UCN, University College of Northern Denmark

IT-programme

AP Degree in Computer Science

Class: dmaj0920

Title: Workshop – Quality & Testing

Participants:
Mihail Gerginov

Teo Stanic

Antonio Kovacevic

Marko Veljkovic

Lucas Inaschwili-Hongre

Supervisors:
Mogens Holm Iversen

Søren Krarup Olesen

Repository link: https://github.com/TheRandomTroll/ucn-2-ws-quality-test

Table of Contents

Introduction 2
Use cases and System Sequence Diagrams 2
Alternative flows and special conditions (Buy Ticket use case) 2
Use-Case Scenario (Buy Ticket) 3
Test cases 3
Test cases for buying a ticket 4
Test cases for database connectivity 4
Test cases for display reset 5
Implementation of tests cases (Writing unit tests) 5
JUnit 5
Test case priority 5
#1: Displaying the proper amount of time for the respective amount of money 5
#2: Ensuring invalid coins do not get accepted by the machine 7
#3: Ensuring illegal coins do not get accepted by the machine 7
#4: Processing the payments and saving them to the database 8
#5: Making sure that the prices per zone get fetched properly, both on database and controller level 10
#6: Ensuring the display gets reset after a payment has been completed/cancelled 10
Group Process 12
Conclusion 12
References 12

Introduction

This workshop is a part of the 2nd semester curriculum for the Computer Science degree at UCN. The premise of the project is to design use-case scenarios, test cases, unit tests and integration tests for a pre-made project. Other tasks include refactoring the code to resolve issues or to improve code quality and performance.

We were contacted by the company Easy Parking A/S, which wants to replace their outdated parking meters. As members of the testing department, we were put in charge of testing the features which were created by the development department.

The company follows Unified Process as its development methodology and UML for the necessary diagrams. The code is written in Java, using Eclipse as an IDE. The framework used for testing is JUnit.

Use cases and System Sequence Diagrams

All the use cases and system sequence diagrams were given in advance to us. However, some of the use cases missed additional information, such as alternate flows and special conditions for them. We had to fill in the blanks to make the use cases more descriptive.

Alternative flows and special conditions (Buy Ticket use case)

One of the first things we immediately noticed is that there was a mismatch between the use case and what the notes from the interview conducted with the company were saying. The notes mentioned some specifications about the ticket machine which were not addressed for in the use case. We have extended the “Buy Ticket” use case to make the alternative flows and special conditions clearer. Figure 1 displays all the additional material that has been added to the use case. The entire use case can be found in the file ***[UC] Buy Ticket.docx***.

Alternative flows			
		6a. The customer enters coins of different, but accepted currencies.	
			1. Registers payment and updates the display which shows how much time he has bought so far.
		6b. The customer enters an illegal coin (not EUR or DKK).	
			1. Does not register payment and returns the coin to the customer, not updating the display.
		9a. The customer presses the button CANCEL.	
			1. Does not register the purchase in the central database. 2. Returns the coins to the customer.

			3. The display is cleared to prepare for another transaction.
Special Requirements		The machine only accepts coins. The allowed currencies are EUR and DKK.	
		The allowed coins are as follows: Euro – 1, 2, 5, 10, 20 & 50 cents; 1 & 2€ DKK – 50 øre; 1, 2, 5, 10 & 20 DKK	
		The response time to update the display should be as soon as the coin is validated.	
		The display should display the BUY and CANCEL buttons after the first coin is inserted.	

Figure 1. Added information to the "Buy Ticket" use case

Use-Case Scenario (Buy Ticket)

A Use-Case Scenario is an instance of a use case. It displays different scenarios with the flows they take through the flows of the case. The following table and explanatory text show all use-case scenarios for the buy ticket use case.

Scenario Name	**Starting Flow**	**Alternate**
Scenario 1 – Successful transaction	Basic Flow	
Scenario 2 – Different currency coins	Basic Flow	
Scenario 3 – Invalid coin	Basic Flow	A1
Scenario 4 – Interrupted transaction	Basic Flow	A2
Alternate flows		
A1	The coin is rejected, and the display does not get updated.	
A2	The coins are returned, and the display gets updated for a new transaction.	

Table 1. Use-Case Scenarios for buying a ticket.

Test cases

A test case is a set of test inputs, execution conditions, and expected results, developed for a particular objective. Its main purpose is to test a specific part of a program, to verify it is running properly and to identify conditions that will be implemented in the form of unit tests. In this project, a majority of the test cases were done using the **black box testing** principle. **Black box testing** is a software testing method in which the functionalities of software applications are tested without having knowledge of internal code structure, implementation details and internal paths. Black Box Testing mainly focuses on input and output of software applications and it is entirely based on software requirements and specifications. [1] For the failing tests that required additional comments, we had to browse through the code in order to specify the reason for the

failure and, therefore, document it in the *Comments* column in the spreadsheet. **Unimplemented functionality is left as-is.**[1]

Test cases for buying a ticket

Test Case no.	Currency = EUR		Currency = DKK		Currency = NOK		Expected result		Actual result		Pass/Fail	Comments
	Cent	€	Øre	Kr.	Øre	Kr.	Per second	Per minute	Per second	Per minute		
	Value	Value	Value	Value	Value	Value						
1	0	0	0	0	0	0					V	
No coins							0	0		0		
2												
Scenario 1a, EUR	5	0	0	0	0	0					V	
One coin, Parking sec. round up							120	2	120	2		
3	0	1	0	0	0	0						
Scenario 1a, EUR	0	2	0	0	0	0					V	
More coins							7200	120	7200			

Table 2. Buy ticket test cases (found in ***[TC] Buy Ticket.xlsx***)

A small amount of the test cases for buying a ticket are shown on Table 2. The entirety of this table is located in the file ***[TC] Buy Ticket.xlsx***.

Test cases for database connectivity

Test Case no.	Variable (and value)	Expected result	Actual result	Pass/Fail	Comments
1					
Can connect to database	*DBConnection* con	DBConnection instance when created, *null* when destroyed	DBConnection instance when created, *null* when destroyed	V	
2					
Insert buy	*PBuy* pBuyInstance, *PPayStation* pPayStationnstance, *DatabasePBuy* dbPBuyInstance, *key* = dbPBuyInstance.insertParkingBuy(pBuyInstance)	*key is a* non-negative value	key is a non-negative value	V	*Non-negative values are returned when the insertion is successful.*
1					
Retrieve price for zone from database layer	*DatabasePPrice* dbInstance, *zoneId* = 2, *PPrice* price = dbInstance.getPriceByZoneId(zoneId), *key* = price.getParkingPrice()	key = 25	Exception is thrown (NullPointerException)	X	*This functionality is not implemented yet.*
2					
Retrieve price from control layer	*ControlPrice* cPriceInstance, *zoneId* = 1, *PPrice* price = dbInstance.getPriceByZoneId(zoneId), *key* = price.getParkingPrice()	key = 35	Exception is thrown (NullPointerException)	X	*This functionality is not implemented yet.*

Table 3. Test cases for database connectivity (found in ***[TC] Database Connectivity.xlsx***)

[1] This decision was made after a consultation with the supervisors.

Test cases for display reset

Test Case no.	Variable (and value)	Expected result	Actual result	Pass/Fail	Comments
1 Display is cleared after purchase	*expectedParkingTime, coinValue*	expectedParkingTime = 0	expectedParkingTime = 0	V	
2 Display is cleared after cancellation	*expectedParkingTime, coinValue*	*expectedParkingTime = 0*	expectedParkingTime = 0	V	

Table 4. Test cases for display reset (found in ***[TC] Reset.xlsx***)

Implementation of tests cases (Writing unit tests)

JUnit

JUnit is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development and is one of a family of unit testing frameworks which is collectively known as xUnit that originated with SUnit. [2] A JUnit test fixture is a Java object. Test methods must be annotated by the **@Test** annotation. If the situation requires it, it is also possible to define a method to execute before (or after) each (or all) of the test methods with the **@BeforeEach** (or **@AfterEach**) and **@BeforeAll** (or **@AfterAll**) annotations. [3]

Test case priority

Before the process of writing unit tests had begun, we had to prioritize the most important features first. This was done by discussion with the group and realization once a conclusion was reached. All test files are in the ***/src/test*** folder.

#1: Displaying the proper amount of time for the respective amount of money

Our priority is to ensure that coins are processed and reflected properly on the display. The realization of the *Buy Ticket* test cases is realized in 4 separate files:

- ***TestCalculationCurrencyDkk.java;***
- ***TestCalculationCurrencyEuro.java;***
- ***TestCalculationCurrencyMixed.java;***
- ***TestIllegalCoin.java.***

These fixtures test all possible cases for currencies and combinations of them. For the sake of avoiding repetition, the first code snippet only shows one of the unit tests, including **@Before**

and **@After** methods. The other 3 fixtures include the exact same **@Before** and **@After** annotated classes.

```
import org.junit.*;

public class TestCalculationCurrencyDkk {
	ControlPayStation ps;

	@Before
	public void setUp() {
		ps = new ControlPayStation();
	}

	@Test
	public void shouldDisplay27MinFor5Dkk() throws
IllegalCoinException {
		// Arrange
		int expectedParkingTime = 27;
		int coinValue = 5;
		Currency.ValidCurrency coinCurrency =
Currency.ValidCurrency.DKK;
		Currency.ValidCoinType coinType =
Currency.ValidCoinType.INTEGER;

		// Act
		ps.addPayment(coinValue, coinCurrency, coinType);

		// Assert
		assertEquals("Should display 27 min for 5 DKK",
expectedParkingTime, ps.readDisplay());
	}

	@After
	public void cleanUp() {
		ps.setReady();
	}
}
```

Figure 2. Unit test for verification of properly displayed time

#2: Ensuring invalid coins do not get accepted by the machine

The ticket machine software should make sure to return invalid coins or coins of invalid currency (e.g. Norwegian krone). The unit test displayed below ensures that invalid Euro coins (in this case - a 3 Euro coin) cannot be accepted. The way it signals an invalid coin has been entered is by throwing an **IllegalCoinException**. This can be caught if "**(expected = *ExceptionName*.class)**" is added immediately after the **@Test** annotation.

```
    @Test(expected = IllegalCoinException.class)
    public void shouldRejectIllegalEuroCoin() throws
IllegalCoinException {
        // Arrange
        int coinValue = 3;
        Currency.ValidCurrency coinCurrency =
Currency.ValidCurrency.EURO;
        Currency.ValidCoinType coinType =
Currency.ValidCoinType.INTEGER;

        // Act
        ps.addPayment(coinValue, coinCurrency, coinType);
        // No need for an assert, it is already done in the
annotation.
    }
```

Figure 3. Unit test for verifying illegal coins are rejected

#3: Ensuring illegal coins do not get accepted by the machine

The ticket machine software should make sure to return invalid coins or coins of invalid currency (e.g. Norwegian krone). The unit test displayed below ensures Norwegian krones cannot be accepted. The assertion is analogous to the previous unit test.

```
    // Norwegian coin
    @Test(expected = IllegalCoinException.class)
    public void shouldRejectIllegalCurrencyNokCoin() throws
IllegalCoinException {
        // Arrange
        int coinValue = 1;
        Currency.ValidCurrency coinCurrency =
Currency.ValidCurrency.NOK;
        Currency.ValidCoinType coinType =
Currency.ValidCoinType.INTEGER;
```

```
            // Act
            ps.addPayment(coinValue, coinCurrency, coinType);
      }
```

Figure 4. Unit test for verifying illegal currencies are rejected

#4: Processing the payments and saving them to the database

Our second priority was to ensure that the records were successfully saved to the database. In general, the file ***TestDatabaseAccess.java*** handles all connectivity-related tests, but we are going to look at only one of the tests in it - the one for properly inserting the file in the database. Once the test file has been inserted and the test has passed, the entry is removed from the database in the **@AfterClass** fixture. The difference between **@After** and **@AfterClass** is that the former is executed after *every* test, whereas the latter is executed *only once - after all tests have been complete*. [4]

```
import org.junit.*;

public class TestDatabaseAccess {
      DBConnection con = null;
      static PBuy tempPBuy;

      @Before
      public void setUp() {
            con = DBConnection.getInstance();
      }

      @Test
      public void wasInsertedBuy() throws DatabaseLayerException {
            // Arrange
            LocalDate timeNow = java.time.LocalDate.now();
            double payedCentAmount = 100;

            tempPBuy = new PBuy();

            PPayStation pStat = new PPayStation(1, "P-423E");
            pStat.setAmount(payedCentAmount);
            tempPBuy.setAssociatedPaystation(pStat);
            tempPBuy.setBuyTime(timeNow);
```

```
            DatabasePBuy dbPbuy = new DatabasePBuy();

            // Act
            int key = dbPbuy.insertParkingBuy(tempPBuy);

            // Assert
            assertTrue("Buy was inserted", key > 0);

      }

      @After
      public void cleanUp() {
            DBConnection.closeConnection();
      }

      @AfterClass
      public static void cleanUpWhenFinish() {
            // Arrange
            DatabasePBuy dbPbuy = new DatabasePBuy();
            int numDeleted = 0;

            // Act
            try {
                  numDeleted = dbPbuy.deleteParkingBuy(tempPBuy);
            } catch(Exception ex) {
                  System.out.println("Error: " + ex.getMessage());
            } finally {
                  DBConnection.closeConnection();
            }

            // Assert
            assertEquals("One row deleted", 1, numDeleted );
      }
}
```

Figure 5. Unit test for verifying payments are inserted correctly

#5: Making sure that the prices per zone get fetched properly, both on database and controller level

One of the company's requirements is that each zone's ticket price gets updated once per night, at 3AM. The tests for fetching the prices are in the same file as the former test. However, now the tests fail because the connection to the database has not been developed yet.

```
	@Test
	public void wasRetrievedPriceDatabaselayer() throws
DatabaseLayerException {
		// Arrange
		PPrice foundPrice = null;
		int pZoneId = 2;
		DatabasePPrice dbPrice = new DatabasePPrice();

		// Act
		foundPrice = dbPrice.getPriceByZoneId(pZoneId);

		// Assert
		assertEquals("Price should be 25 for Zone 2", 25,
foundPrice.getParkingPrice());
	}
```

Figure 6. Unit test for verifying that prices for zones are fetched properly

#6: Ensuring the display gets reset after a payment has been completed/cancelled

After the user has decided if he wants to finalize his transaction, we have to make sure that the display gets reset properly. The unit tests for this case are realized in the file ***TestReset.java***. It follows the same procedure as the previous files of having **@Before** and an **@After** methods for preparing and cleaning-up the tests, respectively. The **shouldClearAfterBuy()** test is analogous to the one showed below, so it is omitted.

```
import org.junit.*;
public class TestReset {
	ControlPayStation ps;

	/** Fixture for pay station testing. */
	@Before
	public void setUp() {
```

```
            ps = new ControlPayStation();
      }

      /**
       * Verify that cancel() clears the pay station
       */
      @Test
      public void shouldClearAfterCancel() throws
IllegalCoinException {
            // Arrange
            int expectedParkingTime = 2; // In minutes
            int coinValue = 5;
            Currency.ValidCurrency coinCurrency =
Currency.ValidCurrency.EURO;
            Currency.ValidCoinType coinType =
Currency.ValidCoinType.FRACTION;

            // Act
            ps.addPayment(coinValue, coinCurrency, coinType);

            // Assert
            assertEquals("Should display 2 mins for 5 cents",
expectedParkingTime, ps.readDisplay());

            // Arrange
            expectedParkingTime = 0;

            // Act
            ps.cancel();

            // Assert
            assertEquals("Should display 0 min when the CANCEL button
is pressed", expectedParkingTime, ps.readDisplay());
      }

      @After
      public void cleanUp() {
            ps.setReady();
      }
```

```
}
```

Figure 7. Unit test for verifying the display gets reset properly on purchase/cancel

Group Process

We got the information about the mini project, met up on Discord and went through all of the materials together. We did all of the work together, for example the one doing it would share the screen to others while explaining what is seen and so on. The program we used to showcase the system development things was Microsoft Excel. We wrote the report using Google Docs so that everybody can be writing at the same time and at the end we error checked it one more time before handing the work in.

Conclusion

With what was presented to us as a project, we were able to create unit tests for almost every edge case that the system could have run into. In terms of future expansion, we can use other Java libraries like Mockito to mock-up the database, which will allow us to test the product more safely when it gets pushed to production, because directly accessing the database in production is considered a serious mistake and a bad practice.

References

[1]: What is BLACK Box Testing? (guru99.com)

[2]: JUnit (Wikipedia)

[3]: Writing Tests (JUnit documentation)

[4]: AfterClass (JUnit documentation)