1. 团队和项目基本信息

团队名称：

Design Innovators Team

团队成员：

李云智 赵子毅 文达

软件项目名称：

Smart Restaurant Order Management System

---

2. 软件项目简介

项目背景：

随着餐饮行业的快速发展和顾客对服务效率要求的提升，许多餐厅开始采用智能化的点餐和订单管理系统。本项目旨在开发一个高效、灵活且易于维护的智能餐厅订单管理系统，能够适应不同餐厅的业务需求，并为顾客提供流畅的点餐体验。

主要功能：

1. **订单管理**：
    - 顾客可以通过菜单选择食品并生成订单。
    - 系统支持订单的修改和删除功能。
2. **食品管理**：
    - 支持多种食品类型（如炸鸡、披萨、薯条等）的管理。
    - 每种食品类型具有独特的属性（如辣度、半径、厚度等）。
3. **折扣系统**：
    - 系统支持多种折扣策略（如无折扣、满减折扣、会员折扣等）。
    - 可以动态调整订单的折扣状态。
4. **PDF 报告生成**：
    - 系统提供订单报告的导出功能，支持多种格式（基于 PDFBox 和 iText 实现）。
5. **历史记录**：
    - 通过 Memento 模式实现订单状态的保存与恢复功能，支持多级撤销。

项目目标：

通过运用多种设计模式对原项目进行重构，以提升系统的模块化设计、可维护性和扩展性。重构后的系统将能够更高效地适应不同业务场景需求。

3.软件设计模式详解

Adapter 模式的应用

1. 修改的源代码文件

Adapter 模式的应用主要集中在以下源代码文件中：

- `Fried_chickenAdapter.java`：适配器类，封装了炸鸡的额外属性（辣度）的提取逻辑。
- `PizzaAdapter.java`：适配器类，封装了披萨的额外属性（半径）的提取逻辑。
- `French_friesAdapter.java`：适配器类，封装了薯条的额外属性（厚度）的提取逻辑。

2. 重构前后的主要差异

未重构前的设计： 在未使用 Adapter 模式时，所有食品类型的特殊属性（如辣度、半径和厚度）解析逻辑直接写在主程序中。这种实现方式存在以下问题：

1. **代码重复**：每次解析特定食品的属性时，都需要重复实现逻辑。
2. **扩展性差**：如果需要添加新的食品类型或解析逻辑，则需要修改主程序。
3. **耦合性高**：食品属性解析与主程序的核心逻辑耦合在一起，难以维护。

示例未重构代码：

```
Element foodElement = ...; // 从 jsoup 获取 HTML 元素
String spiciness = foodElement.getElementsContainingText("Spiciness: ").text();
spiciness = spiciness.substring(11);
Fried_chicken chicken = new Fried_chicken();
chicken.setSpiciness(spiciness);
chicken.TemplateMethod(foodElement);
```

重构后的设计： 通过使用 Adapter 模式，将每种食品类型的额外属性解析逻辑封装到对应的适配器类中。主程序只需要调用适配器，而无需直接处理解析逻辑。

- 每种食品类型都对应一个适配器类（如 `Fried_chickenAdapter`）。
- 在适配器类中实现了 `setAddition` 方法，封装了解析逻辑。
- 提供模板方法 `TemplateMethod`，在调用父类方法的基础上添加属性解析功能。

重构后的代码示例：

```
Fried_chickenAdapter chickenAdapter = new Fried_chickenAdapter();
chickenAdapter.TemplateMethod(foodElement);
```

3. UML 类图

未重构前的设计：


Ad类1

重构后的设计：


Ad类2

---

4. 改动原因与好处

改动原因：

1. **单一职责原则**：解析逻辑和业务逻辑分离，使每个类的职责更加明确。
2. **开闭原则**：当需要支持新的食品类型或属性时，可以通过增加新的适配器类实现扩展，而不需要修改主程序。

好处：

1. **模块化设计**：将食品属性解析逻辑集中在适配器类中，避免主程序中混杂多种逻辑。
2. **提高可扩展性**：添加新食品类型时，只需实现新的适配器类即可。
3. **可维护性提升**：如果解析逻辑需要修改，只需调整适配器中的代码，无需修改主程序。
4. **代码复用性**：不同食品类型共享相同的适配器模式实现，减少代码重复。

Interpreter 模式的应用

1. 修改的源代码文件

Interpreter 模式的应用主要集中在以下源代码文件中：

- 辣度相关：
  - `SpicinessExpression.java`：定义了辣度的表达式接口，包含 `interpret` 和 `greatThan` 方法。
  - `SpicinessTerminalExpression.java`：终结符表达式类，封装了具体的辣度解析和赋值逻辑。
  - `SpicinessNonterminalExpression.java`：非终结符表达式类，组合多个终结符表达式，递归解析辣度属性，并实现辣度的比较功能。
- 厚度相关：
  - `ThicknessExpression.java`：定义了厚度的表达式接口，包含 `interpret` 和 `greatThan` 方法。
  - `ThicknessTerminalExpression.java`：终结符表达式类，封装了具体的厚度解析和赋值逻辑。
  - `ThicknessNonterminalExpression.java`：非终结符表达式类，组合多个终结符表达式，递归解析厚度属性，并实现厚度的比较功能。

2. 重构前后的主要差异

未重构前的设计： 在未使用 Interpreter 模式之前，辣度和厚度的解析和比较逻辑直接嵌入到主程序或相关类中。这种设计存在以下问题：

1. **代码重复**：每次解析和比较辣度或厚度时，都需要实现重复的代码逻辑。
2. **扩展性差**：如果需要支持新的属性解析或更复杂的比较逻辑，需要修改主程序。
3. **耦合性高**：属性解析与核心业务逻辑强耦合，维护起来困难。

示例未重构代码：

```
// 辣度比较逻辑
String spiciness1 = "Medium";
```

```
String spiciness2 = "Hot";
int value1 = 0;
int value2 = 0;

if (spiciness1.equals("Mild")) value1 = 1;
if (spiciness1.equals("Medium")) value1 = 2;
if (spiciness1.equals("Hot")) value1 = 3;

if (spiciness2.equals("Mild")) value2 = 1;
if (spiciness2.equals("Medium")) value2 = 2;
if (spiciness2.equals("Hot")) value2 = 3;

boolean isSpicier = value1 > value2;
```

重构后的设计： 使用 Interpreter 模式后：

1. 定义了统一的接口（如 `SpicinessExpression` 和 `ThicknessExpression`），将解析和比较逻辑抽象化。
2. 解析逻辑被封装在终结符表达式（如 `SpicinessTerminalExpression`）中，每个终结符专注于特定属性的解析。
3. 比较逻辑被封装在非终结符表达式（如 `SpicinessNonterminalExpression`）中，支持组合和递归解析。
4. 主程序调用表达式接口，而无需直接操作解析和比较逻辑。

重构后的代码示例：

```
SpicinessTerminalExpression mild = new SpicinessTerminalExpression("Mild", 1);
SpicinessTerminalExpression medium = new SpicinessTerminalExpression("Medium",
2);
SpicinessNonterminalExpression spicinessExpression = new
SpicinessNonterminalExpression(mild, medium);

Spiciness spiciness1 = new Spiciness("Medium");
Spiciness spiciness2 = new Spiciness("Hot");

boolean isSpicier = spicinessExpression.greatThan(spiciness1, spiciness2) > 0;
```

---

3. UML 类图

未重构前的设计：

重构后的设计：

---

4. 改动原因与好处

改动原因：

1. **单一职责原则**：将辣度和厚度的解析和比较逻辑抽离到专门的表达式类中，使每个类的职责更加明确。
2. **开闭原则**：当需要添加新的解析逻辑或属性时，只需增加新的表达式实现类，无需修改主程序。
3. **组合模式的灵活性**：通过非终结符表达式组合多个终结符表达式，实现递归解析和逻辑复用。

好处：

1. 模块化设计：
   - 解析和比较逻辑集中管理，主程序逻辑清晰且简洁。
2. 扩展性提升：
   - 轻松添加新的属性（如重量）或新的逻辑（如多级解析）。
3. 可维护性增强：
   - 属性值域或解析逻辑变化时，仅需修改对应的表达式类，无需影响主程序。
4. 代码复用性：
   - 不同食品类型可以共享相同的表达式逻辑。

Memento 模式的应用

1. 修改的源代码文件

Memento 模式的实现主要涉及以下源代码文件：

1. `OrderHistory.java`
   - 维护一个历史栈，用于存储 `OrderMemento` 实例。
   - 提供 `push` 和 `pop` 方法，分别用于保存和恢复订单的状态。
2. `OrderMemento.java`
   - 封装订单的状态，包括食品列表。
   - 提供获取已保存食品列表的方法，确保状态的封装性。
3. `Order.java`
   - 代表订单类，负责管理食品列表和折扣状态。
   - 提供 `save` 方法将当前订单状态保存为一个 `OrderMemento` 实例。
   - 提供 `restore` 方法，从 `OrderMemento` 恢复订单状态。

2. 重构前后的主要差异

未重构前的设计： 在未使用 Memento 模式之前，订单的撤销功能可能需要直接操作订单对象，或者通过手动保存食品列表的副本来实现。这种设计存在以下问题：

1. **代码重复**：每次需要保存订单状态时，都需要重复实现保存逻辑。
2. **数据不安全**：状态直接暴露，可能导致不必要的修改。
3. **难以扩展**：无法轻松支持多级撤销或复杂的状态管理。

示例未重构代码：

```
// 保存订单状态
List<Food> previousState = new ArrayList<>(order.getFoods());

// 恢复订单状态
order.getFoods().clear();
order.getFoods().addAll(previousState);
```

重构后的设计： 通过引入 Memento 模式：

1. 将订单状态的保存和恢复逻辑封装在 `OrderMemento` 和 `OrderHistory` 类中。
2. 主程序调用 `Order` 类的 `save` 和 `restore` 方法，而无需直接操作状态，确保了状态的封装性。
3. 支持多级撤销，通过 `OrderHistory` 的栈结构管理状态。

重构后的代码示例：

```
Order order = new Order();
order.addFood(new Food("Pizza", 10));
OrderHistory history = new OrderHistory();

// 保存当前状态
history.push(order.save());

// 修改订单
order.addFood(new Food("Burger", 5));

// 恢复之前的状态
order.restore(history.pop());
```

---

3. UML 类图

未重构前的设计：


M类1

重构后的设计：


M类2

---

4. 改动原因与好处

改动原因：

1. **封装状态管理**：通过 Memento 模式将状态的保存和恢复逻辑封装在独立的类中，避免主程序直接操作状态。
2. **支持多级撤销**：通过栈结构的 `OrderHistory` 类，轻松实现多级撤销功能。
3. **确保状态安全性**：状态的保存和恢复通过 `OrderMemento` 实现，外部无法直接修改状态，确保了数据的完整性。

好处：

1. 模块化设计：
    - 状态保存和恢复逻辑独立于主程序，提高代码的可读性和可维护性。

2. 扩展性增强：
    - 可以轻松扩展支持更多状态管理功能，例如多级撤销、状态日志等。
3. 提升安全性：
    - 状态通过 Memento 对象封装，外部无法直接修改订单数据。
4. 代码复用性：
    - Memento 和 Order 类的设计可以复用于其他需要状态管理的场景。

Facade 模式的应用

1. 修改的源代码文件

Facade 模式的实现主要涉及以下源代码文件：

1. `ITextPDFGenerator.java`
    - 使用 iText 库生成 PDF 报告，封装了具体的 PDF 生成逻辑。
2. `PDFBoxGenerator.java`
    - 使用 Apache PDFBox 库生成 PDF 报告，封装了具体的 PDF 生成逻辑。
3. `PDFGeneratorFacade.java`
    - 提供一个统一的接口，屏蔽了底层的 PDF 生成实现细节。
    - 通过简单的接口调用即可选择不同的 PDF 生成器（如 iText 或 PDFBox）。

---

2. 重构前后的主要差异

未重构前的设计： 在未使用 Facade 模式之前，主程序需要直接与具体的 PDF 生成库交互，并且代码中可能会包含对多个生成器的复杂逻辑。这种设计存在以下问题：

1. **耦合性高**：主程序直接依赖多个具体的 PDF 生成实现。
2. **代码重复**：每次生成 PDF 报告时都需要重复实现生成逻辑。
3. **扩展性差**：如果需要支持新的 PDF 生成器，需要在主程序中添加大量逻辑。

示例未重构代码：

```
// 使用 iText 生成 PDF 报告
Document document = new Document();
try {
    PdfWriter.getInstance(document, new FileOutputStream("output.pdf"));
    document.open();
    // 添加内容...
    document.close();
} catch (Exception e) {
    e.printStackTrace();
}

// 使用 PDFBox 生成 PDF 报告
PDDocument doc = new PDDocument();
PDPage page = new PDPage();
doc.addPage(page);
try (PDPageContentStream content = new PDPageContentStream(doc, page)) {
```

```
    // 添加内容...
}
doc.save("output.pdf");
doc.close();
```

重构后的设计： 通过使用 Facade 模式：

1. 将不同的 PDF 生成器封装为独立的类（`ITextPDFGenerator` 和 `PDFBoxGenerator`）。
2. 提供一个统一的 `PDFGeneratorFacade` 类，让主程序只需通过简单的接口调用即可生成 PDF 报告。
3. 支持动态扩展新的 PDF 生成器，而无需修改主程序。

重构后的代码示例：

```
PDFGeneratorFacade pdfFacade = new PDFGeneratorFacade();
Order order = new Order();
// 添加订单内容...
pdfFacade.generateReport("itext", order, "output_itext.pdf");
pdfFacade.generateReport("pdfbox", order, "output_pdfbox.pdf");
```

---

3. UML 类图

未重构前的设计：

 Fa类1

重构后的设计：

 Fac类2

---

4. 改动原因与好处

改动原因：

1. **降低耦合性**：通过 `PDFGeneratorFacade` 隐藏了底层实现，主程序无需直接依赖具体的 PDF 生成库。
2. **提高扩展性**：可以通过添加新的 PDF 生成器实现类，并在 Facade 中注册来实现扩展。
3. **简化调用流程**：主程序只需通过简单的接口调用即可生成 PDF 报告，无需处理复杂的底层逻辑。

好处：

1. 模块化设计：
    - 各 PDF 生成器的实现被封装到独立的类中，便于维护和替换。
2. 提高可扩展性：
    - 支持动态添加新的 PDF 生成器，而无需修改主程序逻辑。
3. 提高可维护性：
    - 生成器的更改不会影响主程序，只需修改 Facade 类即可。
4. 统一接口调用：
    - 提供一致的接口，让主程序专注于核心逻辑，而非底层实现细节。

Factory 模式的应用

1. 修改的源代码文件

Factory 模式的实现主要集中在以下源代码文件中：

1. `FoodJsoupFactory.java`
    - 实现了一个具体的工厂类，用于通过 `Jsoup` 库解析指定的 HTML 文件（`food.html`），并返回解析后的 `Document` 对象。
    - 隐藏了文件解析的细节，让调用者只需要通过工厂获取 HTML 文档，而无需直接依赖 `Jsoup` 的实现。

2. 重构前后的主要差异

未重构前的设计： 在未使用 Factory 模式之前，主程序需要直接调用 `Jsoup` 库的解析方法来获取 HTML 文档。这种设计存在以下问题：

1. **代码重复**：每次需要解析 HTML 文件时，都需要重复实现解析逻辑。
2. **耦合性高**：主程序直接依赖 `Jsoup` 库，导致代码对具体解析工具的耦合。
3. **扩展性差**：如果需要支持其他数据源（如 JSON 文件或 API 数据），需要修改主程序逻辑。

示例未重构代码：

```
// 主程序直接使用 Jsoup 库解析 HTML
Document document = Jsoup.parse(new File("path/to/food.html"), "UTF-8");
// 直接操作 document 对象
Elements elements = document.select("food");
for (Element element : elements) {
    // 处理元素...
}
```

重构后的设计： 通过引入 Factory 模式：

1. 提供一个 `JsoupFactory` 接口，定义获取 `Document` 对象的方法。
2. 创建具体工厂类 `FoodJsoupFactory`，封装 HTML 文件的解析逻辑。
3. 主程序通过工厂接口获取解析后的 `Document`，实现了解耦。

重构后的代码示例：

```
JsoupFactory factory = new FoodJsoupFactory();
Document document = factory.getDocument();
// 主程序只需要处理解析后的 document 对象
Elements elements = document.select("food");
for (Element element : elements) {
    // 处理元素...
}
```

---

3. UML 类图

未重构前的设计：

 F类1

重构后的设计：

 F类2

---

4. 改动原因与好处

改动原因：

1. **解耦主程序与具体实现**：通过 Factory 模式，主程序只依赖工厂接口，而不直接依赖具体的解析工具（如 `Jsoup`）。
2. **支持多种数据源**：通过不同的工厂实现，可以轻松支持从 HTML 文件、JSON 文件或 API 数据中提取信息，而无需修改主程序。

好处：

1. 模块化设计：
   - HTML 解析逻辑被封装到具体的工厂类中，主程序专注于业务逻辑。
2. 提高扩展性：
   - 未来如果需要支持其他数据源（如 JSON 或 XML 文件），只需创建新的工厂实现类。
3. 提高可维护性：
   - 如果解析逻辑需要修改，只需修改具体的工厂类，而无需修改主程序。
4. 代码复用性：
   - 不同数据源的解析逻辑可以通过工厂模式独立实现并复用。

State 模式的应用

1. 修改的源代码文件

State 模式的实现主要涉及以下源代码文件：

1. `Order.java`
   - 管理订单的折扣状态，使用 `DiscountState` 接口实现动态状态切换。
   - 提供方法 `setDiscountState` 用于设置当前的折扣状态，并通过 `getTotal` 调用当前状态的折扣逻辑。
2. `DiscountState`（假设的接口，未提供实现）
   - 定义了 `calculateTotal` 方法，用于不同折扣状态下计算订单总价。
3. `NoDiscountState`（假设的具体状态，未提供实现）
   - 表示无折扣的状态，直接返回食品总价。
4. 其他假设的状态实现（如 `FiftyPercentOffState`, `OverHundredDiscountState`）
   - 表示不同的折扣策略，例如特定商品半价或满减。

2. 重构前后的主要差异

未重构前的设计： 在未使用 State 模式之前，折扣逻辑通常直接通过条件语句（`if-else` 或 `switch`）实现。这种设计存在以下问题：

1. **代码难以维护**：随着折扣策略的增加，`if-else` 或 `switch` 逻辑会变得复杂且难以阅读。
2. **扩展性差**：添加新的折扣逻辑需要修改主程序，违反了开闭原则。
3. **耦合性高**：订单类与具体的折扣逻辑耦合，降低了代码的模块化程度。

示例未重构代码：

```
public float calculateTotal(List<Food> foods, String discountType) {
    float total = 0;
    for (Food food : foods) {
        total += food.getPrice();
    }

    if (discountType.equals("NO_DISCOUNT")) {
        return total;
    } else if (discountType.equals("FIFTY_PERCENT_OFF")) {
        return total * 0.5f;
    } else if (discountType.equals("OVER_HUNDRED_DISCOUNT") && total > 100) {
        return total - 50;
    }
    return total;
}
```

重构后的设计： 通过引入 State 模式：

1. 定义了 `DiscountState` 接口，封装不同折扣策略的实现。
2. 订单类 `Order` 持有一个 `DiscountState` 对象，表示当前的折扣状态。
3. 调用 `getTotal` 方法时，根据当前的 `DiscountState` 动态计算总价。
4. 添加新的折扣策略只需实现新的 `DiscountState` 类，而无需修改 `Order` 类的代码。

重构后的代码示例：

```
Order order = new Order();
order.addFood(new Food("Pizza", 10));
order.addFood(new Food("Burger", 15));

// 使用无折扣状态
order.setDiscountState(new NoDiscountState());
System.out.println("Total with no discount: " + order.getTotal());

// 使用满减折扣状态
order.setDiscountState(new OverHundredDiscountState());
System.out.println("Total with over-hundred discount: " + order.getTotal());
```

3. UML 类图

未重构前的设计：

 s类1

重构后的设计：

 s类2

---

4. 改动原因与好处

改动原因：

1. **解耦逻辑**：通过将折扣逻辑封装到独立的状态类中，减少了订单类与具体逻辑的耦合。
2. **增强扩展性**：可以轻松添加新的折扣策略，而无需修改现有代码，符合开闭原则。
3. **提高代码可读性**：避免了复杂的 `if-else` 或 `switch` 语句，使得代码更加简洁。

好处：

1. 模块化设计：
   - 折扣逻辑被封装到独立的状态类中，订单类只负责管理状态。
2. 提高可扩展性：
   - 添加新的折扣策略时，只需实现新的状态类，并通过 `setDiscountState` 方法切换状态。
3. 提高可维护性：
   - 订单类的逻辑更加简洁，状态类专注于各自的折扣逻辑，便于调试和维护。
4. 动态行为支持：
   - 通过切换状态对象，可以动态改变订单的折扣逻辑，而无需修改代码。

4. 相关补充信息

4.1 重构中遇到的挑战与解决方案

1. **挑战**：重构过程中，需要在保持系统功能正常运行的同时进行代码大规模的重组。
   - 解决方案：
     - 采用增量式重构方法，将系统按功能模块拆分，并逐步应用设计模式重构。
     - 使用单元测试验证重构后的代码功能。
2. **挑战**：在应用多种设计模式时，保证不同模式之间的协作不会引入额外的复杂性。
   - 解决方案：
     - 使用 UML 类图分析不同模式之间的依赖关系。
     - 定期进行团队代码审查，确保设计的一致性。

4.2 项目成果总结

通过本次重构，系统在以下方面取得了显著提升：

- **模块化设计**：重构后的系统通过引入多种设计模式，实现了各模块之间的低耦合和高内聚。
- **可扩展性**：新增功能时，只需增加新的实现类，而无需修改已有代码。
- **可维护性**：逻辑清晰、职责单一的设计使得系统更易于维护。
- **动态行为支持**：通过 State 和 Interpreter 模式，系统能够灵活处理不同状态和属性解析。