Частное учреждение образования
Колледж бизнеса и права

УТВЕРЖДАЮ
Заведующий
методическим кабинетом
_____Е.В. Паскал
«___»_____2021

Специальность: 2-40 01 01 «Программное обеспечение информационных технологий»	Дисциплина: «Основы кроссплатформенного программирования»

ЛАБОРАТОРНАЯ РАБОТА № 18

Инструкционно-технологическая карта

Тема: «Паттерны в языке. Порождающие паттерны»

Цель: Научиться работать с порождающими паттернами, в частности с фабрикой и абстрактной фабрикой, научиться их различать.

Время выполнения: 2 часа

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Изучить теоретические сведения;
2. Ответить на контрольные вопросы;
3. Откомпилировать примеры программ из раздела «Теоретические сведения»;
4. Выполнить ИДЗ.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ ДЛЯ ВЫПОЛНЕНИЯ РАБОТЫ

Паттерны проектирования (шаблоны проектирования) - это готовые к использованию решения часто возникающих в программировании задач. Это не класс и не библиотека, которую можно подключить к проекту, это нечто большее. Паттерны проектирования, подходящий под задачу, реализуется в каждом конкретном случае. Следует, помнить, что такой паттерн, будучи примененным неправильно или к неподходящей задаче, может принести немало проблем. Тем не менее, правильно примененный паттерн поможет решить задачу легко и просто.

Типы паттернов:

- Порождающие;
- Структурные;
- Поведенческие;

Порождающие паттерны предоставляют механизмы инициализации, позволяя создавать объекты удобным способом.

Порождающие:

Singleton (Одиночка) - ограничивает создание одного экземпляра класса, обеспечивает доступ к его единственному объекту.

Factory (Фабрика) - используется, когда у нас есть суперкласс с несколькими подклассами и на основе ввода, нам нужно вернуть один из подкласса.

Abstract Factory (Абстрактная фабрика) - используем супер фабрику для создания фабрики, затем используем созданную фабрику для создания объектов.

Пример Паттерна одиночки.

```
class Singleton {
   private static Singleton instance = null;
   private Singleton() {}
   public static Singleton getInstance() {
      if (instance == null) {
         instance = new Singleton();
                  }
      return instance;
   }
   public void setUp() {
      System.out.println("setUp");
   }
}
public class SingletonTest {//тест
   public static void main(String[] args){
      Singleton singelton = Singleton.getInstance();
      singelton.setUp();
   }}
```

Factory (Фабрика).

Шаги реализации

1. Приведите все создаваемые продукты к общему интерфейсу.
2. В классе, который производит продукты, создайте пустой фабричный метод. В качестве возвращаемого типа укажите общий интерфейс продукта.
3. Затем пройдитесь по коду класса и найдите все участки, создающие продукты. Поочерёдно замените эти участки вызовами фабричного метода, перенося в него код создания различных продуктов.
4. В фабричный метод, возможно, придётся добавить несколько параметров, контролирующих, какой из продуктов нужно создать.
5. На этом этапе фабричный метод, скорее всего, будет выглядеть удручающе. В нём будет жить большой условный оператор, выбирающий класс создаваемого продукта. Но не волнуйтесь, мы вот-вот исправим это.
6. Для каждого типа продуктов заведите подкласс и переопределите в нём фабричный метод. Переместите туда код создания соответствующего продукта из суперкласса.
7. Если создаваемых продуктов слишком много для существующих подклассов создателя, вы можете подумать о введении параметров в фабричный метод, которые позволят возвращать различные продукты в пределах одного подкласса.

8. Например, у вас есть класс Почта с подклассами АвиаПочта и НаземнаяПочта, а также классы продуктов Самолёт, Грузовик и Поезд. Авиа соответствует Самолётам, но для НаземнойПочты есть сразу два продукта. Вы могли бы создать новый подкласс почты для поездов, но проблему можно решить и по-другому. Клиентский код может передавать в фабричный метод НаземнойПочты аргумент, контролирующий тип создаваемого продукта.

9. Если после всех перемещений фабричный метод стал пустым, можете сделать его абстрактным. Если в нём что-то осталось — не беда, это будет его реализацией по умолчанию.

Преимущества

- 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.

Недостатки

- Может привести к созданию больших параллельных иерархий классов, так как для каждого класса продукта надо создать свой подкласс создателя.

Пример кода:

```
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;
   }
}
interface OS {
   void getOS();
}
class windowsOS implements OS {
   public void getOS () {
      System.out.println("применить для виндовс");
   }
}
class linuxOS implements OS {
   public void getOS () {
      System.out.println("применить для линукс");
   }
}
class macOS implements OS {
   public void getOS () {
```

```
        System.out.println("применить для мак");
    }
}
public class FactoryTest {//тест
    public static void main(String[] args){
        String win = "linux";
        Factory factory = new Factory();
        OS os = factory.getCurrentOS(win);
        os.getOS();
    }
}
```

Abstract Factory (Абстрактная фабрика).

Преимущества

- Гарантирует сочетаемость создаваемых продуктов.
- Избавляет клиентский код 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.

Недостатки

- Усложняет код программы из-за введения множества дополнительных классов.
- Требует наличия всех типов продуктов в каждой вариации.

Проблема

Представьте, что вы пишете симулятор мебельного магазина. Ваш код содержит:

1. Семейство зависимых продуктов. Скажем, Кресло + Диван + Столик.
2. Несколько вариаций этого семейства. Например, продукты Кресло, Диван и Столик представлены в трёх разных стилях: Ар-деко, Викторианском и Модерне.

Вам нужен такой способ создавать объекты продуктов, чтобы они сочетались с другими продуктами того же семейства. Это важно, так как клиенты расстраиваются, если получают несочетающуюся мебель.

Клиенты расстраиваются, если получают несочетающиеся продукты.

Кроме того, вы не хотите вносить изменения в существующий код при добавлении новых продуктов или семейств в программу. Поставщики часто обновляют свои каталоги, и вы бы не хотели менять уже написанный код каждый раз при получении новых моделей мебели.

Решение

Для начала паттерн Абстрактная фабрика предлагает выделить общие интерфейсы для отдельных продуктов, составляющих семейства. Так, все вариации кресел получат общий интерфейс Кресло, все диваны реализуют интерфейс Диван и так далее.

Все вариации одного и того же объекта должны жить в одной иерархии классов.

Далее вы создаёте абстрактную фабрику — общий интерфейс, который содержит методы создания всех продуктов семейства (например, создатьКресло, создатьДиван и создатьСтолик). Эти операции должны возвращать абстрактные типы продуктов, представленные интерфейсами, которые мы выделили ранее — Кресла, Диваны и Столики.

Конкретные фабрики соответствуют определённой вариации семейства продуктов.

Как насчёт вариаций продуктов? Для каждой вариации семейства продуктов мы должны создать свою собственную фабрику, реализовав абстрактный интерфейс. Фабрики создают продукты одной вариации. Например, ФабрикаМодерн будет возвращать только КреслаМодерн,ДиваныМодерн и СтоликиМодерн.

Клиентский код должен работать как с фабриками, так и с продуктами только через их общие интерфейсы. Это позволит подавать в ваши классы любой тип фабрики и производить любые продукты, ничего не ломая.

Для клиентского кода должно быть безразлично, с какой фабрикой работать.

Например, клиентский код просит фабрику сделать стул. Он не знает, какого типа была эта фабрика. Он не знает, получит викторианский или модерновый стул. Для него важно, чтобы на стуле можно было сидеть и чтобы этот стул отлично смотрелся с диваном той же фабрики.

Осталось прояснить последний момент: кто создаёт объекты конкретных фабрик, если клиентский код работает только с интерфейсами фабрик? Обычно программа создаёт конкретный объект фабрики при запуске, причём тип фабрики выбирается, исходя из параметров окружения или конфигурации.

Пример кода (на машинах):

```
interface Lada {
   long getLadaPrice();
}
interface Ferrari {
   long getFerrariPrice();
}
interface Porshe {
   long getPorshePrice();
}
interface InteAbsFactory {
   Lada getLada();
   Ferrari getFerrari();
   Porshe getPorshe();
}
class UaLadaImpl implements Lada {// первая
   public long getLadaPrice() {
      return 1000;
   }
```

```
}
class UaFerrariImpl implements Ferrari {
   public long getFerrariPrice() {
      return 3000;
   }
}
class UaPorsheImpl implements Porshe {
   public long getPorshePrice() {
      return 2000;
   }
}
class UaCarPriceAbsFactory implements InteAbsFactory {
   public Lada getLada() {
      return new UaLadaImpl();
   }
   public Ferrari getFerrari() {
      return new UaFerrariImpl();
   }
   public Porshe getPorshe() {
      return new UaPorsheImpl();
   }
}// первая
class RuLadaImpl implements Lada {// вторая
   public long getLadaPrice() {
      return 10000;
   }
}
class RuFerrariImpl implements Ferrari {
   public long getFerrariPrice() {
      return 30000;
   }
}
class RuPorsheImpl implements Porshe {
   public long getPorshePrice() {
      return 20000;
   }
}
class RuCarPriceAbsFactory implements InteAbsFactory {
   public Lada getLada() {
      return new RuLadaImpl();
   }
   public Ferrari getFerrari() {
      return new RuFerrariImpl();
   }
   public Porshe getPorshe() {
      return new RuPorsheImpl();
   }
}// вторая

public class AbstractFactoryTest {//тест
   public static void main(String[] args) {
```

```
        String country = "UA";
        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());
    }
```

Структурные паттерны определяют различные сложные структуры, которые изменяют интерфейс уже существующих объектов или его реализацию, позволяя облегчить разработку и оптимизировать программу.

Структурные:

Адаптер — это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.

Фасад — это структурный паттерн проектирования, который предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку.

Декоратор — это структурный паттерн проектирования, который позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».

Адаптер (Adapter)

Проблема: необходимо обеспечить взаимодействие несовместимых интерфейсов или создать единый устойчивый интерфейс для нескольких компонентов с разными интерфейсами.

Решение: преобразовать исходный интерфейс компонента к другому виду с помощью промежуточного объекта - адаптера, то есть, добавить специальный объект с общим интерфейсом в рамках данного приложения и перенаправить связи от внешних объектов к этому объекту - адаптеру.

Класс Adapter приводит интерфейс класса Adaptee в соответствие с интерфейсом класса Target (наследником которого является Adapter). Это позволяет объекту Client использовать объект Adaptee (посредством адаптера Adapter) так, словно он является экземпляром класса Target.

Таким образом Client обращается к интерфейсу Target, реализованному в наследнике Adapter, который перенаправляет обращение к Adaptee. Factory (Фабрика).

Применимость

- Когда вы хотите использовать сторонний класс, но его интерфейс не соответствует остальному коду приложения. Адаптер позволяет создать объект-прокладку, который будет превращать вызовы приложения в формат, понятный стороннему классу.
- Когда вам нужно использовать несколько существующих подклассов, но в них не хватает какой-то общей функциональности, причём расширить суперкласс вы не можете. Вы могли бы создать ещё один уровень подклассов и добавить в них недостающую функциональность.

Но при этом придётся дублировать один и тот же код в обеих ветках подклассов.

Более элегантным решением было бы поместить недостающую функциональность в адаптер и приспособить его для работы с суперклассом. Такой адаптер сможет работать со всеми подклассами иерархии. Это решение будет сильно напоминать паттерн Декоратор.

Шаги реализации

1. Убедитесь, что у вас есть два класса с несовместимыми интерфейсами:
   - полезный сервис — служебный класс, который вы не можете изменять (он либо сторонний, либо от него зависит другой код);
   - один или несколько клиентов — существующих классов приложения, несовместимых с сервисом из-за неудобного или несовпадающего интерфейса.
2. Опишите клиентский интерфейс, через который классы приложения смогли бы использовать класс сервиса.
3. Создайте класс адаптера, реализовав этот интерфейс.
4. Поместите в адаптер поле, которое будет хранить ссылку на объект сервиса. Обычно это поле заполняют объектом, переданным в конструктор адаптера. В случае простой адаптации этот объект можно передавать через параметры методов адаптера.
5. Реализуйте все методы клиентского интерфейса в адаптере. Адаптер должен делегировать основную работу сервису.
6. Приложение должно использовать адаптер только через клиентский интерфейс. Это позволит легко изменять и добавлять адаптеры в будущем.

Преимущества:

Отделяет и скрывает от клиента подробности преобразования различных интерфейсов.

Недостатки:

Усложняет код программы из-за введения дополнительных классов

Помещение квадратных колышков в круглые отверстия. Этот простой пример показывает, как с помощью паттерна Адаптер можно совмещать несовместимые вещи.

round/RoundHole.java: Круглое отверстие

```
package refactoring_guru.adapter.example.round;

/**
 * КруглоеОтверстие совместимо с КруглымиКолышками.
 */
public class RoundHole {
   private double radius;

   public RoundHole(double radius) {
```

```
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }

    public boolean fits(RoundPeg peg) {
        boolean result;
        result = (this.getRadius() >= peg.getRadius());
        return result;
    }
}
```

round/RoundPeg.java: Круглый колышек

```
package refactoring_guru.adapter.example.round;

/**
 * КруглыеКолышки совместимы с КруглымиОтверстиями.
 */
public class RoundPeg {
    private double radius;

    public RoundPeg() {}

    public RoundPeg(double radius) {
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }
}
```

square

square/SquarePeg.java: Квадратный колышек

```
package refactoring_guru.adapter.example.square;

/**
 * КвадратныеКолышки несовместимы с КруглымиОтверстиями (они остались в проекте  * после бывших разработчиков). Но мы должны как-то интегрировать их в нашу
 * систему.
 */
public class SquarePeg {
    private double width;
```

```
    public SquarePeg(double width) {
        this.width = width;
    }

    public double getWidth() {
        return width;
    }

    public double getSquare() {
        double result;
        result = Math.pow(this.width, 2);
        return result;
    }
}
```

adapters

adapters/SquarePegAdapter.java: Адаптер квадратных колышков к круглым отверстиям

```
package refactoring_guru.adapter.example.adapters;

import refactoring_guru.adapter.example.round.RoundPeg;
import refactoring_guru.adapter.example.square.SquarePeg;

/**
 * Адаптер позволяет использовать КвадратныеКолышки и КруглыеОтверстия вместе.
 */
public class SquarePegAdapter extends RoundPeg {
    private SquarePeg peg;

    public SquarePegAdapter(SquarePeg peg) {
        this.peg = peg;
    }

    @Override
    public double getRadius() {
        double result;
        // Рассчитываем минимальный радиус, в который пролезет этот колышек.
        result = (Math.sqrt(Math.pow((peg.getWidth() / 2), 2) * 2));
        return result;
    }
}
```

Demo.java: Клиентский код

```
package refactoring_guru.adapter.example;

import refactoring_guru.adapter.example.adapters.SquarePegAdapter;
import refactoring_guru.adapter.example.round.RoundHole;
import refactoring_guru.adapter.example.round.RoundPeg;
import refactoring_guru.adapter.example.square.SquarePeg;

/**
 * Где-то в клиентском коде...
 */
public class Demo {
    public static void main(String[] args) {
        // Круглое к круглому — всё работает.
        RoundHole hole = new RoundHole(5);
        RoundPeg rpeg = new RoundPeg(5);
        if (hole.fits(rpeg)) {
            System.out.println("Round peg r5 fits round hole r5.");
        }

        SquarePeg smallSqPeg = new SquarePeg(2);
        SquarePeg largeSqPeg = new SquarePeg(20);
        // hole.fits(smallSqPeg); // Не скомпилируется.

        // Адаптер решит проблему.
        SquarePegAdapter smallSqPegAdapter = new SquarePegAdapter(smallSqPeg);
        SquarePegAdapter largeSqPegAdapter = new SquarePegAdapter(largeSqPeg);
        if (hole.fits(smallSqPegAdapter)) {
            System.out.println("Square peg w2 fits round hole r5.");
        }
        if (!hole.fits(largeSqPegAdapter)) {
            System.out.println("Square peg w20 does not fit into round hole r5.");
        }
    }
}
```

Фасад (Facade)

Шаблон “фасад” - структурный шаблон проектирования, позволяющий скрыть сложность системы путем сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы.

Проблема: как обеспечить унифицированный интерфейс с набором разрозненных реализаций или интерфейсов, например, с подсистемой, если

нежелательно высокое связывание с этой подсистемой или реализация подсистемы может измениться?

Решение: определить одну точку взаимодействия с подсистемой — фасадный объект, обеспечивающий общий интерфейс с подсистемой, и возложить на него обязанность по взаимодействию с её компонентами. Фасад — это внешний объект, обеспечивающий единственную точку входа для служб подсистемы. Реализация других компонентов подсистемы закрыта и не видна внешним компонентам.

Применимость

Когда вам нужно представить простой или урезанный интерфейс к сложной подсистеме. Часто подсистемы усложняются по мере развития программы. Применение большинства паттернов приводит к появлению меньших классов, но в большем количестве. Такую подсистему проще повторно использовать, настраивая её каждый раз под конкретные нужды, но вместе с тем, применять подсистему без настройки становится труднее. Фасад предлагает определённый вид системы по умолчанию, устраивающий большинство клиентов.

Когда вы хотите разложить подсистему на отдельные слои. Используйте фасады для определения точек входа на каждый уровень подсистемы. Если подсистемы зависят друг от друга, то зависимость можно упростить, разрешив подсистемам обмениваться информацией только через фасады.

Например, возьмём ту же сложную систему видеоконвертации. Вы хотите разбить её на слои работы с аудио и видео. Для каждой из этих частей можно попытаться создать фасад и заставить классы аудио и видео обработки общаться друг с другом через эти фасады, а не напрямую.

Шаги реализации

1. Определите, можно ли создать более простой интерфейс, чем тот, который предоставляет сложная подсистема. Вы на правильном пути, если этот интерфейс избавит клиента от необходимости знать о подробностях подсистемы.
2. Создайте класс фасада, реализующий этот интерфейс. Он должен переадресовывать вызовы клиента нужным объектам подсистемы. Фасад должен будет позаботиться о том, чтобы правильно инициализировать объекты подсистемы.
3. Вы получите максимум пользы, если клиент будет работать только с фасадом. В этом случае изменения в подсистеме будут затрагивать только код фасада, а клиентский код останется рабочим.
4. Если ответственность фасада начинает размываться, подумайте о введении дополнительных фасадов.

Преимущества:

Изолирует клиентов от компонентов сложной подсистемы.

Недостатки:

Фасад рискует стать божественным объектом, привязанным ко всем классам программы.

Простой интерфейс к сложной библиотеке видеоконвертации. В этом примере Фасад упрощает работу клиента со сложной библиотекой видеоконвертации. Фасад предоставляет пользователю лишь один простой метод, скрывая за собой целую систему с видеокодеками, аудиомикшерами и другими не менее сложными объектами.

some_complex_media_library: Сложная библиотека видеоконвертации
some_complex_media_library/VideoFile.java: Класс видеофайла

```
package refactoring_guru.facade.example.some_complex_media_library;

public class VideoFile {
    private String name;
    private String codecType;

    public VideoFile(String name) {
        this.name = name;
        this.codecType = name.substring(name.indexOf(".") + 1);
    }

    public String getCodecType() {
        return codecType;
    }

    public String getName() {
        return name;
    }
}
```

some_complex_media_library/Codec.java: Интерфейс кодека

```
package refactoring_guru.facade.example.some_complex_media_library;

public interface Codec {
}
```

some_complex_media_library/MPEG4CompressionCodec.java: Кодек MPEG4

```
package refactoring_guru.facade.example.some_complex_media_library;

public class MPEG4CompressionCodec implements Codec {
    public String type = "mp4";

}
```

some_complex_media_library/OggCompressionCodec.java: Кодек Ogg

```
package refactoring_guru.facade.example.some_complex_media_library;

public class OggCompressionCodec implements Codec {
    public String type = "ogg";
```

```
}
```

some_complex_media_library/CodecFactory.java: Фабрика видеокодеков кодеков

```
package refactoring_guru.facade.example.some_complex_media_library;

public class CodecFactory {
    public static Codec extract(VideoFile file) {
        String type = file.getCodecType();
        if (type.equals("mp4")) {
            System.out.println("CodecFactory: extracting mpeg audio...");
            return new MPEG4CompressionCodec();
        }
        else {
            System.out.println("CodecFactory: extracting ogg audio...");
            return new OggCompressionCodec();
        }
    }
}
```

some_complex_media_library/BitrateReader.java: Bitrate-конвертер

```
package refactoring_guru.facade.example.some_complex_media_library;

public class BitrateReader {
    public static VideoFile read(VideoFile file, Codec codec) {
        System.out.println("BitrateReader: reading file...");
        return file;
    }

    public static VideoFile convert(VideoFile buffer, Codec codec) {
        System.out.println("BitrateReader: writing file...");
        return buffer;
    }
}
```

some_complex_media_library/AudioMixer.java: Микширование аудио

```
package refactoring_guru.facade.example.some_complex_media_library;

import java.io.File;

public class AudioMixer {
    public File fix(VideoFile result){
        System.out.println("AudioMixer: fixing audio...");
        return new File("tmp");
    }
}
```

facade

facade/VideoConversionFacade.java: Фасад библиотеки работы с видео

```
package refactoring_guru.facade.example.facade;

import refactoring_guru.facade.example.some_complex_media_library.*;

import java.io.File;

public class VideoConversionFacade {
    public File convertVideo(String fileName, String format) {
        System.out.println("VideoConversionFacade: conversion started.");
        VideoFile file = new VideoFile(fileName);
        Codec sourceCodec = CodecFactory.extract(file);
        Codec destinationCodec;
        if (format.equals("mp4")) {
            destinationCodec = new MPEG4CompressionCodec();
        } else {
            destinationCodec = new OggCompressionCodec();
        }
        VideoFile buffer = BitrateReader.read(file, sourceCodec);
        VideoFile intermediateResult = BitrateReader.convert(buffer,
destinationCodec);
        File result = (new AudioMixer()).fix(intermediateResult);
        System.out.println("VideoConversionFacade: conversion completed.");
        return result;
    }
}
```

Demo.java: Клиентский код

```
package refactoring_guru.facade.example;

import refactoring_guru.facade.example.facade.VideoConversionFacade;

import java.io.File;

public class Demo {
    public static void main(String[] args) {
        VideoConversionFacade converter = new VideoConversionFacade();
        File mp4Video = converter.convertVideo("youtubevideo.ogg", "mp4");
        // ...
    }
}
```

Декоратор (Decorator)

Декоратор — структурный шаблон проектирования, предназначенный для динамического подключения дополнительного поведения к объекту. Шаблон

Декоратор предоставляет гибкую альтернативу практике создания подклассов с целью расширения функциональности.
Задача: объект, который предполагается использовать, выполняет основные функции. Однако может потребоваться добавить к нему некоторую дополнительную функциональность, которая будет выполняться до, после или даже вместо основной функциональности объекта.
Решение: шаблон “декоратор” предусматривает расширение функциональности объекта без определения подклассов.

Класс ConcreteComponent — класс, в который с помощью шаблона Декоратор добавляется новая функциональность. В некоторых случаях базовая функциональность предоставляется классами, производными от класса ConcreteComponent. В подобных случаях класс ConcreteComponent является уже не конкретным, а абстрактным. Абстрактный класс Component определяет интерфейс для использования всех этих классов.
Применимость
Когда вам нужно добавлять обязанности объектам на лету, незаметно для кода, который их использует. Объекты помещают в обёртки, имеющие дополнительные поведения. Обёртки и сами объекты имеют одинаковый интерфейс, поэтому клиентам без разницы, с чем работать — с обычным объектом данных или с обёрнутым.
Когда нельзя расширить обязанности объекта с помощью наследования. Во многих языках программирования есть ключевое слово final, которое может заблокировать наследование класса. Расширить такие классы можно только с помощью Декоратора.
Шаги реализации

1. Убедитесь, что в вашей задаче есть один основной компонент и несколько опциональных дополнений или надстроек над ним.
2. Создайте интерфейс компонента, который описывал бы общие методы как для основного компонента, так и для его дополнений.
3. Создайте класс конкретного компонента и поместите в него основную бизнес-логику.
4. Создайте базовый класс декораторов. Он должен иметь поле для хранения ссылки на вложенный объект-компонент. Все методы базового декоратора должны делегировать действие вложенному объекту.
5. И конкретный компонент, и базовый декоратор должны следовать одному и тому же интерфейсу компонента.
6. Теперь создайте классы конкретных декораторов, наследуя их от базового декоратора. Конкретный декоратор должен выполнять свою добавочную функцию, а затем (или перед этим) вызывать эту же операцию обёрнутого объекта.
7. Клиент берёт на себя ответственность за конфигурацию и порядок обёртывания объектов.

Преимущества:

- Большая гибкость, чем у наследования.
- Позволяет добавлять обязанности на лету.
- Можно добавлять несколько новых обязанностей сразу.
- Позволяет иметь несколько мелких объектов вместо одного объекта на все случаи жизни.

Недостатки:

- Трудно конфигурировать многократно обёрнутые объекты.
- Обилие крошечных классов.

Шифрование и сжатие данных. Пример показывает, как можно добавить новую функциональность объекту, не меняя его класса. Сначала класс бизнес-логики мог считывать и записывать только чистые данные напрямую из/в файлы. Применив паттерн Декоратор, мы создали небольшие классы-обёртки, которые добавляют новые поведения до или после основной работы вложенного объекта бизнес-логики. Первая обёртка шифрует и расшифрует данные, а вторая — сжимает и распакует их. Мы можем использовать обёртки как отдельно друг от друга, так и все вместе, обернув один декоратор другим.

decorators/DataSource.java: Интерфейс, задающий базовые операции чтения и записи данных

```
package refactoring_guru.decorator.example.decorators;

public interface DataSource {
   void writeData(String data);
   String readData();
}
```

decorators/FileDataSource.java:

Класс, реализующий прямое чтение и запись данных

```
package refactoring_guru.decorator.example.decorators;
import java.io.*;
public class FileDataSource implements DataSource {
   private String name;

   public FileDataSource(String name) {
      this.name = name;
   }
   @Override
   public void writeData(String data) {
      File file = new File(name);
      try (OutputStream fos = new FileOutputStream(file)) {
         fos.write(data.getBytes(), 0, data.length());
      } catch (IOException ex) {
         System.out.println(ex.getMessage());
      }
```

```
    }

    @Override
    public String readData() {
        char[] buffer = null;
        File file = new File(name);
        try (FileReader reader = new FileReader(file)) {
            buffer = new char[(int) file.length()];
            reader.read(buffer);
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        }
        return new String(buffer);
    }
}
```

decorators/DataSourceDecorator.java: Базовый декоратор

```
package refactoring_guru.decorator.example.decorators;
public class DataSourceDecorator implements DataSource {
    private DataSource wrappee;
    DataSourceDecorator(DataSource source) {
        this.wrappee = source;
    }
    @Override
    public void writeData(String data) {
        wrappee.writeData(data);
    }
    @Override
    public String readData() {
        return wrappee.readData();
    }
}
```

decorators/EncryptionDecorator.java: Декоратор шифрования

```
package refactoring_guru.decorator.example.decorators;
import java.util.Base64;
public class EncryptionDecorator extends DataSourceDecorator {
    public EncryptionDecorator(DataSource source) {
        super(source);
    }
    @Override
    public void writeData(String data) {
        super.writeData(encode(data));
    }
    @Override
    public String readData() {
```

```
        return decode(super.readData());
    }
    private String encode(String data) {
        byte[] result = data.getBytes();
        for (int i = 0; i < result.length; i++) {
            result[i] += (byte) 1;
        }
        return Base64.getEncoder().encodeToString(result);
    }
    private String decode(String data) {
        byte[] result = Base64.getDecoder().decode(data);
        for (int i = 0; i < result.length; i++) {
            result[i] -= (byte) 1;
        }
        return new String(result);
    }
}
```

decorators/CompressionDecorator.java: Декоратор сжатия

```
package refactoring_guru.decorator.example.decorators;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.Base64;
import java.util.zip.Deflater;
import java.util.zip.DeflaterOutputStream;
import java.util.zip.InflaterInputStream;

public class CompressionDecorator extends DataSourceDecorator {
    private int compLevel = 6;
    public CompressionDecorator(DataSource source) {
        super(source);
    }
    public int getCompressionLevel() {
        return compLevel;
    }
    public void setCompressionLevel(int value) {
        compLevel = value;
    }
    @Override
    public void writeData(String data) {
        super.writeData(compress(data));
    }
    @Override
```

```
    public String readData() {
        return decompress(super.readData());
    }
    private String compress(String stringData) {
        byte[] data = stringData.getBytes();
        try {
            ByteArrayOutputStream bout = new ByteArrayOutputStream(512);
            DeflaterOutputStream dos = new DeflaterOutputStream(bout, new
Deflater(compLevel));
            dos.write(data);
            dos.close();
            bout.close();
            return Base64.getEncoder().encodeToString(bout.toByteArray());
        } catch (IOException ex) {
            return null;
        }
    }
    private String decompress(String stringData) {
        byte[] data = Base64.getDecoder().decode(stringData);
        try {
            InputStream in = new ByteArrayInputStream(data);
            InflaterInputStream iin = new InflaterInputStream(in);
            ByteArrayOutputStream bout = new ByteArrayOutputStream(512);
            int b;
            while ((b = iin.read()) != -1) {
                bout.write(b);
            }
            in.close();
            iin.close();
            bout.close();
            return new String(bout.toByteArray());
        } catch (IOException ex) {
            return null;
        }
    }
}
```

Demo.java: Клиентский код

```
package refactoring_guru.decorator.example;
import refactoring_guru.decorator.example.decorators.*;
public class Demo {
    public static void main(String[] args) {
        String salaryRecords = "Name,Salary\nJohn Smith,100000\nSteven
Jobs,912000";
        DataSourceDecorator encoded = new CompressionDecorator(
```

```
                        new EncryptionDecorator(
                            new FileDataSource("out/OutputDemo.txt")));
        encoded.writeData(salaryRecords);
        DataSource plain = new FileDataSource("out/OutputDemo.txt");

        System.out.println("- Input ----------------");
        System.out.println(salaryRecords);
        System.out.println("- Encoded --------------");
        System.out.println(plain.readData());
        System.out.println("- Decoded --------------");
        System.out.println(encoded.readData());
    }
}
```

Поведенческие паттерны определяют взаимодействие между объектами, увеличивая таким образом его гибкость.

Поведенческие:

Наблюдатель — это поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.

Посредник — это поведенческий паттерн проектирования, который позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.

Наблюдатель

Проблема

Представьте, что вы имеете два объекта: Покупатель и Магазин. В магазин вот-вот должны завезти новый товар, который интересен покупателю. Покупатель может каждый день ходить в магазин, чтобы проверить наличие товара. Но при этом он будет злиться, без толку тратя своё драгоценное время.

С другой стороны, магазин может разослать спам каждому своему покупателю. Многих это расстроит, так как товар специфический, и не всем он нужен.

Получается конфликт: либо покупатель тратит время на периодические проверки, либо магазин тратит ресурсы на бесполезные оповещения.

Решение

Давайте называть Издателями те объекты, которые содержат важное или интересное для других состояние. Остальные объекты, которые хотят отслеживать изменения этого состояния, назовём Подписчиками.

Паттерн Наблюдатель предлагает хранить внутри объекта издателя список ссылок на объекты подписчиков, причём издатель не должен вести список подписки самостоятельно. Он предоставит методы, с помощью которых подписчики могли бы добавлять или убирать себя из списка.

Теперь самое интересное. Когда в издателе будет происходить важное событие, он будет проходиться по списку подписчиков и оповещать их об этом, вызывая определённый метод объектов-подписчиков.

Издателю безразлично, какой класс будет иметь тот или иной подписчик, так как все они должны следовать общему интерфейсу и иметь единый метод оповещения.

Увидев, как складно всё работает, вы можете выделить общий интерфейс, описывающий методы подписки и отписки, и для всех издателей. После этого подписчики смогут работать с разными типами издателей, а также получать оповещения от них через один и тот же метод.

Применимость

Когда после изменения состояния одного объекта требуется что-то сделать в других, но вы не знаете наперёд, какие именно объекты должны отреагировать. Описанная проблема может возникнуть при разработке библиотек пользовательского интерфейса, когда вам надо дать возможность сторонним классам реагировать на клики по кнопкам.

Паттерн Наблюдатель позволяет любому объекту с интерфейсом подписчика зарегистрироваться на получение оповещений о событиях, происходящих в объектах-издателях.

Когда одни объекты должны наблюдать за другими, но только в определённых случаях. Издатели ведут динамические списки. Все наблюдатели могут подписываться или отписываться от получения оповещений прямо во время выполнения программы.

Шаги реализации

1. Разбейте вашу функциональность на две части: независимое ядро и опциональные зависимые части. Независимое ядро станет издателем. Зависимые части станут подписчиками.
2. Создайте интерфейс подписчиков. Обычно в нём достаточно определить единственный метод оповещения.
3. Создайте интерфейс издателей и опишите в нём операции управления подпиской. Помните, что издатель должен работать только с общим интерфейсом подписчиков.
4. Вам нужно решить, куда поместить код ведения подписки, ведь он обычно бывает одинаков для всех типов издателей. Самый очевидный способ — вынести этот код в промежуточный абстрактный класс, от которого будут наследоваться все издатели.

Но если вы интегрируете паттерн в существующие классы, то создать новый базовый класс может быть затруднительно. В этом случае вы можете поместить логику подписки во вспомогательный объект и делегировать ему работу из издателей.

5. Создайте классы конкретных издателей. Реализуйте их так, чтобы после каждого изменения состояния они отправляли оповещения всем своим подписчикам.

6. Реализуйте метод оповещения в конкретных подписчиках. Не забудьте предусмотреть параметры, через которые издатель мог бы отправлять какие-то данные, связанные с происшедшим событием.

Возможен и другой вариант, когда подписчик, получив оповещение, сам возьмёт из объекта издателя нужные данные. Но в этом случае вы будете вынуждены привязать класс подписчика к конкретному классу издателя.

7. Клиент должен создавать необходимое количество объектов подписчиков и подписывать их у издателей.

Преимущества:

- Издатели не зависят от конкретных классов подписчиков и наоборот.
- Вы можете подписывать и отписывать получателей на лету.
- Реализует принцип открытости/закрытости.

Недостатки:

- Подписчики оповещаются в случайном порядке.

Подписка и оповещения. В этом примере Наблюдатель используется для передачи событий между объектами текстового редактора. Всякий раз когда объект редактора меняет своё состояние, он оповещает своих наблюдателей. Объекты EmailNotificationListener и LogOpenListener следят за этими уведомлениями и выполняют полезную работу в ответ.

Классы подписчиков не связаны с классом редактора и могут быть повторно использованы в других приложениях если потребуется. Класс Editor зависит только от общего интерфейса подписчиков. Это позволяет добавлять новые типы подписчиков не меняя существующего кода редактора.

Посредник

Проблема

Предположим, что у вас есть диалог создания профиля пользователя. Он состоит из всевозможных элементов управления — текстовых полей, чекбоксов, кнопок. Беспорядочные связи между элементами пользовательского интерфейса. Беспорядочные связи между элементами пользовательского интерфейса.

Отдельные элементы диалога должны взаимодействовать друг с другом. Так, например, чекбокс «у меня есть собака» открывает скрытое поле для ввода имени домашнего любимца, а клик по кнопке отправки запускает проверку значений всех полей формы.

Прописав эту логику прямо в коде элементов управления, вы поставите крест на их повторном использовании в других местах приложения. Они станут слишком тесно связанными с элементами диалога редактирования профиля, которые не нужны в других контекстах. Поэтому вы сможете использовать либо все элементы сразу, либо ни одного.

Решение

Паттерн Посредник заставляет объекты общаться не напрямую друг с другом, а через отдельный объект-посредник, который знает, кому нужно

перенаправить тот или иной запрос. Благодаря этому, компоненты системы будут зависеть только от посредника, а не от десятков других компонентов.
В нашем примере посредником мог бы стать диалог. Скорее всего, класс диалога и так знает, из каких элементов состоит, поэтому никаких новых связей добавлять в него не придётся.
Основные изменения произойдут внутри отдельных элементов диалога. Если раньше при получении клика от пользователя объект кнопки сам проверял значения полей диалога, то теперь его единственной обязанностью будет сообщить диалогу о том, что произошёл клик. Получив извещение, диалог выполнит все необходимые проверки полей. Таким образом, вместо нескольких зависимостей от остальных элементов кнопка получит только одну — от самого диалога.
Чтобы сделать код ещё более гибким, можно выделить общий интерфейс для всех посредников, то есть диалогов программы. Наша кнопка станет зависимой не от конкретного диалога создания пользователя, а от абстрактного, что позволит использовать её и в других диалогах.
Таким образом, посредник скрывает в себе все сложные связи и зависимости между классами отдельных компонентов программы. А чем меньше связей имеют классы, тем проще их изменять, расширять и повторно использовать.
Применимость
Когда вам сложно менять некоторые классы из-за того, что они имеют множество хаотичных связей с другими классами. Посредник позволяет поместить все эти связи в один класс, после чего вам будет легче их отрефакторить, сделать более понятными и гибкими.
Когда вы не можете повторно использовать класс, поскольку он зависит от уймы других классов. После применения паттерна компоненты теряют прежние связи с другими компонентами, а всё их общение происходит косвенно, через объект-посредник.
Когда вам приходится создавать множество подклассов компонентов, чтобы использовать одни и те же компоненты в разных контекстах. Если раньше изменение отношений в одном компоненте могли повлечь за собой лавину изменений во всех остальных компонентах, то теперь вам достаточно создать подкласс посредника и поменять в нём связи между компонентами.
Шаги реализации

1. Найдите группу тесно переплетённых классов, отвязав которые друг от друга, можно получить некоторую пользу. Например, чтобы повторно использовать их код в другой программе.
2. Создайте общий интерфейс посредников и опишите в нём методы для взаимодействия с компонентами. В простейшем случае достаточно одного метода для получения оповещений от компонентов.

Этот интерфейс необходим, если вы хотите повторно использовать классы компонентов для других задач. В этом случае всё, что нужно сделать — это создать новый класс конкретного посредника.

3. Реализуйте этот интерфейс в классе конкретного посредника. Поместите в него поля, которые будут содержать ссылки на все объекты компонентов.
4. Вы можете пойти дальше и переместить код создания компонентов в класс посредника, после чего он может напоминать фабрику или фасад.
5. Компоненты тоже должны иметь ссылку на объект посредника. Связь между ними удобнее всего установить, подавая посредника в параметры конструктора компонентов.
6. Измените код компонентов так, чтобы они вызывали метод оповещения посредника, вместо методов других компонентов. С противоположной стороны, посредник должен вызывать методы нужного компонента, когда получает оповещение от компонента.

Преимущества:

- Устраняет зависимости между компонентами, позволяя повторно их использовать.
- Упрощает взаимодействие между компонентами.
- Централизует управление в одном месте.

Недостатки:

- Посредник может сильно раздуться.

Этот пример показывает, как организовать множество элементов интерфейса при помощи посредника так, чтобы они не знали и не зависели друг от друга.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое паттерн?
2. Что такое паттерн одиночка?
3. Когда удобно использовать одиночку?
4. Что такое фабричный паттерн?
5. Когда стоит использовать фабричный паттерн?
6. Преимущества и недостатки фабричного паттерна.
7. Что такое абстрактная фабрика?
8. Различия фабрики и абстрактной фабрики.
9. Преимущества и недостатки абстрактной фабрики.
10. Что такое паттерн адаптер?
11. Когда удобно использовать адаптер?
12. Что такое паттерн фасад?
13. Когда стоит использовать фасад?
14. Что такое декоратор?
15. Когда стоит использовать декоратор?
16. Преимущества и недостатки адаптера, фасада и декоратора.
17. Что такое паттерн наблюдатель?
18. Когда удобно использовать наблюдатель?
19. Что такое паттерн посредник?

20. Когда стоит использовать посредник?
21. Преимущества и недостатки наблюдателя и посредника.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Задание 1.

Реализовать абстрактную фабрику (тема по вариантам). Вывод интересующей пользователя информации в окне.

Варианты индивидуального задания.

1. Имеется фабрика по производству сыра, молока и сметаны из коровьего, козьего и птичьего молока. Пользователь выбирает нужную ему категорию.

2. Имеется фабрика по производству телефонов, планшетов и компьютеров трех различных торговых марок. Пользователь выбирает нужную ему категорию.

3. Имеется фабрика по производству кистей для художников, маляров и визажистов из конского, лосиного и собачьего волоса. Пользователь выбирает нужную ему категорию.

4. У полководца имеется армия пешая, конная и танковая состоящая из новобранцев, туземцев и профессионалов. Полководец выбирает кто участвует в сегодняшней битве.

Задание 2.

Варианты индивидуального задания.

Варианты индивидуального задания.

1. Разработайте фасад, скрывающий сложную логику системы заказа и оплаты
2. Создайте декоратор, добавляющий следующее поведение системе банковской бухгалтерии: когда клиент снимает со счета более определенной суммы наличными, ему посылается текстовое SMS, оповещающее о снятии

Задание 3.

Вывод интересующей пользователя информации в окне. Реализовать: нечетный вариант паттерн наблюдатель, четный вариант – посредник.

Разработать систему Почтовое отделение. Из издательства в почтовое отделение поступают издаваемые газеты и журналы. Почтовое отделение отправляет полученные печатные издания соответствующим подписчикам. Имеется фабрика по производству телефонов, планшетов и компьютеров трех различных торговых марок. Пользователь выбирает нужную ему категорию.

ДОМАШНЕЕ ЗАДАНИЕ

Индивидуальное задание

ЛИТЕРАТУРА

Альфред В., Ахо Компиляторы. Принципы, технологии и инструментарий, Вильямс, 2017.

Преподаватель А.С.Кибисова

Рассмотрено на заседании цикловой комиссии программного обеспечения информационных технологий
Протокол № ____ от «___» _____ 2021
Председатель ЦК _____В.Ю.Михалевич