2024 年春季学期软件构造

Lab2 实验报告

班号	2211104	学号	2022211917	姓名	周雨凡
开始	2024.6.8	截止	2024.6.15 17：00 截止，电子版 18：00 截止，超时拒收		

目录

1. 实验目标概述

这次实验旨在教授关于软件构造中可复用性和可维护性中的开闭原则的概念和技术。通过实践，学生将学习如何使用抽象数据类型（ADT）及其实现来解决应用问题，涉及多种面向对象的编程概念和设计原则。具体来说，学生将会掌握子类型、泛型、多态、重写、重载、继承、代理和组合等概念，以及面向对象的七原则（SOLID+2EX）。

在实验中，学生将面对两个开放式问题，分别是文件解析和饮料计费系统。对于每个问题，学生需要设计和实现相应的 ADT，并编写适当的说明、规范和测试用例。在设计过程中，要求学生充分考虑未来可能的变化，并确保设计满足开闭原则和复用性。通过这些实践，学生将培养对软件设计和开发中重要概念的理解，以及解决实际问题的能力。

2. 实验环境配置

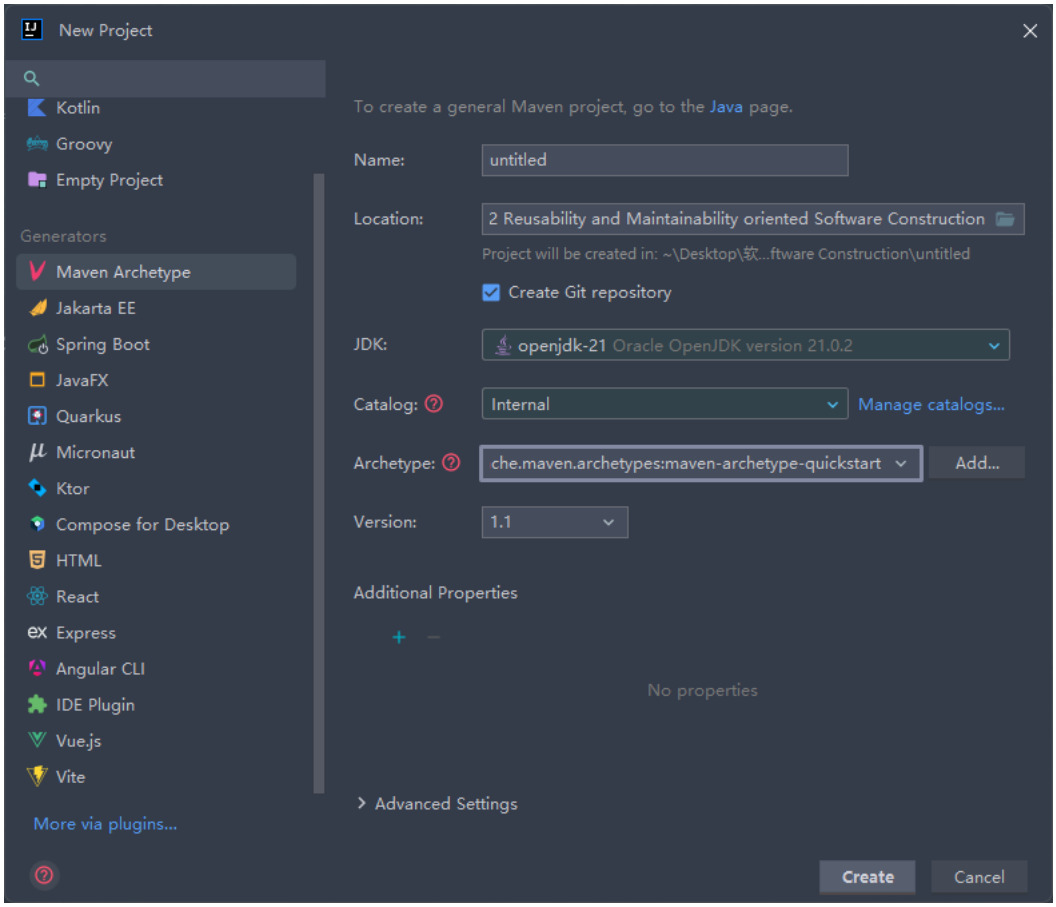

图2-1 实验环境配置

创建时选择 Maven Archetype 创建，在 Catalog 选择 Internal，Archetype 选择 quickstart。

查阅资料得知：

Maven Archetype 是 Maven 项目管理工具中的一个功能，用于快速创建项目的模板或原型。在创建项目时选择 Maven Archetype，是为了使用预先定义好的项目结构和配置，从而加快项目的启动和开发过程。

Catalog 是 Maven Archetype 的一个选项，它指定了 Archetype 的来源。在这种情况下，选择 Internal 意味着使用本地或内部的 Archetype 目录，其中包含了用户自定义或组织内部定义的 Archetype。

而在选择 Archetype 时，选中了 quickstart，这意味着创建一个快速启动的 Maven 项目，该项目包含了基本的结构和配置，可以立即开始编写代码而不必手动配置项目结构。

3. 实验过程

3.1. 待开发的应用场景文件解析（必做）

3.1.1. 设计

简单工厂模式（Simple Factory Pattern）是一种创建对象的设计模式，它定义了一个工厂类，该类可以根据传入的参数决定创建哪种类的实例。虽然不属于 GOF（四人帮）定义的 23 种经典设计模式之一，但它在实际应用中非常常见，被认为是工厂方法模式的一种变体。

这个问题我将使用设计模式中的简单工厂模式。简单工厂模式用于创建对象，但不暴露创建逻辑给客户端，并允许客户端通过使用同一个方法来创建对象。但这种设计模式有一种致命的缺陷，便是不易扩展。每增加一种新产品都需要修改工厂类，违背了开闭原则（对扩展开放，对修改封闭）。而课件中通过添加配置文件的方式解耦，也只是满足了对于工厂类和实现类的 OCP，对于配置文件本身并不满足这个约定；而且每次添加新的类需要手动向配置文件中添加信息，中间的选择器工厂还要扫描配置文件来获取实例类，过于繁琐。（在我的理解看来，对于配置文件的修改实际上比对于代码的修改更优，因为代码内部存在耦合的情况，某部分代码可能与其他地方的代码相互关联，甚至更改任意代码会造成逻辑状态的改变，而对于配置文件仅仅是信息的更改）

解决办法：我设计将直接跳过配置文件这个中间过程，使选择器工厂直接扫描文件目录中的文件来获取实例类，方便快捷，时间复杂度仍是 O（n）。以下函数是我的核心思想，使用泛型静态函数扫描得到 Map 存储的文本标签 Label 与 fileName 的一一对应的映射，当客户端输入 Label 的时候，通过 Map 查询创建具体类的对象，来实现选择功能。通过在选择器类的构造函数中调用这个函数，客户端可以通过一次扫描，就可以一直保留信息重复使用。

核心实现代码样例（文件位于 src/main/java/p2/util/FactoryLoader.java）：

```
public static <T> Map<String, T> scanAndLoadFactories(String directoryPath, String
suffix, String packageName, Class<T> factoryClassType) {
    Map<String, T> factoryMap = new HashMap<>();
    File directory = new File(directoryPath);
    if (!directory.exists() || !directory.isDirectory()) {
        throw new IllegalArgumentException("目录路径无效：  " + directoryPath);
    }

    String[] fileNames = directory.list((dir, name) => name.endsWith(suffix+".java"));
    if (fileNames == null || fileNames.length == 0) {
        throw new IllegalArgumentException("指定目录中没有找到工厂类");
    }
```

```
    for (String fileName : fileNames) {
        try {
            String className = fileName.substring(0, fileName.indexOf(suffix));
            String factoryClassName = packageName + "." + className + suffix;
            Class<?> factoryClass = Class.forName(factoryClassName);
            T factoryInstance =
factoryClassType.cast(factoryClass.getDeclaredConstructor().newInstance());
            factoryMap.put(className.toLowerCase(), factoryInstance);
        } catch (Exception e) {
            System.err.println("无法实例化工厂类：" + fileName + "。错误信息：" +
e.getMessage());
        }
    }
    return factoryMap;
}
```

此设计方法不仅可以封装为类做 API 在 Java 内部使用，还可以作为接口的方式接受前端传递的 json 数据，不论如何，都可以满足在不使用 if-else 的情况下，最大化优化代码逻辑，相较于工厂模式更加适应前后端交互的体验。

相较于工厂模式，在客户端中直接调用的方式，这种方法造成了复杂度的上升到 O(n)，毕竟在工厂模式的环境中可以直接对特定的构造函数进行选择，在这种情况下使用简单工厂方法显然要逊色不少；但是在作为参数的传递上，或者说将选择权交给用户方面，这种方法显然更好。

设计实现过程：

1. 识别类：首先，需要识别出构成系统的类。在类图中，需要设计以下几个类：
   a) Parser 类是抽象的，它定义了所有解析器的通用接口。这意味着任何具体的解析器都必须实现 Parser 接口中的所有方法。这使得 Parser 类可以用于解析任何类型的配置文件。
   b) XXXTypeParser 类是 Parser 接口的具体实现。它们提供了用于创建 JSON 和 XML 等多种解析器的具体方法和实现。
   c) ParserSelectorFactory 类通过扫描预定义目录中的特定 Java 文件（以“TypeParser.java”结尾），动态加载和注册不同的解析器类。它在构造时初始化一个映射，将文件类型与对应的解析器实例关联起来，以便后续可以根据文件类型选择正确的解析器来解析文件。如果在扫描或实例化过程中遇到问题，它会捕获异常并输出错误信息。
   d) Client 类使用 ParserSelectorFactory 类来创建解析器。这意味着 Client 类可以用于解析任何类型的配置文件，而不必知道具体的配置文件类型。
2. 定义类之间的关系：接下来，需要定义类之间的关系。在类图中，有以下几个关系：
   a) ParserSelectorFactory 与 Parser 之间是聚合关系。ParserSelectorFactory 维护了一个 parserFactoryMap 映射，其中键是文件类型，值是相应的 XXXTypeParser

实例。ParserSelectorFactory 根据文件名或文件类型选择并返回具体的 Parser 实例（如 JsonTypeParser、XmlTypeParser 或 YmlTypeParser）。

b) XXXTypeParser 与 Parser 之间是继承关系。XXXTypeParser 是 Parser 的子类。

c) Client 与 ParserSelectorFactory 之间是依赖关系。Client 调用 ParserSelectorFactory 解析特定文件，内部包含 Parser 创建。

3. 定义类的属性和方法：

a) ParserSelectorFactory 类：

i. parserFactoryMap：存储解析器工厂映射的属性。

ii. DIRECTORY_PATH：存储解析器文件的目录地址。

iii. parseFile(String filePath)：解析文件的公共方法。

iv. getFileType(String filePath)：获取文件类型的私有方法。

v. ParserSelectorFactory()：构造函数。

vi. scanAndLoadParser()：搜索解析器文件类型，将特定 String 类型与 XXXTypeParser 对应。

b) XXXTypeParser 类：

i. parser(String path)：创建解析器的具体方法。

ii. printNode(Node node)：打印处理内容的辅助函数。

c) Parser 抽象类：

i. parse()：解析文件的抽象方法。

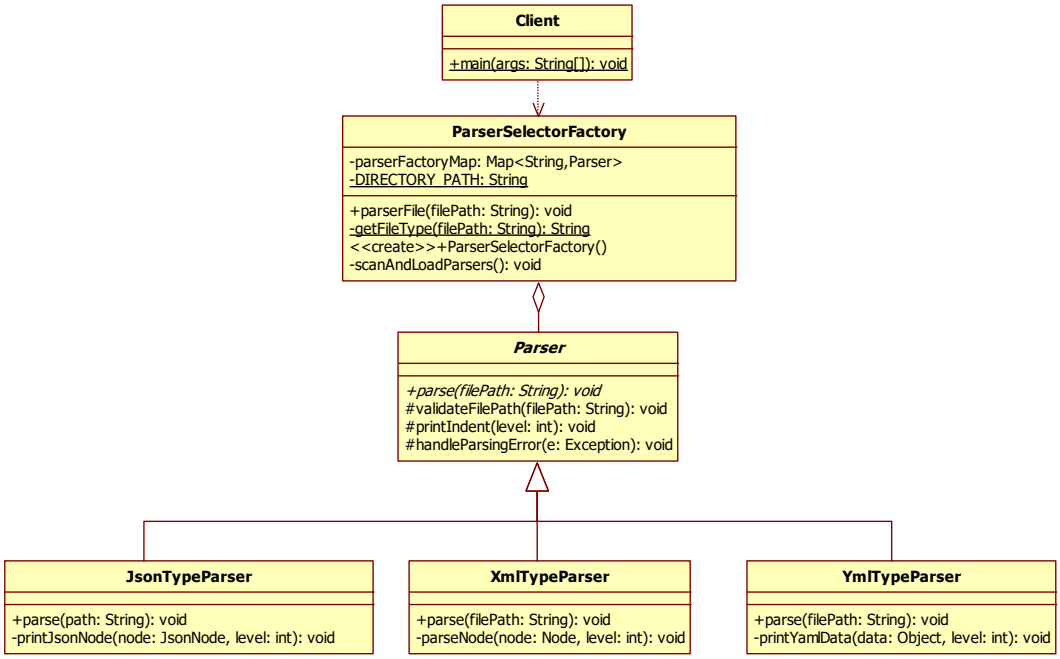

图3.1.1-1 设计类图（UML/p1.jpg）

设计原理解释（个人理解）：从本质上讲，此种方法并不是完全消除了配置文件的作用，而是将配置文件转化成了一参数与名称对应的映射。这种映射在最开始设计的时候就已经确定好了，他的存在如同一种配置文件，只不过程序不会去扫描他，而是已经记住了他，如果在我的方法中参数与文件的名称不对应的时候，仍然需要创建一个如同配置文件作用的文件，来管理这种映射。

满足条件：

复用：是指将代码或设计元素在多个地方使用。

在设计蓝图中，Parser 类作为核心的抽象框架，赋予了系统解析多种配置文件类型的能力。这一灵活性的基石正是多态性的运用——通过派生诸如 JsonTypeParser 和 XmlTypeParser 这样的子类，每种子类都能针对特定格式的配置文件提供专业化的解析服务。这种设计策略不仅确保了代码的简洁与高效，同时也为未来可能的扩展预留了广阔空间。

ParserSelectorFactory 的巧妙之处在于它扮演了一位智慧的中间人，成功构建了一道隔离墙，将具体解析器类（包括但不限于 JsonTypeParser 和 XmlTypeParser）与客户端代码优雅地分离开来。通过它，解析器的选择与实例化被统一管理，从而大大简化了客户端的调用流程，同时保证了这一选择机制的高复用性和灵活性。这意味着，当系统面临新类型配置文件的解析需求时，我仅需在工厂中登记相应的解析器类，无需对客户端代码做出任何调整，便能轻松应对，极大地提升了系统的适应能力和维护效率。

相较于之前两点，参数与文件名的映射尽管显得微不足道，但是相同的名称省下了我添加配置文件的功夫，对于小规模的处理显然比增加配置文件更优。

OCP（Open-Closed Principle）：是指软件应该对扩展开放，对修改封闭。这意味着软件应该能够通过扩展新功能来适应新的需求，而不需要修改现有的代码。

在我的设计中，可以通过创建新的 XXXTypeParser 并继承 Parser 实现来扩展新的解析器类型，而无需修改 Parser 类和已经继承实现的子类们，直接继承 Parser 然后重写 @Override 对应的 parser()方法即可。

对于 ParserSelectorFactory 仍然满足 OCP 原则，ParserSelectorFactory 类中没有具体的工厂类构造函数存在，而是使用扫描文件的形式进行加载，根据目录下代码文件对应的文件类型，动态地选择合适的解析器进行文件解析，从而提高了系统的灵活性和可扩展性，在添加新的类时也不需要对其进行修改。

以下是我的设计的一些优点：

可复用性：我的设计可以复用，因为它允许创建新的 XXXTypeParser 实现来扩展新的配置文件类型，复用 Parser 类和 ParserSelectorFactory 中的选择逻辑。

可维护性：我的设计易于维护，因为它将解析器的逻辑封装在 XXXTypeParser 类中。这使得可以轻松地修改 XXXTypeParser 类而不影响其他代码。

可扩展性：我的设计对于添加新类型文件的解析器同样使用，仅需要添加新的解析器子类即可。

总的来说，我们的设计是一个平衡了复用性、可维护性和可扩展性的良好设计。

3.1.2. 实现

类的实现文档 spec 说明：详细描述了一个文件解析器框架的设计和实现。其中包括了 ParserSelectorFactory 类，负责管理解析器工厂并根据文件类型选择合适的解析器进行解析；Parser 类作为解析器的抽象基类，定义了解析文件的接口；而具体的解析器实现类如

JsonTypeParser、XmlTypeParser 和 YmlTypeParser 则分别实现了解析 JSON、XML 和 YML 文件的逻辑。每个类都经过了严格的设计和规范，包括不变性的定义、防止表示暴露的措施以及方法规格说明，以确保代码的可维护性和可靠性。而 Client 类作为简单的客户端类则用于演示如何使用这些解析器来解析不同类型的文件。

a) ParserSelectorFactory 类：
   i. Mutability/Immutability：该类是不可变的（immutable）。虽然内部有一个映射(parserFactoryMap)，该映射在构造函数中初始化并加载解析器，但在类的生命周期中不再被修改。
   ii. 抽象函数 (AF)：
      1. ParserSelectorFactory 类表示从指定目录中读取解析器类并选择合适的工厂来创建解析器的类。
      2. AF(c) = { directoryPath => “src/main/java/p1/Parser”, parserFactoryMap => 映射文件类型到相应的解析器实例 }
   iii. 表示不变性 (RI)：
      1. DIRECTORY_PATH 必须是一个有效的字符串，不能为 null 或空。
      2. parserFactoryMap 应包含从 DIRECTORY_PATH 目录加载的有效解析器类实例。
      3. parserFactoryMap 不能为 null，初始化后不应再修改其内容。
   iv. 防止表示暴露 (Safety from Rep Exposure)：该类没有提供任何方法暴露其内部状态，且内部状态都是私有的，不会被外部修改。parserFactoryMap 是私有的，在构造函数中初始化后不再被修改，也不会直接暴露给外部访问。
   v. 方法规格说明（Spec）：
      1. 构造方法：
         a) Effects：初始化解析器工厂映射，从指定目录中扫描解析器类。如果无法读取目录或无法实例化工厂类，则输出错误信息。
      2. parseFile 方法：
         a) Requires ：filePath 不是 null 或空字符串。
         b) Effects 解析给定文件路径的文件，使用相应的解析器工厂创建解析器进行解析。如果文件类型不受支持，则抛出 IllegalArgumentException 异常。
      3. getFileType 方法：
         a) Requires： filePath 不是 null 或空字符串。
         b) Effects：返回给定文件路径的文件类型（扩展名）。如果文件没有扩展名，返回空字符串。
      4. scanAndLoadParsers 方法：
         a) Requires： DIRECTORY_PATH 是一个有效的目录路径，不能为 null 或空。
         b) Effects： 从指定目录中扫描解析器类文件，并将其实例化为 Parser 实例，存储在 parserFactoryMap 中。如果目录无效或解析器类加载失败，则输出错误信息。
b) Parser 类（抽象类）：
   i. 方法规格说明 (Spec)：
      1. parse 方法 (抽象方法)
         a) Requires： filePath 是一个非空字符串。

b) Effects： 解析给定文件路径所指示的文件。该方法是抽象的，具体的解析逻辑需要在子类中实现。

2. validateFilePath 方法
   a) Requires： filePath 是一个非空字符串。
   b) Effects： 验证文件路径是否有效。如果文件路径为空或文件不存在，抛出 IllegalArgumentException 异常。
3. printIndent 方法
   a) Requires： level 是一个非负整数。
   b) Effects： 打印指定级别的缩进。
4. handleParsingError 方法
   a) Requires： e 是一个非空的异常对象。
   b) Effects： 打印解析过程中发生的错误信息。

c) JsonTypeParser 类：

i. Mutability/Immutability：JsonTypeParser 类是一个具体的解析器实现类，其中包含了解析 JSON 文件的逻辑。由于类的状态在对象创建后不会改变，因此它是不可变的（immutable）。

ii. 抽象函数 (AF)：
   1. JsonTypeParser 类表示一个具体的解析 JSON 文件的解析器。它继承自抽象类 Parser，实现了解析方法 parse 和辅助方法 printJsonNode。
   2. AF(c) = { filePath => 要解析的 JSON 文件路径, objectMapper => 用于解析 JSON 的 ObjectMapper 实例 }

iii. 表示不变性 (RI)：
   1. filePath 必须是一个非空字符串。
   2. 文件路径所指向的文件必须存在且是一个有效的 JSON 文件。

iv. 防止表示暴露 (Safety from Rep Exposure)：该类没有直接暴露内部状态，因此不存在表示暴露的问题。内部的 ObjectMapper 实例是私有的，并且没有公开的访问方法。

v. 方法规格说明 (Spec)：
   1. printJsonNode 方法 (私有)：
      a) Requires：node 是一个非空的 JsonNode 节点，level 是一个非负整数。
      b) Effects： 递归地打印给定的 JsonNode 节点及其子节点的键值对。如果节点是对象，则打印每个键值对；如果节点是数组，则打印每个元素；如果节点是值节点，则打印节点的值。

d) XmlTypeParser 类：

i. Mutability/Immutability：XmlTypeParser 类是一个具体的解析器实现类，其中包含了解析 XML 文件的逻辑。由于类的状态在对象创建后不会改变，因此它是不可变的（immutable）。

ii. 抽象函数 (AF)：
   1. XmlTypeParser 类表示一个具体的解析 XML 文件的解析器。它继承自抽象类 Parser，实现了解析方法 parse 和辅助方法 parseNode。
   2. AF(c) = { filePath => 要解析的 XML 文件路径 }

iii. 表示不变性 (RI)：
   1. filePath 必须是一个非空字符串。
   2. 文件路径所指向的文件必须存在且是一个有效的 XML 文件。

iv. 防止表示暴露 (Safety from Rep Exposure)：该类没有直接暴露内部状态，因此不

存在表示暴露的问题。

v. 方法规格说明 (Spec)：

1. parseNode 方法 (私有)

a) Requires： node 是一个非空的 Node 节点，level 是一个非负整数。

b) Effects： 递归地遍历给定的 Node 节点及其子节点。如果节点是元素节点，则打印节点名称；如果节点是文本节点且包含非空值，则打印节点值。

e) YmlTypeParser 类：

i. Mutability/Immutability：YmlTypeParser 类是一个具体的解析器实现类，其中包含了解析 YAML 文件的逻辑。由于类的状态在对象创建后不会改变，因此它是不可变的（immutable）。

ii. 抽象函数 (AF)：

1. YmlTypeParser 类表示一个具体的解析 YAML 文件的解析器。它继承自抽象类 Parser，实现了解析方法 parse 和辅助方法 printYamlData。

2. AF(c) = { filePath => 要解析的 YAML 文件路径 }

iii. 表示不变性 (RI)：

1. filePath 必须是一个非空字符串。

2. 文件路径所指向的文件必须存在且是一个有效的 YAML 文件。

iv. 防止表示暴露 (Safety from Rep Exposure)：该类没有直接暴露内部状态，因此不存在表示暴露的问题。

v. 方法规格说明 (Spec)：

1. printYamlData 方法 (私有)：

a) Requires：

i. data 是一个非空的 Object 对象

ii. level 是一个非负整数。

b) Effects： 打印给定的 YAML 数据。如果数据是 Map 类型，则打印键值对；如果数据是 Iterable 类型，则打印每个元素；如果数据是其他类型，则直接打印。

f) Client 类：

i. Mutability/Immutability：Client 类是一个简单的客户端类，没有状态，因此是不可变的（immutable）。

ii. 抽象函数 (AF)：Client 类没有抽象函数。

iii. 表示不变性 (RI)：Client 类没有表示不变性的要求。

iv. 防止表示暴露 (Safety from Rep Exposure)：Client 类没有内部状态，因此不存在表示暴露的问题。

v. 方法规格说明 (Spec)：

1. main 方法：

a) Effects： 创建一个 ParserSelectorFactory 实例，并调用其 parseFile 方法模拟解析不同类型的文件。

3.1.3. 测试

ParserSelectorFactoryTest 测试策略：验证 ParserSelectorFactory 类中的 parseFile 方法在解析不同类型的文件时的正确性和健壮性。

测试方法：

1. testParseFileWithSupportedType 方法：测试 parseFile 方法解析受支持的文件类型时的输出。
   a) 步骤：
      i. 创建 ParserSelectorFactory 实例。
      ii. 创建 ByteArrayOutputStream 捕获输出内容。
      iii. 调用 parseFile 方法解析 XML、JSON 和 YAML 文件。
      iv. 验证输出内容不为空，确保解析正确。
2. testParseFileWithUnsupportedType 方法：测试 parseFile 方法解析不受支持的文件类型时的异常处理。
   a) 步骤：
      i. 创建 ParserSelectorFactory 实例。
      ii. 调用 parseFile 方法解析 TXT 文件，预期抛出 IllegalArgumentException 异常。
      iii. 使用 assertThrows 验证异常是否正确抛出。
3. setUp 方法：设置临时目录，并创建模拟的解析器类文件。
   a) 步骤：
      i. 创建模拟的 TxtTypeParser 和 XmlTypeParser 类文件。
      ii. 将类文件写入临时目录。
      iii. 设置系统属性 DIRECTORY_PATH 为临时目录路径。

测试结果：

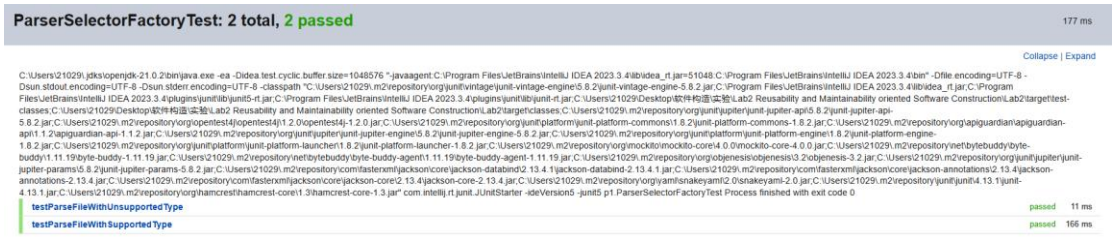

图 3.1.1-1 ParserSelectorFactoryTest 测试结果

测试覆盖率：

表 3.1.1-1 ParserSelectorFactoryTest 测试结果

Class	Class(%)	Method(%)	Branch(%)	Line(%)
ParserSelectorFactory	100% (1/1)	100% (4/4)	62.5% (10/16)	80.8% (21/26)

XmlTypeParserTest 测试策略：验证 XmlTypeParser 类中的 parse 方法在解析 XML 文件时的正确性和健壮性。

测试方法：

1. testValidFilePath 方法：测试 parse 方法使用有效文件路径时的异常处理。
   a) 步骤：
      i. 创建 XmlTypeParser 实例。
      ii. 调用 parse 方法解析有效的 XML 文件路径。
      iii. 使用 assertDoesNotThrow 验证是否没有抛出异常。
2. testEmptyFilePath 方法：测试 parse 方法使用空文件路径时的异常处理。
   a) 步骤：

i. 创建 XmlTypeParser 实例。
ii. 调用 parse 方法解析空的 XML 文件路径。
iii. 使用 assertThrows 验证是否抛出 IllegalArgumentException 异常。

3. parse 方法：描述：测试 parse 方法解析 XML 文件后的输出。
   a) 步骤：
      i. 准备测试数据，包括要解析的 XML 文件路径。
      ii. 重定向 System.out，捕获解析方法的输出内容。
      iii. 调用 parse 方法解析 XML 文件。
      iv. 恢复原始的 System.out。验证解析输出是否符合预期。

测试结果：

图 3.1.1-2 XmlTypeParserTest 测试结果

测试覆盖率：

表 3.1.1-2 XmlTypeParserTest 测试结果

Class	Class(%)	Method(%)	Branch(%)	Line(%)
XmlTypeParser	100% (1/1)	100% (3/3)	80% (8/10)	91.3% (21/23)

JsonTypeParserTest 测试策略：验证 JsonTypeParser 类中的 parse 方法在解析 JSON 文件时的正确性和健壮性。

测试方法：

1. testValidFilePath 方法：测试 parse 方法使用有效文件路径时的异常处理。
   a) 步骤：
      i. 创建 JsonTypeParser 实例。
      ii. 调用 parse 方法解析有效的 JSON 文件路径。
      iii. 使用 assertDoesNotThrow 验证是否没有抛出异常。
2. testEmptyFilePath 方法：测试 parse 方法使用空文件路径时的异常处理。
   a) 步骤：
      i. 创建 JsonTypeParser 实例。
      ii. 调用 parse 方法解析空的 JSON 文件路径。
      iii. 使用 assertThrows 验证是否抛出 IllegalArgumentException 异常。
3. parse 方法：测试 parse 方法解析 JSON 文件后的输出。
   a) 步骤：
      i. 准备测试数据，包括要解析的 JSON 文件路径。
      ii. 重定向 System.out，捕获解析方法的输出内容。
      iii. 调用 parse 方法解析 JSON 文件。
      iv. 恢复原始的 System.out。
      v. 验证解析输出是否符合预期。

测试结果：

图 3.1.1-3 JsonTypeParserTest 测试结果

测试覆盖率：

表 3.1.1-3 JsonTypeParserTest 测试结果

Class	Class(%)	Method(%)	Branch(%)	Line(%)
JsonTypeParser	100% (1/1)	100% (3/3)	100% (8/8)	91.3% (21/23)

YmlTypeParserTest 测试策略：验证 YmlTypeParser 类中的 parse 方法在解析 YAML 文件时的正确性和健壮性。

测试方法：

1. testValidFilePath 方法：测试 parse 方法使用有效文件路径时的异常处理。
   a) 步骤：
      i. 创建 YmlTypeParser 实例。
      ii. 调用 parse 方法解析有效的 YAML 文件路径。
      iii. 使用 assertDoesNotThrow 验证是否没有抛出异常。
2. testEmptyFilePath 方法：测试 parse 方法使用空文件路径时的异常处理。
   a) 步骤：
      i. 创建 YmlTypeParser 实例。
      ii. 调用 parse 方法解析空的 YAML 文件路径。
      iii. 使用 assertThrows 验证是否抛出 IllegalArgumentException 异常。
3. parse 方法：测试 parse 方法解析 YAML 文件后的输出。
   a) 步骤：
      i. 准备测试数据，包括要解析的 YAML 文件路径。
      ii. 重定向 System.out，捕获解析方法的输出内容。
      iii. 调用 parse 方法解析 YAML 文件。
      iv. 恢复原始的 System.out。
      v. 验证解析输出是否符合预期。

测试结果：

图 3.1.1-4 YmlTypeParserTest 测试结果

测试覆盖率：

表 3.1.1-4 YmlTypeParserTest 测试结果

Class	Class(%)	Method(%)	Branch(%)	Line(%)
YmlTypeParser	100% (1/1)	100% (3/3)	100% (8/8)	95.2% (20/21)

3.2. 待开发的应用场景（选做）

相较于问题 1，问题 2 概述了一个软件开发的应用场景，即文件解析。在这个场景中，系统需要根据不同文件类型使用不同的解析器来解析文件内容。具体来说，Json 类型的文件需要用 Json 解析器解析，Xml 类型的文件需要用 Xml 解析器解析，以此类推。设计和实现的过程需要考虑未来可能存在的多种文件格式，例如.yaml、.yml、.properties 等，确保系统能够应对这些变化。要求根据给定的小节顺序和要求，完成场景的设计、实现和测试，并且充分考虑如何满足开放封闭原则（Open-Closed Principle, OCP）。

这个问题的解决方案需要分为两个主要部分：

1. 设计和实现文件解析系统，使其能够灵活处理不同类型的文件，并满足 OCP，即在未来可以方便地扩展支持新的文件格式，而不需要修改现有代码。
2. 在第二部分中，增加了一个要求：根据不同的环境（测试环境和生产环境），对文件进行不同的解析。这进一步挑战了如何设计系统以满足更多的需求，并且如何在满足这些需求的同时保持代码的灵活性和可维护性。

3.2.1. 设计

设计原理：

工厂模式 (Factory Pattern)：工厂模式是一种创建型设计模式，它提供了创建对象的接口，但允许子类决定实例化哪一个类。工厂方法让类的实例化延迟到子类。

依赖注入 (Dependency Injection)：依赖注入是一种设计模式，它使得组件之间的依赖关系外部化，而不是硬编码在组件内部。这有助于解耦合，使得代码更易于测试和维护。

模块化和分层： 设计清晰的模块划分，例如加载工厂、环境选择、解析器等。每个模块负责特定的功能，这有助于代码的组织和管理。

多态性： 具体解析器（如 JsonTypeParser, XmlTypeParser, YmlTypeParser）通过继承 Parser 接口或抽象类，实现了多态性。这意味着相同的接口可以用来处理不同类型的对象，增加了代码的灵活性和可扩展性。

问题分析：

此问题相较于第一问的区别是增加了可选择的环境，在不同的环境中，对于解析文件的格式做了限制。为了满足 OCP，我们首先定义一个抽象父类，表示环境选择的工厂类 EnvironmentFactory，具体的子类都继承这个父类。子类通过实现其中的 judgeType 函数来定义自己的环境中可以解析的文件类型来完成任务。

这里在进行选择的时候，我仍然使用扫描指定目录文件的方式，通过实现特定的环境选择解析器和选择特定的环境这双重的选择来解决这个问题。此种方法的原理已经在第一问中详细阐述过，就不在此赘述。唯一要注意的是，每一种特定的环境都是一种选择器，包括选择环境的选择器，他们都有关于扫描文件的方法是重复的，所以我使用泛型定义的静态函数（已经在问题 1 中展示过）单独提取出来，减少代码的重复和冗余。

设计实现过程：

1. 识别类。在类图中，有以下几个类：
   a) EnvironmentFactory：一个用于扫描目录并选择合适的解析器工厂并解析文件的抽象类。
   b) XXXFactory：是 EnvironmentFactory 的具体实现类，实现了具体的环境配置。
   c) XXXTypeParser：具体的解析器子类，用于创建特定的解析器。包括 JsonTypeParser、XmlTypeParser 和 YmlTypeParser。
   d) Parser：解析器接口，用于解析具体文件。
   e) Client：客户端，调用 EnvironmentSelectorFactory 实现解析文件。
   f) EnvironmentSelectorFactory：用于扫描目录并选择合适的解析器工厂并解析文件的实用类。
   g) FactoryLoader：提取出 scanAndLoadFactories 方法作为静态方法的工具类。
2. 定义类之间的关系：
   a) XXXFactory 与 Parser 之间是聚合关系。XXXFactory 维护了一个 parserFactoryMap 映射，其中键是文件类型，值是相应的 XXXTypeParser 实例。XXXFactory 根据内部的特定判断逻辑选择并返回具体的 Parser 实例（如 JsonTypeParser、XmlTypeParser 或 YmlTypeParser）。
   b) XXXTypeParser 与 Parser 之间是继承关系。XXXTypeParser 是 Parser 的子类。
   c) Client 与 EnvironmentSelectorFactory 之间是依赖关系。Client 调用 EnvironmentSelectorFactory 解析特定文件，内部包含 XXXFactory 创建。
   d) XXXFactory 与 EnvironmentFactory 之间也是继承关系，XXXFactory 通过规定特定的文件类型权限对 EnvironmentFactory 实例化。
   e) EnvironmentSelectorFactory 与 XXXFactory 是聚合关系，EnvironmentSelectorFactory 也是通过维护一个 Map 映射，存储具体的 XXXFactory 类来选择不同的环境。
   f) EnvironmentFactory 和 EnvironmentSelectorFactory 对 FactoryLoader 的关系都是依赖关系，均为在构造函数中调用 FactoryLoader 的静态方法来扫描文件。
3. 定义类的属性和方法：
   a) XXXTypeParser 类：
      i. parser(String path)：创建解析器的具体方法。
      ii. printNode(Node node)：打印处理内容的辅助函数。
   b) EnvironmentFactory 类：
      i. parseFile(String filePath)：解析给定路径的文件。首先检查文件路径是否为空，然后提取文件类型并判断是否能被当前环境解析，最后使用相应的解析器解析文件。
      ii. judgeType(String fileType)：抽象方法，用于判断给定文件类型是否能被当前环境解析。具体的实现由子类提供。
      iii. getFileType(String filePath)：静态私有方法，从文件路径中提取文件类型（扩展名）。
   c) Parser 抽象类：
      i. parse(String filePath)：抽象方法，用于解析给定文件路径所指示的文件。子类需要实现具体的解析逻辑。
      ii. validateFilePath(String filePath)：检查文件路径是否有效。确保文件路径不

为空，并且指示的是一个存在且为文件的有效路径。

iii. printIndent(int level)：打印缩进，用于在解析过程中输出缩进。参数 level 表示缩进级别。

iv. handleParsingError(Exception e)：处理解析过程中发生的异常。输出解析失败的提示信息和异常信息。

d) ProductionFactory 类：

i. judgeType(String fileType)：用于判断文件类型是否被支持，它始终返回 true，表示所有文件类型都被支持。

e) TestFactory 类：

i. judgeType(String fileType)：用于判断文件类型是否被支持。

f) EnvironmentSelectorFactory 类：

i. DIRECTORY_PATH：表示要扫描的目录路径，用于加载环境工厂类。

ii. factoryMap： 是一个映射，用于存储环境工厂类的实例，键是环境名称（小写），值是对应的环境工厂类实例。

iii. EnvironmentSelectorFactory()：是构造函数，用于初始化工厂类实例的映射。

iv. selectEnv(String env, String filePath)：用于选择并解析文件。它接受环境名称和文件路径作为参数，首先检查这些参数是否为空，然后根据给定的环境名称获取对应的环境工厂类实例，并调用其 parseFile(filePath) 方法来解析文件。

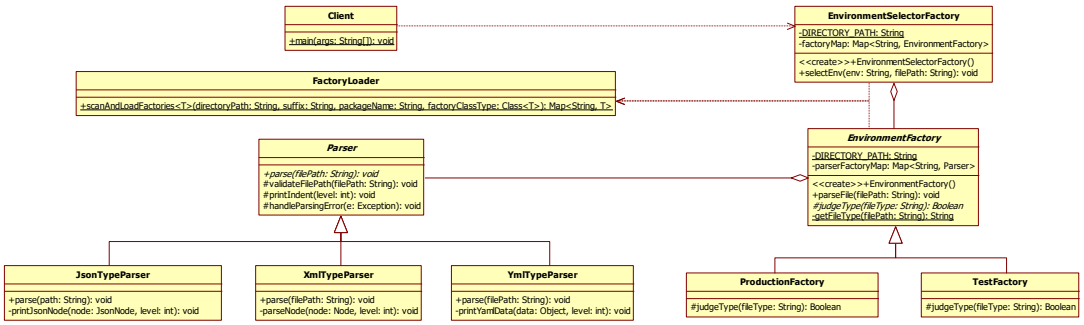

图3.2.1-1 设计类图（UML/p2.jpg）

设计原理解释：此设计采用了嵌套的简单工厂模式，通过 EnvironmentSelectorFactory 加载不同的工厂类，然后由这些工厂类生成对应的解析器来处理不同格式的数据。同时也可以解释成桥接或者策略模式，但是不变的是这种方法从来不具象化要使用的类，而是通过 Map 和动态加载的方式确定。

满足条件：

复用性：由于使用了简单工厂模式，可以方便地增加新的解析器类型而无需修改现有代码。同时我也添加了不同的环境工厂的抽象类，添加新的环境的时候，将必须的方法放到抽象父类中，通过继承的方式实现复用。同时，我将所有选择器类均使用的扫描方法变成静态函数，放到一个单独的工具类中，实现了这个方法的复用。

OCP 原则：开放封闭原则得到了体现，即对扩展开放、对修改关闭。当需要支持其他数据格式时，只需要添加相应的解析器即可，不需要改动已有的代码结构。不论是添加新的

解析器或者是新的环境，均只需要新创一个类即可，不需要修改现有的代码结构。

设计优势：

灵活性：易于扩展新功能或更改已有功能，因为所有操作都基于抽象层进行。

模块化：每个组件都有明确的责任范围，降低了复杂度并提高了可维护性。

3.2.2. 实现

以下文档内容描述了一个文件解析器框架的设计和实现，包括抽象的 Parser 类及其具体实现类 JsonTypeParser、XmlTypeParser 和 YmlTypeParser，用于解析 JSON、XML 和 YAML 文件。框架中的 Client 类用于演示解析过程，而 EnvironmentFactory 类及其具体子类 TestFactory 和 ProductionFactory 则管理解析器实例和解析环境。EnvironmentSelectorFactory 类负责根据环境名称和文件路径选择适当的解析器工厂，而 FactoryLoader 类则用于扫描和加载工厂类。每个类和方法都有详细的设计规范，确保代码的可维护性和可靠性。

a) Parser 类：
   i. 方法规格说明 (Spec)：
      1. parse 方法 (抽象方法)
         a) Requires： filePath 是一个非空字符串。
         b) Effects： 解析给定文件路径所指示的文件。该方法是抽象的，具体的解析逻辑需要在子类中实现。
      2. validateFilePath 方法
         a) Requires： filePath 是一个非空字符串。
         b) Effects： 验证文件路径是否有效。如果文件路径为空或文件不存在，抛出 IllegalArgumentException 异常。
      3. printIndent 方法
         a) Requires： level 是一个非负整数。
         b) Effects： 打印指定级别的缩进。
      4. handleParsingError 方法
         a) Requires： e 是一个非空的异常对象。
         b) Effects： 打印解析过程中发生的错误信息。
b) JsonTypeParser 类：
   i. Mutability/Immutability：JsonTypeParser 类是一个具体的解析器实现类，其中包含了解析 JSON 文件的逻辑。由于类的状态在对象创建后不会改变，因此它是不可变的（immutable）。
   ii. 抽象函数 (AF)：
      a) JsonTypeParser 类表示一个具体的解析 JSON 文件的解析器。它继承自抽象类 Parser，实现了解析方法 parse 和辅助方法 printJsonNode。
      b) AF(c) = { filePath => 要解析的 JSON 文件路径, objectMapper => 用于解析 JSON 的 ObjectMapper 实例 }
   iii. 表示不变性 (RI)：filePath 必须是一个非空字符串。文件路径所指向的文件必须存在且是一个有效的 JSON 文件。
   iv. 防止表示暴露 (Safety from Rep Exposure)：该类没有直接暴露内部状态，因此不存在表示暴露的问题。内部的 ObjectMapper 实例是私有的，并且没有公开的访问

方法。

v. 方法规格说明 (Spec)：

1. printJsonNode 方法 (私有)：

a) Requires：node 是一个非空的 JsonNode 节点，level 是一个非负整数。

b) Effects： 递归地打印给定的 JsonNode 节点及其子节点的键值对。如果节点是对象，则打印每个键值对；如果节点是数组，则打印每个元素；如果节点是值节点，则打印节点的值。

c) XmlTypeParser 类：

i. Mutability/Immutability：XmlTypeParser 类是一个具体的解析器实现类，其中包含了解析 XML 文件的逻辑。由于类的状态在对象创建后不会改变，因此它是不可变的（immutable）。

ii. 抽象函数 (AF)：

a) XmlTypeParser 类表示一个具体的解析 XML 文件的解析器。它继承自抽象类 Parser，实现了解析方法 parse 和辅助方法 parseNode。

b) AF(c) = { filePath => 要解析的 XML 文件路径 }

iii. 表示不变性 (RI)：filePath 必须是一个非空字符串。文件路径所指向的文件必须存在且是一个有效的 XML 文件。

iv. 防止表示暴露 (Safety from Rep Exposure)：该类没有直接暴露内部状态，因此不存在表示暴露的问题。

v. 方法规格说明 (Spec)：

1. parseNode 方法 (私有)

a) Requires： node 是一个非空的 Node 节点，level 是一个非负整数。

b) Effects： 递归地遍历给定的 Node 节点及其子节点。如果节点是元素节点，则打印节点名称；如果节点是文本节点且包含非空值，则打印节点值。

d) YmlTypeParser 类：

i. Mutability/Immutability：YmlTypeParser 类是一个具体的解析器实现类，其中包含了解析 YAML 文件的逻辑。由于类的状态在对象创建后不会改变，因此它是不可变的（immutable）。

ii. 抽象函数 (AF)：

a) YmlTypeParser 类表示一个具体的解析 YAML 文件的解析器。它继承自抽象类 Parser，实现了解析方法 parse 和辅助方法 printYamlData。

b) AF(c) = { filePath => 要解析的 YAML 文件路径 }

iii. 表示不变性 (RI)：filePath 必须是一个非空字符串。文件路径所指向的文件必须存在且是一个有效的 YAML 文件。

iv. 防止表示暴露 (Safety from Rep Exposure)：该类没有直接暴露内部状态，因此不存在表示暴露的问题。

v. 方法规格说明 (Spec)：

1. printYamlData 方法 (私有)：

a) Requires： data 是一个非空的 Object 对象，level 是一个非负整数。

b) Effects： 打印给定的 YAML 数据。如果数据是 Map 类型，则打印键值对；如果数据是 Iterable 类型，则打印每个元素；如果数据是其他类型，则直接打印。

e) Client 类：

i. Mutability/Immutability：Client 类是一个简单的客户端类，没有状态，因此是

不可变的（immutable）。

ii. 抽象函数 (AF)：Client 类没有抽象函数的要求。

iii. 表示不变性 (RI)：Client 类没有表示不变性的要求。

iv. 防止表示暴露 (Safety from Rep Exposure)：Client 类没有内部状态，因此不存在表示暴露的问题。

v. 方法规格说明 (Spec)：

1. main 方法：

a) Effects： 创建一个 ParserSelectorFactory 实例，并调用其 parseFile 方法模拟解析不同类型的文件。

f) EnvironmentFactory 类：

i. 方法规格说明 (Spec)：

1. parseFile 方法：

a) Requires： filePath 是一个非空字符串。

b) Effects： 解析给定文件路径所指示的文件。如果文件路径为空或文件类型不受支持，抛出 IllegalArgumentException 异常。调用抽象方法 judgeType 判断当前环境是否支持解析该文件类型，如果不支持，输出提示信息。

2. judgeType 方法 (抽象方法)：

a) Requires： fileType 是一个非空字符串。

b) Effects： 判断给定文件类型是否能被当前环境解析。如果能解析则返回 true，否则返回 false。具体实现由子类提供。

3. getFileType 方法 (静态方法)：

a) Requires： filePath 是一个非空字符串。

b) Effects： 从文件路径中提取文件类型（扩展名）。如果没有扩展名，返回空字符串。

4. EnvironmentFactory 构造方法

a) Effects： 使用 FactoryLoader 扫描指定目录并加载解析器工厂类，初始化 parserFactoryMap。

g) TestFactory 类：

i. Mutability/Immutability：TestFactory 类是一个具体的工厂类，没有内部状态，因此是不可变的（immutable）。

ii. 抽象函数 (AF)：

1. TestFactory 类实现了 EnvironmentFactory 的抽象方法，用于判断文件类型是否支持解析。

2. AF(c) = { fileType => 支持解析的文件类型, 其中 fileType 为 "xml" 或 "json" }

iii. 表示不变性 (RI)：该类没有特定的不变性要求。

iv. 防止表示暴露 (Safety from Rep Exposure)：由于 TestFactory 没有内部状态，因此不存在表示暴露的问题。

v. 方法规格说明 (Spec)：

1. judgeType 方法 (重写)：

a) Requires： fileType 是一个非空字符串。

b) Effects： 判断给定文件类型是否能被当前环境解析。如果文件类型是 "xml" 或 "json"，则返回 true；否则返回 false。

h) ProductionFactory 类：

i. Mutability/Immutability：ProductionFactory 类是一个具体的工厂类，没有内部状态，因此是不可变的（immutable）。
ii. 抽象函数 (AF)：
    1. ProductionFactory 类表示一个始终返回 true 的环境工厂类，意味着它可以解析任何文件类型。
    2. AF(c) = { fileType => 始终返回 true，表示所有文件类型都支持解析 }
iii. 表示不变性 (RI)：该类没有特定的不变性要求。
iv. 防止表示暴露 (Safety from Rep Exposure)：由于 ProductionFactory 没有内部状态，因此不存在表示暴露的问题。

i) EnvironmentSelectorFactory 类：
i. Mutability/Immutability：EnvironmentSelectorFactory 类是一个包含环境工厂映射的类，映射表在对象创建后不可以更改，因此它是不可变的（Immutability）。
ii. 抽象函数 (AF)：
    1. EnvironmentSelectorFactory 类表示一个从指定目录中读取环境工厂类并选择合适的工厂来创建环境工厂的实用类。
    2. AF(c) = { DIRECTORY_PATH => 被扫描的目录路径, factoryMap => 存储环境名称与环境工厂实例之间映射的映射表 }
iii. 表示不变性 (RI)：目录路径 DIRECTORY_PATH 必须是一个有效的字符串，不能为 null 或空。
iv. 防止表示暴露 (Safety from Rep Exposure)：该类没有暴露其内部状态，因此不存在表示暴露的问题。
v. 方法规格说明 (Spec)：
    1. selectEnv 方法：
        a) Requires： env 是一个非空字符串，表示环境名称；filePath 是一个非空字符串，表示需要解析的文件路径。
        b) Effects： 根据给定的环境名称和文件路径，选择合适的环境工厂并解析文件。如果环境名或文件路径为空，抛出 IllegalArgumentException 异常。如果指定的环境类型不受支持，抛出 IllegalArgumentException 异常。
    2. EnvironmentSelectorFactory 构造方法：
        a) Effects： 使用 FactoryLoader 扫描指定目录并加载环境工厂类，初始化 factoryMap。

j) FactoryLoader 类：
i. Mutability/Immutability：FactoryLoader 类是一个实用工具类，没有内部状态，因此是不可变的（immutable）。
ii. 抽象函数 (AF)：
    1. FactoryLoader 类用于扫描指定目录并加载工厂类，并返回一个包含工厂实例的映射。映射的键为工厂类名，值为工厂实例。
    2. AF(c) = { directoryPath => 要扫描的目录路径, suffix => 工厂类文件名后缀, packageName => 工厂类所在包名, factoryClassType => 工厂类的类型 }
iii. 表示不变性 (RI)：无特定的不变性要求。
iv. 防止表示暴露 (Safety from Rep Exposure)：该类没有内部状态，因此不存在表示暴露的问题。

v. 方法规格说明 (Spec)：

1. scanAndLoadFactories 方法：
   a) Requires：
      i. directoryPath 是一个有效的目录路径字符串。
      ii. suffix 是一个有效的文件名后缀字符串。
      iii. packageName 是一个有效的包名字符串。
      iv. factoryClassType 是一个有效的工厂类类型。
   b) Effects：扫描指定目录并加载具有指定后缀的工厂类。返回一个包含工厂实例的映射，键为工厂类名，值为工厂实例。如果目录路径无效或无法实例化工厂类，抛出 IllegalArgumentException 异常。
   c) Parameters：
      i. directoryPath： 要扫描的目录路径，字符串类型。
      ii. suffix： 工厂类文件名后缀，字符串类型。
      iii. packageName： 工厂类所在包名，字符串类型。
      iv. factoryClassType： 工厂类的类型，Class 类型。
   d) Returns：返回一个 Map<String, T>，其中键为工厂类名，值为工厂实例。
   e) Throws：IllegalArgumentException 如果目录路径无效、无法实例化工厂类，或指定目录中没有找到工厂类。
   f) scanAndLoadFactories 方法的详细行为说明：
      i. 扫描目录： 方法会检查 directoryPath 是否存在且是一个目录。如果目录路径无效，则抛出 IllegalArgumentException。
      ii. 加载文件名： 使用指定的 suffix 扫描目录中的文件名。如果指定目录中没有找到工厂类文件，也会抛出 IllegalArgumentException。
      iii. 实例化工厂类： 对于每个找到的文件，尝试通过反射机制加载相应的工厂类并实例化。如果无法实例化工厂类，抛出带有详细错误信息的 IllegalArgumentException。
      iv. 返回映射： 方法最后返回一个映射，包含了所有成功加载和实例化的工厂类。

3.2.3. 测试

EnvironmentSelectorFactoryTest 测试策略：验证 EnvironmentSelectorFactory 类的各项功能，包括构造函数、选择环境并解析文件的功能，以及异常处理。

测试方法：

1. testConstructor 方法：测试构造函数，确保实例化不会抛出异常。
   a) 步骤：
      i. 通过构造函数创建 EnvironmentSelectorFactory 实例，验证实例不为空。
2. testSelectEnv_NormalCase 方法：测试正常情况下的 selectEnv 方法。
   a) 步骤：
      i. 模拟 EnvironmentFactory 对象。
      ii. 设置工厂映射。
      iii. 调用 selectEnv 方法。
      iv. 验证 parseFile 方法是否被调用。

3. testSelectEnv_EmptyEnv 方法：测试 selectEnv 方法使用空环境名。
   a) 步骤：
      i. 调用 selectEnv 方法传入空的环境名，预期抛出 IllegalArgumentException 异常。
4. testSelectEnv_NullFilePath 方法：测试 selectEnv 方法使用空文件路径。
   a) 步骤：
      i. 调用 selectEnv 方法传入空的文件路径，预期抛出 IllegalArgumentException 异常。
5. testSelectEnv_NonExistingEnv 方法：测试 selectEnv 方法使用不存在的环境名。
   a) 步骤：
      i. 调用 selectEnv 方法传入不存在的环境名 "invalid"。
      ii. 验证是否抛出 IllegalArgumentException 异常。
6. testSelectEnv_NonExistingFile 方法：测试 selectEnv 方法使用不存在的文件路径。
   a) 步骤：
      i. 调用 selectEnv 方法传入不存在的文件路径 "non_existing_file.json"。
      ii. 验证是否抛出 IllegalArgumentException 异常。
7. testSelectEnv_CaseInsensitiveEnv 方法：测试 selectEnv 方法在环境名不区分大小写的情况下。
   a) 步骤：
      i. 模拟 EnvironmentFactory 对象。
      ii. 设置工厂映射。
      iii. 调用 selectEnv 方法传入大写形式的环境名 "Production"。
      iv. 验证 parseFile 方法是否被调用。
8. testFactoryLoading 方法：测试工厂映射的初始化。
   a) 步骤：
      i. 通过构造函数创建 EnvironmentSelectorFactory 实例，验证工厂映射不为空。
9. testFactoryMapContainsMultipleFactories 方法：测试工厂映射是否包含多个工厂类。
   a) 步骤：
      i. 调用 selectEnv 方法传入 "test" 环境名。
      ii. 调用 selectEnv 方法传入 "production" 环境名。
      iii. 确保没有抛出异常。

测试结果：

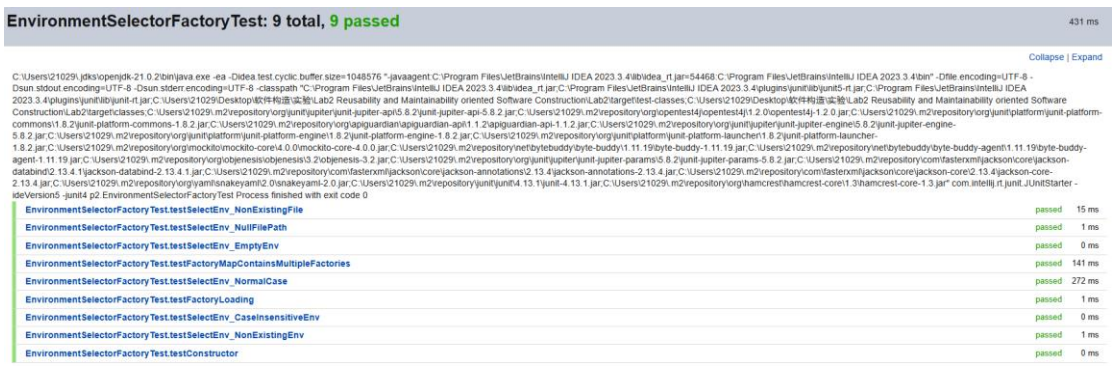

图 3.2.3-1 EnvironmentSelectorFactoryTest 测试结果

测试覆盖率：

表 3.2.3-1 EnvironmentSelectorFactoryTest 测试结果

Class	Class(%)	Method(%)	Branch(%)	Line(%)
EnvironmentSelectorFactory	100% (1/1)	100% (2/2)	80% (8/10)	100% (8/8)

FactoryLoaderTest 测试策略：验证 FactoryLoader 类中的 scanAndLoadFactories 方法在不同情况下的正确性和异常处理能力。

测试方法：

1. testInvalidPath 方法：测试 scanAndLoadFactories 方法在给定无效路径时的异常处理。
   a) 步骤：
      i. 调用 scanAndLoadFactories 方法，传入无效路径。
      ii. 使用 @Test(expected = IllegalArgumentException.class) 验证是否抛出异常。
2. testNotADirectory 方法：测试 scanAndLoadFactories 方法在给定路径不是目录时的异常处理。
   a) 步骤：
      i. 调用 scanAndLoadFactories 方法，传入文件路径而非目录路径。
      ii. 使用 @Test(expected = IllegalArgumentException.class) 验证是否抛出异常。
3. testEmptyDirectory 方法：测试 scanAndLoadFactories 方法在给定空目录时的异常处理。
   a) 步骤：
      i. 调用 scanAndLoadFactories 方法，传入空目录路径。
      ii. 使用 @Test(expected = IllegalArgumentException.class) 验证是否抛出异常。
4. testNoMatchingFiles 方法：测试 scanAndLoadFactories 方法在目录中没有匹配的工厂类文件时的异常处理。
   a) 步骤：
      i. 在临时目录中创建不匹配命名规则的类文件。
      ii. 调用 scanAndLoadFactories 方法，传入该目录路径。
      iii. 使用 @Test(expected = IllegalArgumentException.class) 验证是否抛出异常。
5. testInvalidFactoryClass 方法：测试 scanAndLoadFactories 方法在匹配的工厂类文件不实现指定接口时的异常处理。
   a) 步骤：
      i. 在临时目录中创建符合命名规则但不实现指定接口的类文件。
      ii. 调用 scanAndLoadFactories 方法，传入该目录路径。
      iii. 使用 @Test(expected = IllegalArgumentException.class) 验证是否抛出异常。

测试结果：

图 3.2.3-2 FactoryLoaderTest 测试结果

测试覆盖率：

表 3.2.3-2 FactoryLoaderTest 测试结果

Class	Class(%)	Method(%)	Branch(%)	Line(%)
FactoryLoader	100% (1/1)	50% (1/2)	90% (9/10)	94.1% (16/17)

ProductionFactoryTest 测试策略：验证 FactoryLoader 类中的 scanAndLoadFactories 方法在不同情况下的正确性和异常处理能力。

测试方法：

1. testFactoryLoading_NormalCase 方法：测试 scanAndLoadFactories 方法在正常情况下的表现。
   a) 步骤：
      i. 调用 scanAndLoadFactories 方法，传入正确的目录路径、后缀和包名。
      ii. 验证工厂类映射不为空。
      iii. 验证工厂类映射包含 ProductionFactory。
      iv. 验证加载的类为 EnvironmentFactory 的子类。
2. testFactoryLoading_NonExistingDirectory 方法：测试 scanAndLoadFactories 方法在目录路径不存在的情况下的异常处理。
   a) 步骤：
      i. 调用 scanAndLoadFactories 方法，传入不存在的目录路径。
      ii. 使用 @Test(expected = IllegalArgumentException.class) 验证是否抛出异常。
3. testFactoryLoading_FileAsDirectory 方法：测试 scanAndLoadFactories 方法在目录路径为文件的情况下的异常处理。
   a) 步骤：
      i. 调用 scanAndLoadFactories 方法，传入文件路径而非目录路径。
      ii. 使用 @Test(expected = IllegalArgumentException.class) 验证是否抛出异常。
4. testFactoryLoading_FailedInstantiation 方法：测试 scanAndLoadFactories 方法在无法实例化工厂类的情况下的异常处理。
   a) 步骤：
      i. 调用 scanAndLoadFactories 方法，传入包含无法实例化的工厂类的目录路径。
      ii. 使用 @Test(expected = IllegalArgumentException.class) 验证是否抛出异常。
5. testFactoryLoading_WrongType 方法：测试 scanAndLoadFactories 方法在工厂类不是指定类型的情况下的异常处理。
   a) 步骤：

i. 调用 scanAndLoadFactories 方法，传入包含不实现指定接口的类文件的目录路径。

ii. 使用 @Test(expected = IllegalArgumentException.class) 验证是否抛出异常。

测试结果：

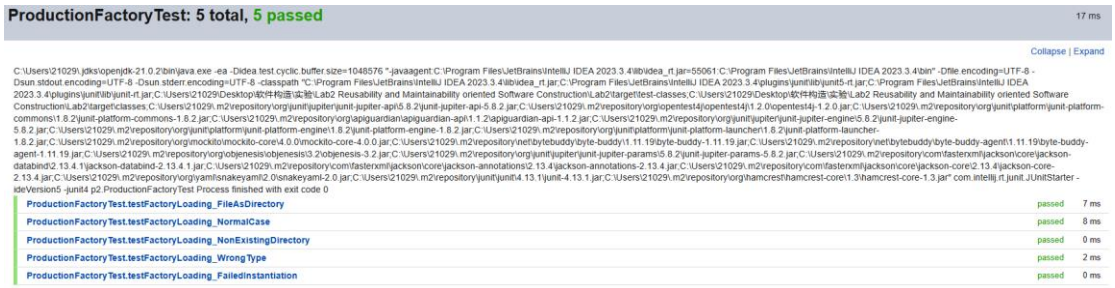

图 3.2.3-3 ProductionFactoryTest 测试结果

测试覆盖率：

表 3.2.3-3 ProductionFactoryTest 测试结果

Class	Class(%)	Method(%)	Line(%)
ProductionFactory	100% (1/1)	100% (2/2)	100% (2/2)

TestFactoryTest 测试策略：验证 FactoryLoader 类中的 scanAndLoadFactories 方法在不同情况下的正确性和异常处理能力。

测试方法：

1. testFactoryLoading_NormalCase 方法：测试 scanAndLoadFactories 方法在正常情况下的表现。
   a) 步骤：
      i. 调用 scanAndLoadFactories 方法，传入正确的目录路径、后缀和包名。
      ii. 验证工厂类映射不为空。
      iii. 验证工厂类映射包含 TestFactory。
      iv. 验证加载的类为 EnvironmentFactory 的子类。
2. testFactoryLoading_NonExistingDirectory 方法：测试 scanAndLoadFactories 方法在目录路径不存在的情况下的异常处理。
   a) 步骤：
      i. 调用 scanAndLoadFactories 方法，传入不存在的目录路径。
      ii. 使用 @Test(expected = IllegalArgumentException.class) 验证是否抛出异常。
3. testFactoryLoading_FileAsDirectory 方法：测试 scanAndLoadFactories 方法在目录路径为文件的情况下的异常处理。
   a) 步骤：
      i. 调用 scanAndLoadFactories 方法，传入文件路径而非目录路径。
      ii. 使用 @Test(expected = IllegalArgumentException.class) 验证是否抛出异常。
4. testFactoryLoading_FailedInstantiation 方法：测试 scanAndLoadFactories 方法在无法实例化工厂类的情况下的异常处理。
   a) 步骤：
      i. 调用 scanAndLoadFactories 方法，传入包含无法实例化的工厂类的目录路径。

ii. 使用 @Test(expected = IllegalArgumentException.class) 验证是否抛出异常。

5. testFactoryLoading_WrongType 方法：测试 scanAndLoadFactories 方法在工厂类不是指定类型的情况下的异常处理。
   a) 步骤：
      i. 调用 scanAndLoadFactories 方法，传入包含不实现指定接口的类文件的目录路径。
      ii. 使用 @Test(expected = IllegalArgumentException.class) 验证是否抛出异常。

测试结果：

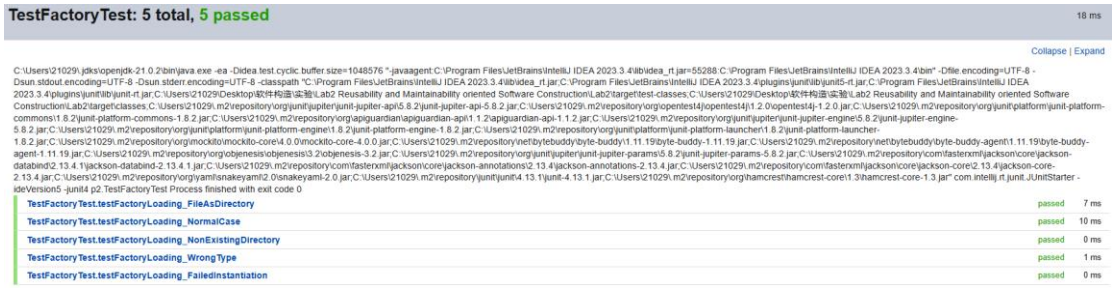

图 3.2.3-4 TestFactoryTest 测试结果

测试覆盖率：

表 3.2.3-4 TestFactoryTest 测试结果

Class	Class(%)	Method(%)	Branch(%)	Line(%)
TestFactory	100% (1/1)	100% (2/2)	50% (1/2)	75% (3/4)

XmlTypeParserTest 测试策略：验证 XmlTypeParser 类中的 parse 方法在解析 XML 文件时的正确性和健壮性。

测试方法：

1. testValidFilePath 方法：测试 parse 方法使用有效文件路径时的异常处理。
   a) 步骤：
      i. 创建 XmlTypeParser 实例。
      ii. 调用 parse 方法解析有效的 XML 文件路径。
      iii. 使用 assertDoesNotThrow 验证是否没有抛出异常。
2. testEmptyFilePath 方法：测试 parse 方法使用空文件路径时的异常处理。
   a) 步骤：
      i. 创建 XmlTypeParser 实例。
      ii. 调用 parse 方法解析空的 XML 文件路径。
      iii. 使用 assertThrows 验证是否抛出 IllegalArgumentException 异常。
3. parse 方法：描述：测试 parse 方法解析 XML 文件后的输出。
   a) 步骤：
      i. 准备测试数据，包括要解析的 XML 文件路径。
      ii. 重定向 System.out，捕获解析方法的输出内容。
      iii. 调用 parse 方法解析 XML 文件。
      iv. 恢复原始的 System.out。验证解析输出是否符合预期。

测试结果：

图 3.2.3-5 XmlTypeParserTest 测试结果

测试覆盖率：

表 3.2.3-5 XmlTypeParserTest 测试结果

Class	Class(%)	Method(%)	Branch(%)	Line(%)
XmlTypeParser	100% (1/1)	100% (3/3)	80% (8/10)	91.3% (21/23)

JsonTypeParserTest 测试策略：验证 JsonTypeParser 类中的 parse 方法在解析 JSON 文件时的正确性和健壮性。

测试方法：

1. testValidFilePath 方法：测试 parse 方法使用有效文件路径时的异常处理。
   a) 步骤：
      i. 创建 JsonTypeParser 实例。
      ii. 调用 parse 方法解析有效的 JSON 文件路径。
      iii. 使用 assertDoesNotThrow 验证是否没有抛出异常。
2. testEmptyFilePath 方法：测试 parse 方法使用空文件路径时的异常处理。
   a) 步骤：
      i. 创建 JsonTypeParser 实例。
      ii. 调用 parse 方法解析空的 JSON 文件路径。
      iii. 使用 assertThrows 验证是否抛出 IllegalArgumentException 异常。
3. parse 方法：测试 parse 方法解析 JSON 文件后的输出。
   a) 步骤：
      i. 准备测试数据，包括要解析的 JSON 文件路径。
      ii. 重定向 System.out，捕获解析方法的输出内容。
      iii. 调用 parse 方法解析 JSON 文件。
      iv. 恢复原始的 System.out。
      v. 验证解析输出是否符合预期。

测试结果：

图 3.2.3-6 JsonTypeParserTest 测试结果

测试覆盖率：

表 3.2.3-6 JsonTypeParserTest 测试结果

Class	Class(%)	Method(%)	Branch(%)	Line(%)
JsonTypeParser	100% (1/1)	100% (3/3)	100% (8/8)	91.3% (21/23)

YmlTypeParserTest 测试策略：验证 YmlTypeParser 类中的 parse 方法在解析 YAML 文件时的正确性和健壮性。

测试方法：

1. testValidFilePath 方法：测试 parse 方法使用有效文件路径时的异常处理。
   a) 步骤：
      i. 创建 YmlTypeParser 实例。
      ii. 调用 parse 方法解析有效的 YAML 文件路径。
      iii. 使用 assertDoesNotThrow 验证是否没有抛出异常。
2. testEmptyFilePath 方法：测试 parse 方法使用空文件路径时的异常处理。
   a) 步骤：
      i. 创建 YmlTypeParser 实例。
      ii. 调用 parse 方法解析空的 YAML 文件路径。
      iii. 使用 assertThrows 验证是否抛出 IllegalArgumentException 异常。
3. parse 方法：测试 parse 方法解析 YAML 文件后的输出。
   a) 步骤：
      i. 准备测试数据，包括要解析的 YAML 文件路径。
      ii. 重定向 System.out，捕获解析方法的输出内容。
      iii. 调用 parse 方法解析 YAML 文件。
      iv. 恢复原始的 System.out。
      v. 验证解析输出是否符合预期。

测试结果：

图 3.2.3-7 YmlTypeParserTest 测试结果

测试覆盖率：

表 3.2.3-7 YmlTypeParserTest 测试结果

Class	Class(%)	Method(%)	Branch(%)	Line(%)
YmlTypeParser	100% (1/1)	100% (3/3)	100% (8/8)	95.2% (20/21)

3.3. 待开发的应用场景-饮料计费系统（必做）

3.3.1. 设计

该问题旨在实现一个咖啡订购系统，该系统允许用户选择不同类型的咖啡和调料，并计算总成本。我使用策略模式与桥接方式相结合的办法来实现这一目标。这种设计方法提供了高可扩展性和可维护性，使得我们可以轻松添加新的咖啡类型或调料。

解决办法：我们通过定义基本的 Beverage 类和 CondimentDecorator 类，以及对应的不同的具体实现来表示各种咖啡类型和调料类型，从而实现继承和复用的效果。对于组合的设计我使用桥接模式和装饰器模式来实现调料的动态组合。通过简单工厂模式和策略模式创建决策器，以便客户端代码可以方便地构建和定制咖啡订单。具体办法是将 Beverage 通过聚合的方式作为内部属性传递给 CondimentDecorator，然后通过不断调用 Beverage 的 cost() 方法返回重写的花费，从而实现装饰叠加的效果。

具体实现如下：

```
public Double select(String beverageName, List<CondimentSelector>
condimentSelectorList) {
    Beverage beverage = selectBeverageMap.get(beverageName.toLowerCase());
    if (beverage == null) {
        throw new IllegalArgumentException("未找到指定的饮料：" + beverageName);
    }

    for (CondimentSelector condimentSelector ： condimentSelectorList) {
        CondimentDecorator condimentDecorator =
selectCondimentMap.get(condimentSelector.condimentName().toLowerCase());
        if (condimentDecorator == null) {
            throw new IllegalArgumentException("未找到指定的调料：" +
condimentSelector.condimentName());
        }
        condimentDecorator.setCondiment(beverage, condimentSelector.num());
        beverage = condimentDecorator;
    }

    return beverage.cost();
}
```

具体种类的选择仍然采用前两问使用的策略模式，首先获取到具体的 Beverage 种类，然后对于传入的各种调料和对应的数量，使用查询到的对应具体的 Condiment 类，具体的 Condiment 类通过重写 cost()函数来达到增加价格的目的。这里的 CondimentSelector 是一个 record 类，作为记录来维持参数的传递。在最后使用 setter 方法为调料对象中的 beverage 属性赋新值，并且将其赋值回原始的 beverage 对象，这里最后的 beverage 对象就会仅仅保存已经重写的 cost()方法，从而实现装饰的效果。

```
public abstract class Beverage {
    public abstract double cost();
}
```

通过查看 Beverage 类可以看到其中仅仅有指向 cost()函数的索引而已。

一种具体的 Condiment 实现：

```
public class SteamedMilk extends CondimentDecorator {
    @Override
    public double cost() {
        return beverage.cost() + 0.12 * num;
    }
}
```

通过代码可以看出，SteamedMilk 类在重写 cost()函数的时候同时也调用了其中属性 beverage 的 cost()函数，因为这里的 beverage 属性存储着之前所有的 price 累加结果。

原因是在 select()函数中的这段代码：

```
beverage = condimentDecorator;
```

他将已经实现的具体 condiment 类赋值给父类对象，实现了信息传递的功能。

设计过程：

1. 识别类：
   a) Beverage： 表示基本的咖啡类型。
   b) CondimentDecorator： 表示调料的抽象类。
   c) 具体咖啡类： DarkRoast, HouseBlend, Decaf, Espresso。
   d) 具体调料类： Whip, Mocha, Soy, SteamedMilk。
   e) SelectorFactory： 工厂类，用于创建咖啡和调料选择器。
   f) CondimentSelector： 用于表示调料选择的类。
   g) Client： 客户端类，用于运行程序。
2. 定义类之间的关系：
   a) Beverage 抽象类类是所有具体咖啡类的父类。
   b) CondimentDecorator 抽象类继承自 Beverage，并作为所有具体调料类的父类。
   c) SelectorFactory 类将 Beverage 和 CondimentDecorator 作为内部属性，是聚合关系，同时他依赖 CondimentSelector 来获取 Client 传递的信息。
   d) Client 类依赖 SelectorFactory 来构建饮料和调料列表，依赖 CondimentSelector 将信息传递给工厂 SelectorFactory。
3. 定义类的属性和方法：
   a) Beverage
      i. 方法：
         1. cost()： double
   b) CondimentDecorator
      i. 属性：
         1. num： int
         2. beverage： Beverage
      ii. 方法：
         1. setCondiment(beverage： Beverage, num： int)： void
         2. cost()： double
   c) 具体咖啡类 (DarkRoast, HouseBlend, Decaf, Espresso)
      i. 方法：
         1. cost()： double
   d) 具体调料类 (Whip, Mocha, Soy, SteamedMilk)
      i. 方法：

1. cost()： double

e) SelectorFactory

i. 属性：

1. CONDIMENT_DIRECTORY_PATH： String
2. BEVERAGE_DIRECTORY_PATH： String

ii. 方法：

1. selectCondimentMap(map： Map<String, CondimentDecorator>)
2. selectBeverageMap(map： Map<String, Beverage>)
3. scanAndLoadClasses<T>(directoryPath： String, map： Map<String, T>, expectedClass： Class<T>)： void
4. select(beverageName： String
5. condimentSelectorList： List<CondimentSelector>)： Beverage

f) CondimentSelector

i. 属性：

1. condimentName： String
2. num： Integer

ii. 方法：create(condimentName：String, num：Integer)：CondimentSelector

1. condimentName()： String
2. num()： Integer
3. equals(object： Object)： boolean
4. hashCode()： int
5. toString()： String

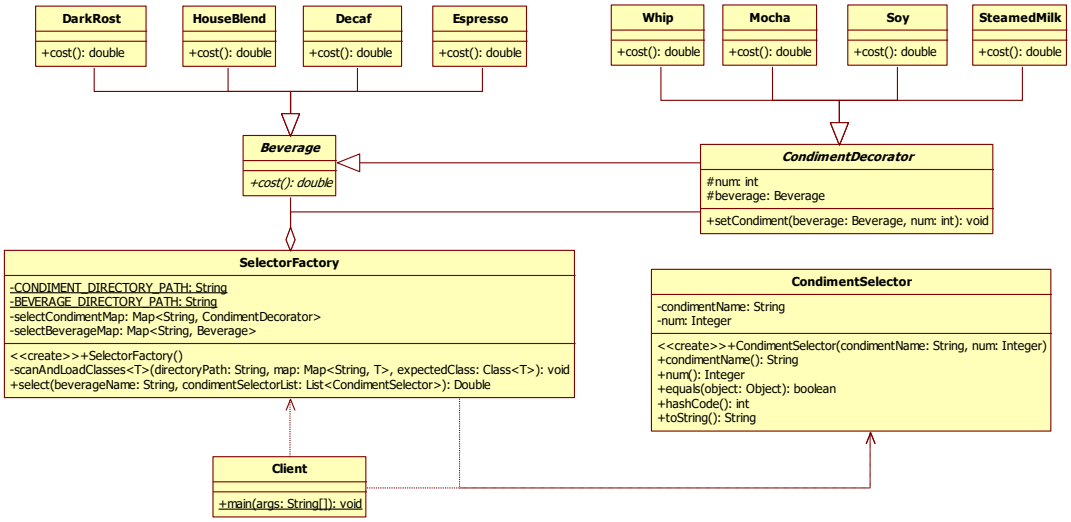

图 3.3.1-1 设计类图(UML/p3.jpg)

设计原理解释：

桥接模式：用于动态地将调料添加到基本咖啡类型中，从而灵活地组合各种调料和咖啡。

工厂模式：用于创建具体的 Beverage 和 CondimentDecorator 实例，以实现更高的灵活性和可扩展性。

单一职责原则：每个类仅负责一项任务，例如，Beverage 类负责表示咖啡，CondimentDecorator 类负责表示调料，SelectorFactory 类负责创建选择器。

满足条件：

复用：通过使用装饰器模式和工厂模式，可以方便地添加新的咖啡类型和调料，而无需修改现有代码，从而实现代码的复用性和可扩展性。

OCP：系统设计遵循开放-封闭原则（OCP），即对扩展开放，对修改封闭。可以通过增加新的具体 Beverage 类和 CondimentDecorator 类来扩展系统，而不需要修改现有类的代码。

设计优势：

灵活性：可以任意组合咖啡和调料。

可扩展性：添加新的咖啡类型和调料非常简单。

可维护性：系统遵循单一职责原则，使得每个类的职责单一且明确，便于维护和扩展。

3.3.2. 实现

a) Beverage 类：
   i. 方法规格说明 (Spec)：
      1. cost 方法 (抽象)：
         a) Effects： 子类必须实现此方法以返回具体饮料的成本。

b) CondimentDecorator 类：
   i. 方法规格说明 (Spec)：
      1. setCondiment 方法：
         a) Requires：
            i. beverage 是一个有效的饮料对象。
            ii. num 是一个有效的调料数量。
         b) Effects：
            i. 设置被装饰的饮料为指定的饮料对象。
            ii. 设置调料的数量为指定的数量。
         c) Parameters：
            i. beverage： 被装饰的饮料对象，Beverage 类型。
            ii. num： 调料的数量，整数类型。
         d) Modifies：CondimentDecorator 的内部状态 beverage 和 num。

c) DarkRost 类（包括其余 Decaf、Espresso、HouseBlend 类）：
   i. Mutability/Immutability：DarkRost 类是 Beverage 的一个具体实现类，表示一种饮料，它的价格是固定的，没有内部状态需要改变，因此是不可变的（immutable）。
   ii. 抽象函数 (AF)：
      1. DarkRost 类表示一种名为 DarkRost 的饮料，其固定价格为 0.99。
      2. AF(c) = { cost() => 返回 DarkRost 饮料的价格，固定为 0.99，double 类型 }
   iii. 表示不变性 (RI)：DarkRost 类的价格必须始终为 0.99。
   iv. 防止表示暴露 (Safety from Rep Exposure)：DarkRost 类没有可变的内部状态，因此不存在表示暴露的问题。
   v. 方法规格说明 (Spec)：

1. cost 方法：
    a) Effects： 返回 DarkRost 饮料的价格，固定为 0.99。
    b) Returns： 返回 DarkRost 饮料的价格，double 类型。

d) Mocha 类（包括其余 Soy、SteamedMilk、Whip 类）：
   i. Mutability/Immutability：Mocha 类是 CondimentDecorator 的一个具体实现类，表示一种摩卡调料。在他的父类中，存在 setCondiment()方法，可以修改 num 和 beverage 属性，因此 Mocha 类是可变的（mutable）。
   ii. 抽象函数 (AF)：
      1. Mocha 类表示一种摩卡调料，可以添加到饮料中，并根据添加的数量增加饮料的成本。
      2. AF(c) = { cost() => 返回 Mocha 调料添加到饮料中的成本，成本等于被装饰的饮料的成本加上添加的摩卡调料的成本，double 类型 }
   iii. 表示不变性 (RI)：
      1. beverage 必须是一个有效的 Beverage 实例。
      2. num 必须是一个非负整数。
   iv. 防止表示暴露 (Safety from Rep Exposure)：beverage 通过构造函数传递，并且没有提供修改其引用的方法。num 的值只能通过类的内部逻辑修改。
   v. 方法规格说明 (Spec)：
      1. cost 方法：
         a) Effects： 返回 Mocha 调料添加到饮料中的成本，成本等于被装饰的饮料的成本加上添加的摩卡调料的成本。
         b) Returns： 返回 Mocha 调料添加到饮料中的成本，double 类型。

e) SelectorFactory 类：
   i. Mutability/Immutability：SelectorFactory 类用于扫描指定目录中的 Beverage 和 Condiment 类并实例化它们，同时提供计算饮料及其调料总价格的方法。SelectorFactory 是不可变（Immutability）的，因为它的 selectCondimentMap 和 selectBeverageMap 映射可以通过构造函数或 scanAndLoadClasses 方法进行修改，但是在扫描完成之后就固定了。
   ii. 抽象函数 (AF)：
      1. SelectorFactory 表示一个用于扫描指定目录中的饮料和调料类并实例化它们的工厂类，以及提供计算饮料及其调料总价格的方法。
      2. AF(c) = {select(beverageName, condimentSelectorList) => 计算饮料 beverageName 和 condimentSelectorList 中指定的调料的总价格,selectCondimentMap => 包含所有已扫描的 CondimentDecorator 实例的映射,selectBeverageMap => 包含所有已扫描的 Beverage 实例的映射}
   iii. 表示不变性 (RI)：
      1. 目录路径 CONDIMENT_DIRECTORY_PATH 和 BEVERAGE_DIRECTORY_PATH 必须是有效的字符串，不能为 null 或空。
      2. selectCondimentMap 和 selectBeverageMap 映射中的值必须是有效的 CondimentDecorator 和 Beverage 实例。
      3. selectCondimentMap 和 selectBeverageMap 映射中的键必须是类名的小写形式。
   iv. 防止表示暴露 (Safety from Rep Exposure)：该类没有暴露其内部状态给外部，

因此不存在表示暴露的问题。

v. 方法规格说明 (Spec)：

1. SelectorFactory 构造函数：
   a) Requires： 目录路径 CONDIMENT_DIRECTORY_PATH 和 BEVERAGE_DIRECTORY_PATH 必须是有效的字符串，不能为 null 或空。
   b) Effects： 扫描并加载饮料和调料类，将它们的实例存储在 selectCondimentMap 和 selectBeverageMap 映射中。
2. scanAndLoadClasses 方法：
   a) Requires： directoryPath 是一个有效的目录路径，map 是一个有效的映射，expectedClass 是一个有效的类类型。
   b) Effects： 扫描指定目录并将实例加载到 map 中。
   c) Modifies： map。
   d) Throws： IllegalArgumentException 如果目录路径无效或指定目录中没有找到类。
3. select 方法：
   a) Requires： beverageName 是一个有效的饮料名称，condimentSelectorList 是一个有效的调料选择列表。
   b) Effects： 计算指定饮料及其调料的总价格。
   c) Returns： 饮料及其调料的总价格，double 类型。
   d) Throws： IllegalArgumentException 如果未找到指定的饮料或调料。

3.3.3. 测试

BeverageTest 测试策略：验证具体饮料类（DarkRost、Decaf、Espresso、HouseBlend）的功能是否符合预期。

测试方法：

1. testDarkRostCost 方法：测试 DarkRost 饮料的成本。
   a) 步骤：
      i. 创建 DarkRost 实例。
      ii. 调用 cost 方法计算成本。
      iii. 使用 assertEquals 验证成本与预期值是否相等。
2. testDecafCost 方法：测试 Decaf 饮料的成本。
   a) 步骤：
      i. 创建 Decaf 实例。
      ii. 调用 cost 方法计算成本。
      iii. 使用 assertEquals 验证成本与预期值是否相等。
3. testEspressoCost 方法：测试 Espresso 饮料的成本。
   a) 步骤：
      i. 创建 Espresso 实例。
      ii. 调用 cost 方法计算成本。
      iii. 使用 assertEquals 验证成本与预期值是否相等。
4. testHouseBlendCost 方法：测试 HouseBlend 饮料的成本。
   a) 步骤：
      i. 创建 HouseBlend 实例。

ii. 调用 cost 方法计算成本。
iii. 使用 assertEquals 验证成本与预期值是否相等。

测试结果：

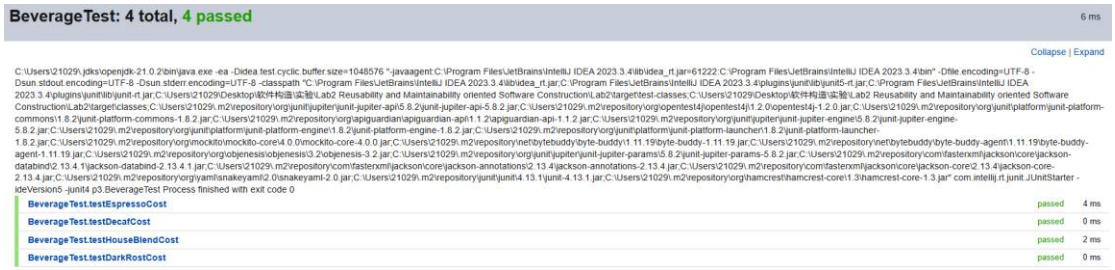

图 3.3.3-1 BeverageTest 测试结果

测试覆盖率：

表 3.3.3-1 BeverageTest 测试结果

Class	Class(%)	Method(%)	Line(%)
DarkRost	100% (1/1)	100% (2/2)	100% (2/2)
Decaf	100% (1/1)	100% (2/2)	100% (2/2)
Espresso	100% (1/1)	100% (2/2)	100% (2/2)
HouseBlend	100% (1/1)	100% (2/2)	100% (2/2)

CondimentDecoratorTest 测试策略：验证具体调料类的功能，确保每种调料的成本计算正确。

测试方法：

1. testMochaCost()方法：测试 Mocha 调料的成本计算。首先创建一个 DarkRost 饮料实例，然后将两份 Mocha 添加到其中。最后，验证返回的成本是否与预期值相等。
   a) 步骤：
      i. 创建一个 DarkRost 饮料实例。
      ii. 创建一个 Mocha 调料实例。
      iii. 将两份 Mocha 调料添加到 DarkRost 饮料中。
      iv. 调用 DarkRost 的 cost() 方法计算成本。
      v. 使用断言验证成本与预期值的相等性。
2. testSoyCost()方法：测试 Soy 调料的成本计算。首先创建一个 Decaf 饮料实例，然后将三份 Soy 添加到其中。最后，验证返回的成本是否与预期值相等。
   a) 步骤：
      i. 创建一个 Decaf 饮料实例。
      ii. 创建一个 Soy 调料实例。
      iii. 将三份 Soy 调料添加到 Decaf 饮料中。
      iv. 调用 Decaf 的 cost() 方法计算成本。
      v. 使用断言验证成本与预期值的相等性。
3. testSteamedMilkCost()方法：测试 SteamedMilk 调料的成本计算。首先创建一个 Espresso 饮料实例，然后将一份 SteamedMilk 添加到其中。最后，验证返回的成本是否与预期值相等。
   a) 步骤：
      i. 创建一个 Espresso 饮料实例。

ii. 创建一个 SteamedMilk 调料实例。
iii. 将一份 SteamedMilk 调料添加到 Espresso 饮料中。
iv. 调用 Espresso 的 cost() 方法计算成本。
v. 使用断言验证成本与预期值的相等性。

4. testWhipCost()方法：测试 Whip 调料的成本计算。首先创建一个 HouseBlend 饮料实例，然后将两份 Whip 添加到其中。最后，验证返回的成本是否与预期值相等。
   a) 步骤：
      i. 创建一个 HouseBlend 饮料实例。
      ii. 创建一个 Whip 调料实例。
      iii. 将两份 Whip 调料添加到 HouseBlend 饮料中。
      iv. 调用 HouseBlend 的 cost() 方法计算成本。
      v. 使用断言验证成本与预期值的相等性。
5. testMultipleCondiments()方法：测试多种调料组合后的成本计算。首先创建一个 Espresso 饮料实例，然后依次添加一份 Mocha、一份 Soy 和一份 Whip，最后检查返回的成本是否与预期值相等。
   a) 步骤：
      i. 创建一个 Espresso 饮料实例。
      ii. 创建一个 Mocha、Soy 和 Whip 调料实例。
      iii. 分别将 Mocha、Soy 和 Whip 调料添加到 Espresso 饮料中。
      iv. 调用 Espresso 的 cost() 方法计算成本。
      v. 使用断言验证成本与预期值的相等性。
6. 环境设置：
   a) 设置 DELTA 为成本比较的容差值。

测试结果：

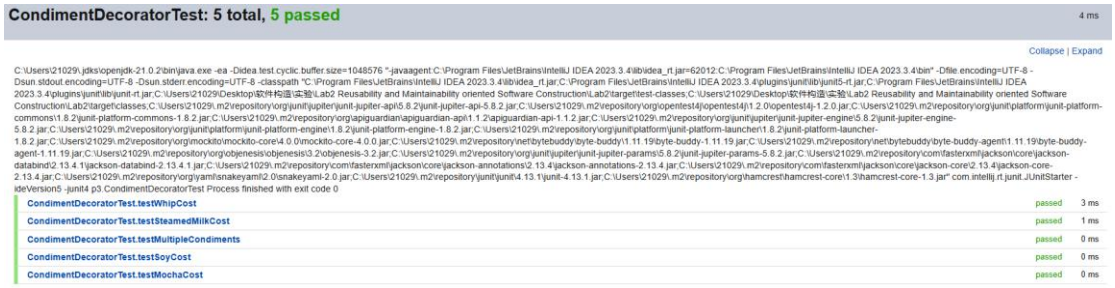

图 3.3.3-2 CondimentDecoratorTest 测试结果

测试覆盖率：

表 3.3.3-2 CondimentDecoratorTest 测试结果

Class	Class(%)	Method(%)	Line(%)
Mocha	100% (1/1)	100% (2/2)	100% (2/2)
Soy	100% (1/1)	100% (2/2)	100% (2/2)
SteamedMilk	100% (1/1)	100% (2/2)	100% (2/2)
Whip	100% (1/1)	100% (2/2)	100% (2/2)

SelectorFactoryTest 测试策略：验证 SelectorFactory 类中的 select 方法在计算饮料及其调料的成本时的正确性和健壮性，以及在输入无效饮料或调料名时是否能正确抛出异常。

测试方法：

1. testSelect 方法：测试 SelectorFactory 类中的 select 方法是否能正确计算饮料及其调料的成本。
   a) 步骤：
      i. 创建 SelectorFactory 实例。
      ii. 测试 DarkRost 配 2 份 Mocha 的成本：
      iii. 创建 Mocha 调料选择器（数量为 2）。
      iv. 调用 select 方法选择 DarkRost 配 Mocha。
      v. 使用 assertEquals 方法验证返回的成本是否为 1.39。
      vi. 测试 Decaf 配 3 份 Soy 的成本：
      vii. 创建 Soy 调料选择器（数量为 3）。
      viii. 调用 select 方法选择 Decaf 配 Soy。
      ix. 使用 assertEquals 方法验证返回的成本是否为 1.35。
      x. 测试 Espresso 配 1 份 SteamedMilk 的成本：
      xi. 创建 SteamedMilk 调料选择器（数量为 1）。
      xii. 调用 select 方法选择 Espresso 配 SteamedMilk。
      xiii. 使用 assertEquals 方法验证返回的成本是否为 2.11。
      xiv. 测试 HouseBlend 配 2 份 Whip 的成本：
      xv. 创建 Whip 调料选择器（数量为 2）。
      xvi. 调用 select 方法选择 HouseBlend 配 Whip。
      xvii. 使用 assertEquals 方法验证返回的成本是否为 1.09。
      xviii. 测试 Espresso 配多种调料的成本：
      xix. 创建 Mocha、Soy、Whip 调料选择器（数量分别为 1）。
      xx. 调用 select 方法选择 Espresso 配 Mocha、Soy、Whip。
      xxi. 使用 assertEquals 方法验证返回的成本是否为 2.39。
2. testInvalidBeverage 方法：测试当输入无效的饮料名时是否抛出 IllegalArgumentException 异常。
   a) 步骤：
      i. 创建 SelectorFactory 实例。
      ii. 调用 select 方法选择无效饮料名 "InvalidBeverage" 配 Mocha。
      iii. 使用 @Test(expected = IllegalArgumentException.class) 注解验证是否抛出 IllegalArgumentException 异常。
3. testInvalidCondiment 方法：测试当输入无效的调料名时是否抛出 IllegalArgumentException 异常。
   a) 步骤：
      i. 创建 SelectorFactory 实例。
      ii. 调用 select 方法选择 Espresso 配无效调料名 "InvalidCondiment"。
      iii. 使用 @Test(expected = IllegalArgumentException.class) 注解验证是否抛出 IllegalArgumentException 异常。
4. testEmptyBeverageName 方法：测试当输入空的饮料名时是否抛出 IllegalArgumentException 异常。
   a) 步骤：
      i. 创建 SelectorFactory 实例。
      ii. 调用 select 方法选择空饮料名 "" 配 Mocha。
      iii. 使用 @Test(expected = IllegalArgumentException.class) 注解验证是否抛出 IllegalArgumentException 异常。

5. testEmptyCondimentName 方法：测试当输入空的调料名时是否抛出 IllegalArgumentException 异常。
   a) 步骤：
      i. 创建 SelectorFactory 实例。
      ii. 调用 select 方法选择 Espresso 配空调料名 ""。
      iii. 使用 @Test(expected = IllegalArgumentException.class) 注解验证是否抛出 IllegalArgumentException 异常。

测试结果：

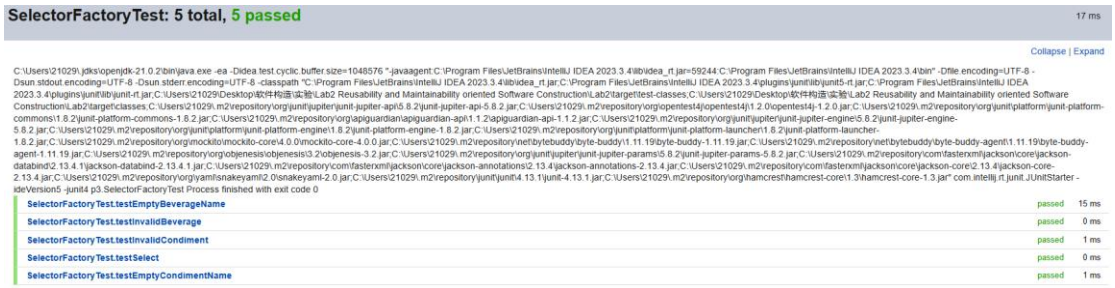

图 3.3.3-3 SelectorFactoryTest 测试结果

测试覆盖率：

表 3.3.3-3 SelectorFactoryTest 测试结果

Class	Class(%)	Method(%)	Branch(%)	Line(%)
SelectorFactory	100% (1/1)	100% (3/3)	75% (12/16)	86.2% (25/29)

4. 总结

在本次软件构造实验中，我深刻体会到了软件设计与开发过程中的关键概念和技术的重要性。实验不仅让我掌握了抽象数据类型（ADT）的实际应用，还通过实践加深了对面向对象编程原理的理解，尤其是对子类型、泛型、多态、重写、重载、继承、代理和组合等概念的运用。

在设计阶段，我尝试使用了简单工厂模式，尽管它在扩展性方面存在不足，但我创新地通过扫描文件目录而非依赖配置文件的方式来动态加载解析器，解决了原有模式的缺点，提高了系统的灵活性和可扩展性。

我认识到设计模式在软件工程中的价值，它们提供了解决常见问题的标准化方案，而工厂模式和依赖注入模式的运用，帮助我构建了更为模块化、解耦合的系统。同时，接口隔离原则的遵守确保了系统组件间的紧密配合，而多态性的应用增强了代码的灵活性和复用性。

总之，本次实验是一次宝贵的学习经历，不仅提升了我对软件设计和开发的理论认识，还锻炼了我的实际编程技能，尤其是在解决实际问题时应用面向对象设计原则的能力。我期待在未来的课程和实践中，继续深化这些知识，不断提高我的软件构造能力。