

2024 年春季学期软件构造

Lab2 实验报告

班号	2211107	学号	2022212029	姓名	唐炜堤
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	1
2. 实验环境配置	1
3. 实验过程	1
3.1. 待开发的应用场景文件解析（必做）	1
3.1.1. 设计	1
3.1.2. 实现	2
3.1.3. 测试	5
3.2. 待开发的应用场景（选做）	7
3.2.1. 设计	7
3.2.2. 实现	7
3.2.3. 测试	8
3.3. 待开发的应用场景-饮料计费系统（必做）	9
3.3.1. 设计	9
3.3.2. 实现	10
3.3.3. 测试	12
4. 总结	13

1. 实验目标概述

本次实验覆盖课程第 5、6、7 和 8 讲的内容。目标是编写具有可复用性和可维护性的软件，主要使用一下软件构造技术：

- (1) 使用 ADT 及其实现，为应用问题开发程序；
- (2) 子类型、泛型、多态、重写、重载；
- (3) 继承、代理、组合；
- (4) 面向对象七原则：SOLID+2EX
- (5) 设计模式

2. 实验环境配置

- (1) 安装 JDK21，并配置环境变量
- (2) 安装 JetBrains IDEA
- (3) 使用 Maven 创建项目
- (4) 使用 Junit 4.13.2 进行测试
- (5) 这里给出 GitHub 上 Lab2 实验的 URL：
https://github.com/2364058719/Lab2_2022212029_tangweidi.git

3. 实验过程

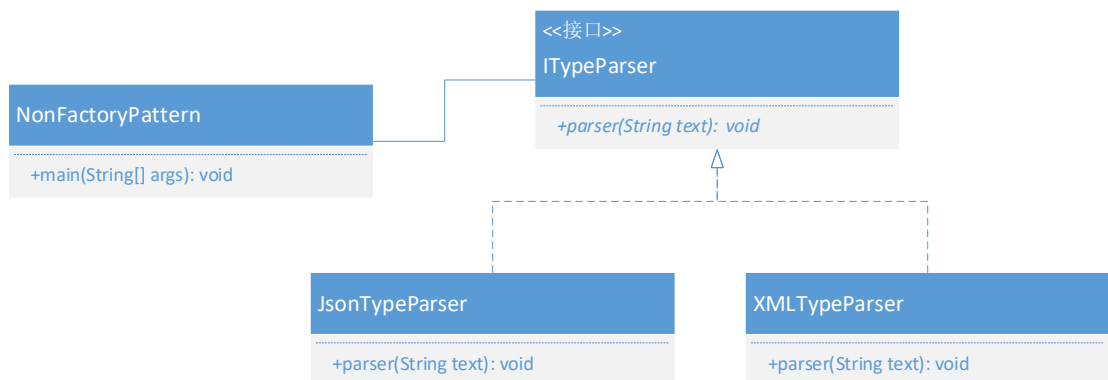
请仔细对照实验手册，针对每一项任务，在下面各节中记录你的实验过程、阐述你的设计思路和问题求解思路，可辅之以示意图或关键源代码加以说明（但千万不要把你的源代码全部粘贴过来！）。

3.1. 待开发的应用场景文件解析（必做）

3.1.1. 设计

在设计过程中，如果一步到位非常困难，可以采用逐步迭代的办法实现。因此，在本小结中，可以将设计图按照你的设计步骤来撰写。

1、设计 1（不使用工厂模式）



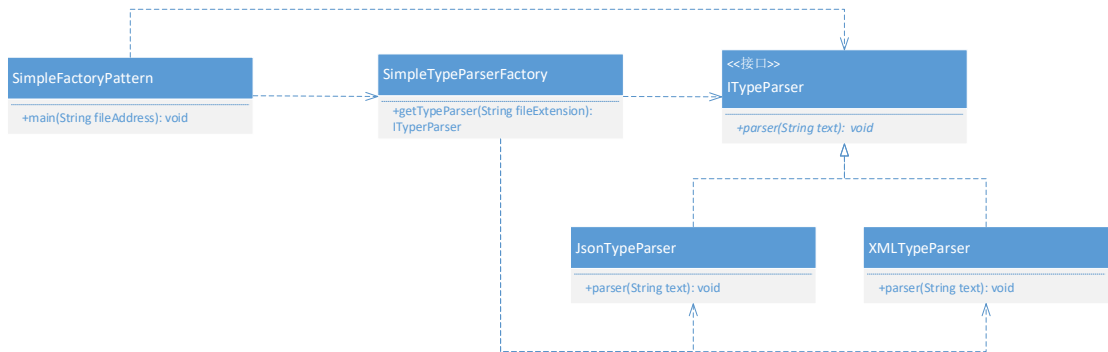
实现了 `ITypeParser` 的层次结构，有利于扩展 `ITypeParser` 的种类。

但是存在一些问题：

(1) 获取文件解析器对象的代码写在 `main` 方法中，即业务方法中，若在同一个业务里，需要多次用到文件解析器，则很多地方会出现重复的方法，不符合 ReUSE 的原则；

(2) 获取文件解析器的对象都在 if-else 中，如果要增加一两个，那么得增加 if-else 中的代码，不符合 OCP 原则。

2、设计 2（简单工厂模式）

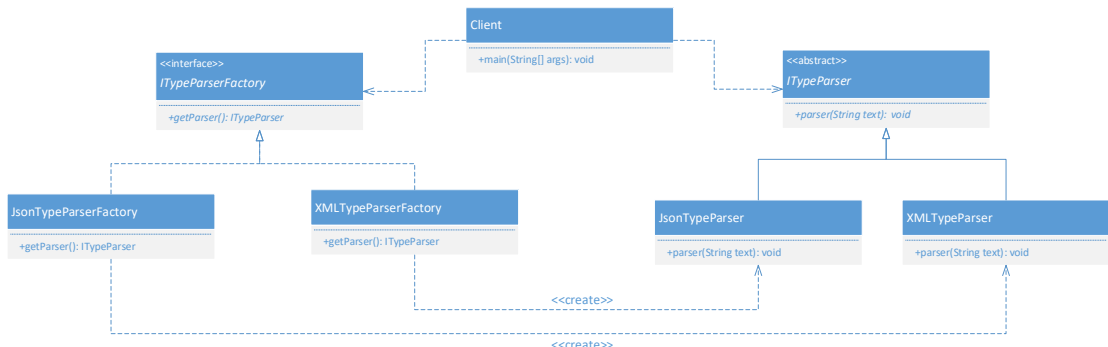


提供了创建 ITypeParser 的方法，调用者通过该方法来获取 ITypeParser。业务代码中不必处理复杂的实例化操作，直接用 SimpleTypeParserFactory 对象产生 ITypeParser 对象。

但存在的问题：

(1) 每次增加一个解析器，是需要在工厂类中去添加一个 if-else，即需要修改这个类中的代码，违反了 OCP 原则。并且在一个类中产生多种类型的 ITypeParser，属于多职责，不符合单一职责原则。

3、设计 3（工厂模式）



3.1.2. 实现

在实现过程中，请为你设计和实现的 ADT 撰写 mutability/immutability 说明、AF、RI、safety from rep exposure。给出各 ADT 中每个方法的 spec。在实际操作中，ADT 对应接口或者抽象类，因此相关的 spec 定义应该在此编写；而 AF、RI、safety from rep exposure 需要在抽象类的子类、接口的实现类中编写。

在本小结，可以仅仅实现最终的设计，其他设计演化的版本不必写。

1、实现 ITypeParser

ITypeParser 的 spec 如下所示：

```

/**
 * immutable 文件解析器
 */
public abstract class ITypeParser {

    /**

```

```

    * 根据给定的配置文件的内容，返回解析之后的文件的内容
    * @param text 配置文件解析之前的内容
    * @return 将配置文件内容经过解析之后的解析内容返回
    */
    public abstract String parser(String text);
}

```

(1) JsonTypeParser 单例类，其中的 AF、RI、safty from rep exposure 如下所示：

```

/**
 * immutable 饿汉式单例类，Json 文件解析器的一个实现
 */
public class JsonTypeParser extends ITypeParser{

    private static final JsonTypeParser jsonTypeParser = new
JsonTypeParser();

    // Abstraction function:
    // AF(jsonTypeParser) = json 文件解析器单例

    // Representation invariant:
    // jsonTypeParser json 文件解析器，单一实例，类被初始化时就被创建，
数量在程序的生命周期中一直为1

    // Safety from rep exposure:
    // 每个字段都是被private 和 final 修饰
    // 所以它们不能被外部直接访问，或者被再分配
}

```

(2) XMLTypeParser 单例类，其中的 AF、RI、safty from rep exposure 如下所示：

```

/**
 * immutable 饿汉式单例类，XML 文件解析器的实现
 */
public class XMLTypeParser extends ITypeParser{

    private static final XMLTypeParser xmlTypeParser = new
XMLTypeParser();

    // Abstraction function:
    // AF(smlTypeParser) = xml 文件解析器单例

    // Representation invariant:
    // xmlTypeParser xml 文件解析器，单一实例，类被初始化时就被创建，数量
在程序的生命周期中一直为1

    // Safety from rep exposure:
    // 每个字段都是被private 和 final 修饰
    // 所以它们不能被外部直接访问，或者被再分配
}

```

```
}
```

2、实现 ITypeParserFactory

ITypeParserFactory 的 spec 如下所示:

```
/**
 * 一个文件解析器工厂，用于生产特定的文件解析器
 * immutable
 */
public interface ITypeParserFactory {
    /**
     * 根据对应的工厂，调用对应的工厂方法，返回对应类型的文件解析器 Parser
     *
     * @return 返回跟工厂类型对应的文件解析器
     */
    public ITypeParser getTypeParser();
}
```

(1) JsonParserFactory 中的 AF、RI、safty from rep exposure 如下所示:

```
/**
 * ITypeParserFactory 的一个实现
 * immutable
 */
public class JsonTypeParserFactory implements
ITypeParserFactory{

    // Abstraction function:
    // None

    // Representation invariant:
    // 在整个生命周期中都能返回 JsonTypeParser

    // Safety from rep exposure:
    // None

}
```

(2) XMLTypeParserFactory 中的 AF、RI、safty from rep exposure 如下所示:

```
/**
 * ITypeParserFactory 的一个实现
 * immutable
 */
public class XMLTypeParserFactory implements ITypeParserFactory
{

    // Abstraction function:
    // None

    // Representation invariant:
```

```

// 在整个生命周期中都能返回 XMLTypeParser

// Safety from rep exposure:
// None
}

```

3.1.1. 测试

1、书写测试 `ITypeParser` 层次的 `testing strategy`。主要对每个需要测试的函数进行输入空间的划分，然后结合输入空间的划分进行“最少一次覆盖”的策略进行测试。其中测试策略如下：

Test <code>JsonTypeParser.paser</code>	Partition for inputs of <code>JsonTypeParser.parser(inputs):</code> * <code>ITypeParser: JsonTypeParser</code> * <code>inputs: normal text, empty text</code>
Test <code>JsonTypeParser.getInstance</code>	Partition for <code>JsonTypeParser.getInstance()</code> * <code>ITypeParser: JsonTypeParser</code>
Test <code>XMLTypeParser.paser</code>	Partition for inputs of <code>XMLTypeParser.parser(inputs):</code> * <code>ITypeParser: XMLTypeParser</code> * <code>inputs: normal text, empty text</code>
Test <code>XMLTypeParser.getInstance</code>	Partition for <code>XMLTypeParser.getInstance()</code> * <code>ITypeParser: XMLTypeParser</code>

2、书写测试 `ITypeParserFactory` 层次的 `testing strategy`。主要对每个需要测试的函数进行输入空间的划分，然后结合输入空间的划分进行“最少一次覆盖”的策略进行测试。其中测试策略如下：

Test <code>getTypeParser</code>	Partition for <code>ITypeParserFactory.getTypeParser()</code> * <code>ITypeParserFactory:</code> <code>JsonTypeParserFactory,</code> <code>XMLTypeParserFactory</code>
---------------------------------	--

3、进行 `TypeParserTest` 测试：

Coverage TypeParserTest x				
Element ^				
Class, %	Method...	Line, %	Branch, %	
com.awei.Parser.P1	33% (2/6)	72% (8/11)	38% (8/21)	0% (0/8)
Client	0% (0/1)	0% (0/1)	0% (0/11)	0% (0/8)
ITypParser	0% (0/1)	100% (0/0)	100% (0/0)	100% (0/0)
ITypParserFactory	100% (0...	100% (0/0)	100% (0/0)	100% (0/0)
JsonTypeParser	100% (1...	100% (4/4)	100% (4/4)	100% (0/0)
JsonTypeParserFactory	0% (0/1)	0% (0/1)	0% (0/1)	100% (0/0)
XMLTypeParser	100% (1...	100% (4/4)	100% (4/4)	100% (0/0)
XMLTypeParserFactory	0% (0/1)	0% (0/1)	0% (0/1)	100% (0/0)

Cover TypeParserTest x	
TypeParserTest (com.awei)	
testJsonTypeParser	4 ms
testXMLTypeParser	0 ms
testXMLTypeParserWithoutIn	1 ms
testJsonTypeParserWithoutIn	0 ms

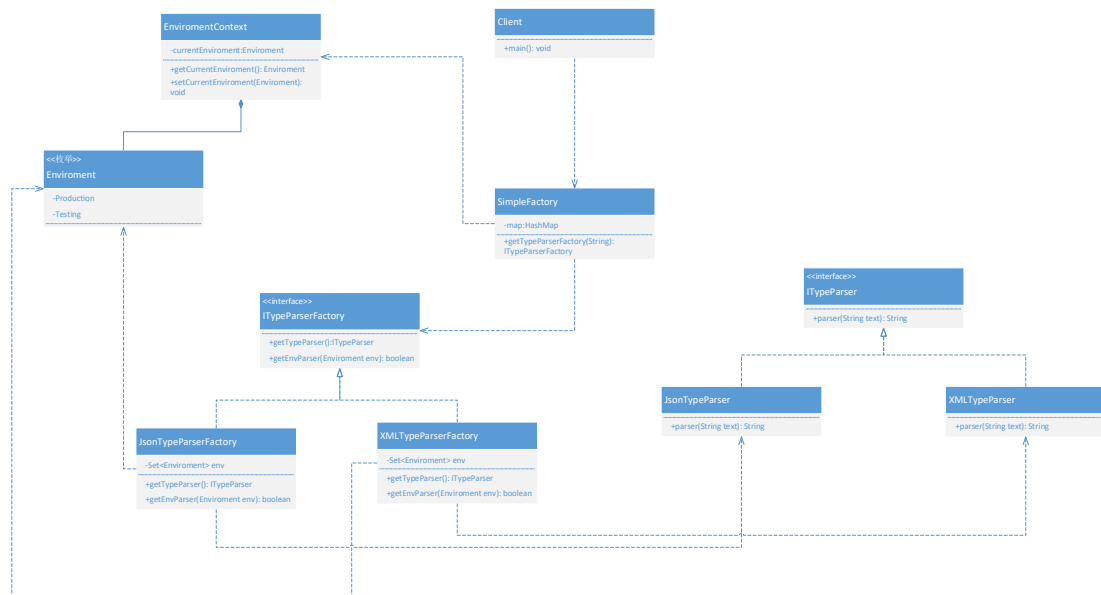
4、进行 TypeParserFactoryTest 测试:

Coverage TypeParserFactoryTest x				
Element ^				
Class, %	Method...	Line, %	Branch, %	
com.awei.Parser.P1	66% (4/6)	72% (8/11)	38% (8/21)	0% (0/8)
Client	0% (0/1)	0% (0/1)	0% (0/11)	0% (0/8)
ITypParser	0% (0/1)	100% (0/0)	100% (0/0)	100% (0/0)
ITypParserFactory	100% (0...	100% (0/0)	100% (0/0)	100% (0/0)
JsonTypeParser	100% (1...	75% (3/4)	75% (3/4)	100% (0/0)
JsonTypeParserFactory	100% (1...	100% (1/1)	100% (1/1)	100% (0/0)
XMLTypeParser	100% (1...	75% (3/4)	75% (3/4)	100% (0/0)
XMLTypeParserFactory	100% (1...	100% (1/1)	100% (1/1)	100% (0/0)

Cover TypeParserFactoryTest x	
TypeParserFactoryTest (com.awei)	
testJsonTypeParserFactory	7 ms
testXMLTypeParserFactory	2 ms

3.2. 待开发的应用场景（选做）

3.2.1. 设计



使用 Enviroment 枚举类来枚举各种环境，通过 EnvironmentContext 来获取当前的环境。SimpleFactory 类中的 map 通过读取 bean.properties 配置文件来获取类，实现 OCP。工厂类中的 set 集合通过读取 env.properties 配置文件来获取其对应的可解析环境。

3.2.2. 实现

修改 ITypeParserFactory 类，增加 getEnvParser 方法：

```
/**
 * 根据给定的环境来判断是否能返回当前类型的文件解析器
 *
 * @param enviroment 给定的当前环境，不为 null
 * @return 如果当前环境能使用这个 Parser 则返回 true，否则返回 false
 */
public boolean getEnvParser(Enviroment enviroment);
```

ITypeParserFactory 的具体实现的 AF、RI、safty from rep exposure 更改为如下所示：

```
public class JsonTypeParserFactory implements ITypeParserFactory {
    private static final Set<Enviroment> env = new HashSet<>();

    // Abstraction function:
    // AF(env) = 能使用该文件解析器的环境集合

    // Representation invariant:
    // 在整个生命周期中都能返回 JsonTypeParser

    // Safety from rep exposure:
    // 每个字段都是被 private 和 final 修饰
    // 所以它们不能被外部直接访问，或者被再分配
```

```
}
```

实现 `EnviromentContext` 类来表示当前的环境，其 AF,RI,safety from rep exposure 如下所示：

```
/**
 * mutable
 */
public class EnviromentContext {
    private static Enviroment currentEnviroment =
Enviroment.Testing;

    /**
     * Abstraction function:
     * AF(currentEnviroment) = 当前使用的环境
     *
     * Representation invariant:
     * 在整个程序运行周期都能返回属于合法 Enviroment 中的一个环境
     *
     * Safety from rep exposure:
     * 使用 private 修饰，外界不能直接修改该 field
     */
}
```

实现 `SimpleFactory`，通过 `SimpleFactory` 来获取对应的文件解析器工厂，运用外观模式，其 AF,RI,safety from rep exposure 如下所示：

```
/**
 * immutable
 */
public class SimpleFactory {
    private static final HashMap<String, ITypeParserFactory> map =
new HashMap<>();

    /**
     * Abstraction function:
     * AF(map) = 一个<K,V>键值对代表一个文件解析器名称与其对应的工厂类
     *
     * Representation invariant:
     * map 不为空，在类被加载时则将 map 赋值，之后不会改变
     *
     * Safety from rep exposure:
     * 使用 private 和 final 修饰，外界无法直接访问或分配其值
     */
}
```

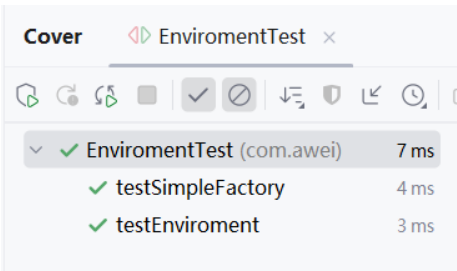
3.2.3. 测试

书写测试 `SimpleFactory` 和 `EnviromentContext` 层次的 testing strategy。主要对每个

需要测试的函数进行输入空间的划分，然后结合输入空间的划分进行“最少一次覆盖”的策略进行测试。其中测试策略如下：

Test getCurrentEnviroment	Partition for <code>EnviromentContext.getCurrentEnviroment()</code> : * 调用该方法查看是否能得到正确返回
Test setCurrentEnviroment	Partition for <code>EnviromentContext.setCurrentEnviroment()</code> : * 调用该方法查看是否能正确将 <code>Enviroment</code> 赋值进去
Test getTypeParserFactory	Partition for <code>SimpleFactory.getTypeParserFactory(inputs)</code> : * 通过给定的环境与文件解析器名称，测试能否正确得到文件解析器

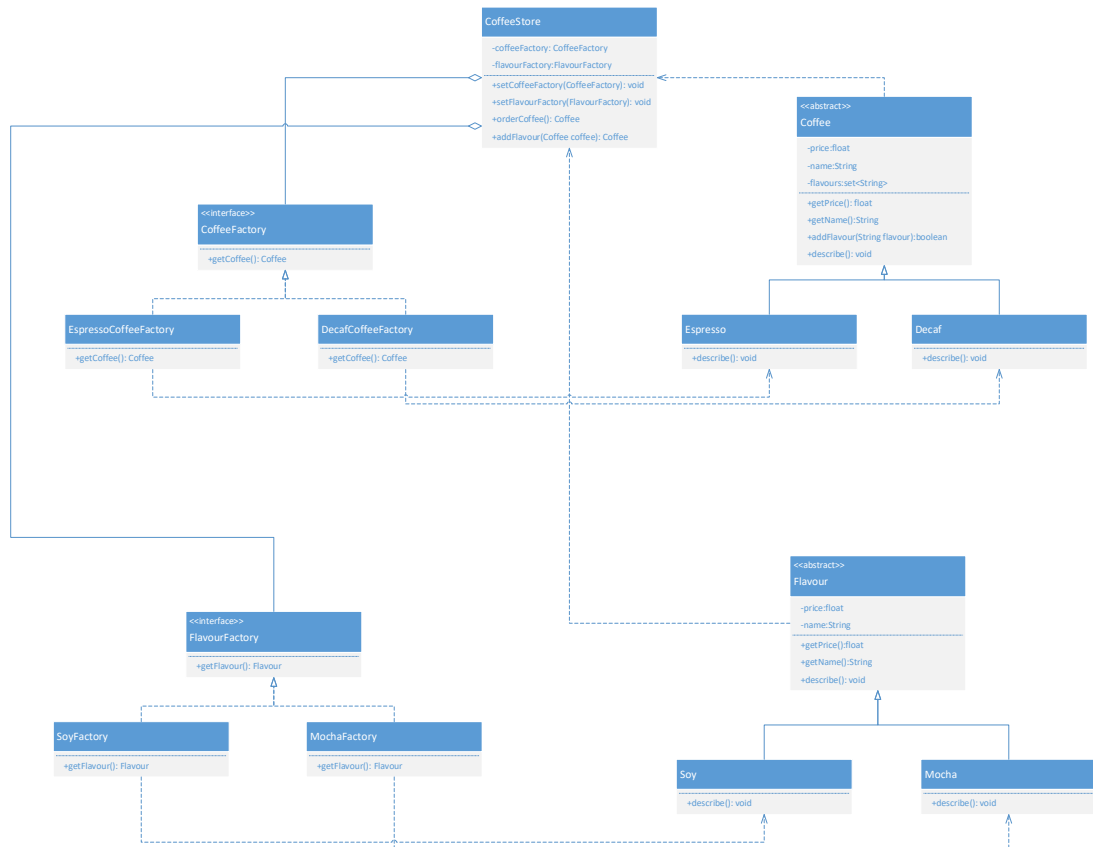
进行 `EnviromentTest` 测试



3.3. 待开发的应用场景-饮料计费系统（必做）

3.3.1. 设计

使用工厂模式，创建 `CoffeeFactory` 和 `FlavourFactory` 两个工厂类。通过抽象类 `Coffee` 和 `Flavour` 两个抽象类，符合 `ReUSE` 原则，将重复的代码写入抽象类。并且在以后加入 `Coffee` 和 `Flavour` 时不需要修改内部代码，只需要增加实体类和工厂类即可，符合 `OCP` 原则。



在 CoffeeStore 类中负责 orderCoffee 和 addFlavour 等。

3.3.2. 实现

1、实现 Coffee 类，其 AF、RI、safty from rep exposure 如下所示：

```
/**
 * mutable
 */
public abstract class Coffee {
    private final String name;
    private int price;
    private final Map<String,Integer> flavours = new
HashMap<String, Integer>();
    /**
     * Abstraction function:
     * AF(name,price,flavours) = 名字叫做 name, 含有 flavours 这些口味, 并
    且价格为 price 的一杯咖啡
     * flavours 中 (key, value), key 表示一种口味, 该口味放了 value 份
     *
     * Representation invariant:
     * name 不为 null, 且不会更改
     * flavours 中每个口味都不相同
     * price 为非负, 即 price >= 0
     */
}
```

```

    * Safety from rep exposure:
    * 每个方法都用 private 修饰, 防止外部访问
    * name 和 flavours 都用 final 修饰, 防止外部再向它分配值
    *
    */
}

```

2、实现 Flavour 类, 其 AF、RI、safty from rep exposure 如下所示:

```

/**
 * immutable
 */
public abstract class Flavour {
    private final String name;
    private final int price;

    /*
     * Abstraction function:
     * AF(name,price) = 名称为 name, 价格为 price 的一个调料口味
     *
     * Representation invariant:
     * name 不为 null, 且不会更改
     * price 为非负, 即 price >= 0
     *
     * Safety from rep exposure:
     * 每个方法都用 private 和 final 修饰, 防止外部访问和再分配
     */
}

```

3、实现 CoffeeStore 类, 其 AF、RI、safty from rep exposure 如下所示:

```

/**
 * mutable 咖啡店对象
 */
public class CoffeeStore {
    private CoffeeFactory coffeeFactory = new DecafCoffeeFactory();
    private FlavourFactory flavourFactory = new
    SoyFlavourFactory();

    /*
     * Abstraction function:
     * AF(coffeeFactory, flavourFactory) = 生产咖啡和生产调料的咖啡店
     *
     * Representation invariant:
     * coffeeFactory 和 flavourFactory 指向一个特定的工厂, 不为 null
     *
     * Safety from rep exposure:
     * 用 private 修饰对象, 防止外部调用、访问该 field
     */
}

```

```
}
```

其中，两个核心业务方法，点咖啡与加调料如下所示：

```
/**
 * 点咖啡
 * @return 返回当前咖啡工厂类型的咖啡对象
 */
public Coffee orderCoffee() {
    return coffeeFactory.getCoffee();
}

/**
 * 向指定咖啡加入调料
 * @param coffee 不为 null，需要添加调料的咖啡
 * @return 返回添加完调料的咖啡
 */
public Coffee orderFlavour(Coffee coffee) {
    //    assert coffee != null;
    checkRep(coffee);
    Flavour flavour = flavourFactory.getFlavour();
    coffee.addFlavour(flavour.getName());
    coffee.addPrice(flavour.getPrice());
    return coffee;
}
```

3.3.3. 测试

1、书写测试 Coffee 层次的 testing strategy。主要对每个需要测试的函数进行输入空间的划分，然后结合输入空间的划分进行“最少一次覆盖”的策略进行测试。其中测试策略如下：

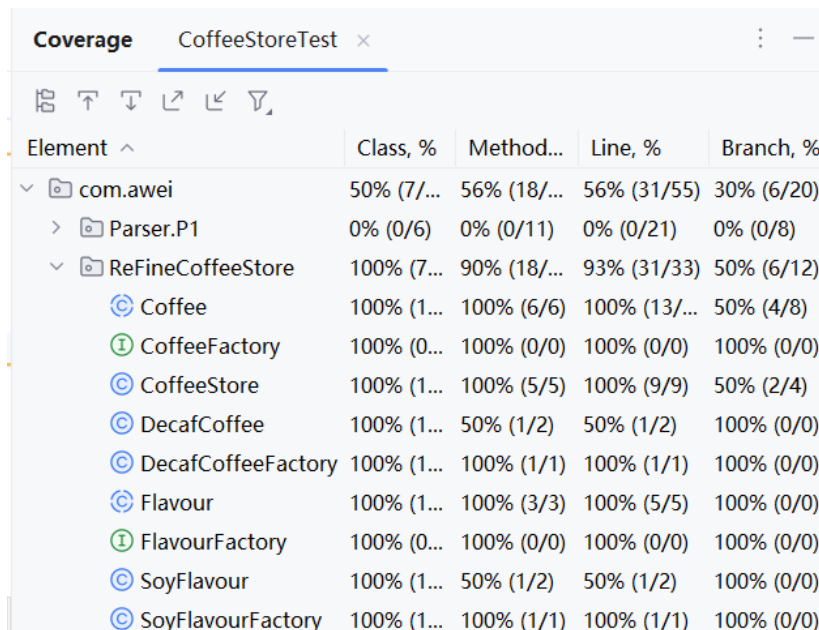
Test addPrice	<i>Partition for the inputs of Coffee.addPrice(inputs):</i> * Coffee:DecafCoffee * inputs:price<0, price>=0
Test addFlavour	<i>Partition for the inputs of Coffee.addFlavour(inputs):</i> * Coffee:DecafCoffee * inputs: empty inputs,normal inputs,repeated inputs

2、书写测试 CoffeeStore 层次的 testing strategy。主要对每个需要测试的函数进行输入空间的划分，然后结合输入空间的划分进行“最少一次覆盖”的策略进行测试。其中测试策略如下：

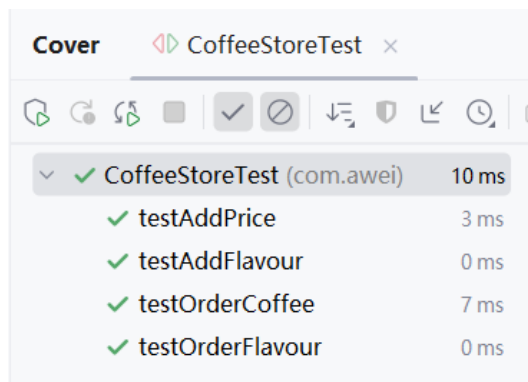
Test orderCoffee	<i>Partition for CoffeeStore.orderCoffee():</i> *检测是否能产生 Coffee 对象（不为
------------------	--

	<code>null</code>)，并且其 <code>field</code> 符合 <code>CoffeeFactory</code> 设置
Test orderFlavour	Partition for the inputs of <code>CoffeeStore.orderFlavour(inputs)</code> : * <code>inputs: normal inputs, empty inputs</code>

3、进行 CoffeStoreTest 测试



Element ^	Class, %	Method...	Line, %	Branch, %
com.awei	50% (7/...	56% (18/...	56% (31/55)	30% (6/20)
> Parser.P1	0% (0/6)	0% (0/11)	0% (0/21)	0% (0/8)
ReFineCoffeeStore	100% (7...	90% (18/...	93% (31/33)	50% (6/12)
Coffee	100% (1...	100% (6/6)	100% (13/...	50% (4/8)
CoffeeFactory	100% (0...	100% (0/0)	100% (0/0)	100% (0/0)
CoffeeStore	100% (1...	100% (5/5)	100% (9/9)	50% (2/4)
DecafCoffee	100% (1...	50% (1/2)	50% (1/2)	100% (0/0)
DecafCoffeeFactory	100% (1...	100% (1/1)	100% (1/1)	100% (0/0)
Flavour	100% (1...	100% (3/3)	100% (5/5)	100% (0/0)
FlavourFactory	100% (0...	100% (0/0)	100% (0/0)	100% (0/0)
SoyFlavour	100% (1...	50% (1/2)	50% (1/2)	100% (0/0)
SoyFlavourFactory	100% (1...	100% (1/1)	100% (1/1)	100% (0/0)



Test Method	Execution Time
✓ CoffeeStoreTest (com.awei)	10 ms
✓ testAddPrice	3 ms
✓ testAddFlavour	0 ms
✓ testOrderCoffee	7 ms
✓ testOrderFlavour	0 ms

4. 总结

通过本次实验，融会贯通了简单工厂方法，工厂方法，抽象工厂方法设计模式与其具体实现。

1、工厂方法：

- (1) 意图：定义一个创建对象（创建对象的工厂）的接口，让具体的工厂类去实现接口，将创建对象的过程延申到了子类。
- (2) 主要解决：接口选择问题。
- (3) 何时使用：计划在不同条件下使用不同的产品（创建不同的实例）。
- (4) 如何解决：让子类（具体的工厂）去实现工厂接口，返回一个抽象产品。

(5) 优点:

扩展性高, 如果想增加一个产品, 只要扩展一个工厂类就可以。

屏蔽产品的具体实现, 调用者只关心产品的接口。

(6) 缺点:

每次增加一个产品时, 都需要增加一个具体类和对象实现工厂, 使得系统中类的个数成倍增加, 在一定程度上增加了系统的复杂度, 同时也增加了系统具体类的依赖。

2、工厂方法的好处:

(1) 封装变化: 创建逻辑有可能变化, 封装成工厂类之后, 创建逻辑的变更对调用者透明。

(2) 代码复用: 创建代码抽离到独立的工厂类之后可以复用。

(3) 隔离复杂性: 封装复杂的创建逻辑, 调用者无需了解如何创建对象

(4) 控制复杂度: 将创建代码抽离出来, 让原本的函数或类职责更单一, 代码更简洁。