软件工程 Lab 3

石睿欣 张逸涵 胡宵宵 宋怡景

一、 阐述本次实验中需求变化对开发工作带来的影响：

1. 对于旧的功能：
   ① 对于已经实现的功能——登录与注册，以及对一种咖啡搭配计算价格的方法，本次 lab 中增加了对这些方法的测试，是开发过程和开发结果的可信度和稳定性更高
   ② 在上一次 lab 的开发中，本组成员在注册函数中调用了自己写的 checkUsername 函数，在此函数中，同时包含了对用户名为空的检测和对用户名是否已存在的检测。而在这次 lab 新增的功能对用户名合法性的检查中，这样写没有真正实现检查用户名是否已存在和检查用户名是否符合规范这个两部分的逻辑上的分离。所以，在这次 lab 中，本组成员将检查用户名是否已存在部分抽出来，形成函数 checkUserExisted，而对于用户名是否符合规范的检查，就放在 checkName 函数中
   ③ 在此次开发中，由于加入了对咖啡总价的测试，本组成员找到了一些之前没有发现的漏洞，并修改了这些漏洞：在实际计算价格之前，先判断传进来的 map 参数是否为 null，或者长度为 0； 同时也增加了对进来的参数中咖啡数量、种类、价格、杯型的合法性的检查，更好地利用到防御式编程的思想，提高了程序的稳定性
2. 对于新的功能：
   ① 采用与之前不同的开发策略，之前的开发过程着重于先实现，再测试，而本次开发采用测试驱动开发，对于用户名和密码的检查，先写测试用例，不仅帮助我们更深入地了解了需求，并且由于对用户名和密码不同输入情况的系统性地全方位的考虑，使得在对用户名和密码的检查的具体实现中，考虑情况覆盖面更全，程序的可靠性更高
   ② 对于这次所写的测试用例，在后续的开发过程中可能还会反复用到，在以后的开发测试中每次增加一个小的增量都对之前的内容做一遍测试，保证每次的开发都是安全的

二、 阐述本次实验中单元测试的设计与实现思想：

0．总体测试设计思路：

1）jmock 在登陆、注册函数的测试中的应用

在登陆和注册时，需要用到数据库中的用户信息。而在对于函数的单元测试中，我们想要剥离函数对于这个外部对象的依赖，于是要用 jmock 来创建 mock 对象模拟 UserRepository 接口的一些行为，如 getUser、createUser。

也就是说，在测试中，我们不关心与数据库连接的问题，直接使用 mock 对象模拟与数据库连接这一过程，我们可以直接设定某一用户是存在的，其密码是多少；也可以直接设定某一用户不存在；也可以调用这个模拟对象创建用户但是不对 csv 文件产生任何影响。

2）jmock 在咖啡对象创建测试中的应用

在测试 cost 函数时，注意到咖啡对象的生成是依赖于 csv 文件的，所以用 jmock 生成咖啡对象，剥离函数 getCoffee（）对外部文件的依赖问题。

3）jmock 具体使用过程：

在@before 时就创建 mock 对象去模拟用户数据库操作接口，设置 checking() 函数来规定模拟对象的行为，最后将该参数传入 accountService,使得在 accountService 中，我们通过这个模拟的对象去调用 getUser、createUser 等方法。

对于模拟对象的行为设计完全依赖于测试用例。如在测试 login 时，我们希望 “starbb_hu” 这个用户是存在的，就设置 getUser(“starbb_hu”)的返回值是一个 user，该 user 的名字是 “starbb_hu” ，密码是 “huhu_123” 。于是，我们就达到了模拟对象行为的效果。

1．实现接口，支持售货员的注册操作：

1)设计思想：

因为注册是通过建立 user 对象进行的，所以先检测传入的参数为 null，以及对象中有 null 值的错误情况；同时，在注册时有对于用户名和密码的要求，针对这些要求进行测试。最后，测试注册成功的情况。

*因为注册涉及到了 UserRepositoryImpl 类中的操作，所以使用 jmock 模拟了真实的创建用户和 isExisted（）函数的实现，避免使得测试中的注册直接改变了 user.csv 文件。

2)实现思想：

测试场景：系统在注册用户时传值 null

输入与期待的输出：

输入	期待输出
传入用户对象为 null	运行异常 “There is a null pointer as the function's parameter”
传入用户对象中的姓名为 null，密码不为 null	运行异常 “There is a null pointer as the function's parameter”
传入用户对象中的姓名为 null，密码也为 null，但此对象不为 null	运行异常 “There is a null pointer as the function's parameter”
传入对象中的姓名不为 null，密码为 null	运行异常 “There is a null pointer as the function's parameter”

期待的 log 信息：异常内容写入 logger

测试场景：售货员注册时有错误输入

输入与期待的输出：

输入	期待输出
传入用户对象中用户名和密码为空（“”）	运行异常 “nvalid username! “
传入用户对象用户名为空，密码正常	运行异常 “nvalid username! “
传入用户对象用户名正常，密码为空	运行异常 “Invalid password!”
传入用户对象密码正常，用户名不以 starbb_开头	运行异常 “nvalid username! “
传入用户对象密码正常，用户名中包含非法字符	运行异常 “nvalid username! “
传入用户对象密码正常，用户名长度小于 8	运行异常 “nvalid username! “
传入用户对象密码正常，用户名长度大于 50	运行异常 “nvalid username! “
传入用户对象用户名正常，密码缺少必须成分	运行异常 “Invalid password!”
传入用户对象用户名正常，密码存在非法字符	运行异常 “Invalid password!”
传入用户对象用户名正常，密码长度小于 8	运行异常 “Invalid password!”
传入用户对象用户名正常，密码长度大于 100	运行异常 “Invalid password!”

期待的 log 信息：异常内容写入 logger

测试场景：售货员注册是用户名已经存在
输入与期待的输出：

输入	期待输出
传入用户对象中用户名已经存在	运行异常 “User already exists, name: {name}”

期待的 log 信息：异常内容写入 logger

测试场景：售货员成功注册
输入与期待的输出：

输入	期待输出	期待的 log 信息
传入正确的用户对象	无异常	“User signup successfully, name: “

2．实现接口，支持售货员的登陆操作：

1）设计思想

由于登陆是通过 user 对象进行的，所以先判断 user 对象为 null 以及 user 对象里有 null 值的错误情况。再测试用户登陆时将密码用户名为空（“”）的情况，再测试当用户名输入正确，密码输错的情况，再测试用户名根本不存在的错误。最后再测试登陆正确的情况。

*因为登陆涉及到了 UserRepositoryImpl 类中的操作，所以使用 jmock 模拟了其中函数（比如 isExisted（））的实现。

2）实现思想

测试场景：系统在注册用户时传值 null

输入与期待的输出：

输入	期待输出
传入用户对象为 null	运行异常 “There is a null pointer as the function's parameter”
传入用户对象中的姓名为 null，密码不为 null	运行异常 “There is a null pointer as the function's parameter”
传入用户对象中的姓名为 null，密码也为 null，但此对象不为 null	运行异常 “There is a null pointer as the function's parameter”
传入对象中的姓名不为 null，密码为 null	运行异常 “There is a null pointer as the function's parameter”

期待的 log 信息：异常内容写入 logger

测试场景：售货员登陆时有错误输入

输入与期待的输出：

输入	期待输出
传入用户对象中用户名和密码为空（“”）	运行异常 “User not found!”
传入用户对象用户名为空，密码正常	运行异常 “User not found!”
传入用户对象用户名不存在，密码正确	运行异常 “User not found!”
传入用户对象用户名存在，密码错误	运行异常 “Username or password error.”
传入用户名对象用户名存在，密码正确但前后多了空格	运行异常 “Username or password error.”

期待的 log 信息：异常内容写入 logger

测试场景：售货员正确登陆

输入与期待的输出

输入	期待的输出	期待的 log 信息
传入可以正确登陆的用户对象	无异常	“User login successfully,name:”

3．实现功能，支持售货员的咖啡搭配操作：

由于只需要搭配一下咖啡算出总价，我们问过汪昕助教表示只需要自己搭配一下（因为需求中并没有要求写一个售货员搭配咖啡订单的函数，只有算咖啡总价的函数），所以没有对其的测试用例，在本测试代码中，将咖啡搭配抽象成如下：

使用 jmock 模拟咖啡搭配操作

```
private Cappuccino jmockCappuccino(int size) {
    Mockery context = new JUnit4Mockery();
    CappuccinoRepository test = context.mock(CappuccinoRepository.class);
    context.checking(new Expectations() {
        {
            Cappuccino cappuccino = new Cappuccino();
            cappuccino.setName("cappuccino");
            cappuccino.setDescription("cappuccino");
            cappuccino.setPrice(20);
            cappuccino.setSize(size);
            allowing(test).getCappuccino( name: "cappuccino");
            will(returnValue(cappuccino));
        }
    });
    return test.getCappuccino( name: "cappuccino");
}
```

4．实现接口，支持咖啡订单的价格计算操作：

1）设计思想：由于咖啡订单的价格计算涉及到选择的咖啡数量、型号、咖啡名称，所以针对这些情况分类讨论错误情况。在此之外同时考虑传入 null 值的错误情况；以及订单先创建成功后又被修改的情况；以及测试创建成功的咖啡对象是否可以返回正确的费用值；最后测试成功的例子；

2）实现思想：

对 priceService 的 cost 函数分析，发现需要参数 Map<Coffee,Integer>order，则测试时需要创建该 order 则需要创建 coffee 对象，coffee 是一个 Javabean 仅存储数据，而它的创建是通过调用 CappuccinoRepositoryImpl 和 EspressoRepositoryImpl 的 getCappuccino 和 getEspressRepository 函数，这两个函数需要对外部文件 cappuccino.csv 和 espress.csv 读取，存在对文件的外部依赖，所以用 jmock 模拟读取文件创建返回 coffee 对象。在测试不同情况时，调用上图的 mock 函数。

测试场景：售货员计算用户点单总价

输入与期待输出

输入	期待输出	期待的 log 信息
正确订单 2 类咖啡每种杯型 1 个	150.0	对每一单咖啡对象 logger “name: {coffee}, size: {size}, number: {number}, price: {price}$”

测试场景：售货员在记录用户点一杯咖啡的订单

输入与期待的输出：

输入	期待输出

1 杯小杯 cappuccino	22.0
1 杯中杯 cappuccino	24.0
1 杯大杯 cappuccino	26.0
1 杯小杯 espresso	24.0
1 杯中杯 espresso	26.0
1 杯大杯 espresso	28.0

期待的 log 信息： “name: {coffee}, size: {size}, number: {number}, price: {price}$”

测试场景：售货员在记录用户订单时把杯型输错了
输入与期待的输出：

输入	期待输出
点了一杯杯型-1 的咖啡	运行异常 "The size of coffee is invalid."
点了一杯杯型超过已有选项的咖啡	运行异常 "The size of coffee is invalid."

期待的 log 信息：异常内容写入 logger

测试场景：售货员在记录用户订单时把杯数写错了
输入与期待输出：

输入	期待输出
杯数写成负数	运行异常 "The number of coffee is negative or zero."
杯数写成 0	运行异常 "The number of coffee is negative or zero."

期待的 log 信息：异常内容写入 logger

测试场景：传入后端的订单
输入与期待输出：

输入	期待输出
表示订单的 map 为 null	运行异常 "The map of order is null."
表示订单的 map 长度为 0，即其中没有咖啡	运行异常 "The number of coffee is negative or zero."
加入订单 map 中的咖啡为 null，杯数不为空	运行异常 "The order has something wrong."

期待的 log 信息：异常内容写入 logger

测试场景：订单建立好了之后被服务员修改出错
输入与期待输出：

输入	期待输出

订单中咖啡之后被修改，咖啡名为 null	运行异常 "Coffee name is wrong."
订单中咖啡之后被修改，咖啡名错误	运行异常 "Coffee name is wrong."
订单中咖啡被修改，使咖啡描述为 null	运行异常 "Coffee description is wrong."
订单中咖啡被修改，使咖啡描述错误	运行异常 "Coffee description is wrong."
订单中咖啡被修改，使价格低于最低价	运行异常 "The order has something wrong."
订单中咖啡被修改，使价格高于最高价	运行异常 "The order has something wrong."

期待的 log 信息：异常内容写入 logger

测试场景：营业员正确输入

输入与期待输出

输入	期待输出
正确输入	无异常，正确的 logger

测试场景：营业员建立咖啡对象时输错

输入与期待输出

输入	期待输出
用错误的名称建立咖啡对象（非 cappuccino 也非 espresso）	运行异常 “Object not found, name: {name}”

期待的 log 信息：异常内容写入 logger

5)实现接口，支持系统对登陆状态的检查：

实现思想：

测试场景：系统查看登陆信息

输入与期待输出：

输入	期待输出	期待的 log 信息
正确登陆后检查当前登陆状态	True	“User has logged in”
未登陆时检查当前登陆状态	False	“Please login”

6．实现接口，支持系统对账户名合法性检查：

1）设计思想：由于对用户名既有前缀的要求也有长度与合法字符的要求，所以在实现中首先要考虑用户名为 null 的情况，并且要对以上三个条件检测。

2）实现思想：

测试场景：系统检测用户名的合法性

输入与期待的输出：

输入	期待输出	期待的 log 信息
Null	False	Invalid username!
正确开头用户名	True	Username is valid:{name}

开头不正确的用户名	False	Invalid username!
用户名开头正确，含有非法字符	False	Invalid username!
用户名开头正确，后缀只含有字母数字下划线中的一种	True	Username is valid:{name}
结构合法，输入密码长度为 0	False	Invalid username!
结构合法，输入密码长度为 7	False	Invalid username!
结构合法，输入密码长度为 8	True	Username is valid:{name}
结构合法，输入密码长度在范围中	True	Username is valid:{name}
结构合法，输入密码长度为 49	True	lUsername is valid:{name}
结构合法，输入密码长度为 50	False	Invalid username!
结构合法，输入密码长度大于 50	False	Invalid username!

7．实现接口，支持系统对密码合法性的检查：

1）设计思想：由于密码必须仅包括字母数字下划线并且三者都要有，长度大于等于 8 小于 100，所以考虑测试密码正确和密码不符合这个规范的情况。在这些情况外，再考虑传入密码为 null 值的错误情况。

2）实现思想：

测试场景：系统检测密码的合法性

输入与期待的输出：

输入	期待输出	期待的 log 信息
Null	False	Invalid password!
正确密码	True	Password is valid
长度合法，不包含合法字符的密码	False	Invalid password!
长度合法，既包含合法字符又包含非法字符的密码	False	Invalid password!
长度合法，输入密码只包括字母数字	False	Invalid password!
长度合法，输入密码只包括字母下划线	False	Invalid password!
长度合法，输入密码只包括数字下划线	False	Invalid password!
长度合法，输入密码只包含数字	False	Invalid password!
长度合法，输入密码只包含字母	False	Invalid password!
长度合法，输入密码只包含下划线	False	Invalid password!
结构合法，输入密码长度为 0	False	Invalid password!
结构合法，输入密码长度为 3	False	Invalid password!
结构合法，输入密码长度为 8	True	Password is valid
结构合法，输入密码长度为 10	True	Password is valid
结构合法，输入密码长度为 100	False	Invalid password!
结构合法，输入密码长度为 101	False	Invalid password!

三、 阐述通过本次实验对单元测试和测试驱动开发的理解：

单元测试：

1.对单元测试额外的感悟：

1）单元测试真的要注重写的时候输入的分类。

如果写输入组合没有一个逻辑，写起来就会杂乱无章。此条尤见于写 checkPassword 的测试驱动开发，按照助教给的不符合情况分类写测试用例，就会非常清楚，而且确保对需求的每种情况了然于心。

2）单元测试书写也需要注意代码可读性和质量。

因为我们最后有同学审阅，如果代码写的不好，很难读。

同时，在我们这次的单元测试中特别还注意了代码风格的一致性。尽可能保证函数处理方式类似。

3）写单元测试需要仔细思考哪些地方需要有单元测试，哪些地方又没必要。

函数有很多，大函数里面还会调用子函数。全部拆开测试是很累的。我感觉那些报错点比较多也比较容易报错的，或者后期需求还会修改的以及逻辑比价复杂的比较需要单元测试。

4）写单元测试可以用来确保不会破坏其他人的代码或者可以用于验证重构后的代码。

如果单元测试写得好，我让队友帮我改一个地方，他们改完只需要跑一遍单元测试，通过了我就很放心了。

5）如果结构设计不好，那么写单元测试就很痛苦。

我们组员对某个函数(cost)的需求有一些争议，归根到底是对这个函数本身业务逻辑的内容产生了分歧。Cost 函数到底该不该负责检查里面 coffee 函数，如果我们对整体业务逻辑架构有足够深刻理解或者之前有一个明确的结构设计，则这种问题就可以得到明确。

2.对单元测试意义的理解：

1）单元测试可以帮助我们更好理解需求。

编写测试时的过程可以帮助自己理解和细化需求，因为在不断分解问题，此条详见于测试驱动开发。

2）单元测试帮助发现代码中的问腿，而且隔离了代码单元和其它部分的依赖，对错误定位的速度也更快。

单元测试考虑出错的情况，就让我发现了之前代码里没有考虑 order 为空的情况。

而且一个测试函数测试一种情况，一旦哪个函数错了就知道是哪边写的有问题。

3.对单元测试本身方法的理解：

1）软件的基本单元是模块、类、方法等，而单元测试就是测试这些单元，

比如测试 priceService 的 cost 函数。

2）对单元的测试需要隔绝外部依赖。

Cost 函数创建 coffee 对象，隔绝了对 csv 文件的依赖。

在 signup 等函数里也特地模拟文件读写接口，隔绝对 use.csv 的依赖。

3）对有状态的对象可能需要一些初始化准备。预期结果判断除了输出结果还需要考虑对象状态变化、异常抛出、外部数据（如文件）变化等。

在测试价格计算的时候要准备好 order 订单。

测试这些函数，都会大量抛出异常，我们考虑了所有异常抛出，还精确的检测抛出异常的信息对不对。

4）单元测试需要注意覆盖率和可重复性。

我们一开始测试用户名是否存在，由于当时没有隔绝依赖，是直接对文件进行用户名写入的，一遍测试结束后没有恢复原来状态，第一次测试通过，但是第

二次就不行了。

上面说到用单元测试验证代码和重构的正确性。如果覆盖率达不到其实就不行了。

测试驱动开发（测试很多都与单元测试相同，这里仅列举一点点不同之处）：

1.对测试驱动开发的额外感悟：

1）测试驱动开发其实就是在设计和编码开始之前就编写测试，但是这需要对接口的定义非常清晰。当然由于我们这次的函数过于简单，其实很容易先写测试用例。

2）测试驱动开发尤其在注意对需求的理解，只有需求理解到位，才能写出覆盖率全条理清楚的测试用例。然后就以写出代码通过测试用例为目标而努力就行了。

3）如果需求变化比较频繁或者程序结构变化频繁，那么写的测试就要经常变化，需求是第一生产力。

四、阐述实现中遇到的问题与解决方案：

1. 在进行注册测试时，我们发现注册测试只可以运行一遍。因为再次运行时，之前的信息都已经写入 csv 文件，会导致“用户已存在”错误。问题的根源就是我们没有将外部文件读写与函数逻辑隔离，导致我们的测试影响了程序的状态，这是不应该出现的。我们的解决方法就是使用 jmock 去模拟外部文件读写的过程，从而实现隔离。

2. 在小组讨论中，我们认为对于 userRepository 这个接口使用 jmock 是必要的，因为我们想要分离文件读写这个外部对象和内部函数。但是，我们发现，在 lab2 的实现中，accountServiceImpl 里判断用户是否存在，我们也直接调用了 Fileutil 来判断用户是否存在。如果要完全隔离文件读写，那我们就必须对 fileutil 也进行 mock。最终，我们参照 java web 中的 DAO 设计模式，认为在我们的 Lab 中，UserRepository 等接口起到的作用就相当于 DAO 接口。即这些 Repository 接口是负责对数据库进行操作的，即增删改查。故我们认为判断用户是否存在，也应该放在这个接口里，因为这也是查的一部分，是对数据库数据操作的一种。因此，我们在 UserRepository 接口里添加了 isExisted()方法，并在接口实现类中调用了 fileutil 里的 exist 方法来实现该函数。这样处理，减少了直接与数据库交互的环节，增加了安全性；同时，也使得程序整体架构的逻辑更加分明。而 fileutil 就是个工具包，当前 lab 里用来封装一些文件读写操作，之后扩展为数据库，可以放一些 sql 操作等。因此，我们不希望对 fileutil 进行 mock。

3. 刚开始写 jmock 的时候，一直会报错。在 debug 的过程中，我们发现 mock 模拟的对象并没有被真正使用，也就是说，我们的 mock 是无效的。导致这种错误的原因是，我们在@before 创建了 mock 模拟对象，但是我们并没有把它传进 accountService 里。此时的 accoutService 里在调用对数据库的操作时，还是使用的匿名对象，如(new UserRepositoryImpl).getUser(“starbb_hu”)。找到了根本原因，也就找到了解决方案。我们在 accountService 里加了一个全局变量：userRepository=new UserRepositoryImpl(); 并暴露了对这个全局变量 set 的接口。我们创建完 mock 对象之后，直接通过该 set 方法传进 mock 对象即可。

4. 在我们之前的实现中，User 类的 setName 方法中，如果名字为空字符串，会报错。因为我们测试的时候需要用到 setName 方法去设置我们的测试用例，如果该函数也报

错的话，我们就无法判断错误的原因是 setName，还是我们要测试的函数。因此，我们删去了这个空字符串的判断，对于用户名的判断一并放在了 checkName 里。密码的处理与之相同。

五、 小组分工：

姓名-commit 用户名

石睿欣 -- achillessanger
胡宵宵 -- 胡宵宵 / HXX5656
宋怡景 -- songyijing
张逸涵 -- zyhlx

Task：

测试：实现接口支持售货员的注册操作 胡宵宵
测试：实现接口，支持售货员的登陆操作 胡宵宵
测试：实现功能，支持售货员的咖啡搭配操作 张逸涵
测试：实现接口，支持咖啡订单的价格计算操作 张逸涵
测试：实现接口，支持系统对登陆状态的检查 石睿欣
测试+实现：实现接口，支持系统对账户名合法性的检查 宋怡景
测试+实现：实现接口，支持系统对密码合法性的检查 石睿欣