позволяет создавать семейства связанных объектов разного типа (мебель (кресло, диван, стол) разных стилях) при помощи одного интерфейса.

Плюсы:

· гарантирует сочетаемость создаваемых продуктов;
· реализует второй принцип SOLID (принцип открытости/закрытости);
· упрощает добавление новых продуктов;
· избавляет клиентский код от привязки к конкретным классам продуктов;
· выделяет код производства продуктов в одно место, упрощая поддержку кода.

Минусы:

- много дополнительных классов;
- требует наличия всех типов продуктов в каждой вариации.

В Java — `newInstance()` (`javax.xml.parsers.DocumentBuilderFactory`).

```
1.  public interface Lada {
2.
3.      long getLadaPrice();
4.  }
```

```
1.  class UaLadaImpl implements Lada {
2.
3.      public long getLadaPrice() {
4.          return 1000;
5.      }
6.  }
```

```
1.  class RuLadaImpl implements Lada {
2.
3.      public long getLadaPrice() {
4.          return 10000;
5.      }
6.  }
```

```
1.  interface Ferrari {
2.
3.      long getFerrariPrice();
4.  }
```

```
1.  class UaFerrariImpl implements Ferrari {
2.
3.      public long getFerrariPrice() {
4.          return 3000;
5.      }
6.  }
```

```
1.  class RuFerrariImpl implements Ferrari {
2.
3.      public long getFerrariPrice() {
4.          return 30000;
```

```
    }
}
```

```
interface Porshe {

    long getPorshePrice();
}
```

```
class UaPorsheImpl implements Porshe {

    public long getPorshePrice() {
        return 2000;
    }
}
```

```
class RuPorsheImpl implements Porshe {

    public long getPorshePrice() {
        return 20000;
    }
}
```

```
interface InteAbsFactory {

    Lada getLada();

    Ferrari getFerrari();

    Porshe getPorshe();
}
```

```
class UaCarPriceAbsFactory
        implements InteAbsFactory {

    public Lada getLada() {
        return new UaLadaImpl();
    }

    public Ferrari getFerrari() {
        return new UaFerrariImpl();
    }

    public Porshe getPorshe() {
        return new UaPorsheImpl();
```

```
    }
}
```

```
class RuCarPriceAbsFactory
        implements InteAbsFactory {

    public Lada getLada() {
        return new RuLadaImpl();
    }

    public Ferrari getFerrari() {
        return new RuFerrariImpl();
    }

    public Porshe getPorshe() {
        return new RuPorsheImpl();
    }
}
```

Объект интерфейса `InteAbsFactory` позволяет создавать объекты всех автомобилей одной реализации. В нашем случаю формируются цены для России.

```
public class AbstractFactoryTest {

    public static void main(String[] args) {
        String country = "RU";
        InteAbsFactory ifactory = null;
        if (country.equals("UA")) {
            ifactory = new UaCarPriceAbsFactory();
        } else if (country.equals("RU")) {
            ifactory = new RuCarPriceAbsFactory();
        }

        Lada lada = ifactory.getLada();
        System.out.println(lada.getLadaPrice());
    }
}
```

Фабричный метод (factory method) – делегирует создание объектов наследникам родительского класса. Это позволяет

использовать в коде программы не специфические классы, а манипулировать абстрактными объектами на более высоком уровне.

Плюсы:

· избавляет класс от привязки к конкретным классам продуктов;
· выделяет код производства продуктов в одно место, упрощая поддержку кода;
· упрощает добавление новых продуктов в программу;
· реализует принцип открытости/закрытости.

Минус:

· может привести к созданию больших параллельных иерархий классов, т. к. для каждого класса продукта надо создать свой подкласс создателя.

В java — `toString()` (`Object`).

```
1.  interface OS {
2.
3.      void getOS();
4.  }
```

```
1.  class windowsOS implements OS {
2.
3.      public void getOS () {
4.          System.out.println(
5.                  "применить для виндовс");
6.      }
7.  }
```

```
1.  class linuxOS implements OS {
2.
3.      public void getOS () {
4.          System.out.println(
5.                  "применить для линукс");
```

```
    }
}
```

```
class macOS implements OS {

    public void getOS () {
        System.out.println("применить для мак");
    }
}
```

```
class Factory {

    public OS getCurrentOS(String inputos) {
        OS os = null;
        if (inputos.equals("windows")) {
            os = new windowsOS();
        } else if (inputos.equals("linux")) {
            os = new linuxOS();
        } else if (inputos.equals("mac")) {
            os = new macOS();
        }
        return os;
    }
}
```

```
public class FactoryTest {

    public static void main(String[] args){
        String win = "linux";
        Factory factory = new Factory();
        OS os = factory.getCurrentOS(win);
        os.getOS();
    }
}
```

Строитель (builder) — интерфейс для создания сложного объекта (строительство дома, авто).

Плюсы:

- позволяет создавать продукты пошагово;

· позволяет использовать один и тот же код для создания различных продуктов;
· изолируетет сложный код сборки продукта от его основной бизнес-логики.

Минусы:
· усложняет код программы из-за введения дополнительных классов;

- клиент будет привязан к конкретным классам строителей, т. к. в интерфейсе строителя может не быть метода получения результата.

В java — `StringBuilder`.

```
class Car {

    public void buildBase() {
        print("Делаем корпус");
    }

    public void buildWheels() {
        print("Ставим колесо");
    }

    public void buildEngine(Engine engine) {
        print(
                "Ставим движок: "
                    + engine.getEngineType());
    }

    private void print(String msg){
        System.out.println(msg);
    }
}
```

```
interface Engine {

    String getEngineType();
}
```

```
class OneEngine implements Engine {
```

```
2.
3.      public String getEngineType() {
4.          return "Первый двигатель";
5.      }
6.  }
```

```
1.  class TwoEngine implements Engine {
2.
3.      public String getEngineType() {
4.          return "Второй двигатель";
5.      }
6.  }
```

```
1.  abstract class Builder {
2.      protected Car car;
3.
4.      public abstract Car buildCar();
5.  }
```

```
1.  class OneBuilderImpl extends Builder {
2.
3.      public OneBuilderImpl(){
4.          car = new Car();
5.      }
6.
7.      public Car buildCar() {
8.          car.buildBase();
9.          car.buildWheels();
10.         Engine engine = new OneEngine();
11.         car.buildEngine(engine);
12.         return car;
13.     }
14. }
```

```
1.  class TwoBuilderImpl extends Builder {
2.
3.      public TwoBuilderImpl(){
4.          car = new Car();
5.      }
6.
7.      public Car buildCar() {
8.          car.buildBase();
9.          car.buildWheels();
10.         Engine engine = new OneEngine();
11.         car.buildEngine(engine);
12.         car.buildWheels();
```

```
13.             engine = new TwoEngine();
14.             car.buildEngine(engine);
15.             return car;
16.         }
17.     }
```

```
 1.  class Build {
 2.      private Builder builder;
 3.
 4.      public Build(int i) {
 5.          if (i == 1) {
 6.              builder = new OneBuilderImpl();
 7.          } else if (i == 2) {
 8.              builder = new TwoBuilderImpl();
 9.          }
10.      }
11.
12.      public Car buildCar() {
13.          return builder.buildCar();
14.      }
15.  }
```

```
 1.  public class BuilderTest {
 2.
 3.      public static void main(String[] args) {
 4.          Build build = new Build(1);
 5.          build.buildCar();
 6.      }
 7.  }
```

Прототип (prototype) — создание объекта не при помощи конструктора, а при помощи клонирования.

Пример в java — метод `clone()` (`Object`).

```
1.  interface Copyable {
2.
3.      Copyable copy();
4.  }
```

```
 1.  class ComplicatedObject implements Copyable {
 2.      private Type type;
 3.
 4.      public enum Type {
 5.          ONE, TWO
 6.      }
 7.
 8.      public ComplicatedObject copy() {
 9.          ComplicatedObject complicatedobject =
10.                  new ComplicatedObject();
11.          return complicatedobject;
12.      }
13.
14.      public void setType(Type type) {
15.          this.type = type;
16.      }
17.  }
```

```
 1.  public class PrototypeTest {
 2.
 3.      public static void main(String[] args) {
 4.          ComplicatedObject prototype =
 5.                  new ComplicatedObject();
 6.          ComplicatedObject clone =
 7.                  prototype.copy();
 8.          clone.setType(ComplicatedObject.Type.ONE);
 9.      }
10.  }
```

Одиночка (singleton) — класс, который может иметь только один экземпляр.

`getDesktop()` (`java.awt Desktop`).

```
 1.  class Singleton {
 2.      private static Singleton instance = null;
 3.
 4.      private Singleton() {}
 5.
 6.      public static Singleton getInstance() {
 7.          if (instance == null) {
```

```
            instance = new Singleton();
        }
        return instance;
    }

    public void setUp() {
        System.out.println("setUp");
    }
}
```

```
public class SingletonTest {

    public static void main(String[] args){
        Singleton singelton =
                Singleton.getInstance();
        singelton.setUp();
    }
}
```

Примеры структурных шаблонов проектирования

Адаптер (adapter) — объект, который обеспечивает взаимодействие двух несовместимых (с разными интерфейсами) объектов (наследование класса и поле с типом объекта).

`Arrays.asList(array)` — создает `List` из массива; при изменении `List` или массива — изменения будет применяться к обоим контейнерам.

```
class PBank {
    private int balance;

    public PBank() {
        balance = 100;
    }
```

```
 7.
 8.      public void getBalance() {
 9.          System.out.println(
10.                  "PBank balance = " + balance);
11.      }
12.  }
```

```
 1.  class ABank {
 2.      private int balance;
 3.
 4.      public ABank() {
 5.          balance = 200;
 6.      }
 7.
 8.      public void getBalance() {
 9.          System.out.println(
10.                  "ABank balance = " + balance);
11.      }
12.  }
```

```
 1.  class PBankAdapter extends PBank {
 2.      private ABank abank;
 3.
 4.      public PBankAdapter(ABank abank) {
 5.          this.abank = abank;
 6.      }
 7.
 8.      public void getBalance() {
 9.          abank.getBalance();
10.      }
11.  }
```

```
 1.  public class AdapterTest {
 2.
 3.      public static void main(String[] args) {
 4.          PBank pbank = new PBank();
 5.          pbank.getBalance();
 6.          PBankAdapter abank =
 7.                  new PBankAdapter(new ABank());
 8.          abank.getBalance();
 9.      }
10.  }
```

Декоратор (decorator) — класс, расширяющий функциональность другого класса без использования наследования (наследование и объект в конструктор).

`java.io.InputStream`, `OutputStream`, `Reader` и `Writer`.

```
interface Car {

    void draw();
}
```

```
class SportCar implements Car {

    public void draw() {
        System.out.println("SportCar");
    }
}
```

```
class UnknownCar implements Car {

    public void draw() {
        System.out.println("UnknownCar");
    }
}
```

```
abstract class CarDecorator implements Car {
    protected Car decorated;

    public CarDecorator(Car decorated){
        this.decorated = decorated;
    }

    public void draw(){
        decorated.draw();
    }
}
```

```
class BlueCarDecorator extends CarDecorator {

    public BlueCarDecorator(Car decorated) {
        super(decorated);
    }

    public void draw() {
```

```
        decorated.draw();
        setColor();
    }

    private void setColor(){
        System.out.println("Color: red");
    }
}
```

```
public class DecoratorTest {

    public static void main(String[] args) {
        Car sportCar = new SportCar();
        Car blueUnknownCar =
                new BlueCarDecorator(
                    new UnknownCar());
        sportCar.draw();
        System.out.println();
        blueUnknownCar.draw();
    }
}
```

Заместитель (proxy) — объект, который является посредником для другого объекта, т. е. заместитель ограничивает/реализует доступ к объекту.

`javax.persistence.PersistenceContext` над `EntityManager`.

```
interface Image {

    void display();
}
```

```
class RealImage implements Image {
    private String file;

    public RealImage(String file){
        this.file = file;
        load(file);
    }
```

```
 8.
 9.      private void load(String file){
10.          System.out.println("Загрузка " + file);
11.      }
12.
13.      public void display() {
14.          System.out.println("Просмотр " + file);
15.      }
16.  }
```

```
 1.  class ProxyImage implements Image {
 2.      private String file;
 3.      private RealImage image;
 4.
 5.      public ProxyImage(String file){
 6.          this.file = file;
 7.      }
 8.
 9.      public void display() {
10.          if(image == null){
11.              image = new RealImage(file);
12.          }
13.          image.display();
14.      }
15.  }
```

```
 1.  public class ProxyTest {
 2.
 3.      public static void main(String[] args) {
 4.          Image image = new ProxyImage("test.jpg");
 5.          image.display();
 6.          image.display();
 7.      }
 8.  }
```

Адаптер, декоратор и прокси (обобщение)

Если обобщить, то заместитель притворяется оригинальным классом, декоратор расширяет функциональность (умный

заместитель), а адаптер заменяет интерфейс оного из несовместимого объекта.

Также можно добавить, что в Java паттерны адаптер и декоратор существуют из-за отсутствия множественного наследования.

Мост (bridge) — разделяет реализацию и абстракцию (т. е. один класс наследуется от абстрактного, а второй реализует интерфейс), дает возможность изменять их свободно друг от друга. Делает конкретные классы независимыми от классов реализации интерфейса.

В java — `newSetFromMap()` (`Collections`).

```
interface Engine {

    void setEngine();
}
```

```
abstract class Car {
    protected Engine engine;

    public Car(Engine engine){
        this.engine = engine;
    }

    abstract public void setEngine();
}
```

```
class SportCar extends Car {

    public SportCar(Engine engine) {
        super(engine);
    }

    public void setEngine() {
        System.out.print("SportCar engine: ");
        engine.setEngine();
```

```
    }
}
```

```
class UnknownCar extends Car {

    public UnknownCar(Engine engine) {
        super(engine);
    }

    public void setEngine() {
        System.out.print("UnknownCar engine: ");
        engine.setEngine();
    }
}
```

```
class SportEngine implements Engine {

    public void setEngine(){
        System.out.println("sport");
    }
}
```

```
class UnknownEngine implements Engine {

    public void setEngine(){
        System.out.println("unknown");
    }
}
```

```
public class BridgeTest {
    public static void main(String[] args) {
        Car sportCar =
                new SportCar(new SportEngine());
        sportCar.setEngine();
        System.out.println();
        Car unknownCar = new UnknownCar(
                new UnknownEngine());
        unknownCar.setEngine();
    }
}
```

Компоновщик (composite) — объект, который объединяет в себе объекты, подобные ему самому (внутри коллекции).

B java — add(Component) (java.awt.Container).

```
1.  interface Car {
2.
3.      void draw(String color);
4.  }
```

```
1.  class SportCar implements Car {
2.
3.      public void draw(String color) {
4.          System.out.println(
5.                  "SportCar color: " + color);
6.      }
7.  }
```

```
1.  class UnknownCar implements Car {
2.
3.      public void draw(String color) {
4.          System.out.println(
5.                  "UnknownCar color: " + color);
6.      }
7.  }
```

```
1.  class Drawing implements Car {
2.      private List<Car> cars = new ArrayList<Car>();
3.
4.      public void draw(String color) {
5.          for(Car car : cars) {
6.              car.draw(color);
7.          }
8.      }
9.
10.     public void add(Car s){
11.         this.cars.add(s);
12.     }
13.
14.     public void clear(){
15.         System.out.println();
16.         this.cars.clear();
17.     }
18. }
```

```
1.  public class CompositeTest {
2.
```

```
    public static void main(String[] args) {
        Car sportCar = new SportCar();
        Car unknownCar = new UnknownCar();
        Drawing drawing = new Drawing();
        drawing.add(sportCar);
        drawing.add(unknownCar);
        drawing.draw("green");
        drawing.clear();
        drawing.add(sportCar);
        drawing.add(unknownCar);
        drawing.draw("white");
    }
}
```

Фасад (facade) — работа с несколькими классами при помощи одного (использование в искомом классе поля с типами других классов).

В Java — `javax.faces.context.ExternalContext`, который используется внутри `ServletContext`, `HttpSession`, `HttpServletRequest`, `HttpServletResponse` и т. д.

```
interface Car {

    void start();

    void stop();
}
```

```
class Key implements Car {

    public void start() {
        System.out.println("Вставить ключи");
    }

    public void stop() {
        System.out.println("Вытянуть ключи");
    }
}
```

```
 1. class Engine implements Car {
 2.
 3.     public void start() {
 4.         System.out.println("Запустить двигатель");
 5.     }
 6.
 7.     public void stop() {
 8.         System.out.println(
 9.                 "Остановить двигатель");
10.     }
11. }
```

```
 1. class Facade {
 2.     private Key key;
 3.     private Engine engine;
 4.
 5.     public Facade() {
 6.         key = new Key();
 7.         engine = new Engine();
 8.     }
 9.
10.     public void startCar() {
11.         key.start();
12.         engine.start();
13.     }
14.
15.     public void stoptCar() {
16.         key.stop();
17.         engine.stop();
18.     }
19. }
```

```
 1. public class FacadeTest {
 2.
 3.     public static void main(String[] args) {
 4.         Facade facade = new Facade();
 5.         facade.startCar();
 6.         System.out.println();
 7.         facade.stoptCar();
 8.     }
 9. }
```

Приспособленец (flyweight) — вместо создания большого количества похожих объектов, объекты используются повторно (создание объекта внутри другого). Экономит память.

В Java — пул строк, а также метод `valueOf(int)` (`java.lang.Integer`, `Boolean`, `Byte`, `Character`, `Short`, `Long` и `BigDecimal`).

```
 1. class Flyweight {
 2.     private int row;
 3.
 4.     public Flyweight(int row) {
 5.         this.row = row;
 6.         System.out.println("ctor: " + this.row);
 7.     }
 8.
 9.     void report(int col) {
10.         System.out.print(" " + row + col);
11.     }
12. }
```

```
 1. class Factory {
 2.     private Flyweight[] pool;
 3.
 4.     public Factory(int maxRows) {
 5.         pool = new Flyweight[maxRows];
 6.     }
 7.
 8.     public Flyweight getFlyweight(int row) {
 9.         if (pool[row] == null) {
10.             pool[row] = new Flyweight(row);
11.         }
12.         return pool[row];
13.     }
14. }
```

```
 1. public class FlyweightTest {
```

```
 2.
 3.      public static void main(String[] args) {
 4.          int rows = 5;
 5.          Factory theFactory = new Factory(rows);
 6.          for (int i = 0; i < rows; i++) {
 7.              for (int j = 0; j < rows; j++) {
 8.                  theFactory.getFlyweight(i)
 9.                      .report(j);
10.              }
11.              System.out.println();
12.          }
13.      }
14.  }
```

Примеры поведенческих шаблонов проектирования

Цепочка обязанностей (chain of responsibility) — избегание жесткой зависимости отправителя запроса от получателя; запрос может быть обработан несколькими объектами.

Плюсы:

· уменьшает зависимость между клиентом и обработчиками;

· реализует 2 принципа SOLOD: единственной обязанности и открытости/закрытости.

Минусы:

· Запрос может остаться никем не обработанным.

В Java — `log()` (`java.util.logging.Logger`).

```
 1.  interface Payment {
 2.
 3.      void setNext(Payment payment);
```

```
4.
5.      void pay();
6.  }
```

```
1.  class VisaPayment implements Payment {
2.      private Payment payment;
3.
4.      public void setNext(Payment payment) {
5.          this.payment = payment;
6.      }
7.
8.      public void pay() {
9.          System.out.println("Visa Payment");
10.     }
11. }
```

```
1.  class PayPalPayment implements Payment {
2.      private Payment payment;
3.
4.      public void setNext(Payment payment) {
5.          this.payment = payment;
6.      }
7.
8.      public void pay() {
9.          System.out.println("PayPal Payment");
10.     }
11. }
```

```
1.  public static void main(String[] args) {
2.      Payment visaPayment = new VisaPayment();
3.      Payment payPalPayment =
4.              new PayPalPayment();
5.      visaPayment.setNext(payPalPayment);
6.      visaPayment.pay();
7.  }
```

Итератор (iterator) — объект, позволяющий последовательно обходить элементы объекта-агрегата (составного объекта), не раскрывая их внутреннего представления.

Плюсы:

· упрощает классы хранения данных;

· позволяет реализовать различные способы обхода структуры данных;

· позволяет одновременно перемещаться по структуре данных в разные стороны.

Минусы:

· не оправдан, если можно обойтись простым циклом.

В Java — все реализации `java.util.Iterator`.

```
1.  interface Iterator {
2.
3.      boolean hasNext();
4.
5.      Object next();
6.  }
```

```
1.  class Numbers {
2.      public int num[] = {1 , 2, 3};
3.
4.      public Iterator getIterator() {
5.          return new NumbersIterator();
6.      }
7.
8.      private class NumbersIterator
9.              implements Iterator {
10.         int ind;
11.
12.         public boolean hasNext() {
13.             if(ind < num.length) return true;
14.             return false;
15.         }
16.
17.         public Object next() {
18.             if(this.hasNext()) return num[ind++];
```

```
            return null;
        }
    }
}
```

```
public static void main(String[] args) {
    Numbers numbers = new Numbers();
    Iterator iterator = numbers.getIterator();
    while (iterator.hasNext()) {
        System.out.println(iterator.next());
    }
}
```

Шаблонный метод (template method) — определяет основу алгоритма (***1***: 7) и позволяет наследникам переопределять некоторые шаги алгоритма (***2***, ***3***), не изменяя его структуру в целом (***1***).

Плюсы:

· облегчает переиспользование кода.

Минусы:

· ограничены скелетом алгоритма;
· можем нарушить 3 принцип SOLID (подстановки Барбары Лисков);
· с ростом количества шагов сложно поддерживать.

В Java — все не абстрактные методы `java.io.InputStream`, `OutputStream`, `Reader` и `Writer`.

1

```
abstract class Car {

```

```
 3.      abstract void startEngine();
 4.
 5.      abstract void stopEngine();
 6.
 7.      public final void start() {
 8.          startEngine();
 9.          stopEngine();
10.      }
11.  }
```

2, 3

```
 1.  class OneCar extends Car {
 2.
 3.      public void startEngine() {
 4.          System.out.println("Start engine.");
 5.      }
 6.
 7.      public void stopEngine() {
 8.          System.out.println("Stop engine.");
 9.      }
10.  }
```

```
 1.  class TwoCar extends Car {
 2.
 3.      public void startEngine() {
 4.          System.out.println("Start engine.");
 5.      }
 6.
 7.      public void stopEngine() {
 8.          System.out.println("Stop engine.");
 9.      }
10.  }
```

```
 1.  public static void main(String[] args) {
 2.      Car car1 = new OneCar();
 3.      car1.start();
 4.      System.out.println();
 5.      Car car2 = new TwoCar();
 6.      car2.start();
 7.  }
```

Команда (command) — позволяет инкапсулировать различные операции в отдельные объекты.

В Java — все реализации `java.lang.Runnable`.

```
1.  interface Command {
2.
3.      void execute();
4.  }
```

```
1.  class StartCar implements Command {
2.      Car car;
3.
4.      public StartCar(Car car) {
5.          this.car = car;
6.      }
7.
8.      public void execute() {
9.          car.startEngine();
10.     }
11. }
```

```
1.  class StopCar implements Command {
2.      Car car;
3.
4.      public StopCar(Car car) {
5.          this.car = car;
6.      }
7.
8.      public void execute() {
9.          car.stopEngine();
10.     }
11. }
```

```
1.  class Car {
2.
3.      public void startEngine() {
4.          System.out.println("запустить двигатель");
5.      }
6.
7.      public void stopEngine() {
```

```
        System.out.println(
                "остановить двигатель");
    }
}
```

```
class CarInvoker {
    public Command command;

    public CarInvoker(Command command){
        this.command = command;
    }

    public void execute(){
        this.command.execute();
    }
}
```

```
public static void main(String[] args) {
    Car car = new Car();
    StartCar startCar = new StartCar(car);
    StopCar stopCar = new StopCar(car);
    CarInvoker carInvoker =
            new CarInvoker(startCar);
    carInvoker.execute();
}
```

Интерпретатор (interpreter) — решает часто встречающуюся, но подверженную изменениям, задачу.

В Java — все подклассы `java.text.Format`.

```
interface Expression {

    String interpret(Context context);
}
```

```
class LowerExpression implements Expression {
    private String s;

    public LowerExpression(String s) {
        this.s = s;
```

```
    }

    public String interpret(Context context) {
        return context.getLowerCase(s);
    }
}
```

```
class UpperExpression implements Expression {
    private String s;

    public UpperExpression(String s) {
        this.s = s;
    }

    public String interpret(Context context) {
        return context.getUpperCase(s);
    }
}
```

```
class Context {

    public String getLowerCase(String s){
        return s.toLowerCase();
    }

    public String getUpperCase(String s){
        return s.toUpperCase();
    }
}
```

```
public static void main(String[] args) {
    String str = "TesT";
    Context context = new Context();
    Expression lowerExpression =
            new LowerExpression(str);
    str = lowerExpression.interpret(context);
    System.out.println(str);
    Expression upperExpression =
            new UpperExpression(str);
    str = upperExpression.interpret(context);
    System.out.println(str);
}
```

Посредник (mediator) — обеспечивает взаимодействие множества объектов, формируя при этом слабую связанность и избавляя объекты от необходимости явно ссылаться друг на друга.

В java — `execute()` (`java.util.concurrent.Executor`).

```
1.  class Mediator {
2.
3.      public static void sendMessage(
4.              User user,
5.              String msg) {
6.          System.out.println(
7.                  user.getName() + ": " + msg);
8.      }
9.  }
```

```
1.  class User {
2.      private String name;
3.
4.      public User(String name){
5.          this.name = name;
6.      }
7.
8.      public String getName() {
9.          return name;
10.     }
11.
12.     public void sendMessage(String msg){
13.         Mediator.sendMessage(this, msg);
14.     }
15. }
```

```
1.  public static void main(String[] args) {
2.      User user1 = new User("user1");
3.      User user2 = new User("user2");
4.      user1.sendMessage("message1");
```

```
    user2.sendMessage("message2");
}
```

Хранитель (memento) — позволяет, не нарушая инкапсуляцию, зафиксировать и сохранить внутренние состояния объекта так, чтобы позднее восстановить его в этих состояниях.

В Java — все реализации `java.io.Serializable`.

```
class User {
    private String name;
    private int age;

    public User(String name, int age) {
        this.name = name;
        this.age = age;
        System.out.println(String.format(
                "create: name = %s, age = %s",
                name,
                age));
    }

    public Memento save(){
        System.out.println(String.format(
                "save: name = %s, age = %s",
                name,
                age));
        return new Memento(name, age);
    }

    public void restore(Memento memento){
        name = memento.getName();
        age = memento.getAge();
        System.out.println(String.format(
                "restore: name = %s, age = %s",
                name,
                age));
    }
}
```

```
 1. class SaveUser {
 2.     private List<Memento> list =
 3.             new ArrayList<Memento>();
 4.
 5.     public void add(Memento memento){
 6.         list.add(memento);
 7.     }
 8.
 9.     public Memento get(int ind){
10.         return list.get(ind);
11.     }
12. }
```

```
 1. class Memento {
 2.     private String name;
 3.     private int age;
 4.
 5.     public Memento(String name, int age){
 6.         this.name = name;
 7.         this.age = age;
 8.     }
 9.
10.     public String getName() {
11.         return name;
12.     }
13.
14.     public int getAge() {
15.         return age;
16.     }
17. }
```

```
 1. public static void main(String[] args) {
 2.     SaveUser saveUser = new SaveUser();
 3.     User user1 = new User("Peter", 17);
 4.     User user2 = new User("Ian", 19);
 5.     saveUser.add(user1.save());
 6.     user1.restore(saveUser.get(0));
 7. }
```

Наблюдатель (observer) — определяет зависимость типа «один ко многим» между объектами т. о., что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.

В Java — все реализации `java.util.EventListener` (практически во всем Swing).

```
1.  interface Observer {
2.
3.      void event(List<String> strings);
4.  }
```

```
1.  class Director implements Observer {
2.
3.      public void event(List<String> strings) {
4.          System.out.println(
5.                  "The list of students has"
6.                      + " changed: "
7.                  + strings);
8.      }
9.  }
```

```
1.  class University {
2.      private List<Observer> observers =
3.              new ArrayList<Observer>();
4.      private List<String> students =
5.              new ArrayList<String>();
6.
7.      public void addStudent(String name) {
8.          students.add(name);
9.          notifyObservers();
10.     }
11.
12.     public void removeStudent(String name) {
13.         students.remove(name);
14.         notifyObservers();
15.     }
16.
```

```
17.     public void addObserver(Observer observer){
18.         observers.add(observer);
19.     }
20.
21.     public void removeObserver(
22.             Observer observer) {
23.         observers.remove(observer);
24.     }
25.
26.     public void notifyObservers(){
27.         for (Observer observer : observers) {
28.             observer.event(students);
29.         }
30.     }
31. }
```

```
1.  public static void main(String[] args) {
2.      University university = new University();
3.      Director director = new Director();
4.      university.addStudent("Vaska");
5.      university.addObserver(director);
6.      university.addStudent("Anna");
7.      university.removeStudent("Vaska");
8.  }
```

Состояние (state) – используется в тех случаях, когда во время выполнения программы объект должен менять свое поведение в зависимости от своего состояния.

В Java – `execute()` (`javax.faces.lifecycle.LifeCycle`) (контролируется `FacesServlet`, поведение зависит от текущей фазы (состояния) жизненного цикла JSF).

```
1.  interface State {
2.
3.      void doAction();
```

```
4.  }
```

```
1.  class StartPlay implements State {
2.
3.      public void doAction() {
4.          System.out.println("start play");
5.      }
6.  }
```

```
1.  class StopPlay implements State {
2.
3.      public void doAction() {
4.          System.out.println("stop play");
5.      }
6.  }
```

```
1.  class PlayContext implements State {
2.      private State state;
3.
4.      public void setState(State state){
5.          this.state = state;
6.      }
7.
8.      public void doAction() {
9.          this.state.doAction();
10.     }
11. }
```

```
1.  public static void main(String[] args) {
2.      PlayContext playContext =
3.              new PlayContext();
4.      State startPlay = new StartPlay();
5.      State stopPlay = new StopPlay();
6.      playContext.setState(startPlay);
7.      playContext.doAction();
8.      playContext.setState(stopPlay);
9.      playContext.doAction();
10. }
```

Стратегия (strategy) — определяет ряд алгоритмов позволяя взаимодействовать между ними. Алгоритм стратегии может быть изменен во время выполнения программы.

В Java — `compare()` (`java.util.Comparator`), выполненный среди других методов `sort()` (`Collections`).

```
1.  interface Strategy {
2.
3.      void download(String file);
4.  }
```

```
1.  class DownloadWindownsStrategy
2.          implements Strategy {
3.
4.      public void download(String file) {
5.          System.out.println(
6.                  "windows download: " + file);
7.      }
8.  }
```

```
1.  class DownloadLinuxStrategy implements Strategy {
2.
3.      public void download(String file) {
4.          System.out.println(
5.                  "linux download: " + file);
6.      }
7.  }
```

```
1.  class Context {
2.      private Strategy strategy;
3.
4.      public Context(Strategy strategy){
5.          this.strategy = strategy;
6.      }
7.
8.      public void download(String file){
9.          strategy.download(file);
10.     }
11. }
```

```
1.  public static void main(String[] args) {
2.      Context context = new Context(
3.              new DownloadWindownsStrategy());
4.      context.download("file.txt");
```

```
5.      context = new Context(
6.              new DownloadLinuxStrategy());
7.      context.download("file.txt");
8.  }
```

Посетитель (visitor) — описывает операцию, которая выполняется над объектами других классов (при изменении класса Visitor нет необходимости изменять обслуживаемые классы).

В Java — `java.nio.file.FileVisitor` и `SimpleFileVisitor`.

```
1.  interface Car {
2.
3.      void accept(Visitor visitor);
4.  }
```

```
1.  class Engine implements Car {
2.
3.      public void accept(Visitor visitor) {
4.          visitor.visit(this);
5.      }
6.  }
```

```
1.  class Whell implements Car {
2.
3.      public void accept(Visitor visitor) {
4.          visitor.visit(this);
5.      }
6.  }
```

```
1.  class SportCar implements Car {
2.      Car[] cars;
3.
4.      public SportCar(){
5.          cars = new Car[]{
6.                  new Engine(),
7.                  new Whell()};
8.      }
9.
10.     public void accept(Visitor visitor) {
11.         for (int i = 0; i < cars.length; i++) {
12.             cars[i].accept(visitor);
```

```
13.         }
14.         visitor.visit(this);
15.     }
16. }
```

```
 1. interface Visitor {
 2.
 3.     void visit(SportCar sportCar);
 4.
 5.     void visit(Engine engine);
 6.
 7.     void visit(Whell whell);
 8. }
```

```
 1. class CarVisitor implements Visitor {
 2.
 3.     public void visit(SportCar computer) {
 4.         print("car");
 5.     }
 6.
 7.     public void visit(Engine engine) {
 8.         print("engine");
 9.     }
10.
11.     public void visit(Whell whell) {
12.         print("whell");
13.     }
14.
15.     private void print(String string) {
16.         System.out.println(string);
17.     }
18. }
```

```
1. public static void main(String[] args) {
2.     Car computer = new SportCar();
3.     computer.accept(new CarVisitor());
4. }
```

Антипаттерн (anti-pattern)

Подход к решению проблем, являющийся неэффективным, рискованным или непродуктивным.

Полтергейсты (poltergeists) — классы с ограниченной ответственностью и ролью в системе, чье единственное предназначение — передавать информацию в другие классы. Их эффективный жизненный цикл непродолжителен. Полтергейсты нарушают стройность архитектуры программного обеспечения, создавая избыточные (лишние) абстракции, они чрезмерно запутаны, сложны для понимания и трудны в сопровождении. Обычно такие классы задумываются как классы-контроллеры, которые существуют только для вызова методов других классов, зачастую в предопределенной последовательности.

Признаки появления и последствия антипаттерна:

· избыточные межклассовые связи;

· временные ассоциации;

· классы без состояния (содержащие только методы и константы);

· временные объекты и классы (с непродолжительным временем жизни);

· классы с единственным методом, который предназначен только для создания или вызова других классов посредством временной ассоциации;

· классы с именами методов в стиле «управления», такие как startProcess.

Типичные причины:

· отсутствие объектно-ориентированной архитектуры (архитектор не понимает объектно-ориентированной парадигмы);

· неправильный выбор пути решения задачи;

· предположения об архитектуре приложения на этапе анализа требований (до объектно-ориентированного анализа) могут также вести к проблемам на подобии этого антипаттерна.

Внесенная сложность (introduced complexity): необязательная сложность дизайна. Вместо одного простого класса выстраивается целая иерархия интерфейсов и классов. Типичный пример «Интерфейс — Абстрактный класс — Единственный класс реализующий интерфейс на основе абстрактного».

Инверсия абстракции (abstraction inversion): Сокрытие части функциональности

от внешнего использования, в надежде на то, что никто не будет его использовать.

Неопределенная точка зрения (ambiguous viewpoint): Представление модели без спецификации ее точки рассмотрения.

Большой комок грязи (big ball of mud): Система с нераспознаваемой структурой.

Божественный объект (god object): Концентрация слишком большого количества функций в одной части системы (классе).

Затычка на ввод данных (input kludge): Забывчивость в спецификации и выполнении поддержки возможного неверного ввода.

Раздувание интерфейса (interface bloat): Разработка интерфейса очень мощным и очень сложным для реализации.

Волшебная кнопка (magic pushbutton): Выполнение результатов действий пользователя в виде неподходящего (недостаточно абстрактного) интерфейса. Например, написание прикладной логики в обработчиках нажатий на кнопку.

Перестыковка (re-coupling): Процесс внедрения ненужной зависимости.

Дымоход (stovepipe system): Редко поддерживаемая сборка плохо связанных компонентов.

Состояние гонки (race hazard):
непредвидение возможности наступления событий в порядке, отличном от ожидаемого.

Членовредительство (mutilation):
Излишнее «затачивание» объекта под определенную очень узкую задачу таким образом, что он не способен будет работать с никакими иными, пусть и очень схожими задачами.

Сохранение или смерть (save or die):
Сохранение изменений лишь при завершении приложения.

Dependency Injection (внедрение зависимости)

Набор паттернов и принципов разработки программного обеспечения, которые позволяют писать слабосвязный код. В полном соответствии с принципом единой обязанности объект отдает заботу о построении требуемых ему зависимостей внешнему, специально предназначенному для этого общему механизму.

Паттерны в Spring Framework

Singleton — Creating beans with default scope.

Factory — Bean Factory classes

Prototype — Bean scopes

Adapter — Spring Web and Spring MVC

Proxy — Spring Aspect Oriented Programming support

Template Method — JdbcTemplate, HibernateTemplate etc

Front Controller — Spring MVC DispatcherServlet

Data Access Object — Spring DAO support

Dependency Injection and Aspect Oriented Programming

Паттерны в Hibernate

Domain Model — объектная модель предметной области, включающая в себя как поведение, так и данные.

Data Mapper — слой мапперов (Mappers), который передает данные между объектами и БД, сохраняя их независимыми друг от друга и себя.

Proxy — применяется для ленивой загрузки.

Factory — используется в `SessionFactory`.