作业 1 白盒测试

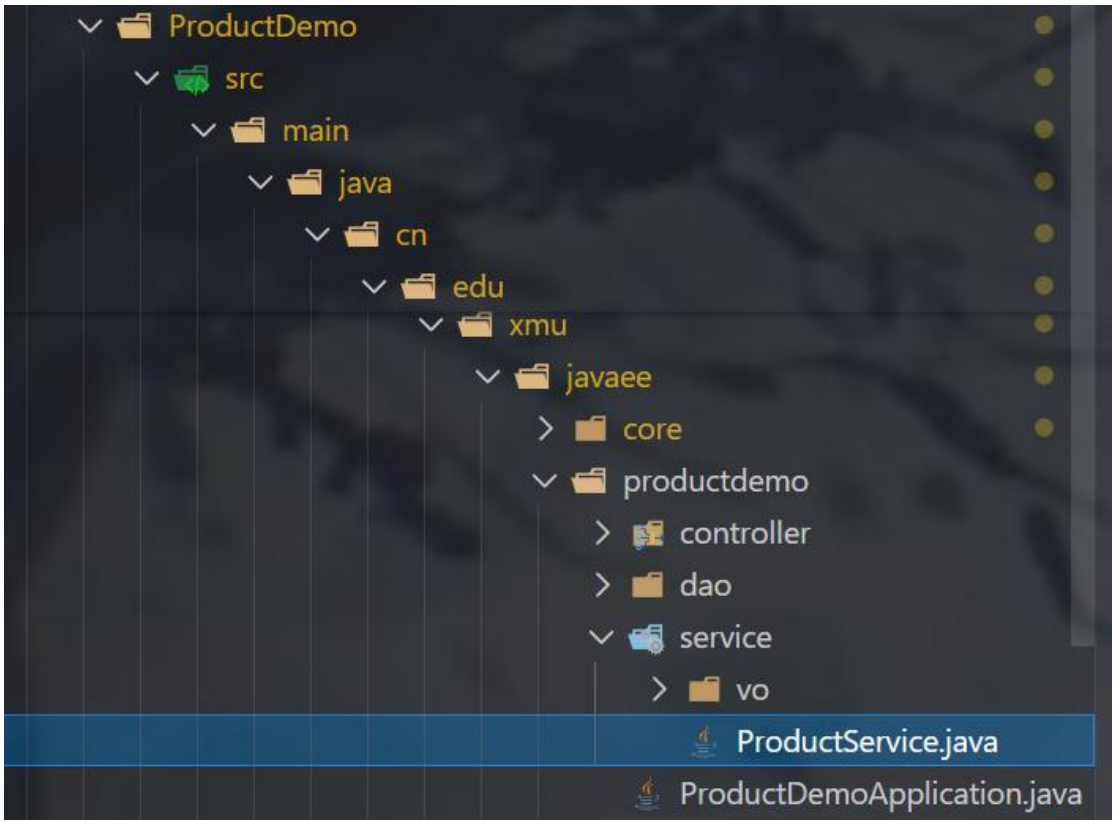
- 白盒测试：
  - 选择一个项目作业中的类，设计单元测试用例，提交测试结果文档截图

被测类: cn.edu.xmu.javaee.productdemo.service.ProductService

该类的功能有：商品草稿（Draft）的创建、查询、删除和更新逻辑。

这边用 jupiter 进行测试

具体工程结构和类代码如下：



```java
package cn.edu.xmu.javaee.productdemo.service;


import cn.edu.xmu.javaee.productdemo.dao.bo.Product;
import cn.edu.xmu.javaee.productdemo.dao.ProductDao;
import cn.edu.xmu.javaee.productdemo.service.vo.ProductVo;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Service;

@Slf4j
@RequiredArgsConstructor
@Service
public class ProductService {

    private final ProductDao productDao;

    public ProductVo createDraft(Product product){
        Product draft =  this.productDao.insert(product);
        ProductVo draftVo = ProductVo.builder().id(draft.getId()).name(draft.getName()).originalPrice(draft.getOriginalPrice()).originPlace(draft.getOriginPlace()).build();
        return draftVo;
    }

    public ProductVo getDraft(Long shopId, Long draftId){
        Product draft = this.productDao.findById(shopId, draftId);
        log.debug("getDraft: draft = {}", draft);
        ProductVo draftVo = ProductVo.builder().id(draft.getId()).name(draft.getName()).originalPrice(draft.getOriginalPrice()).originPlace(draft.getOriginPlace()).build();
        return draftVo;
    }

    public void delDraft(Long shopId, Long draftId){
        this.productDao.findById(shopId, draftId);
        this.productDao.deleteById(draftId);
    }

    public void updateDraft(Long shopId, Product draft){
        this.productDao.findById(shopId, draft.getId());
        this.productDao.updateById(draft);
    }
}

```

对这四个功能分别写测试用例。

采用白盒测试中的单元测试方法。

由于 ProductService 依赖于 ProductDao 进行数据库操作，为了隔离测试目标（Service 层逻辑），我们使用 Mockito 框架对 ProductDao 进行模拟。

Mock 对象: @Mock private ProductDao productDao;

注入对象: @InjectMocks private ProductService productService;

验证点:

验证 Service 方法的返回值是否符合预期。

验证 Service 方法是否正确调用了 Dao 层的相应方法（参数匹配、调用次数）。

测试流程和验证方式

测试创建草稿 (testCreateDraft):

测试目标: 验证 createDraft 方法能否正确调用 Dao 层插入数据，并返回正确的视图对象（VO）。

前置条件 (Given):

构造一个输入的 Product 对象（无 ID）。

构造一个模拟 Dao 层返回的 Product 对象（包含生成的 ID 1L）。

Mock 行为: 当调用 productDao.insert 时，返回带有 ID 的 Product 对象。

执行操作 (When): 调用 productService.createDraft(product)。

断言验证 (Then):

验证返回的 ProductVo 的 ID 是否为 1L。

验证返回的 ProductVo 的属性（名称、价格、产地）是否与输入一致。

验证 productDao.insert 方法被调用了 1 次。

代码如下:

```java
package cn.edu.xmu.javaee.productdemo.service;

import cn.edu.xmu.javaee.productdemo.dao.ProductDao;
import cn.edu.xmu.javaee.productdemo.dao.bo.Product;
import cn.edu.xmu.javaee.productdemo.service.vo.ProductVo;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.Mockito.*;

@ExtendWith(MockitoExtension.class)
public class ProductServiceTest {

    @Mock
    private ProductDao productDao;

    @InjectMocks
    private ProductService productService;

    @Test
    public void testCreateDraft() {
        Product product = Product.builder().name(name: "Test Product").originalPrice(originalPrice: 100L).originPlace(originPlace: "Xiamen").build();
        Product savedProduct = Product.builder().id(id: 1L).name(name: "Test Product").originalPrice(originalPrice: 100L).originPlace(originPlace: "Xiamen").build();

        when(productDao.insert(any(Product.class))).thenReturn(savedProduct);

        ProductVo result = productService.createDraft(product);

        assertEquals(1L, result.getId());
        assertEquals("Test Product", result.getName());
        assertEquals(100L, result.getOriginalPrice());
        assertEquals("Xiamen", result.getOriginPlace());
        verify(productDao, times(1)).insert(any(Product.class));
    }
```

测试查询草稿 (testGetDraft):

测试目标: 验证 getDraft 方法能否根据 ID 正确查询数据并转换为 VO 对象。

前置条件 (Given):

定义 shopId = 1L, draftId = 1L。

Mock 行为: 当调用 productDao.findById(1L, 1L) 时，返回一个预设的 Product 对象。

执行操作 (When): 调用 productService.getDraft(1L, 1L)。

断言验证 (Then):

验证返回对象的 ID、名称、价格等属性是否与 Mock 数据一致。

验证 productDao.findById 方法被调用了 1 次。

代码如下:

```java
@Test
public void testGetDraft() {
    Long shopId = 1L;
    Long draftId = 1L;
    Product product = Product.builder().id(draftId).name(name: "Test Product").originalPrice(originalPrice: 100L).originPlace(originPlace: "Xiamen").build();

    when(productDao.findById(shopId, draftId)).thenReturn(product);

    ProductVo result = productService.getDraft(shopId, draftId);

    assertEquals(draftId, result.getId());
    assertEquals("Test Product", result.getName());
    assertEquals(100L, result.getOriginalPrice());
    assertEquals("Xiamen", result.getOriginPlace());
    verify(productDao, times(1)).findById(shopId, draftId);
}
```

测试删除草稿 (testDelDraft):

测试目标: 验证 delDraft 方法是否执行了存在性检查并调用了删除操作。

前置条件 (Given): 定义 shopId = 1L, draftId = 1L。

执行操作 (When): 调用 productService.delDraft(1L, 1L)。

断言验证 (Then):

验证 productDao.findById(shopId, draftId) 被调用 1 次（模拟业务逻辑中的存在性检查）。

验证 productDao.deleteById(draftId) 被调用 1 次（模拟实际删除操作）。

代码如下:

```java
@Test
public void testDelDraft() {
    Long shopId = 1L;
    Long draftId = 1L;

    productService.delDraft(shopId, draftId);

    verify(productDao, times(1)).findById(shopId, draftId);
    verify(productDao, times(1)).deleteById(draftId);
}
```

测试更新草稿 (testUpdateDraft):

测试目标: 验证 updateDraft 方法是否执行了存在性检查并调用了更新操作。

前置条件 (Given):

定义 shopId = 1L。

构造一个待更新的 Product 对象（ID 为 1L）。

执行操作 (When): 调用 productService.updateDraft(shopId, draft)。

断言验证 (Then):

验证 productDao.findById(shopId, 1L) 被调用 1 次。

验证 productDao.updateById(draft) 被调用 1 次。

代码如下:

```java
@Test
public void testUpdateDraft() {
    Long shopId = 1L;
    Product draft = Product.builder().id(id: 1L).name(name: "Updated Product").build();

    productService.updateDraft(shopId, draft);

    verify(productDao, times(1)).findById(shopId, draft.getId());
    verify(productDao, times(1)).updateById(draft);
}
}
```

测试结果

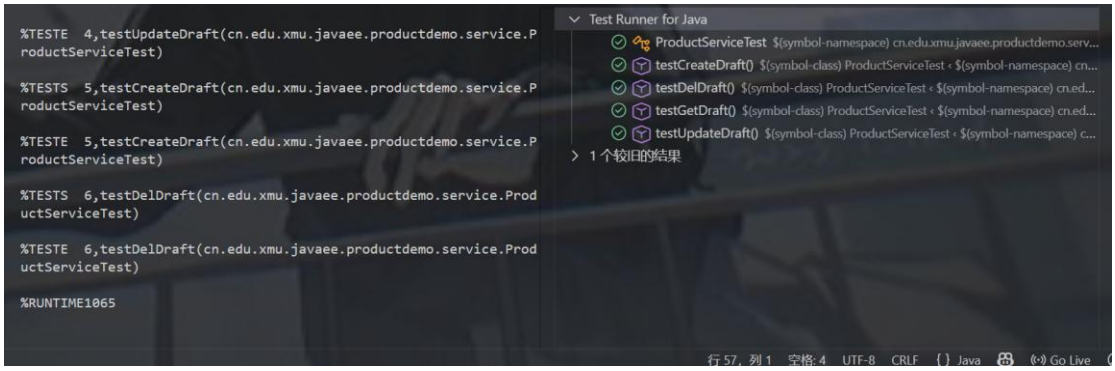
所有单元测试用例均通过 (Passed)。

总用例数: 4

通过数: 4

失败数: 0

覆盖率: 覆盖了 ProductService 中所有主要的增删改查逻辑路径。

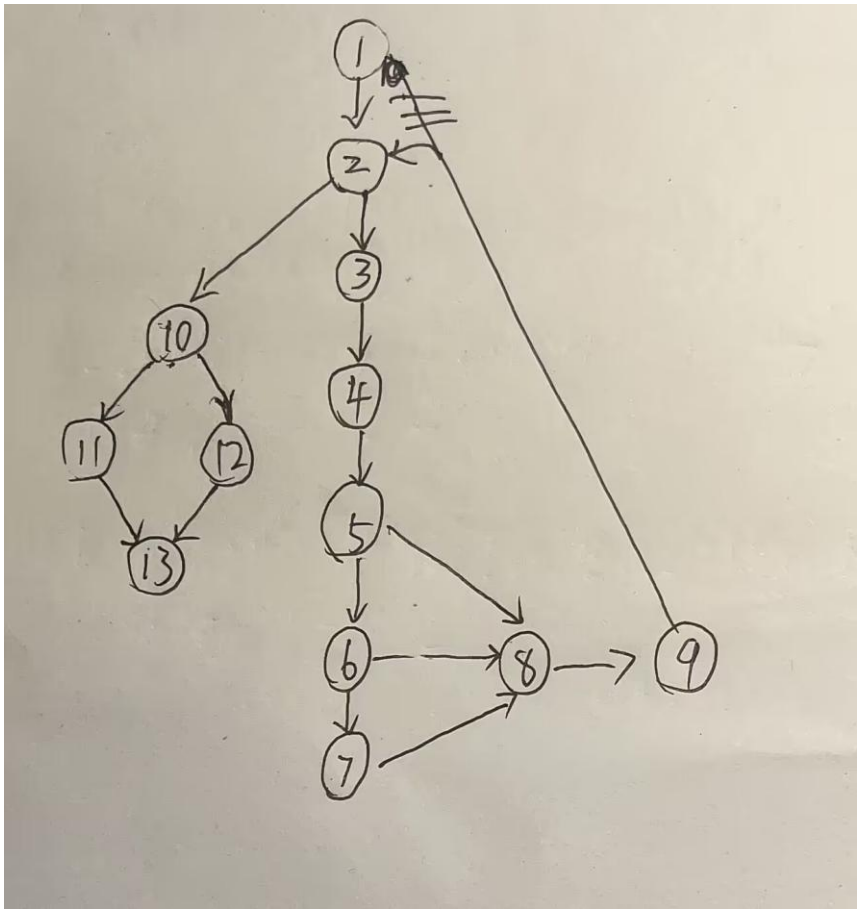


作业 2

- 采用基本路径测试方法为下面的程序设计测试用例，并写明主要过程。

```
Float sum=0; total=0;
Float average (value , minimum , maximum)
Float value[100];
Int minimum , maximum ;
{int inputnum , i;
    float aver;
    i=1 ; inputnum=0;
    While (value[i] !=-999 && inputnum<100)
    { inputnum ++;
      if (value[i]>=minimum && value[i]<= maximum)
      { total ++;
        sum=sum + value[i];}
      i++;
    }
    if (total >0)
        aver=sum / total ;
    else  aver = -999;
return (aver)
}
```

根据提供的代码画出流图如下



可以找出来 6 条路径，如下：

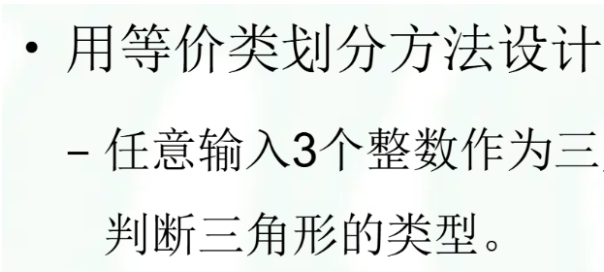
1. 1—2—10—11—13

2. 1—2—10—12—13
3. 1—2—3—10—12—13
4. 1—2—3—4—5—6—7—8—9—2—10—11—13
5. 1—2—3—4—5—6—8—9—2—10—12—13
6. 1—2—3—4—5—8—9—2—10—12—13

再对这六条路径设计测试用例：

1. 路径 1 无法设计测试用例
2. Value[1] = -999，maximum =0 ,minimum=0
3. 路径 3 无法设计测试用例
4. Value[0] =-50，value[1] = -999，maximum = -30,minimum= -60
5. Value[0] =50，value[1] = -999， maximum = 40,minimum=30
6. Value[0] =50，value[1] = -999， ，maximum =30 ,minimum= 80

作业 3



等价类划分及测试用例设计

基于“输入 3 个整数作为三角形三边，判断类型”的需求，等价类分为**有效等价类**（满足三角形条件）和**无效等价类**（不满足三角形条件 / 输入不合法），测试用例如表所示：

用例编号	输入数据	预期结果	覆盖等价类
1	3,4,5	是三角形，类型为不等	有效等价类（不等边）

用例编号	输入数据	预期结果	覆盖等价类
		边三角形	
2	2,2,3	是三角形，类型为等腰三角形	有效等价类（等腰）
3	5,5,5	是三角形，类型为等边三角形	有效等价类（等边）
4	0,1,2	不是三角形	无效等价类（含 0）
5	-1,2,3	不是三角形	无效等价类（含负数）
6	1,2,3	不是三角形	无效等价类（两边和等于第三边）
7	1,2,4	不是三角形	无效等价类（两边和小于第三边）