250 028 663

CS5001: PRACTICAL-2 REPORT

OVERVIEW

The purpose of this practical was to gain hands-on experience with object-oriented programming principles, coding to predefined interfaces, and writing effective unit tests. I developed full implementations for the vending machine, product, and product record interfaces within the impl package library, and created corresponding JUnit 5 tests using the "Test Package for Java" VS Code extension to verify their functionality. My implementation successfully covered all core requirements, including handling product creation, stock tracking, vending operations, and exception cases for unavailable items.

The strongest areas of my work were the clarity and consistency of my unit tests, which supported a test-driven development approach and included printing to the console at times for improved clarity as well. One area for improvement would be refining certain edge-case handling to make the implementation more efficient and maintainable as well as creating more error-specific exception handling within in the exceptions package folder.

DESIGN AND IMPLEMENTATION

The design followed a clear object-oriented approach based on the provided interfaces, ensuring each class had a single, well-defined responsibility.

- **VendingMachineProduct**: Represents a specific product line in a vending machine, identified by a lane code (e.g. "A1") and a description (e.g. "Coca Cola"). The class ensures data integrity by validating its inputs and implements equality and hashing based on the lane code to prevent duplication. I also added checks to ensure lane codes were valid.
- **ProductRecord**: Maintains information about stock levels and sales for a given product. It implements the IProductRecord interface and stores counts of available items and total sales, providing methods to restock (addItem) and purchase (buyItem). Exception handling is built in via ProductUnavailableException to ensure robustness when stock runs out.
- **availableProducts**: Tracks all products currently known to the system. I chose a List over an array to allow dynamic resizing and easy addition/removal during tests.
- **Factory**: Acts as a simple object creator that abstracts the instantiation of different implementation classes. This design follows the Factory Method pattern, helping to decouple the test code from specific constructor calls and enforce coding to interfaces.

Overall, the system is composed of classes connected by interface implementation rather than inheritance, ensuring flexibility and compliance with the assignment's requirement to code to given interfaces. The design prioritised simplicity and clarity, with data structures chosen for readability and maintainability rather than performance optimisation.

INTERESTING FEATURES

While the implementation closely followed the given specification, I made several design choices and enhancements to improve functionality, maintainability, and test reliability beyond a basic version.

- The ProductRecord class maintains a static list of all available products across all records. This provides a convenient overview of all items currently known to the vending system, allowing cross-checks and validation between tests.
- Input validation in the VendingMachineProduct constructor prevents invalid lane codes or products from being created.
- The equals and hashCode methods in VendingMachineProduct are overridden to ensure that each product is uniquely identified by its lane code. This ensures two products are not saved on the same lane.
- The testing method I chose to create focussed on small methods that isolated specific behaviours. This made debugging easier as I knew exactly what the issue was (compared to testing multiple methods and actions at once).

PROBLEMS ENCOUNTERED

I had difficulty initially setting up the "Test Runner for Java" VS Code extension and getting it to work on my tests. However, this error seemed to stem larger from the simple need to close and reopen VS Code entirely.

Additionally, I had a hard time with my overall system function test as products were somehow persisting from previous tests. This meant that when run alone, my test passed, but when run all together: it failed. I went around this problem by creating a variable that counts all the products in the cache before doing the equals validation for product count.

RUNNING THE PROGRAM

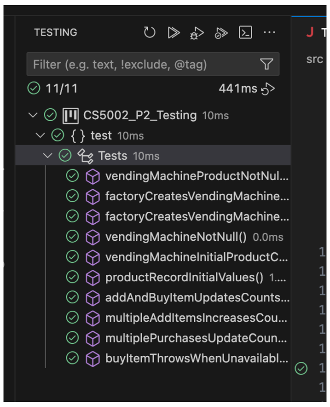
The program can be run via the JUnit test class Tests.java using the "Test Runner for Java" VS Code extension.

Testing
A comprehensive set of 10 tests were developed to verify that each class behaved according to the specification and handled both normal and exceptional conditions correctly. The tests followed a Test-Driven Development style, with each test focusing on one clearly defined aspect of the vending machine functionality.

I divided the testing process was divided into categories:

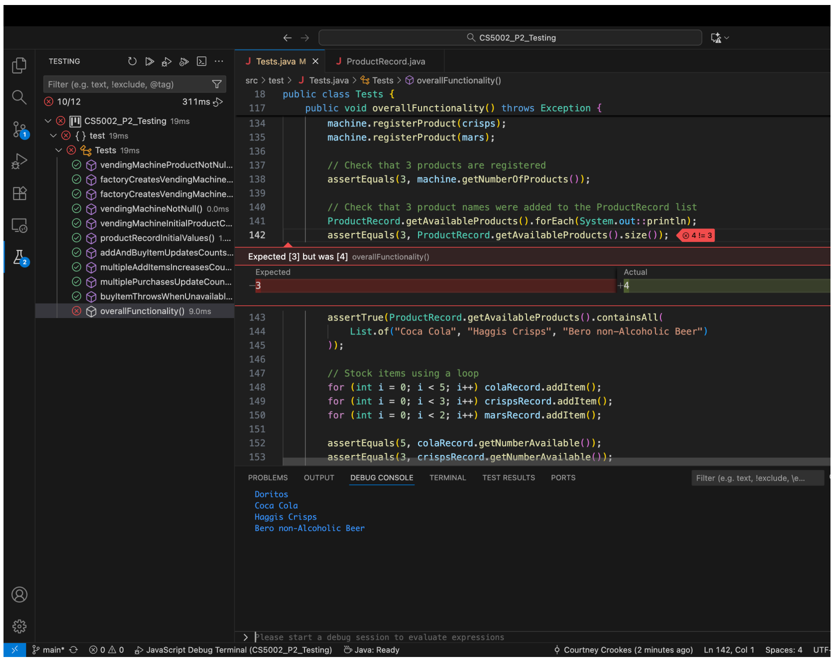
- **Factory Tests:** Verified that the factory methods correctly created non-null instances of all required components. These ensured that the Factory class correctly linked to constructors without direct instantiation in the tests.
- **Product Record Tests:** Checked that a new ProductRecord started with zero available items and zero sales. Tests such as addAndBuyItemUpdatesCounts and multipleAddItemsIncreasesCountCorrectly confirmed that item counts and sales updated accurately under normal operation. Boundary conditions were also explored—for example, buying when stock was zero to confirm that a ProductUnavailableException was thrown.
- **Vending Machine Tests:** Confirmed that the machine initially contained zero products and correctly updated its product count after registration.
- **Edge and Exception Cases:** Specific tests (such as buyItemThrowsWhenUnavailable) checked exception handling, ensuring the program behaved predictably when users attempted invalid operations, such as purchasing unavailable products.
- **Integration Test:** The overallFunctionality test simulated a complete vending session. From product creation and stocking to purchasing and exception handling. This verified that all components interacted correctly and that the shared static availableProducts list remained consistent across records. It also exposed a persistence issue between my tests, demonstrating how I needed take care with static and carefully managed it to avoid state leakage between runs.

All tests passed successfully when run independently, and after adjustments to isolate static data, they also passed as a suite. The results demonstrated that the implementation handled normal operation, boundary conditions, and error cases reliably, providing strong confidence in the robustness of the design.



```
TESTING
11/11                                   441ms
CS5002_P2_Testing 10ms
  { } test 10ms
    Tests 10ms
      vendingMachineProductNotNul...
      factoryCreatesVendingMachine...
      factoryCreatesVendingMachine...
      vendingMachineNotNull() 0.0ms
      vendingMachineInitialProductC...
      productRecordInitialValues() 1....
      addAndBuyItemUpdatesCounts...
      multipleAddItemsIncreasesCou...
      multiplePurchasesUpdateCoun...
      buyItemThrowsWhenUnavailabl...
```

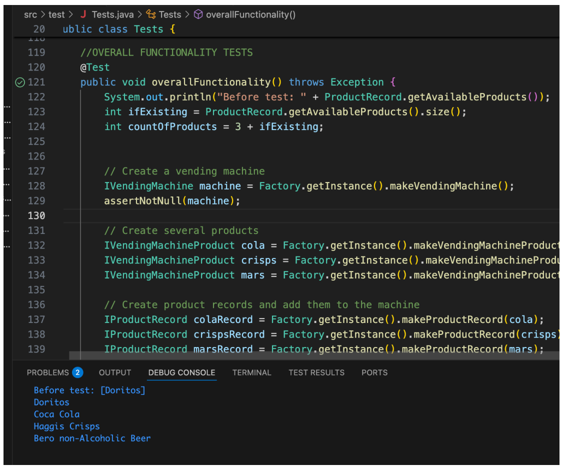
Left: All tests passing



```
TESTING
10/12                                   311ms
CS5002_P2_Testing 19ms
  { } test 19ms
    Tests 19ms
      vendingMachineProductNotNul...
      factoryCreatesVendingMachine...
      factoryCreatesVendingMachine...
      vendingMachineNotNull() 0.0ms
      vendingMachineInitialProductC...
      productRecordInitialValues() 1....
      addAndBuyItemUpdatesCounts...
      multipleAddItemsIncreasesCou...
      multiplePurchasesUpdateCoun...
      buyItemThrowsWhenUnavailabl...
      overallFunctionality() 9.0ms
```

```java
public class Tests {
    public void overallFunctionality() throws Exception {
        machine.registerProduct(crisps);
        machine.registerProduct(mars);

        // Check that 3 products are registered
        assertEquals(3, machine.getNumberOfProducts());

        // Check that 3 product names were added to the ProductRecord list
        ProductRecord.getAvailableProducts().forEach(System.out::println);
        assertEquals(3, ProductRecord.getAvailableProducts().size());   4 != 3
Expected [3] but was [4] overallFunctionality()
Expected                                  Actual
-3                                        +4
        assertTrue(ProductRecord.getAvailableProducts().containsAll(
            List.of("Coca Cola", "Haggis Crisps", "Bero non-Alcoholic Beer")
        ));

        // Stock items using a loop
        for (int i = 0; i < 5; i++) colaRecord.addItem();
        for (int i = 0; i < 3; i++) crispsRecord.addItem();
        for (int i = 0; i < 2; i++) marsRecord.addItem();

        assertEquals(5, colaRecord.getNumberAvailable());
        assertEquals(3, crispsRecord.getNumberAvailable());
```

```
Doritos
Coca Cola
Haggis Crisps
Bero non-Alcoholic Beer
```



```java
ublic class Tests {

    //OVERALL FUNCTIONALITY TESTS
    @Test
    public void overallFunctionality() throws Exception {
        System.out.println("Before test: " + ProductRecord.getAvailableProducts());
        int ifExisting = ProductRecord.getAvailableProducts().size();
        int countOfProducts = 3 + ifExisting;


        // Create a vending machine
        IVendingMachine machine = Factory.getInstance().makeVendingMachine();
        assertNotNull(machine);

        // Create several products
        IVendingMachineProduct cola = Factory.getInstance().makeVendingMachineProduct
        IVendingMachineProduct crisps = Factory.getInstance().makeVendingMachineProdu
        IVendingMachineProduct mars = Factory.getInstance().makeVendingMachineProduct

        // Create product records and add them to the machine
        IProductRecord colaRecord = Factory.getInstance().makeProductRecord(cola);
        IProductRecord crispsRecord = Factory.getInstance().makeProductRecord(crisps
        IProductRecord marsRecord = Factory.getInstance().makeProductRecord(mars);
```

```
Before test: [Doritos]
Doritos
Coca Cola
Haggis Crisps
Bero non-Alcoholic Beer
```

Left: State persistence issue.

Above: Correction of the issue by adding items from previous tests to the counter.