# Part 1 – Unit testing

## Task 1 - Setup

As required, a Git repository was created on github, and the user **mark-uom** was added as a contributor in order to have access. The repository can be found at :

<https://github.com/LaraBrockdorff/CPS3230_SoftwareTesting>

Commits were made to this repository when changes were made.

In order to easily identify the differences between tasks 2 and 3 of this past, a release was set after the end of task 2.

In order to get familiar with the system it was first run from the runner present, and tested manually. Through this process, it was noted how this was not possible to test all the possible edge cases in a structured way. Understanding how cumbersome and easily fallible it would be to go through all this manual process every time, shed light on how value writing structured tests is.

## Task 2 – Initial unit testing

### Testing the system undertest

While writing unit tests for the system under test, the principles behind unit testing were kept in mind, and tests would only test one specific thing, and will always return the same result, independent of other tests. This approach of independent tests also makes debugging future issues that arise when changes are made, and only the concerned points will fail (if bugs are present) and not other dependencies. These tests were kept simple and well commented to even act as documentation.

### Commenting on the coverage

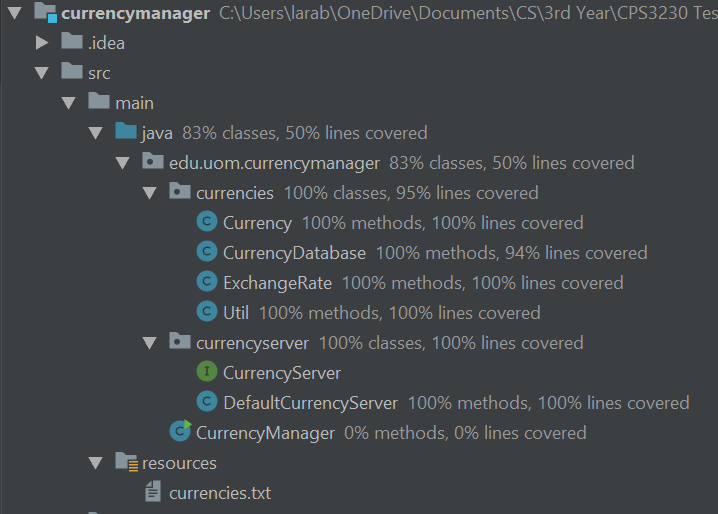


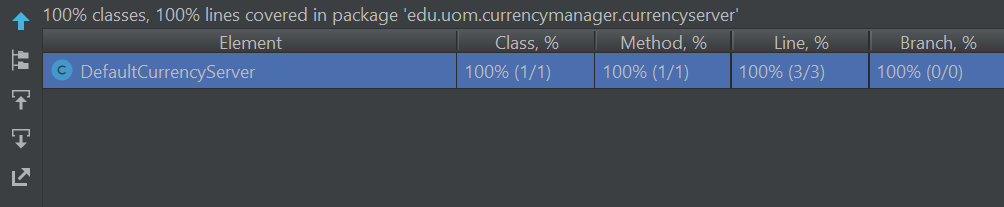
Figure Initial coverage results

**Class coverage**

The first thing which was noted is that a class coverage of 83 % was initially achieved. This was due to the fact that the CurrencyManager runner class was not tested. This is because the runner in the main menu is dependant on user input with various possible paths, and the other helper methods are dependant on interactions with the *CurrencyDatabase* class. Extracting more of the logic from the methods in this runner class would increase overall coverage.

**CurrencyServer package**

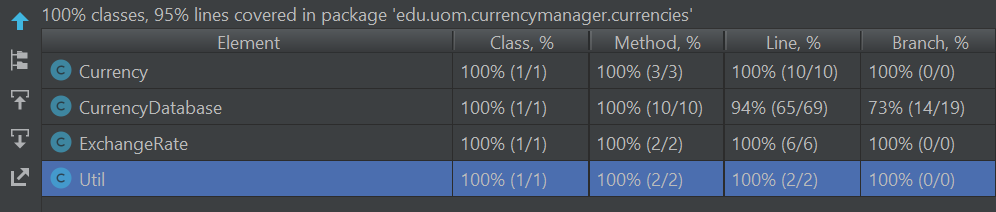
This package consists of the *CurrencyServer* interface as well as an implementation called *DefaultCurrencyServer* calls that makes use of a random number generator inside the class within the *getExchangeRate* method, making the returned value untestable. Despite this, a test that merely invokes the method and asserts that the value returned is not null was implemented to assert that any errors that could cause the program to crash while that method is being executed will be caught.



The fact that this class has full coverage, but yet its internal workings are not properly testable, sheds light on how coverage results are in no way a guarantee that the system is bug-free.

A test dummy of this method was created where the random value was replaced with a fixed value, in order to be able to predict the returned value with confidence in order to test the methods that make use of the class.

**Currencies package**



*Util class*

This util class containing only the *formatAmount* method was tested by testing the outputted formatting of different numbers (23.567, 3.4, 0000) which required rounding, adding of a last decimal place, and formatting of excess 0 values respectively. These numbers were chosen from different regions of the number set (different decimal place formats) valid in this context since it is not feasibly possible to test all possible doubles.

This class has full method, line and branch coverage.

*ExchangeRate class*

In this class, only the *toString* method was tested, even though not much logic was present in it. This was done by passing the test data and asserting that it matches the output of the *toString* method.

This class has full method, line and branch coverage

*Currency class*

This class is mainly used to create the object type of currency which contains a *fromString* method and a *toString* method. These were both tested by populating the objects with test data and comparing the output. In the case of the *fromString* method a valid currency was first tested to make sure it was populated correctly, then an invalid currency containing null data was passed to assure that a null pointer exception is thrown as expected.

This class has full method, line and branch coverage.

*CurrencyDatabase class*

Similarly to the previous cases, the contained methods were tested by asserting that returned results matched the expected results. This class posed challenges while being tested due to the way the methods interact with CurrencyServer and with the currencies.txt (database).

Due to the interaction with the currencies.txt file, the init method was not directly tested, reducing both the line and branch coverage in the class since not all the possibly thrown exceptions were tested for, as testing for this would involve manually altering the text file, which would go against the structure of unit testing.

Within the getExchangeRate method, one branch relating to the timeLastChecked was not tested, reducing the branch coverage. This is because of the way that the time is used in the ExchangeRate class.

Despite principles being followed related to the atomicity of unit tests, the tests do not function if the currency database/txt file is not present in the correct directory. This is due to the design of the system.

This class has 100% method coverage, 94% line coverage and 73% branch coverage.

### Noted testability Issues

As mentioned above, testability issues include:

* If the currencies.txt file is not present, all the tests in the CurrencieDatabaseTests class fail, making the test dependant on this file.
* The functionality within the DefaultServer could not be tested due to the dependency on the Random number generated within the method.
* The timeLastChecked value within the ExcahngeRate class cannot be manipulated easily when tested due to the way it is set from the System time within the constructor method.

## Task 3 – Advanced unit Testing

Aim of this section was to implement changes to the system under test to make the system simpler to test.

In order to make the system more efficiently to test, the following three main changes were effected;

1. **Refactor of Random number in *DefaultCurrencyServer***

While originally testing this class, it was noted that its testability is limited due to the use of a random number generator within the *getExchangeRate* method. It was an instance of a hardcoded dependency.

In order to address this issue and **the Dependency Injection-Parameter Injection** design pattern was implemented. This allowed for depended-on-components to be set at runtime from outside the method itself and to be passed as a parameter.

1. **Refactor of ExchangeRateTime**

Similarly to the previous case, the use of System.currentTimeMillis in the timeLastChecked field was refactored using and **the Dependency Injection of Constructor injection** design pattern, allowing for the timeLastChecked value to be passed in the constructor.

This allows for the value to be set from the constructor, independent od the System method.

1. **Refactor of reading from file**

Another issue previously mentioned was that there existed a dependency between the CurrencyDatabase class and the currencies.txt file being read from. This meant that if this file was not present, none of the related CurrencyDatabase would run successfully. This was amended by abstracting the reading of the file to another class, which would later be mocked, using the **DepandancyLookup** design pattern.

These changes lead to a large increase in the coverage rates of all classes in the currencies and currencySever packages, which now have coverage of 100 % (both branch and line)

Despite this, due to the abstraction of the fileReader (that was previously being called as a side effect of the init method in the CurrencyDatabase class), the overall line coverage decreased. Since the file reader class deals mainly with external communication (to an external file), more focus should be given to the integration testing of this component.

Overall, the changes made will also make future testing of the system easier as these changes allow for more modular code and less hardcoded dependencies.

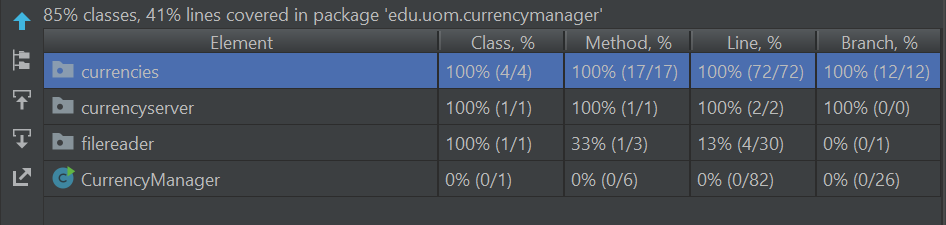


Figure Updated Coverage results