深圳大学实验报告

课程名称：软件体系结构与设计模式

实验项目名称：实验 4 行为型设计模式分析与应用

学院：计算机与软件学院

专业：软件工程

指导教师：毛斐巧

报告人：郑彦薇 学号：2020151022 班级：软件工程 01 班

实验时间：2023 年 5.24，31，6.7，14，21 日（周三）

实验报告提交时间：2023/6/4

教务部制

一、 实验目的与要求：

1. 对 GoF 23 个设计模式中的行为型设计模式的 UML 图形表示，进行结构分析，发现与总结出反复出现的 UML 图形结构。

2. 结合实例，熟练绘制常见的行为型设计模式结构图。

3. 理解每一种行为型设计模式的模式动机，掌握模式结构，学习如何使用代码实现这些设计模式。

二、实验内容

1. 使用辅助工具绘制职责链模式、命令模式、迭代器模式、观察者模式、状态模式、策略模式和模板方法模式这 7 种行为型模式的类结构图，并进行分析，指出包含的各角色类。另外，这 7 种行为型设计模式的使用频率如何？哪种使用频率最高？哪种最低？

2. 在某 web 框架中采用职责链模式来组织数据过滤器，不同的数据过滤器提供了不同的功能，例如字符编码转换过滤器、数据类型转换过滤器、数据校验过滤器等，可以将多个过滤器连接成一个过滤器链，进而对数据进行多次处理。根据以上描述，绘制对应的类图并编程模拟实现。

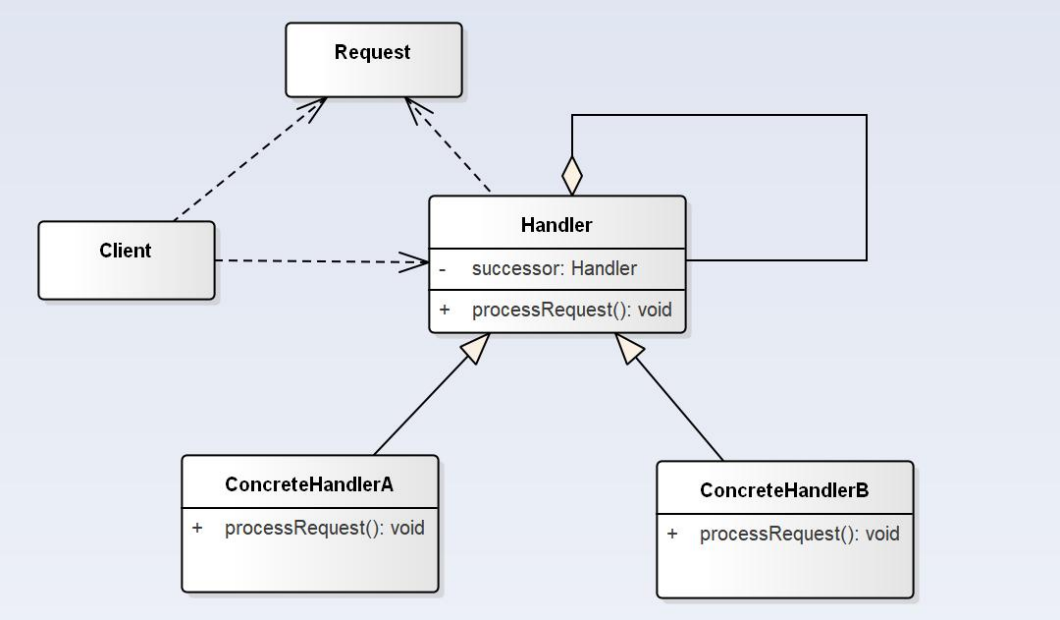
3. 在某云计算模拟平台中提供了多种虚拟机迁移算法，例如动态迁移算法中的 Pre Copy(预拷贝)算法、Post-Copy(后拷贝)算法、CR/RT-Motion 算法等，用户可以灵活地选择所需的虚拟机迁移算法，也可以方便地增加新算法。现采用策略模式进行设计，绘制对应的类图并编程模拟的实现。

4. 在某数据挖掘工具的数据分类模块中，数据处理流程包括 4 个步骤，分别是(1)读取数据，(2)二转换数据格式，(3)调用数据分类算法，(4)显示数据分类结果。对于不同的分类算法而言，第 1 步、第 2 步和第 4 步是相同的，主要区别在于第 3 步，第 3 步将调用算法库中已有的分类算法实现，例如朴素贝叶斯分类(Nave Bayes)算法，决策树（Decision Tree）算法、K 最近邻(K-Nearest Neighbor,KNN)算法等。现采用模板方法模式和适配器模式设计该数据分类模块，绘制对应的类图并编程模拟实现。

三、理解和分析报告、实践过程及结果等

1. 答：（1）职责链模式中，通常每个接收者都包含对另一个接收者的引用，如果一个对象不能处理该请求，那么它会把相同的请求传给下一个接收者。

类结构图绘制如下：



角色分析：

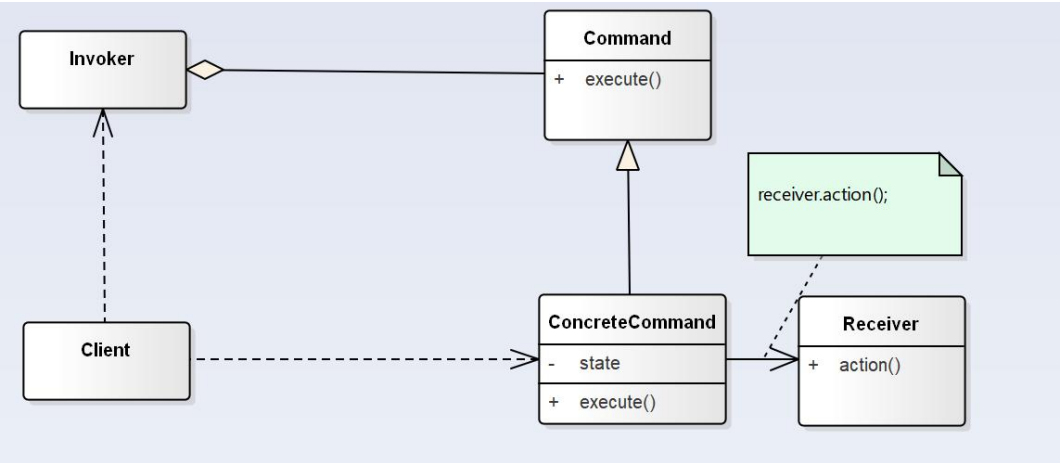
Handler 是抽象的处理者，它定义了一个处理请求的接口，同时含有另外 Handler。

ConcreteHandlerA/B 是具体的处理者，处理它自己负责的请求，可以访问它的后继者（即下一个处理者），如果可以处理当前请求则处理，否则就将该请求交给后继者去处理，从而形成一个职责链。

Request 表示一个请求，含有很多属性。

（2）命令模式的核心在于引入了抽象命令类和具体命令类，通过命令类来降低发送者和接收者的耦合度，请求发送者只需指定一个命令对象，再通过命令对象来调用请求接收者的处理方法。

类结构图绘制如下：



角色分析：

Command 是抽象命令类，一般是一个抽象类或接口，在其中声明了用于执行请求的 execute() 等方法，通过这些方法可以调用请求接收者的相关操作。

ConcreteCommand 是具体命令类，它是抽象命令类的子类，实现了在抽象命令类中声明的方

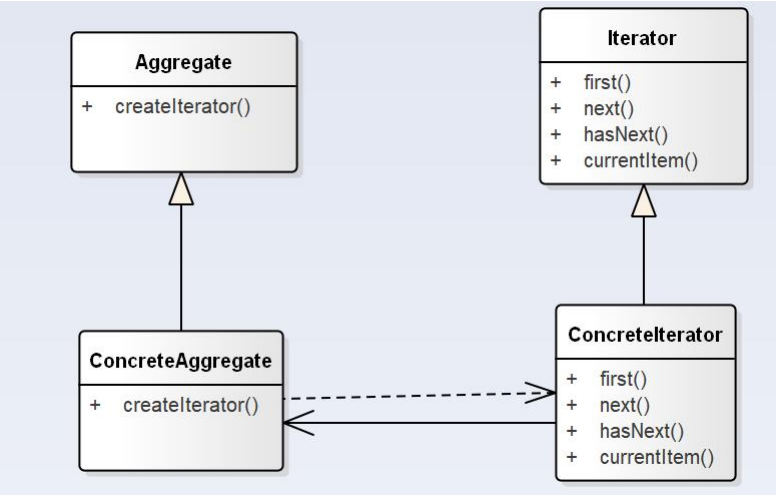
法，它对应具体的接收者对象，将接收者对象的动作绑定其中，具体命令类在实现 execute()方法时将调用接收者对象的相关操作。

Invoker 是调用者，即发送请求者，它通过命令对象来执行请求。一个调用者并不需要在设计时确定其接收者，因此它只与抽象命令类之间存在关联关系。在程序运行时可以将一个具体命令对象注入其中，再调用具体命令对象的 execute()方法，从而实现间接调用请求接收者的相关操作。

Receiver 是接收者，接收者执行与请求相关的操作，具体实现对请求的业务处理。

（3）迭代器模式提供一个外部的迭代器来对聚合对象进行访问和遍历，迭代器定义了一个访问该聚合元素的接口，并且可以跟踪当前遍历的元素，了解哪些元素已经遍历过而哪些没有。

类结构图绘制如下：



角色分析：

Iterator 是抽象迭代器，定义了访问和遍历元素的接口，一般声明这几个方法：用于获取第一个元素的 first()，用于访问下一个元素的 next()，用于判断是否还有下一个元素的 hasNext()，用于获取当前元素的 currentItem()，在其子类中实现这些方法。

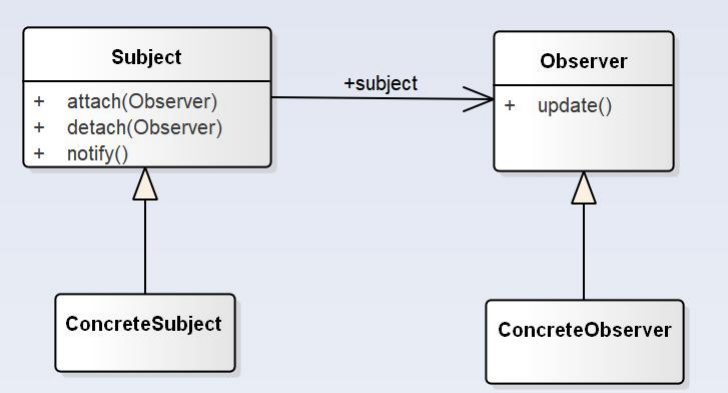
ConcreteIterator 具体迭代器类，它实现了抽象迭代器接口，完成对聚合对象的遍历，同时在对象聚合进行遍历时跟踪其当前位置。

Aggregate 抽象聚合类用于存储对象，并定义创建相应迭代器对象的接口，声明一个 createIterator()方法用于创建一个迭代器对象。

ConcreteAggregate 是具体聚合类，实现了创建相应迭代器的接口，实现了在聚合类中声明的 createIterator()方法，该方法返回一个与该具体聚合对应的具体迭代器 ConcreteIterator 实例。

（4）观察者模式属于行为型模式的一种，它定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态变化时，会通知所有的观察者对象，使他们能够自动更新自己。

类结构图绘制如下：



角色分析：

Subject 是抽象主题，或抽象被观察者。抽象主题角色把所有观察者对象保存在一个集合里，每个主题都可以有任意数量的观察者，抽象主题提供一个接口，可以增加和删除观察者对象。

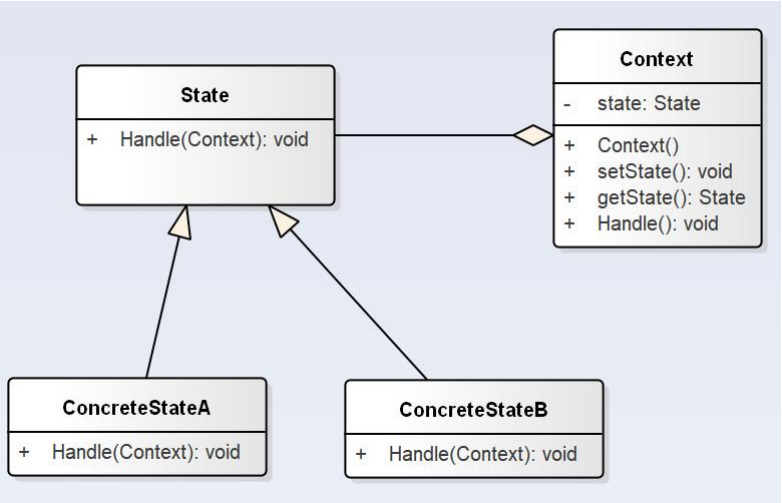
ConcreteSubject 是具体主题，或具体被观察者。该角色将有关状态存入具体观察者对象，在具体主题的内部状态发生改变时，给所有注册过的观察者发送通知。

Observer 抽象观察者是观察者的抽象类，它定义了一个更新接口，使得在得到主题更改通知时更新自己。

ConcreteObserver 具体观察者实现抽象观察者定义的更新接口，以便在得到主题更改通知时更新自身的状态。

（5）状态模式用于解决系统中复杂对象的状态转换以及不同状态下行为的封装问题。它将一个对象的状态从该对象中分离出来，封装到专门的状态类中，使得对象状态可以灵活变化，对于客户端而言，无需关心对象状态的转换以及对象所处的当前状态，无论对于何种状态的对象，客户端都可以一致处理。

类结构图绘制如下：



角色分析：

Context 环境类，又称为上下文类，它是拥有多种状态的对象。由于环境类的状态存在多样性且在不同状态下对象的行为有所不同，因此将状态独立出去形成单独的状态类。在环境类

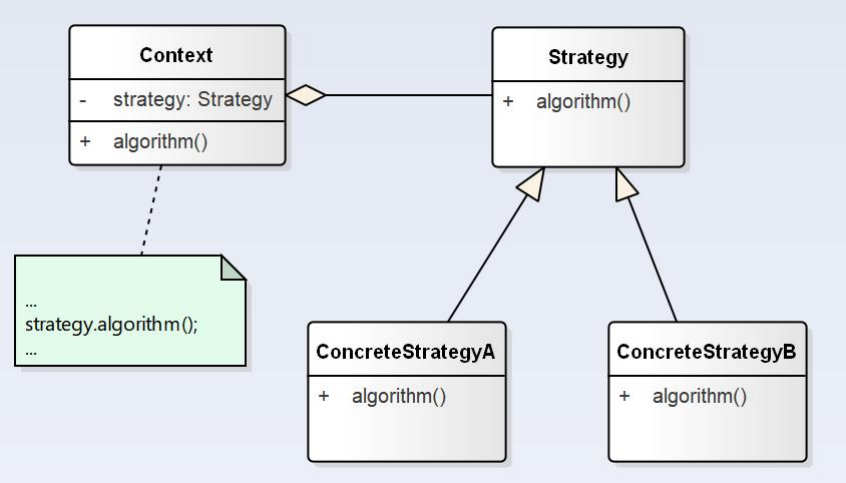
中维护一个抽象状态类 State 的实例，这个实例定义当前状态，在具体实现时，它是一个 State 子类的对象。

State 抽象状态角色，它用于定义一个接口以封装与环境类的一个特定状态相关的行为，在抽象状态类中声明了各种不同状态对应的方法，而在其子类中实现类这些方法，由于不同状态下对象的行为可能不同，因此在不同子类中方法的实现可能存在不同，相同的方法可以写在抽象状态类中。

ConcreteState 具体状态角色，它实现抽象状态所对应的行为，并且在需要的情况下进行状态切换。

（6）策略模式定义一些独立的类来封装不同的算法，每一个类封装一种具体的算法，在这里每一个封装算法的类都可以称为一种策略为了保证这些策略在使用时具有一致性，一般会提供一个抽象的策略类来做算法的声明，而每种算法对应于一个具体策略类。

类结构图绘制如下：



角色分析：

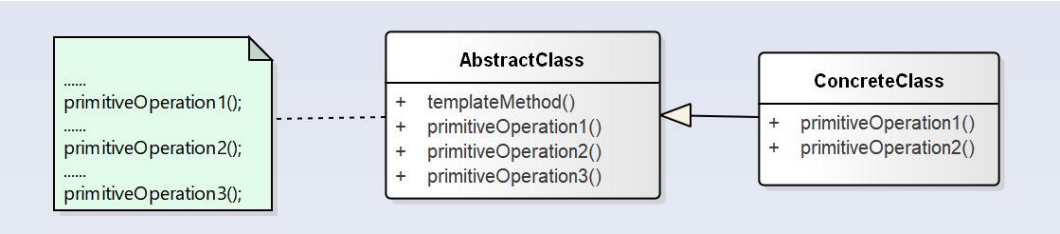
Context 环境类是使用算法的角色，它在解决某个问题时可以采用多种策略。在环境类中维持一个对抽象策略类的引用实例，用于定义所采用的策略。

Strategy 是抽象策略类，为所支持的算法声明了抽象方法，是所有策略类的父类，它可以是抽象类或具体类，也可以是接口。环境类通过抽象策略类中声明的方法在运行时调用具体策略类中实现的算法。

ConcreteStrategy 具体策略类实现了在抽象策略类中声明的算法，在运行时具体策略类将覆盖在环境类中定义的抽象策略类对象，使用一种具体的算法实现某个业务功能。

（7）模板方法模式定义一个操作中算法的框架，而将一些步骤延迟到子类中。模板方法模式使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。它是结构最简单的行为型设计模式，在其结构中只存在父类和子类的继承关系。

类结构图绘制如下：



角色分析：

AbstractClass 抽象类，在抽象类中定义了一系列基本操作（Primitive Operations），这些基本操作可以是具体的，也可以是抽象的，每一个基本操作对应算法的一个步骤，在其子类中可以重定义或实现这些步骤。同时在抽象类中实现了一个模板方法（Template Method），用于定义一个算法的框架，模板方法不仅可以调用在抽象类中实现的基本方法，也可以调用在抽象类的子类中实现的基本方法，还可以调用其他对象中的方法。

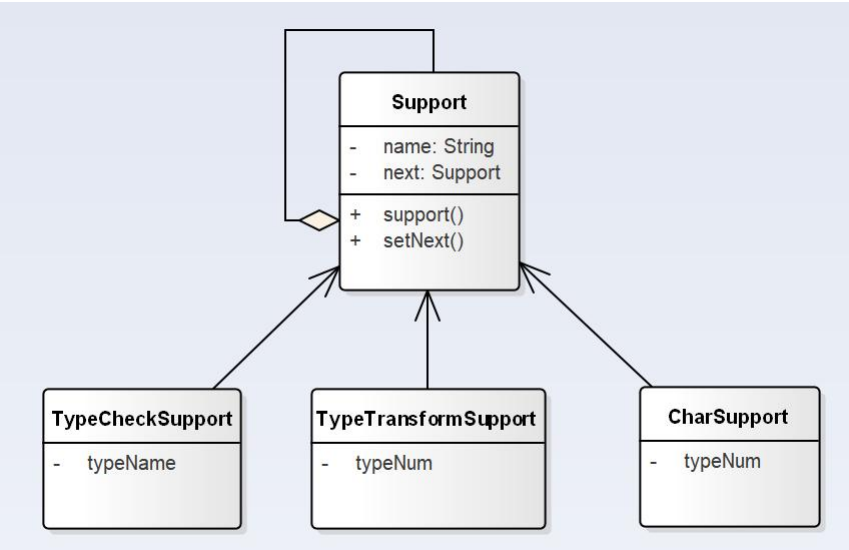
ConcreteClass 具体子类，它是抽象类的子类，用于实现在父类中声明的抽象。

（8）这 7 种行为型设计模式的使用频率如何？哪种使用频率最高？哪种最低？

根据项目的具体需求不同，不同的设计模式也具有不同的使用场景和使用频率。相对而言，策略模式的使用频率较其他行为型设计模式高，它可以让算法的变化独立于使用算法的客户端，提高代码的灵活性和可维护性。职责链模式的使用频率较其他行为型设计模式低，主要是在事件处理和异常处理的设计中使用。其他行为型设计模式的使用频率的大致排名为观察者模式、模板方法模式、迭代器模式、命令模式、状态模式。

2. 给出应用职责链模式的结构图（类图）和实现代码。

（1）根据问题描述可以绘制类图如下：



（2）给出模拟编程实现：

定义一个 Trouble 类，用 number 来对应过滤器类型

```
public class Trouble {
    private int number;//定义 trouble 的类型，对应过滤器类型
```

```
    public Trouble(int number){
        this.number = number;
    }
    public int getNumber(){
        return this.number;
    }
    public String toString(){
        switch (number) {
            case 1:
                return "[Trouble " + number + " CharTrouble]";
            case 2:
                return "[Trouble " + number + " TypeTransformTrouble]";
            case 3:
                return "[Trouble " + number + " TypeCheckTrouble]";
            default:
                return "No such trouble";
        }
    }
}
```

Support 类

```
public abstract class Support {
    private String name;
    private Support next;
    public Support(String name){
        this.name = name;
    }
    public Support setNext(Support next){ //设置下一个支持对象
        this.next = next;
        return next;
    }
    public final void support(Trouble trouble){
        if(resolve(trouble)){
            done(trouble);
        } else if (next!=null) {
            next.support(trouble);
        } else{
            fail(trouble);
        }
    }
    public String toStirng(){
        return "["+name+"]";
    }
    protected abstract boolean resolve(Trouble trouble); //解决 trouble
    protected void done(Trouble trouble){ //完成 trouble
```

```
            System.out.println(trouble+"is resolved by "+this);
        }
        protected void fail(Trouble trouble){ //未能解决 trouble
            System.out.println(trouble + " cannot be resolved");
        }
}
```

CharSupport 类（另外两个类同理，修改 typeNum 值对应 Trouble 定义的对应值即可）

```
public class CharSupport extends Support{
    private int typeNum = 1; //其他两个修改此处的值为 trouble 中对应的值
    public CharSupport(String name) {
        super(name);
    }

    @Override
    protected boolean resolve(Trouble trouble) {
        if(trouble.getNumber() == typeNum){
            return true;
        }else {
            return false;
        }
    }
}
```

Test 类

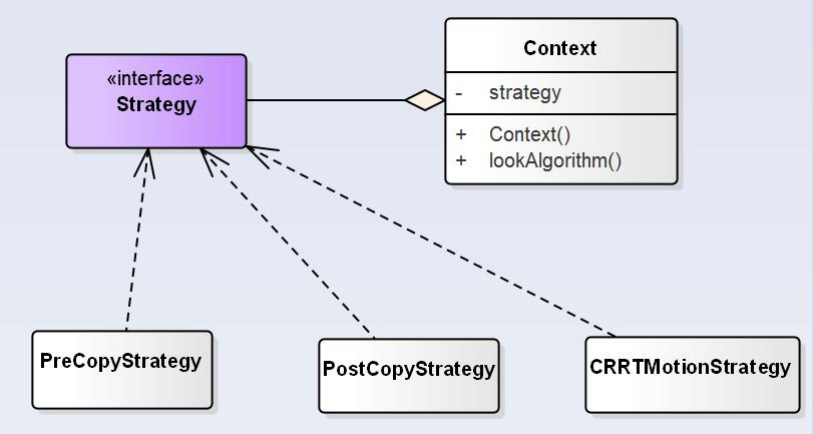
```
public class Test {
    public static void main(String[] args) {
        Support charSupport = new CharSupport("charSupport");
        Support typeTransformSupport = new TypeTransformSupport("typeTransformSupport");
        Support typeCheckSupport = new TypeCheckSupport("typeCheckSupport");

        charSupport.setNext(typeTransformSupport).setNext(typeCheckSupport);

        for(int i = 1; i <= 4; i++){
            charSupport.support(new Trouble(i));
        }
    }
}
```

运行 Test 类，得到输出如下

```
[Trouble 1 CharTrouble]is resolved by Q1.CharSupport@2ef9b8bc
[Trouble 2 TypeTransformTrouble]is resolved by Q1.TypeTransformSupport@5d624da6
[Trouble 3 TypeCheckTrouble]is resolved by Q1.TypeCheckSupport@1e67b872
No such trouble cannot be resolved
```

3. 给出应用策略模式的类图和编程模拟实现代码。

（1）根据问题描述可以给出类图如下



（2）编程模拟实现

定义 Strategy 接口，包含一个 algorithm 方法

```
public interface Strategy {
    public void algorithm();
}
```

定义算法类 PreCopyStrategy（其他两个同理，修改输出信息即可）

```
public class PreCopyStrategy implements Strategy{
    @Override
    public void algorithm() {
        System.out.println("Use Pre-Copy algorithm");
    }
}
```

定义 Context 实现策略模式框架

```
public class Context {
    private Strategy strategy;
    public Context(Strategy strategy){
        this.strategy = strategy;
    }
    public void lookAlgorithm(){ //调用 strategy
        strategy.algorithm();
    }
}
```

Test 类

```
public class Test {
    public static void main(String[] args) {
        Strategy preStrategy = new PreCopyStrategy();
        //..其他两个算法类同理
        Context preContext = new Context(preStrategy);
        //..其他两个算法类同理
        preContext.lookAlgorithm();
```

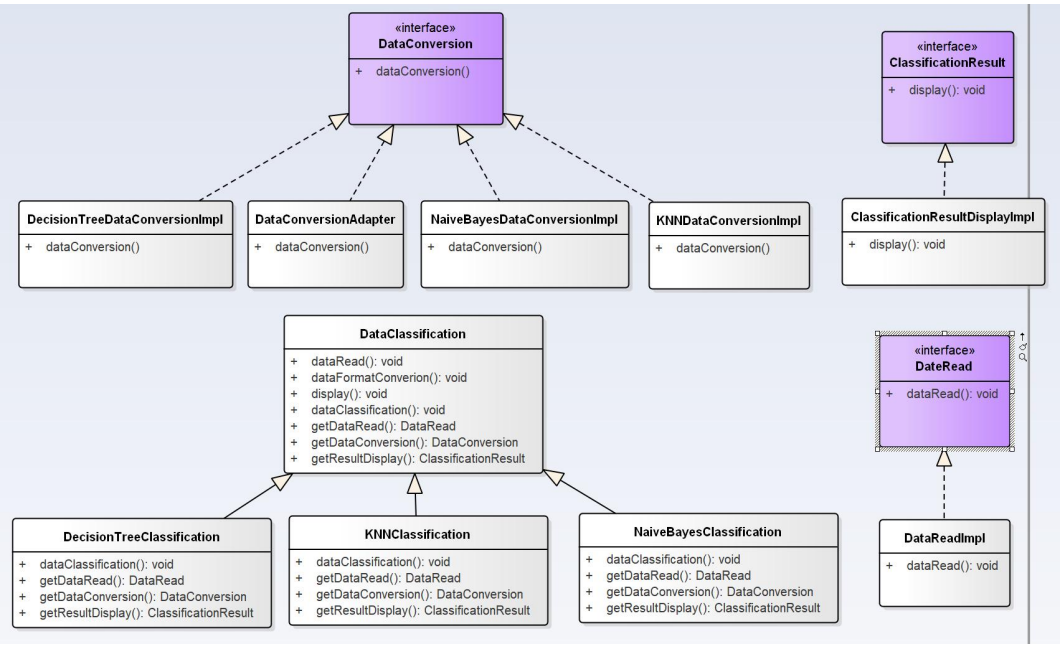
```
        //..其他两个算法类同理
    }
}
```

运行 Test 类，可以得到输出如下

```
Use Pre-Copy algorithm
Use Post-Copy algorithm
Use CR/RT-Motion algorithm
```

4. 给出应用模板方法模式和适配器模式的类图和编程模拟实现代码。

（1）根据问题描述可以给出类图如下



（2）编程模拟实现

数据分类模块抽象类 DataClassification

```
public abstract class DataClassification {
    public void dataRead() {// 调用具体的数据读取实现类
        DataRead dataRead = getDataRead();
        dataRead.dataRead();
    }
    public void dataFormatConversion() { // 调用具体的数据转换实现类
        DataConversion dataConversion = getDataConversion();
        dataConversion.dataConversion();
    }
    public void display() {// 调用具体的分类结果显示实现类
        ClassificationResult resultDisplay = getResultDisplay();
        resultDisplay.display();
    }
    // 调用具体的分类算法
```

```
    public abstract void dataClassification();
    // 数据读取
    public abstract DataRead getDataRead();
    // 数据转换
    public abstract DataConversion getDataConversion();
    // 显示
    public abstract ClassificationResult getResultDisplay();
}
```

定义算法类NaiveBayesClassification实现抽象方法调用具体的分类算法（其他两种算法同理，修改调用的具体分类算法即可）

```
public class NaiveBayesClassification extends DataClassification {
    // 调用具体的分类算法
    @Override
    public void dataClassification() {
        System.out.println("使用朴素贝叶斯分类算法进行分类");
    }
    // 数据读取
    @Override
    public DataRead getDataRead() {
        return new DataReadImpl();
    }
    // 数据转换
    @Override
    public DataConversion getDataConversion() {
        return new DataConversionAdapter(new NaiveBayesDataConversionImpl());
    }
    // 显示分类结果
    @Override
    public ClassificationResult getResultDisplay() {
        return new ClassificationResultDisplayImpl();
    }
}
```

定义适配器类 DataConversionAdapter 将数据转换为算法库中分类算法所需要的格式

```
public class DataConversionAdapter implements DataConversion {
    private DataConversion dataConversion;
    public DataConversionAdapter(DataConversion dataConversion) {
        this.dataConversion = dataConversion;
    }
    // 实现接口方法，调用具体的数据转换实现类
    @Override
    public void dataConversion() {
        dataConversion.dataConversion();
    }
```

```
}
```

定义数据转换的抽象接口 DataConversion

```
public interface DataConversion {
    void dataConversion();
}
```

不同算法的数据转换具体实现类（其它两个算法同理）

```
public class NaiveBayesDataConversionImpl implements DataConversion {
    @Override
    public void dataConversion() {
        System.out.println("将数据转换为朴素贝叶斯分类算法所需的格式");
    }
}
```

定义数据读取的抽象接口 DataRead

```
public interface DataRead {
    void dataRead();
}
```

数据读取抽象接口的具体实现类 DataReadImpl

```
public class DataReadImpl implements DataRead {
    @Override
    public void dataRead() {
        System.out.println("读取数据");
    }
}
```

定义 ClassificationResult 接口显示分类结果

```
public interface ClassificationResult {
    void display();
}
```

显示结果接口的具体实现类 ClassificationResultImpl

```
public class ClassificationResultImpl implements ClassificationResult {
    @Override
    public void display() {
        System.out.println("显示分类结果");
    }
}
```

Test 类

```
public class Test {
    public static void main(String[] args) {
        // 使用朴素贝叶斯分类算法进行分类
        DataClassification naiveBayesClassification = new NaiveBayesClassification();
        naiveBayesClassification.dataRead();
        naiveBayesClassification.dataFormatConversion();
        naiveBayesClassification.dataClassification();
```

```
            naiveBayesClassification.display();
            System.out.println();
            //...其他两种算法同理
        }
}
```

运行 Test 类，可以得到输出如下

```
读取数据
将数据转换为朴素贝叶斯分类算法所需的格式
使用朴素贝叶斯分类算法进行分类
显示分类结果

读取数据
将数据转换为决策树分类算法所需的格式
使用决策树分类算法进行分类
显示分类结果

读取数据
将数据转换为K最近邻分类算法所需的格式
使用K最近邻分类算法进行分类
显示分类结果
```

四、实验总结与体会

1. 通过本次实验，对 7 种行为型设计模式有了更深的掌握，学习了其简单类结构图的绘制以及其中的所包含的角色。
2. 通过具体的问题解决，对职责链模式、策略模式、模板方法模式和适配器模式进行了应用，学会了如何利用这些模式的特点设计相应问题的类图和模拟编程实现，对这三个模式有了更好的掌握。

五、成绩评定及评语

1.指导老师批阅意见：

2.成绩评定：

指导教师签字：毛斐巧

2023 年　　月　　日