

CS2001/CS2101 Week 5 Practical

JUnit and TDD

12th October 2020

Matriculation Number: 200002548

Tutor: Alex Konovalov

Contents

1	Introduction	2
2	Design	2
2.1	VendingMachineProduct Class	2
2.2	ProductRecord Class	3
2.3	VendingMachine Class	3
3	Testing	4
3.1	IVendingMachineProduct Tests	4
3.2	IProductRecord Tests	6
3.3	IVendingMachine Tests	8
4	Conclusion	12

1 Introduction

The aim of this practical was to follow the Test Driven Development (TDD) methodology, to create an implementation of an ADT interface in Java using the JUnit framework. The ADT interfaces in question were `IVendingMachine`, `IProductRecord` and `IVendingMachineProduct`. They outlined the operation of a basic vending machine, whereby different products were stored in discrete lanes, and products could be bought or sold from a specific lane.

Test Driven Development is a type of software development whereby unit tests are written and ran before implementation. Therefore the implementation is forced to adhere to the test specification, producing a robust program. These unit tests should be focused and independent from each other, the environment they are ran in, and the order of execution.

Therefore, unit tests and implementation of the interfaces were carried out sequentially, for all methods in each interface. The unit tests were to cover the full range of of cases. Defining what should happen in normal, edge and exceptional cases.

2 Design

There were three different classes `VendingMachineProduct`, `ProductRecord` and `VendingMachine` used to implement the respective `IVendingMachineProduct`, `IProductRecord` and `IVendingMachine` interfaces. Overall implementation was designed to be as maintainable and efficient as possible. There were some considerations that had to be made to design where the specification was vague:

- A lane is considered to be still open to a certain type of product even if there are no products left in the lane.
- Products with the same description can feature on multiple different lanes.
- There is no max limit on the number of items in each lane.
- `IVendingMachineProduct`'s can have null `laneCodes` and `desriptions`.
- A vending machine can initially have no products registered.
- If `IVendingMachines`'s `getMostPopular` method comes across two `IProductRecords` that have the same number of sales the most popular product is chosen to be the first product it encounters.

2.1 VendingMachineProduct Class

This class implements the `IVendingMachineProduct` interface. This class has a basic implementation, it stores two private string attributes, `laneCode` and `description` used to describe and identify a certain product that could feature

in the `VendingMachine`. It has two methods overridden from the interface that each return the `laneCode` or description. No restrictions were placed on the `laneCode` and `description` because it was not outlined as a requirement in the specification and would increase complexity without any real benefit.

2.2 ProductRecord Class

This class implements the `IProductRecord` interface. This class has three private attributes. The `vendingMachineProduct` variable of type `IVendingMachineProduct` stores the product it is related to. Two integers are also used to keep track of the number of items available and the number sold. The `buyItem()` method increments the number of sales and decrements the number available by one. However if there are no products of that type available the method throws a `ProductUnavailableException`. The `addItem` method does the opposite increments the number of items available by one.

2.3 VendingMachine Class

This class implements the `IVendingMachine` interface and has two private attributes. This first is a `HashMap` called `products`. This is the collection used for storing the `IProductRecord`'s stored by the `VendingMachine`. The interface specified that a considerable number of methods would have as a parameter the string `laneCode`. Therefore, it made sense to use a `HashMap` with `laneCode`'s as keys as this would allow for easy access to a specific `laneCode` for these methods. This means that implementation was a lot easier than an array based implementation as `contains()` operations were faster, taking constant time - $O(1)$. The second attribute is an integer to store the number of items within the entire `VendingMachine`. This was added because it allows for constant time - $O(1)$ - `getTotalNumberOfItems()` calls, since now the method can just return this number. Without it, the method would have to iterate over every `IProductRecord` in the `products` `HashMap` to add up all the separate integers.

The method `registerProduct()` checks to see if a product is in `products` before adding it. If the `laneCode` for the product is already in `products` then the function throws a `LaneCodeAlreadyInUs eException`. `unregisterProduct()` checks if the `laneCode` is already in `products` and if it is removes the product from the machine. If not, it throws `LaneCodeNotRegisteredException`.

Both `addItem()` and `removeItem()` increment and decrement the number of a specific product by one, however if the `laneCode` specifying the product is not in `products` they throw a `LaneCodeNotRegisteredException`. Furthermore, `getNumberOfItems` and `getNumberOfsales` return the sales and number of items available of a particular product. Again, if `laneCode` is not present in `products` the same exception is thrown.

The `getMostPopular()` finds the `IProductRecord` in `records` with the most

number of sales. If multiple products have the same number of sales the method picks the most popular product to be the first product it comes across. As per the interface, the method throws a `LaneCodeNotRegisteredException` if there are no products in `products`.

3 Testing

There are in total 53 different tests associated with this project. However, if we remove parametrised tests this number falls to 41. Implementation has proven to be successful as all 53 tests passed, as shown in figure 1 and 2. The tests were split into separate test classes with each test class testing each implementation of the interfaces. They were split up like this as it makes them easier to manage and diagnose which implementation is failing. In `IVendingMachineProductTests` there are 6 tests, `IProductRecordTests` has 11, and `IVendingMachineTests` includes 24.

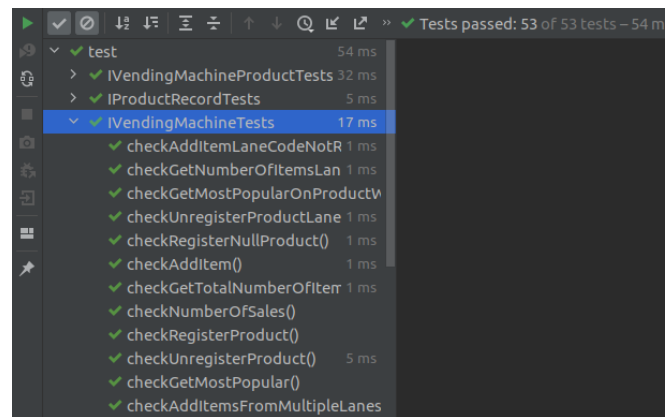
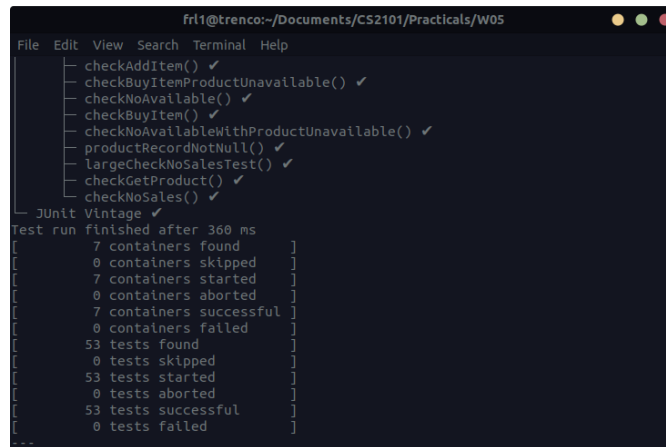


Figure 1: JUnit IntelliJ integration showing all tests passing

3.1 IVendingMachineProduct Tests

- `vendingMachineProductNotNull()`
 - Tests that the factory was able to call a sensible constructor to get a non-null instance of `IVendingMachineProduct`.
- `checkDescriptionAndLaneCode()`
 - Tests that the `laneCode` and `description` are present in the instance of `IVendingMachineProduct`.
 - Included as we need to make sure that the `laneCode` and `description` are present in the object as otherwise this would cause other errors



```
fr1@treno:~/Documents/CS2101/Practicals/W05
File Edit View Search Terminal Help
- checkAddItem() ✓
- checkBuyItemProductUnavailable() ✓
- checkNoAvailable() ✓
- checkBuyItem() ✓
- checkNoAvailableWithProductUnavailable() ✓
- productRecordNotNull() ✓
- largeCheckNoSalesTest() ✓
- checkGetProduct() ✓
- checkNoSales() ✓
- JUnit Vintage ✓
Test run finished after 360 ms
[ 7 containers found ]
[ 0 containers skipped ]
[ 7 containers started ]
[ 0 containers aborted ]
[ 7 containers successful ]
[ 0 containers failed ]
[ 53 tests found ]
[ 0 tests skipped ]
[ 53 tests started ]
[ 0 tests aborted ]
[ 53 tests successful ]
[ 0 tests failed ]
```

Figure 2: JUnit ran on stacsccheck showing all tests passing

when trying to include them in the products HashMap in IVendingMachine.

- `checkDescription()`
 - Tests that different descriptions (including single character and empty strings) are accepted by IVendingMachineProduct.
 - This is a parametrised test that takes in different product descriptions: "Haggis Crisps", "Chocolate", "", "a", " "
 - Included as we need to make sure that when the method is implemented it can handle any string, whether it be empty, one character, one word or multiple words.
- `descriptionNull()`
 - Tests that the IVendingMachineProduct description can be null.
 - Included as we need to ensure that we can handle null values in our description.
- `checkLaneCode()`
 - Tests that different laneCodes (including single character and empty strings) are accepted by IVendingMachineProduct.
 - This is a parametrised test that takes in different lineCodes: "A1", "A2", "A49320", "A0", "0A", "A0B", "394fj04", "", " "
 - Included as there is no formal restrictions in the specification about the format of laneCodes and so should be able to be a string of any form. Here we consider if it is empty, one character, or multiple characters.

- `laneCodeNull()`
 - Tests that the `IVendingMachineProduct` `laneCode` can be null.
 - Included as we need to ensure that we can handle null values as `laneCodes`.

3.2 IProductRecord Tests

This test class features a `@BeforeEach` method that runs creates a new `IVendingMachineProduct` and `IProductRecord` before every test. This allows for each test to use standard variables that will be reset after every test, therefore removing dependency between tests.

- `productRecordNotNull()`
 - Tests that the factory was able to call a sensible constructor to get a non-null instance of `IProductRecord`.
- `checkGetProduct()`
 - Tests that the `IProductRecord` can still retrieve the correct `IVendingMachineProduct` using the `getProduct()` method and that the `IVendingMachineProduct` holds the correct `laneCode` and description.
 - Included since receiving the `IVendingMachineProduct` needs to be correct since so that methods can work correctly later on within the `IVendingMachine` implementation.
- `checkBuyItem()`
 - Tests that the `buyItem()` method works in buying an item that is available.
 - Included since we need to make sure that an item can be bought for future tests and within the `IVendingMachine` implementation.
- `checkAddItem()`
 - Tests that the `addItem()` method works to add one new item to `IProductRecord`.
 - Included since we need an item to be added to the `IVendingMachine` so that it can be later bought.
- `checkBuyItemProductUnavailable()`
 - Tests that the `buyItem()` method throws a `ProductUnavailableException` when buying an item that not available. It checks this before any items have been added, and after an item has been bought and is no longer available.

- Included since buying an item when it is not available should throw this exception up to the `IVendingMachine` to let the user know that there are no items left and that more need to be added.
- `checkNoAvailable()`
 - Tests that the `getNumberAvailable()` method correctly returns the number of products available after multiple `buyItem()` and `addItem()` method calls.
 - Included since we need to make sure that the `IProductRecord` implementation counts the number of items added and bought correctly.
- `checkNoSales()`
 - Tests that the `getNumberOfSales()` method correctly returns the number of products sold after multiple items have been bought.
 - Included since we need to make sure that the `IProductRecord` implementation counts the number of items bought correctly.
- `checkNoSalesWithProductUnavailable()`
 - Tests that the number of sales does not change after the `buyItem()` method throws a `ProductUnavailableException` if a product has tried to be sold that is not available.
 - Included since we need to make sure that the number of sales does not change if we receive a `ProductUnavailableException` since if this error has been generated a product should not be able to be bought.
- `checkNoAvailableWithProductUnavailable()`
 - Tests that the amount of product available does not change after the `buyItem()` method throws a `ProductUnavailableException` if a product has tried to be sold that is not available.
 - Included since we need to make sure that the number of products available does not change if we receive a `ProductUnavailableException`. If we do not check this then the number available could be going into negative numbers since no available could be decremented when nothing has been sold.
- `largeCheckNoAvailableTest()`
 - Tests that `IProductRecord` can add lots of the same item and can return the correct number of items available.
 - Included since there should be no limit (or to a high a limit to notice there is one) on the number of products that can be available in the `IVendingMachine`. This was specified in the design.
- `largeCheckNoSalesTest()`

- Tests that `IProductRecord` can buy lots of the same item and can return the correct number of products sold.
- Included since there should be no limit (or to a high a limit to notice there is one) on the number of products that can be sold in the `IVendingMachine`. This was specified in the design.

3.3 IVendingMachine Tests

This test class also features a `@BeforeEach` method that runs creates a new `IVendingMachine` and two `IProductRecord`'s before every test. This allows for each test to use standard variables that will be reset after every test, therefore removing dependency between tests.

- `vendingMachineNotNull()`
 - Tests that the factory was able to call a sensible constructor to get a non-null instance of `IVendingMachine`.
- `checkRegisterProduct()`
 - Tests that a product has been successfully registered within the `IVendingMachine` in a specific lane.
 - Included since products should be able to be registered to the `IVendingMachine` if a `IProductRecord` object is passed in.
- `checkRegisterNullProduct()`
 - Tests that `IVendingMachine` throws a `NullPointerException` when trying to register a null product.
 - Included since if a null value is trying to be registered as a product this should throw an error.
- `checkRegisterProductLaneAlreadyInUse()`
 - Tests that the `IVendingMachine` throws a `LaneCodeAlreadyInUseException` when trying to register a product that has already been registered to that lane code in the machine.
 - Included since if a product lane is already in use and another product is trying to be registered in that product lane this should cause a `LaneCodeAlreadyInUseException` and not register two products in one lane.
- `checkUnregisterProduct()`
 - Tests that a product has been successfully unregistered from the `IVendingMachine` and is no longer available.
 - Included since we need to check if a product has been unregistered so that we can register products back to the lane later.

- `checkUnregisterProductLaneCodeNotRegistered()`
 - Tests that the `IVendingMachine` throws a `LaneCodeNotRegisteredException` when trying to unregister a product that does not exist.
 - Included since we should not be able to unregister a product if it does not exist in the `IVendingMachine`.
- `checkAddItem()`
 - Tests that a `IVendingMachine` adds an item on the specified lane.
 - Included since we need to know if items are being added onto the specified lane correctly.
- `checkAddItemsFromMultipleLanes()`
 - Tests that `IVendingMachine` adds two different products on separate lanes correctly.
 - Included since we need to know if items are being added onto the specified lanes correctly when dealing with adding to multiple different lanes at the same time.
- `checkAddItemLaneCodeNotRegistered()`
 - Tests that the `IVendingMachine` throws a `LaneCodeNotRegisteredException` when trying to add an item to a lane code that does not exist.
 - Included since it should not be possible to add an item to a lane code and `IPProductRecord` that does not exist.
- `checkBuyItem()`
 - Tests that a `IVendingMachine` buys an item from the specified lane.
 - Included since we need to know if items are being bought onto the specified lane correctly.
- `checkBuyItemLaneCodeNotRegistered()`
 - Tests that the `IVendingMachine` throws a `LaneCodeNotRegisteredException` when trying to buy an item from a lane code that does not exist.
 - Included since it should not be possible to buy an item from a lane code and `IPProductRecord` that does not exist.
- `checkBuyItemsFromMultipleLanes()`
 - Tests that `IVendingMachine` buys two different products from separate lanes correctly.

- Included since we need to know if items are being bought from the specified lanes correctly when dealing with adding to multiple different lanes at the same time.
- `checkBuyItemsProductUnavailable()`
 - Tests that `IVendingMachine` throws a `ProductUnavailableException` when trying to buy an item from a lane in which there are no products left to buy.
 - Included since if there are no products left to buy then the `ProductUnavailableException` should be thrown and no items of the product should be bought since the amount of products available cannot be a negative number.
- `checkGetNumberOfProducts()`
 - Tests that the correct value for the number of products registered is returned by the `IVendingMachine` class.
- `checkGetTotalNumberOfItems()`
 - Tests that `IVendingMachine` calculates the correct value for the total number of items over all products in the machine. The test checks if this value is correct when we add and buy multiple items from multiple lanes in the `IVendingMachine`.
 - Included since we need to check that the system for incrementing the `noOfItems` variable works correctly. This is needed since we included this variable rather than iterate over all of the products.
- `checkGetNumberOfItems()`
 - Tests that `IVendingMachine` returns the correct value for the total number of items of one product. The test checks if this value is correct when we add and buy multiple items from multiple lanes (which should not update the value if not in the chosen lane) in the `IVendingMachine`.
- `checkGetNumberOfItemsLaneCodeNotRegistered()`
 - Tests that `IVendingMachine` throws a `LaneCodeNotRegisteredException` when trying to check for the number of items of a product in a lane code that does not exist.
 - Included since a `LaneCodeNotRegisteredException` should occur because the number of items in an unregistered lane is undefined.
- `checkNumberOfSales()`

- Tests that `IVendingMachine` returns the correct value for the total number of sales of one product. The test checks if this value is correct when buying multiple items from multiple lanes (which should not update the value if not in the chosen lane) in the `IVendingMachine`.
- `checkGetNumberOfSalesLaneCodeNotRegistered()`
 - Tests that `IVendingMachine` throws a `LaneCodeNotRegisteredException` when trying to check for the number of sales of a product in a lane code that does not exist.
 - Included since a `LaneCodeNotRegisteredException` should occur because the number of sales in an unregistered lane is undefined.
- `checkGetMostPopular()`
 - Tests that the most popular product is calculated to be the product bought the most number of times.
 - Included since we need an initial test to see that the `getMostPopular` method works in a normal test case where there is a clear most popular product out of multiple products.
- `checkGetMostPopularLaneCodeNotRegistered()`
 - Tests that `IVendingMachine` throws a `LaneCodeNotRegisteredException` when trying to check for the most popular product when no products have registered yet.
 - Included since if no products have been registered yet then a `LaneCodeNotRegisteredException` should be thrown because there is no `IProductRecord` to return.
- `checkGetMostPopularOnProductWithNoItems()`
 - Tests that `IVendingMachine` can return the most popular product when products with no items are present in the machine.
 - Included since we need to test extreme cases where the most popular product is default the only one in the `IVendingMachine`, even though it has no sales.
- `checkGetMostPopularOnProductWithOneProduct()`
 - Tests that `IVendingMachine` can return the most popular product when only one product is present but has items in the machine.
 - Included since this extreme test should return the most popular product to be the only product that has sold.
- `checkGetMostPopularWithTiedProducts()`

- Tests that `IVendingMachine` can return the most popular product when multiple products have the same number of sales. This should return the first product it comes across as the behaviour is undefined.
- Included since this extreme test should return the first product it comes across that is one of the most popular products and not anything else.

4 Conclusion

In this practical I followed TDD methodology whilst using the JUnit framework to implement the three ADT interfaces, `IVendingMachine`, `IProductRecord` and `IVendingMachineProduct`. I wrote test methods to first establish the behaviour of methods I was implementing, and ran these implementations through the test methods I defined. These tests cover the full range of test cases, from normal to extreme and exceptional test cases. I learnt a lot about the TDD process, JUnit and factories. Using IntelliJ with JUnit was excellent as it had inbuilt testing features, making it easy find out where and why some tests failed without leaving the IDE.

Overall, implementation in this project was not too challenging. The difficult aspect of the practical was in designing tests that covered all normal, extreme and exceptional test cases. I enjoyed using JUnit and found that different assertions and `@BeforeEach` made it extremely easy to test my various implementations. While I can now appreciate the benefits of TDD, I would still be unlikely to use it over the traditional software development process. I feel like it is suited to a smaller, subset of programming projects, where there can only be a small number of outcomes. That said, if I was building a small implementation of an ADT again, that I needed to ensure was as robust as possible, TDD would be a good approach to utilise.

References

- [1] Stefan Bechtold, Sam Brannen, Johannes Link, Matthias Merdes, Marc Philipp, Juliette de Rancourt, Christian Stein, *JUnit 5 User Guide V5.7.0*, 14-08-2020, accessed 13-10-2020, <https://junit.org/junit5/docs/current/user-guide/>
- [2] JetBrains, *Prepare for testing*, 19-08-2020, accessed 13-10-2020, <https://www.jetbrains.com/help/idea/testing.html>