Практическая работа № 24. Паттерны проектирования. порождающие паттерны: абстрактная фабрика, фабричный метод

Цель: научиться применять порождающие паттерны при разработке программ на Java. В данной практической работе рекомендуется использовать следующие паттерны: Абстрактная фабрика и фабричный метод.

Теоретические сведения. Понятие паттерна.

Паттерны (или шаблоны) проектирования описывают типичные способы решения часто встречающихся проблем при проектировании программ.

Некоторые из преимуществ использования шаблонов проектирования:

1) Шаблоны проектирования уже заранее определены и обеспечивают стандартный отраслевой подход к решению повторяющихся в программном коде проблем, вследствие этого разумное применение шаблона проектирования экономит время на разработку.
2) Использование шаблонов проектирования способствует реализации одного из преимуществ ООП – повторного использования кода, что приводит к более надежному и удобному в сопровождении коду. Это помогает снизить общую стоимость владения программного продукта.
3) Поскольку шаблоны проектирования заранее определены то это упрощает понимание и отладку нашего кода. Это приводит к более быстрому развитию проектов, так как новые новые члены команды понимают код.

Шаблоны проектирования Джава делятся на три категории: порождающие, структурные и поведенческие шаблоны проектирования. Так же есть еще шаблоны проектировани, например MVC – model-view-controller.

Шаблон проектирования Фабрика (factory), его также называют фабричный метод используется, когда у нас есть суперкласс с несколькими подклассами, и на основе ввода нам нужно вернуть один из подклассов.

Этот шаблон снимает с себя ответственность за создание экземпляра класса из клиентской программы в класс фабрики. Давайте сначала узнаем, как реализовать фабричный шаблон проектирования в java, а затем мы рассмотрим преимущества фабричного шаблона. Мы увидим некоторые примеры использования фабричного шаблона проектирования в JDK. Обратите внимание, что этот шаблон также известен как шаблон проектирования фабричный метод.

Суперкласс в шаблоне проектирования Фабрика может быть интерфейсом, абстрактным классом или обычным классом Java. Для нашего примера шаблона проектирования фабрики у нас есть абстрактный суперкласс с переопределенным toString() методом для целей тестирования см. листинг 24.1

Листинг 24.1 Пример абстрактного класса Computer.java

```
package ru.mirea.it;

public abstract class Computer {

    public abstract String getRAM();
    public abstract String getHDD();
    public abstract String getCPU();

    @Override
    public String toString(){
        return "RAM= "+this.getRAM()+",
HDD="+this.getHDD()+", CPU="+this.getCPU();
    }
}
```

Создадим производные классы шаблона фабрики. Допустим, у нас есть два подкласса ПК и сервер с реализацией ниже см листинг 24.2.

Листинг 24.2 –Дочерний класс PC фабричного шаблона.

```
package ru.mirea.it;

public class PC extends Computer {

    private String ram;
    private String hdd;
    private String cpu;

    public PC(String ram, String hdd, String cpu){
```

```
        this.ram=ram;
        this.hdd=hdd;
        this.cpu=cpu;
    }
    @Override
    public String getRAM() {
        return this.ram;
    }

    @Override
    public String getHDD() {
        return this.hdd;
    }

    @Override
    public String getCPU() {
        return this.cpu;
    }

}
```

На листинге 24.3 представлен еще один дочерний класс шаблона фабрика.

Листинг 24.3 –Дочерний класс Server фабричного шаблона.

```
package ru.mirea.it
public class Server extends Computer {

    private String ram;
    private String hdd;
    private String cpu;

    public Server(String ram, String hdd, String
cpu){
        this.ram=ram;
        this.hdd=hdd;
        this.cpu=cpu;
    }
```

```
    @Override
    public String getRAM() {
        return this.ram;
    }

    @Override
    public String getHDD() {
        return this.hdd;
    }

    @Override
    public String getCPU() {
        return this.cpu;
    }

}
```

Теперь, когда у нас есть готовые суперклассы и подклассы, мы можем написать наш фабричный класс. На листинге 24.4 представлена его базовая реализация.

Листинг 24.4 Базовая реализация фабричного класса

```
package ru.mirea.it;

import ru.mirea.it.Computer;
import ru.mirea.it.PC;
import ru.mirea.it.Server;

public class ComputerFactory {

    public static Computer getComputer(String type,
String ram, String hdd, String cpu){
        if("PC".equalsIgnoreCase(type)) return new
PC(ram, hdd, cpu);
        else if("Server".equalsIgnoreCase(type))
return new Server(ram, hdd, cpu);

        return null;
```

```
    }
}
```

Отметим некоторые важные моменты метода Factory Design Pattern;

Мы можем оставить класс Factory Singleton или оставить метод, возвращающий подкласс, как статический .

Обратите внимание, что на основе входного параметра создается и возвращается другой подкласс. getComputer() является заводским методом.

Простая тестовая клиентская программа, в которой используется приведенная выше реализация шаблона проектирования factory представлена на листинге 24.5.

Листинг 24.5 Класс Тестер

```
package ru.mirea.it.abstractfactory;
import ru.mirea.it.ComputerFactory;
import ru.mirea.it.Computer;

public class TestFactory {

    public static void main(String[] args) {
        Computer pc =
ComputerFactory.getComputer("pc","2 GB","500 GB","2.4
GHz");
        Computer server =
ComputerFactory.getComputer("server","16 GB","1
TB","2.9 GHz");
        System.out.println("Factory PC Config::"+pc);
        System.out.println("Factory Server
Config::"+server);
    }

}
```

Результат работы тестовой программы:

```
    Factory PC Config::RAM= 2 GB, HDD=500 GB, CPU=2.4
GHz
    Factory Server Config::RAM= 16 GB, HDD=1 TB,
CPU=2.9 GHz
```

Преимущества шаблона Фабрика

Шаблон проектирования Factory обеспечивает подход к коду для интерфейса, а не для реализации.

Фабричный шаблон удаляет экземпляры реальных классов реализации из клиентского кода. Фабричный шаблон делает наш код более надежным, менее связанным и легко расширяемым. Например, мы можем легко изменить реализацию класса ПК (PC), потому что клиентская программа не знает об этом.

Основное преимущество, котрое мы получаем - Фабричный шаблон обеспечивает абстракцию между реализацией и клиентскими классами посредством наследования.

Примеры шаблонов проектирования Factory в JDK

Методы java.util.Calendar, ResourceBundle и NumberFormat getInstance() используют шаблон Factory. Методы valueOf()метод в классах-оболочках, таких как Boolean, Integer и т. д. также используют шаблон Factory

Паттерн Abstract Factory похож на паттерн Factory и представляет собой фабрику фабрик. Если вы знакомы с шаблоном проектирования factory в java, вы заметите, что у нас есть один класс Factory, который возвращает различные подклассы на основе предоставленных входных данных, и для достижения этого класс factory использует операторы if-else или switch. В шаблоне абстрактной фабрики мы избавляемся от блока if-else и создаем класс фабрики для каждого подкласса, а затем класс абстрактной фабрики, который возвращает подкласс на основе входного фабричного класса.

Мы рассмотрели на листингах 24.1, 24.2, 24.3 классы, теперь прежде всего нам нужно создать интерфейс Abstract Factory или абстрактный класс .ComputerAbstractFactory.java см. листинг 24.6

Листинг 24.6 Абстрактный класс .ComputerAbstractFactory.java

```
package ru.mirea.it.abstractfactory;

import ru.mirea.it.Computer;

public interface ComputerAbstractFactory {

    public Computer createComputer();

}
```

братите внимание, что метод createComputer() возвращает экземпляр суперкласса Computer. Теперь наши фабричные классы будут реализовывать этот интерфейс и возвращать соответствующий подкласс PCFactory.java см.листинг 24.7

Листинг 24.7 Фабричный класс PCFactory.java

```
package ru.mirea.it.abstractfactory;

import ru.mirea.it.Computer;
import ru.mirea.it.PC;

public class PCFactory implements
ComputerAbstractFactory {

    private String ram;
    private String hdd;
    private String cpu;

    public PCFactory(String ram, String hdd, String
cpu){
        this.ram=ram;
        this.hdd=hdd;
        this.cpu=cpu;
    }
    @Override
    public Computer createComputer() {
        return new PC(ram,hdd,cpu);
    }

}
```

точно так же у нас будет фабричный класс для Server подкласса ServerFactory.java см листинг 24.8

Листинг 24.8 Фабричный класс ServerFactory.java

```
package ru.mirea.it.abstractfactory;

import ru.mirea.it.Computer;
import ru.mirea.it.Server;
```

```
public class ServerFactory implements
ComputerAbstractFactory {

    private String ram;
    private String hdd;
    private String cpu;

    public ServerFactory(String ram, String hdd,
String cpu){
        this.ram=ram;
        this.hdd=hdd;
        this.cpu=cpu;
    }

    @Override
    public Computer createComputer() {
        return new Server(ram,hdd,cpu);
    }

}
```

Теперь мы создадим клиентский класс, который предоставит клиентским классам точку входа для создания подклассов ComputerFactory.java см листинг 24.9

Листинг 24.9 Класс ComputerFactory

```
package ru.mirea.it.abstractfactory;

import ru.mirea.it.Computer;

public class ComputerFactory {

    public static Computer
getComputer(ComputerAbstractFactory factory){
        return factory.createComputer();
    }
}
```

Обратите внимание, что это простой класс и метод getComputer, принимающий аргумент ComputerAbstractFactory и возвращающий объект Computer. На этом этапе реализацияуже должна быть понятной. Давайте напишем простой тестовый метод и посмотрим, как использовать абстрактную фабрику для получения экземпляров подклассов TestDesignPatterns.java см листинг 24.10.

Листинг 24.10 Тестовый класс TestDesignPatterns.java

```
Package  ru.mirea.it.design.test;

import ru.mirea.it.abstractfactory.PCFactory;
import
ru.mirea.it.design.abstractfactory.ServerFactory;
import ru.mirea.it..design.factory.ComputerFactory;
import ru.mirea.it.design.model.Computer;

public class TestDesignPatterns {

    public static void main(String[] args) {
        testAbstractFactory();
    }

    private static void testAbstractFactory() {
        Computer pc =
com.journaldev.design.abstractfactory.ComputerFactory
.getComputer(new PCFactory("2 GB","500 GB","2.4
GHz"));
        Computer server =
com.journaldev.design.abstractfactory.ComputerFactory
.getComputer(new ServerFactory("16 GB","1 TB","2.9
GHz"));
        System.out.println("AbstractFactory PC
Config::"+pc);
        System.out.println("AbstractFactory Server
Config::"+server);
    }
}
```

Результат работы тестовой программы

```
    AbstractFactory PC Config::RAM= 2 GB, HDD=500 GB,
CPU=2.4 GHz
    AbstractFactory Server Config::RAM= 16 GB, HDD=1
TB, CPU=2.9 GHz
```

Ниже представлена UML диаграмма классов с использованием шаблона Фабрика см рис 21.1

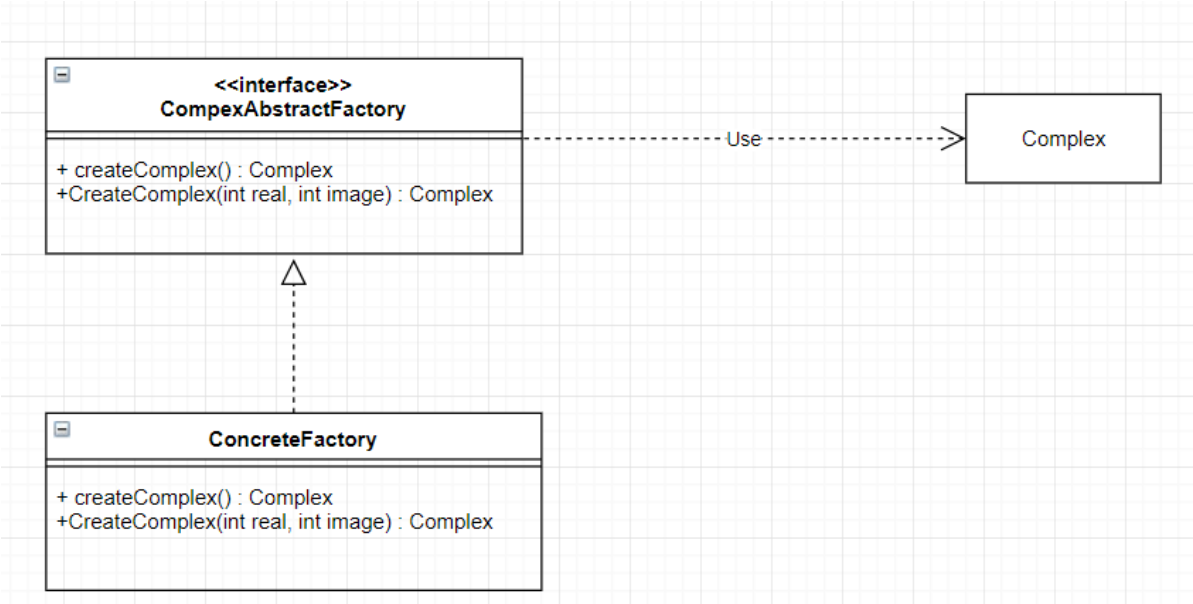

Рисунок 24.1. UML диаграмма проекта ComplexNumber с обработкой исключений

Задания на практическую работу № 21

Задание 1.

Разработать программную реализацию по UML диаграмме, представленной на рис.24.1с использованием изучаемых паттернов.

Задание 2.

Реализовать класс Абстрактная фабрика для различных типов стульев: Викторианский стул, Многофункциональный стул, Магический стул, а также интерфейс Стул, от которого наследуются все классы стульев, и класс Клиент, который использует интерфейс стул в своем методе Sit (Chair chair).

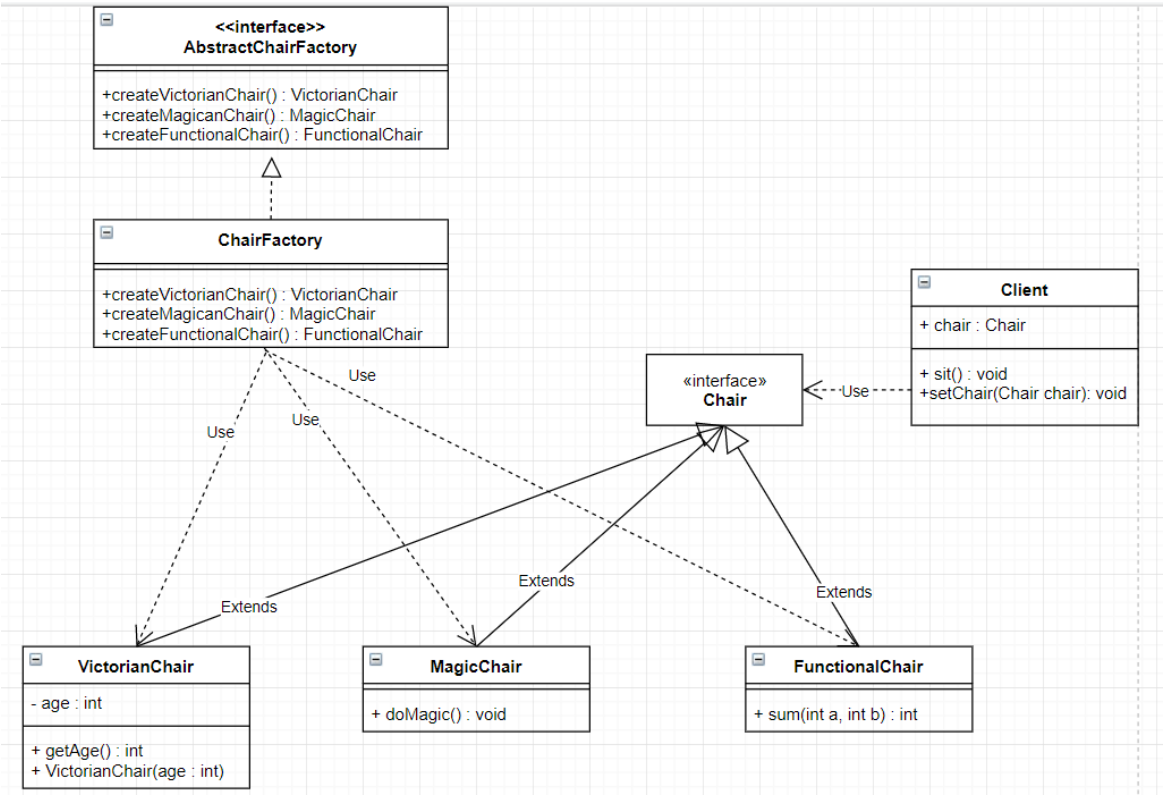

Рисунок 21.2. UML диаграмма проекта для компании XXX

Задание 3.

Легенда “Компании XXX нужно написать редактор текста, редактор изображений и редактор музыки. В трёх приложениях будет очень много общего: главное окно, панель инструментов, команды меню будут весьма схожими. Чтобы не писать повторяющуюся основу трижды, вам поручили разработать основу (каркас) приложения, которую можно использовать во всех трёх редакторах.”

На совещании в компании была принята следующая архитектура проекта:

Исходные данные. Есть некий базовый интерфейс IDocument, представляющий документ неопределённого рода. От него впоследствии будут унаследованы конкретные документы: TextDocument, ImageDocument, MusicDocument и т.п. Интерфейс IDocument перечисляет общие свойства и операции для всех документов.

- При нажатии пунктов меню File -> New и File -> Open требуется создать новый экземпляр документа (конкретного подкласса). Однако каркас не должен быть привязан ни к какому конкретному виду документов.

- Нужно создать фабричный интерфейс ICreateDocument. Этот интерфейс содержит два абстрактных фабричных метода: CreateNew и CreateOpen, оба возвращают экземпляр IDocument
- Каркас оперирует одним экземпляром IDocument и одним экземпляром ICreateDocument. Какие конкретные классы будут подставлены сюда, определяется во время запуска приложения.

Требуется:

1. создать перечисленные классы. Создать каркас приложения — окно редактора с меню File. В меню File реализовать пункты New, Open, Save, Exit.

продемонстрировать работу каркаса на примере текстового редактора. Потребуется создать конкретный унаследованный класс TextDocument и фабрику для него — CreateTextDocument.