# 期末大作业

通过查询并参考课外资料，自选3个日常生活中与软件设计模式有关的典型案例。对应每个案例，分别重点使用1个设计模式以完成该案例的实验设计与实现。所使用的设计模式必须来源于本课程学习到的共计24个设计模式的其中3个，且满足属于创建型设计模式、结构型设计模式和行为型设计模式各1个。以解决实际案例为核心，撰写并提交一份篇幅不少于4500字的word格式的实验报告文档。

该实验报告应至少包含针对不同案例的问题说明、方案描述、结果展示、结论归纳等基本要素。同时，可结合环境搭建、代码解释、类图绘制以及心得体会等方面进一步丰富实验报告内容。此外，实验报告中务必要体现与突出使用设计模式相较于未使用设计模式的优势所在。原则上，不强制要求上交代码附件，但需要在文档中提供实验相关截图。

本期末大作业的提交截止时间为第16周的周日结束前，请合理安排时间并及时上传系统。

## 案例一创建型设计模式

### 1案例介绍

#### 问题说明

在日常生活中，手机套餐的销售是一项普遍且充满商业智慧的活动。它不仅吸引顾客购买，还蕴含着丰富的商业策略。如何引导顾客做出最佳选择，并为他们提供最合适的手机套餐，包括内存、颜色、性能等因素，是提升顾客满意度和品牌价值的关键。

我打算运用计算机软件设计中的创建型设计模式中的建造者模式，可以有效地解决这一问题。建造者模式封装一个复杂对象构造过程，并允许按步骤构造。在手机套餐销售的背景下，我们可以利用这一模式来设计一个系统，该系统能够根据顾客的具体需求，如内存大小、颜色偏好、性能要求等，动态地构建和推荐最合适的手机套餐。

#### 方案描述

建造者模式来描述这个手机销售方案：

1.最终的产品是手机，每款手机有多个属性：

* **配色**（Color）：例如红色、蓝色、黑色等。
* **存储容量**（Storage）：如128GB、256GB、512GB等。
* **套餐**（Plan）：如基础套餐、标准套餐、高级套餐等。

2.用建造者接口来定义创建手机的各个步骤，每个步骤负责设置手机的某一属性：设置手机配色、手机存储容量、套餐选择、获取最终构建好的手机

3.用具体的建造者实现了建造者接口，并通过它来构建不同配置的手机。

4.用手机类用于表示最终的手机产品，包含了配色、存储容量、套餐等信息。

5.导演类负责指导具体建造者按照顺序一步步构建手机，并将构建好的手机交给客户端使用。它根据不同的需求来构建不同配置的手机。

6.在客户端中，我们可以通过导演类来构建各种不同配置的手机。客户端通过指定配色、存储和套餐来获取符合需求的手机。

#### 模式说明与优点

1. **Phone** 类表示最终的手机产品，它包含了配色、存储和套餐等属性。
2. **PhoneBuilder** 接口定义了构建手机的步骤，而 **ConcretePhoneBuilder** 类实现了这些步骤，用于构建不同配置的手机。
3. **PhoneDirector** 类负责协调建造者，按照需求构建手机。
4. **客户端代码** 通过与 **PhoneDirector** 的交互，方便地构建不同配置的手机。

通过这种方式，建造者模式使得手机的各个配置（如配色、存储、套餐）可以灵活组合，而客户端则不需要关心如何构建手机的细节。

这种模式特别适用于当产品复杂，且需要不同通过建造者模式，在构建复杂对象时具有以下几个明显的优点：

1. **分离构建过程与表示**：

客户端只需要关心如何通过建造者来创建所需的手机，而不需要关心创建的细节。所有的配置和组合逻辑都被封装在了建造者和导演类中，减少了代码的耦合度。

1. **灵活的配置组合**：

不同的配色、存储容量和套餐可以自由组合，通过多次调用建造者的不同方法，可以构建出多种不同配置的手机，且无需修改已有的代码。

1. **易于扩展**：

若后续需要添加新的配置项（例如增加更多配色或新的套餐选项），只需要在 PhoneBuilder 接口和其实现类中添加新的方法即可，不会影响到已有的客户端代码。

1. **增强代码可维护性**：

将构建过程分解成多个小步骤，代码更清晰、更易于维护。每个方法的职责单一，符合单一职责原则。

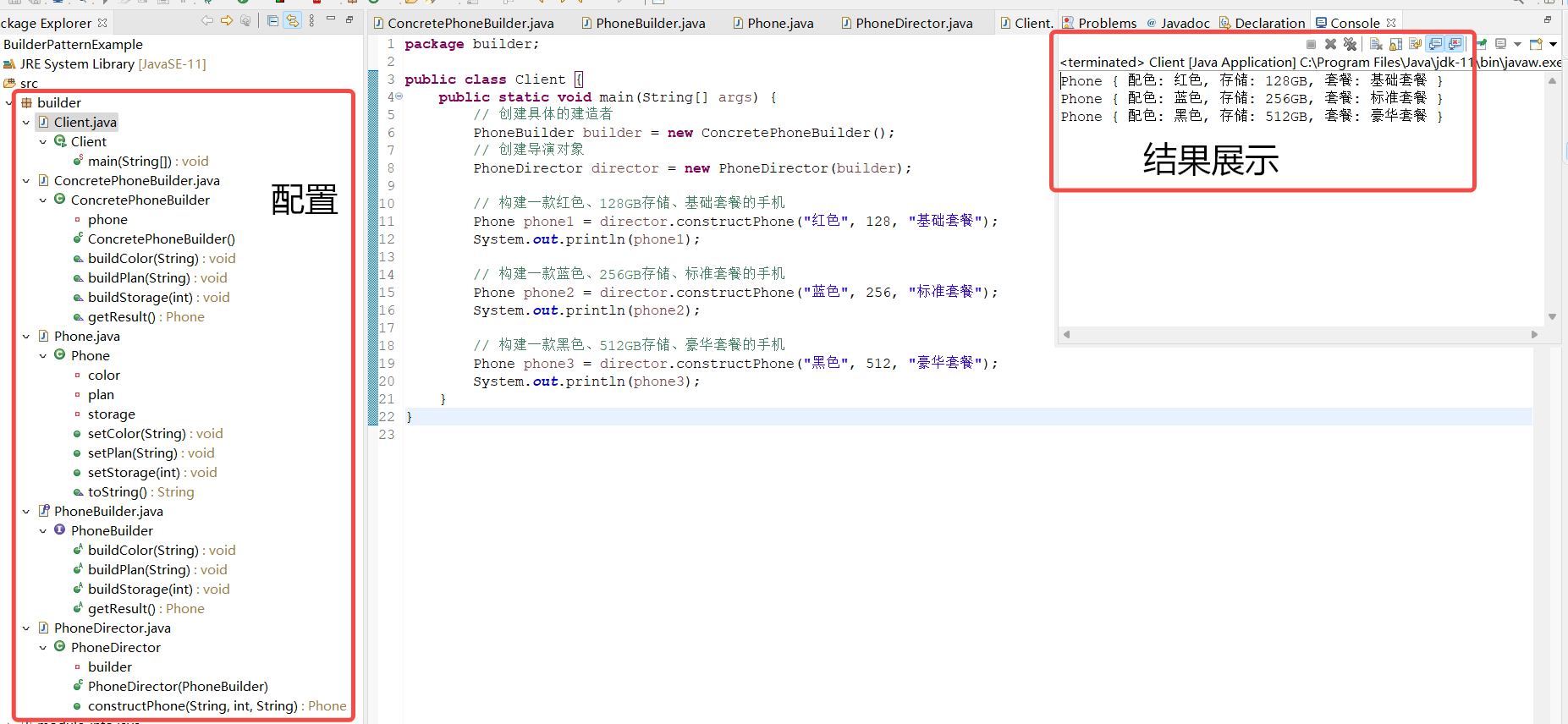
1. **解耦合客户端与构建过程**：

客户端不需要了解如何一步步构建手机，只需指定手机的配色、存储和套餐，建造者模式通过封装复杂的构建过程，避免了客户端与构建细节的耦合。

### 2成果展示

#### 环境搭建

运用Eclipse 展示：



#### 代码解释

**建造者接口（PhoneBuilder）**：

public interface PhoneBuilder {

void buildColor(String color); // 配色

void buildStorage(int storage); //存储容量

void buildPlan(String plan); //套餐

Phone getResult(); //构建手机

}

**具体建造者（ConcretePhoneBuilder）**：

public class ConcretePhoneBuilder implements PhoneBuilder {

private Phone phone;

public ConcretePhoneBuilder() {

phone = new Phone(); //创建一个新的手机实例

}

@Override

public void buildColor(String color) {

phone.setColor(color); //配色

}

@Override

public void buildStorage(int storage) {

phone.setStorage(storage); //存储容量

}

@Override

public void buildPlan(String plan) {

phone.setPlan(plan); //套餐

}

@Override

public Phone getResult() {

return phone; //构建手机

}

}

**产品类（Phone）：**

public class Phone {

private String color;

private int storage;

private String plan;

// 设置方法

public void setColor(String color) {

this.color = color;

}

public void setStorage(int storage) {

this.storage = storage;

}

public void setPlan(String plan) {

this.plan = plan;

}

// 获取套餐详细信息

@Override

public String toString() {

return "Phone { 配色: " + color + ", 存储: " + storage + "GB, 套餐: " + plan + " }";

}

}

**导演类（PhoneDirector）**：

public class PhoneDirector {

private PhoneBuilder builder;

public PhoneDirector(PhoneBuilder builder) {

this.builder = builder;

}

// 通过不同配置来构建手机

public Phone constructPhone(String color, int storage, String plan) {

builder.buildColor(color);

builder.buildStorage(storage);

builder.buildPlan(plan);

return builder.getResult();

}

}

**客户端代码（Client）**：

public class Client {

public static void main(String[] args) {

// 创建具体的建造者

PhoneBuilder builder = new ConcretePhoneBuilder();

// 创建导演对象

PhoneDirector director = new PhoneDirector(builder);

// 构建一款红色、128GB存储、基础套餐的手机

Phone phone1 = director.constructPhone("红色", 128, "基础套餐");

System.out.println(phone1);

// 构建一款蓝色、256GB存储、标准套餐的手机

Phone phone2 = director.constructPhone("蓝色", 256, "标准套餐");

System.out.println(phone2);

// 构建一款黑色、512GB存储、豪华套餐的手机

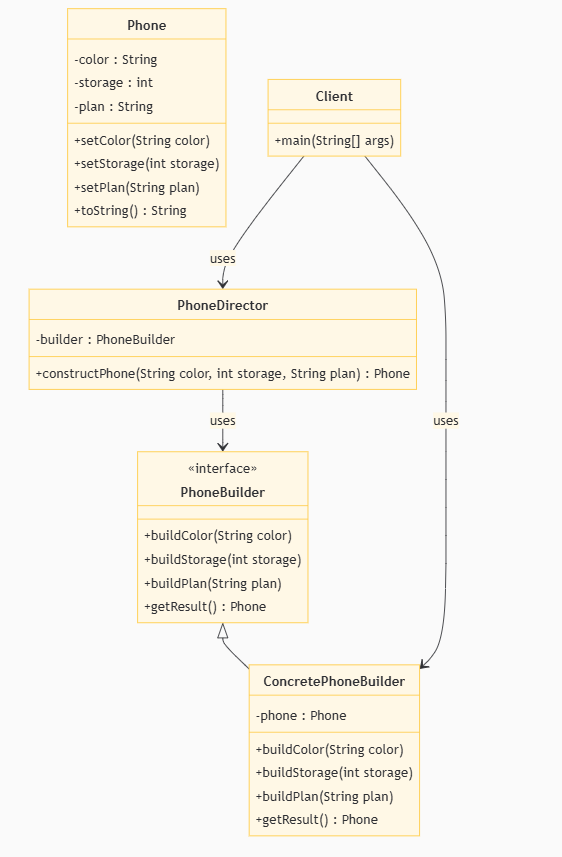
Phone phone3 = director.constructPhone("黑色", 512, "豪华套餐");

System.out.println(phone3);

}

}

#### 类图绘制

**关系说明**

* **PhoneBuilder（建造者接口）**：定义了建造手机的接口，包括设置颜色、存储容量、套餐和获取最终产品的方法。
* **ConcretePhoneBuilder（具体建造者）**：实现了PhoneBuilder接口，具体负责构建Phone对象。它持有一个Phone对象，并提供设置颜色、存储和套餐的方法。getResult()方法返回构建完成的Phone对象。
* **Phone（产品类）**：表示最终要构建的产品，即手机。包含颜色、存储容量和套餐属性，并提供设置这些属性的方法。toString()方法返回手机的详细信息。
* **PhoneDirector（导演类）**：持有一个PhoneBuilder对象，并指导如何构建Phone对象。它通过调用PhoneBuilder的方法来构建Phone对象，并返回最终构建的产品。
* **Client（客户端代码）**：客户端代码使用PhoneDirector和ConcretePhoneBuilder来构建不同的Phone对象。客户端不需要知道具体的构建细节，只需要通过导演类来获取最终的产品。

在这个建造者模式中，ConcretePhoneBuilder是具体建造者，负责具体的建造过程；PhoneDirector是导演类，负责指导建造过程；Phone是产品类，是建造过程的结果；Client是客户端代码，它使用导演类和建造者来获取最终的产品。这种模式将产品的构建过程和表示分离，使得相同的构建过程可以创建不同的产品。

### 3结论归纳

#### 总结与心得体会

在软件开发中，尤其是面向对象的设计中，经常需要处理具有复杂结构的对象的创建问题。例如在手机销售方案中，消费者可以选择不同的配色、存储容量和套餐，而这些配置的组合可能导致不同的手机产品。若使用传统方式进行设计，每次都需要重新编写代码来创建各种不同配置的手机，这样会导致代码重复和耦合度较高，维护困难。为了解决这一问题，建造者模式提供了一种灵活的解决方案。

建造者模式是一种创建型设计模式，旨在将复杂对象的构建过程与其表示分离，使得同样的构建过程可以创建不同的表示。通过将构建过程分解成多个步骤，可以逐步构建出一个复杂的对象，而客户端不需要关心这些构建的细节。

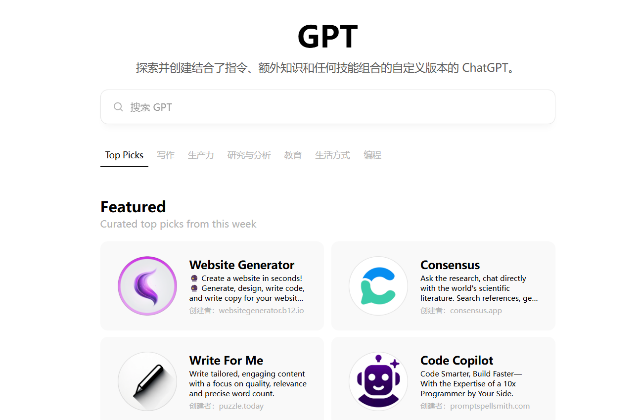
通过采用建造者模式，可以非常灵活地处理类似手机销售方案中不同配置组合的问题。建造者模式将产品的构建过程与具体的产品表示解耦，不仅使得代码更具可读性和可维护性，而且可以根据需求自由组合不同的配置选项，大大提高了代码的扩展性和灵活性。对于复杂的产品构建过程，建造者模式是一个非常理想的解决方案。

## 案例二结构型设计模式

### 1案例介绍

#### 问题说明

日常生活当中，我们都会使用到AI，但大家知道AI是怎样进行运行的，帮助我们进行解决问题，我们将采用外观设计模式进行回答！

在日常生活中，AI 技术被应用在各种场景中，例如聊天机器人、图像生成、文本总结、代码生成等。用户通常希望通过简单的操作即可调用复杂的 AI 功能，而不需要了解具体的技术细节。

#### 方案描述

外观设计模式（Facade Pattern）是一种结构型设计模式，旨在为子系统提供一个统一的接口，它通过隐藏复杂的子系统接口，为用户提供一个更为简单的使用方式。

在 AI 场景中，外观模式可以帮助我们：将复杂的 AI 功能模块封装成一个统一的接口，使用户无需直接与底层子系统交互。提供灵活、简化的操作体验，降低用户使用的复杂性。

我们以一个简化的 AI 服务系统为例，该系统包含以下功能：

1. **生成图片：**根据用户的需求生成图片。
2. 总结文本**：**对给定的文本进行总结。
3. **代码生成：**生成用户指定的代码。
4. **数据分析：**对用户的数据进行分析。

我们将通过外观设计模式构建一个统一的接口，用户只需调用 AIService 类，而无需关心具体各个功能模块的实现细节。

#### 模式说明与优点

外观模式（Facade Pattern）提供了一个统一的接口，用来访问一个子系统中的一群接口。AIServiceFacade 类作为外观类，封装了四个子系统：ImageGenerator、TextSummarizer、CodeGenerator 和 DataAnalyzer。通过这个外观类，客户端可以方便地访问这些子系统，而不需要直接与复杂的子系统接口打交道。

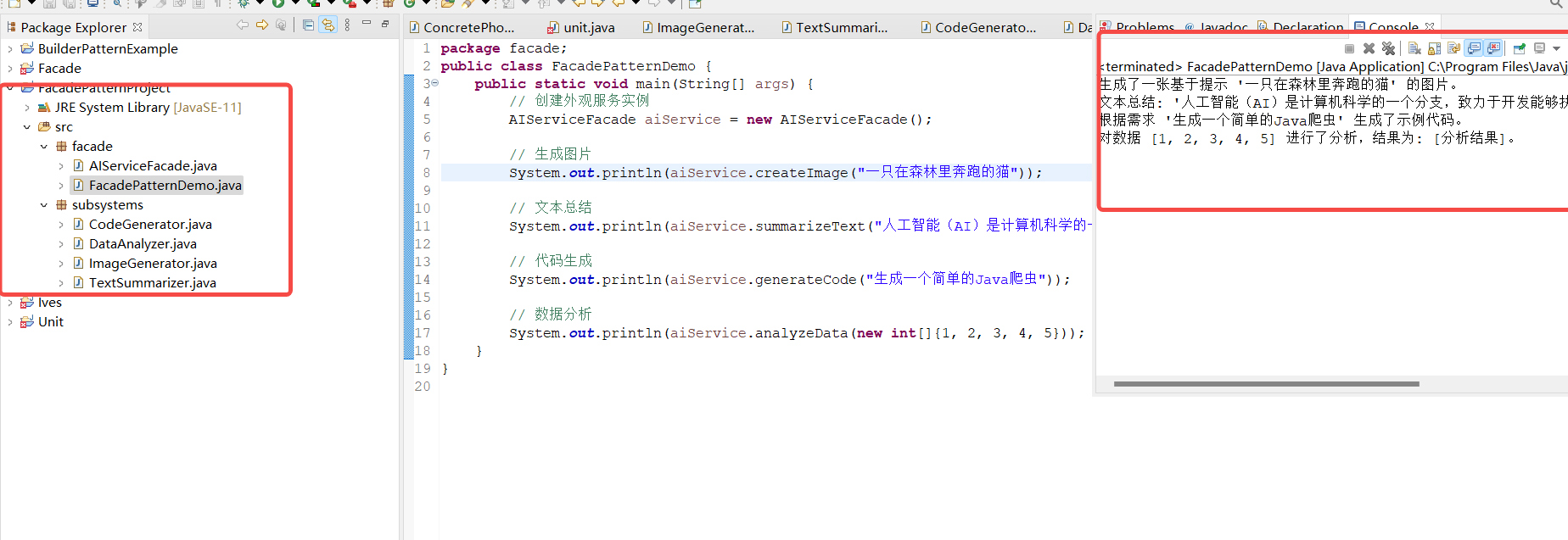
外观模式有如下优点

1. **简化接口**：为复杂的子系统提供一个客户端可以访问的简单接口。
2. **降低耦合度**：客户端与子系统的耦合度降低，提高系统的模块化。
3. **易于使用和维护**：客户端代码更简洁，易于理解和维护。
4. **良好的层次结构**：通过外观类，可以清晰地分离系统的不同层次。
5. **灵活性：**可以切换子系统的具体实现，而不影响客户端。

### 2成果展示

#### 环境搭建

运用Eclipse 展示：



#### 代码解释

**子系统 1: 图片生成模块(ImageGenerator)**

class ImageGenerator {

public String generateImage(String prompt) {

return "生成了一张基于提示 '" + prompt + "' 的图片。";

}

}

**子系统 2: 文本总结模块(TextSummarizer)**

class TextSummarizer {

public String summarizeText(String text) {

return "文本总结: '" + (text.length() > 50 ? text.substring(0, 50) + "..." : text) + "'";

}

}

**子系统 3: 代码生成模块(CodeGenerator)**

class CodeGenerator {

public String generateCode(String requirement) {

return "根据需求 '" + requirement + "' 生成了示例代码。";

}

}

**子系统 4: 数据分析模块(DataAnalyzer)**

class DataAnalyzer {

public String analyzeData(int[] data) {

StringBuilder result = new StringBuilder("对数据 [");

for (int i = 0; i < data.length; i++) {

result.append(data[i]);

if (i < data.length - 1) result.append(", ");

}

result.append("] 进行了分析，结果为: [分析结果]。");

return result.toString();

}

}

**外观实现类: 提供AI服务（AIServiceFacade）**

class AIServiceFacade {

private ImageGenerator imageGenerator;

private TextSummarizer textSummarizer;

private CodeGenerator codeGenerator;

private DataAnalyzer dataAnalyzer;

// 构造方法初始化所有子系统

public AIServiceFacade() {

this.imageGenerator = new ImageGenerator();

this.textSummarizer = new TextSummarizer();

this.codeGenerator = new CodeGenerator();

this.dataAnalyzer = new DataAnalyzer();

}

// 封装图片生成服务

public String createImage(String prompt) {

return imageGenerator.generateImage(prompt);

}

// 封装文本总结服务

public String summarizeText(String text) {

return textSummarizer.summarizeText(text);

}

// 封装代码生成服务

public String generateCode(String requirement) {

return codeGenerator.generateCode(requirement);

}

// 封装数据分析服务

public String analyzeData(int[] data) {

return dataAnalyzer.analyzeData(data);

}

}

**客户端: 使用外观（FacadePatternDemo）**

public class FacadePatternDemo {

public static void main(String[] args) {

// 创建外观服务实例

AIServiceFacade aiService = new AIServiceFacade();

// 生成图片

System.out.println(aiService.createImage("一只在森林里奔跑的猫"));

// 文本总结

System.out.println(aiService.summarizeText("人工智能（AI）是计算机科学的一个分支，致力于开发能够执行通常需要人类智能的任务的系统。"));

// 代码生成

System.out.println(aiService.generateCode("生成一个简单的Java爬虫"));

// 数据分析

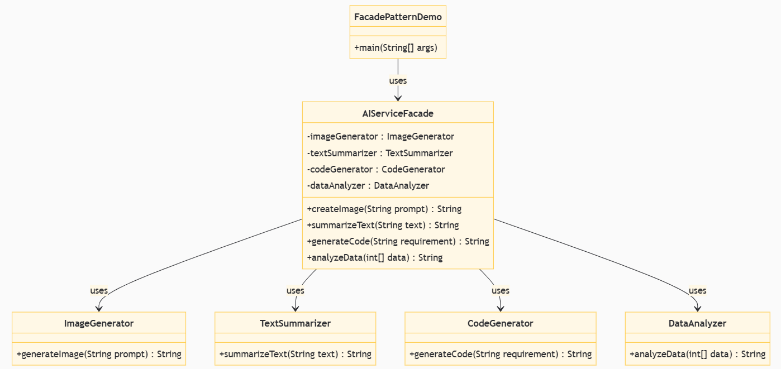
System.out.println(aiService.analyzeData(new int[]{1, 2, 3, 4, 5}));

}

}

#### 类图绘制

**关系说明**

* **ImageGenerator（图片生成模块）**：子系统类，负责生成基于提示的图片。
* **TextSummarizer（文本总结模块）**：子系统类，负责对文本进行总结。
* **CodeGenerator（代码生成模块）**：子系统类，负责根据需求生成代码。
* **DataAnalyzer（数据分析模块）**：子系统类，负责对数据进行分析。
* **AIServiceFacade（外观类）**：外观类，提供了一个统一的接口来访问子系统。它封装了对子系统对象的创建和方法调用，使得客户端代码可以简洁地访问这些子系统的功能。
* **FacadePatternDemo（客户端代码）**：客户端代码使用外观类来访问子系统的功能。它不需要知道子系统的具体实现细节，只需要与外观类交互。

在这个外观模式中，AIServiceFacade作为外观类，它为客户端提供了一个简化的接口来访问复杂的子系统。FacadePatternDemo客户端代码通过AIServiceFacade来使用各个子系统的功能，而不需要直接与这些子系统交互。这样，客户端代码的复杂性被大大降低，同时也隐藏了子系统的复杂性。

### 3结论归纳

#### 总结与心得体会

在实际开发中，外观模式是一种非常实用的设计模式。它能够隐藏系统的复杂性，为客户端提供一个清晰、简洁的接口。这不仅有助于新用户快速上手，也便于系统的扩展和维护。

外观模式通过引入一个外观类，封装了子系统的复杂性，使得客户端可以通过一个简单的接口与系统交互，这有助于提高代码的可维护性和可扩展性。

这种模式特别适合于那些拥有多个子系统和复杂接口的大型系统，能够有效地简化客户端的交互过程。在实际应用中，外观模式的灵活性体现在可以轻松替换子系统的实现，而不影响客户端。这种灵活性对于需要适应快速变化需求的系统尤为重要。

外观模式体现了“单一职责原则”和“开放封闭修改原则”，即系统应该对扩展开放，对修改封闭。这有助于构建更加健壮和可维护的系统架构，同时也为系统的未来发展提供了便利。

更多案例使用：ChatGPT 的功能整合：提供统一的服务入口，支持各种 AI 功能（如图像生成、代码生成、文本总结等）。跨平台 API 网关：通过外观模式封装多个 API，使开发者可以快速调用各种服务。

## 案例三行为型设计模式

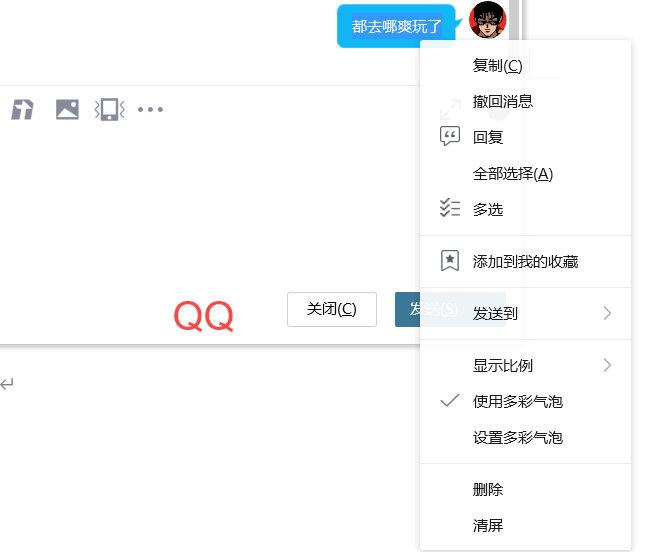
### 1案例介绍

#### 问题说明

在现代聊天工具（如微信、QQ、飞书）中，右键点击信息框后，会弹出特定功能菜单。不同工具可能有相似的功能（如复制、收藏），但具体实现的行为和表现形式不同。例如：

* **微信**：提供功能如“放大阅读、翻译、搜一搜”等。
* **QQ**：提供功能如“全部选择功能、发送到、显示比例功能”等。
* **飞书**：提供功能如“创建话题、Pin、添加任务”等。
* **钉钉：**提供功能如“DING、执行日程、搜一搜”等。

由于这些聊天工具的右键菜单功能具有多样性和灵活性，难以直接使用固定逻辑实现。需要一种设计模式使得可以方便扩展和定制功能。





#### 方案描述

在现代聊天工具中，右键菜单提供了一种快速访问特定功能的便捷方式。由于不同聊天工具可能具有不同的功能和用户界面，使用策略模式可以灵活地管理和扩展这些右键菜单的功能。以下是使用策略模式设计聊天工具右键菜单的方案描述：

1. 定义抽象策略类（Strategy），定义一个抽象策略类，为右键菜单中的每个功能定义一个接口。这个接口可以包含一个方法，比如 execute()，用于执行具体的功能。

2. 创建具体策略类（Concrete Strategy），为右键菜单中的每个功能实现具体的策略类。这些类实现了抽象策略类中的接口，并提供了具体的功能实现。

3. 环境类（Context）负责维护一个策略对象的引用，提供方法来设置和执行策略。这个类使得客户端可以在运行时动态地改变策略。

4. 客户端使用策略，可以根据用户的选择或配置，动态地设置环境类的策略对象。这样，当用户触发右键菜单时，环境类将执行相应的策略。

5.策略模式的核心思想，定义一系列算法，将每种算法封装到独立的策略类中，并使这些策略类可以互换。在本案例中，每种右键功能（如复制、翻译、撤回）都被视为一个策略类。

#### 模式说明与优点

行为设计模式的策略模式，它使得算法的变化独立于使用算法的客户端，被用于定义一系列算法（右键菜单功能），把它们一个个封装起来，并且使它们可以相互替换。这种模式让算法的变化不会影响到使用算法的客户。

**设计结构：**

1. **环境类**：

负责根据具体的聊天工具应用不同的功能策略，并动态调用具体功能。

1. **抽象策略类**：

定义功能策略的接口，统一所有功能的行为。具体策略类将实现这个公共接口。

3.**具体策略类**：

实现了抽象策略定义的接口，为不同功能（如复制、翻译、搜一搜、清屏、创建话题等）提供具体的实现。

**设计优点：**

1. **算法的独立性**：

策略模式允许算法独立于使用它的客户端变化，使得算法可以独立变化。

1. **简化客户端代码**：

客户端不需要知道算法的具体实现，只需要与抽象策略接口交互。

1. **高扩展性**：

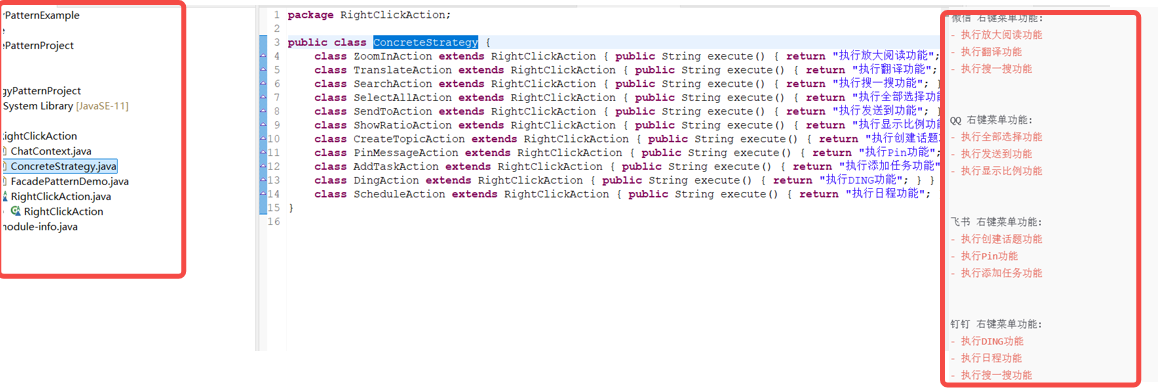
可以方便地添加新功能（如删除、标记等），仅需实现接口。新的算法可以很容易添加到系统中，无需修改现有代码。不同聊天工具共享同样的功能逻辑，无需重复编写代码。

1. **灵活组合**：

不同工具可以灵活选择功能策略，符合实际需求，客户端可以在运行时选择使用哪个具体策略。

### 2成果展示

#### 环境搭建



#### 代码解释

import java.util.ArrayList;

import java.util.List;

**抽象策略类：右键功能接口**

abstract class RightClickAction {

public abstract String execute();

}

**具体策略类：不同功能的具体实现**

class ZoomInAction extends RightClickAction { public String execute() { return "执行放大阅读功能"; } }

class TranslateAction extends RightClickAction { public String execute() { return "执行翻译功能"; } }

class SearchAction extends RightClickAction { public String execute() { return "执行搜一搜功能"; } }

class SelectAllAction extends RightClickAction { public String execute() { return "执行全部选择功能"; } }

class SendToAction extends RightClickAction { public String execute() { return "执行发送到功能"; } }

class ShowRatioAction extends RightClickAction { public String execute() { return "执行显示比例功能"; } }

class CreateTopicAction extends RightClickAction { public String execute() { return "执行创建话题功能"; } }

class PinMessageAction extends RightClickAction { public String execute() { return "执行Pin功能"; } }

class AddTaskAction extends RightClickAction { public String execute() { return "执行添加任务功能"; } }

class DingAction extends RightClickAction { public String execute() { return "执行DING功能"; } }

class ScheduleAction extends RightClickAction { public String execute() { return "执行日程功能"; } }

**环境类：负责调用策略**

class ChatContext {

List<RightClickAction> actions;

public ChatContext(List<RightClickAction> actions) {

this.actions = actions;

}

public void show\_menu() {

System.out.println("聊天工具右键菜单功能:");

for (RightClickAction action : actions) {

System.out.println("- " + action.execute());

}

}

}

**客户端：**

public class FacadePatternDemo {

public static void main(String[] args) {

ChatContext wechatMenu = new ChatContext(new ArrayList<RightClickAction>() {{

add(new ZoomInAction());

add(new TranslateAction());

add(new SearchAction());

}});

ChatContext qqMenu = new ChatContext(new ArrayList<RightClickAction>() {{

add(new SelectAllAction());

add(new SendToAction());

add(new ShowRatioAction());

}});

ChatContext feishuMenu = new ChatContext(new ArrayList<RightClickAction>() {{

add(new CreateTopicAction());

add(new PinMessageAction());

add(new AddTaskAction());

}});

ChatContext dingtalkMenu = new ChatContext(new ArrayList<RightClickAction>() {{

add(new DingAction());

add(new ScheduleAction());

add(new SearchAction());

}});

wechatMenu.show\_menu();

System.out.println("\n");

qqMenu.show\_menu();

System.out.println("\n");

feishuMenu.show\_menu();

System.out.println("\n");

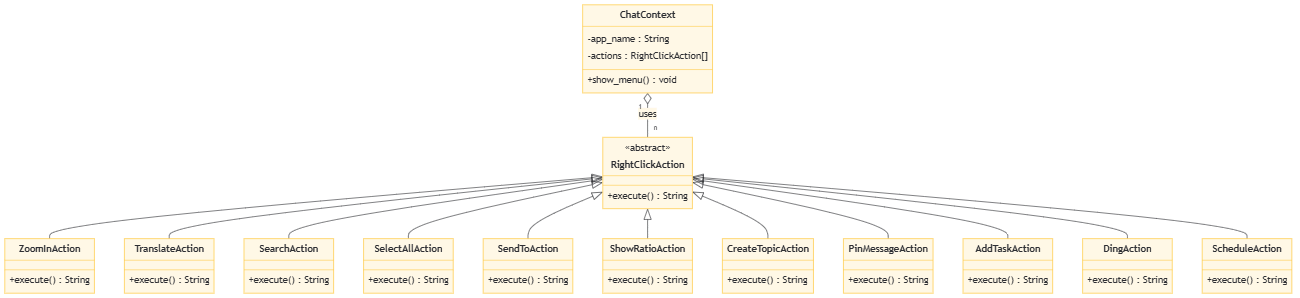
dingtalkMenu.show\_menu();

}

}

#### 类图绘制

**关系说明**

* **RightClickAction（抽象策略类）**：定义了右键菜单功能的接口，所有具体策略类都必须实现这个接口。
* **具体策略类**：包括 ZoomInAction、TranslateAction、SearchAction、SelectAllAction、SendToAction、ShowRatioAction、CreateTopicAction、PinMessageAction、AddTaskAction、DingAction 和 ScheduleAction。这些类实现了 RightClickAction 接口，提供了不同右键菜单功能的具体实现。
* **ChatContext（环境类）**：负责调用策略。它持有一个聊天工具的名称和一个策略列表。show\_menu() 方法遍历策略列表，调用每个策略的 execute() 方法来显示右键菜单的功能列表。
* **具体聊天工具的右键菜单配置**：包括 create\_wechat\_menu()、create\_qq\_menu()、create\_feishu\_menu() 和 create\_dingtalk\_menu()。这些函数分别创建了微信、QQ、飞书和钉钉的右键菜单配置，通过组合不同的具体策略类来实现。

在这个策略模式中，不同的聊天工具可以通过组合不同的具体策略类来实现各自的右键菜单功能，而不需要修改环境类 ChatContext 的代码。这样，新的右键菜单功能可以通过添加新的具体策略类来实现，而不会影响现有的代码，从而满足了开闭原则。

### 3结论归纳

#### 总结与心得体会

在项目中，策略模式被应用于实现一个灵活且可扩展的右键菜单功能系统。

策略模式有效地将客户端与具体策略的实现解耦合。这意味着，如果需要添加新的右键菜单功能，只需添加一个新的具体策略类而无需修改现有代码。同时通过策略模式，可以轻松地为应用程序添加新的行为，而不影响现有功能。这使得系统更加模块化，易于管理和扩展。

策略模式提供了一种灵活的方式来动态选择算法或行为。在项目中，这意味着可以根据不同的上下文或用户偏好动态地改变右键菜单的行为。通过将行为封装在独立的类中，策略模式使得代码更加清晰和易于理解。每个具体策略类负责一个特定的右键菜单功能，这简化了代码的维护。

环境类（`ChatContext`）提供了一个简洁的接口来访问所有策略。这简化了客户端代码，因为客户端不需要知道具体策略的实现细节。

通过这个项目，我体会到了设计模式在解决特定问题上的重要性。策略模式不仅提高了代码的可读性和可维护性，还提高了系统的灵活性。策略模式提高了代码的重用性。通过定义统一的接口，可以创建多个实现该接口的类，从而在不同情况下重用这些类。

虽然策略模式的概念和实现相对直观，但正确地应用它需要对设计模式有深入的理解。这个项目提供了一个实践这些概念的机会。策略模式鼓励开发者在设计阶段就考虑多种可能的行为，这有助于在项目初期做出更灵活的架构决策。

通过这个项目，我更加深入地理解了策略模式的价值，以及它如何在实际开发中应用来提高代码质量和系统的可维护性。