

SOFT2412: Assignment 1- Team 47

ASSIGNMENT/PROJECT COVERSHEET - GROUP ASSESSMENT

Unit of Study: Agile Software Development Practices (SOFT2412/COMP9412)

Assignment name: Group project Assignment 1 – Tools for Agile Software Development

Tutorial time: Tuesday 10-12

Tutor name: Ken Liu

DECLARATION

We the undersigned declare that we have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure, and except where specifically acknowledged, the work contained in this assignment/project is our own work, and has not been copied from other sources or been previously submitted for award or assessment.

We understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to severe penalties as outlined under Chapter 8 of the University of Sydney By-Law 1999 (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgement from other sources, including published works, the internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.

We realise that we may be asked to identify those portions of the work contributed by each of us and required to demonstrate our individual knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.

Student name	Student ID	Participated	Agree to Share	Signature
Darby Durack	480386401	Yes	Yes	DD
Benjamin Phua	450132759	Yes	Yes	BP
Elizabeth Andrews	450255955	Yes	Yes	EA
Sarah Liu	470527100	Yes	Yes	SL

1.Github Integration

A single shared repository on Github was created, where each group member was included as a collaborator. First we initialised an empty repository on Github. Then we initialized our local git repository in our gradle project using

git init

After the git repository was created, we added the empty repository on Github as the remote repository using the command

git remote add origin https://github.sydney.edu.au/SOFT2412-2019S2/Agile_47.git

Afterwards, everyone cloned the Github repository into their local computers using

git clone https://github.sydney.edu.au/SOFT2412-2019S2/Agile_47.git

During the duration of developing the currency convertor we typically created our own branches after cloning the master branch, or

git pull origin master

if our local branch was behind the remote master branch to update on the commits we were behind on.

We created our own branch and then switched to the branch using

git checkout -b <branch_name>

After branching and adding our individual functionalities and test cases, we would stage the changes using

git add .

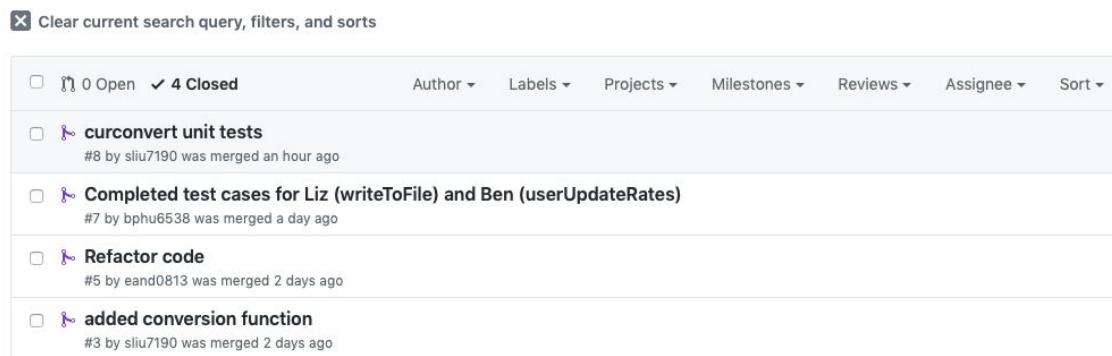
then commit changes

git commit -m <commit message>

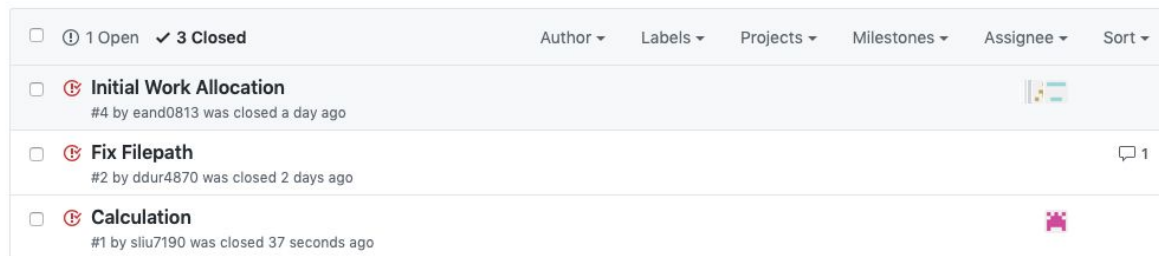
Lastly, we would push our branch to the remote Github repository using

git push origin <branch_name>

This was done to create pull requests on Github so that all merges in master can be seen and checked by other group members. If one of our local branches was not building properly, we can easily switch back to the working master branch and rebuild our tests or features again. After reviewing that there are no merge conflicts after submitting a pull request, we will merge the changes into master on Github. It should be noted that there were no merge conflicts in this assignment.



Each one of us were assigned tasks during each work iteration posted on Github issues. This was done so it was clear what tasks were expected from each group member. Additional features, requests for fixes between meetings were posted on the issue board, as seen below.



💡 ProTip! Ears burning? Get @sliu7190 mentions with [mentions:sliu7190](#).

Github was also used to help each other when one of us experienced bugs or test failures on our branch.

git pull origin <branch_name>

would pull the branch with the bug.

The branch with the bug will be pushed onto Github while the rest can pull and help resolve the issue by looking at the code and push a fix onto Github.

```
vlan ---- - - - - - Agile_47 sarah$ git pull origin darby
remote: Enumerating objects: 22, done.
remote: Counting objects: 100% (22/22), done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 22 (delta 6), reused 22 (delta 6), pack-reused 0
Unpacking objects: 100% (22/22), done.
From https://github.sydney.edu.au/SOFT2412-2019S2/Agile_47
* branch      darby      -> FETCH_HEAD
* [new branch] darby      -> origin/darby
```

2. Gradle Integration

Gradle was used in the assignment to automate the build of the project. JUnit was integrated with Gradle so that the tests would automatically run every time the project was built. This was to ensure every new build passed the test cases and no functionality had been broken by changes. Gradle was also used to integrate with Jenkins so that each build can be automatically tested.

1. List of relevant Gradle commands that include building a jar file
The command used to build/test the project was: **gradle clean build**

```

Lizs-MacBook-Pro:Agile_47 lizandrews$ gradle clean build

> Task :test

Agile_47.AppTest > testFileWriteThreeRates() PASSED
Agile_47.AppTest > testFileWriteInvalidCurrency() PASSED
Agile_47.AppTest > testPromptUserInvalidSelection() PASSED
Agile_47.AppTest > testFileWriteOneRateLastLine() PASSED
Agile_47.AppTest > testUserUpdateRatesInvalidCurrencyInput() PASSED
Agile_47.AppTest > testGradlePrintToStandardOut() STANDARD_OUT
    Gradle is capturing System.out
Agile_47.AppTest > testGradlePrintToStandardOut() PASSED
Agile_47.AppTest > testUserUpdateRatesInvalidRateInput() PASSED
Agile_47.AppTest > testFileWriteOneRateFirstLine() PASSED
Agile_47.AppTest > testPromptUserToChangeRates() PASSED
Agile_47.AppTest > testUserUpdateRatesValidInput() PASSED
Agile_47.AppTest > testUserUpdateRatesInvalidNumberInput() PASSED
Agile_47.AppTest > testWriteSameCurrencyTwice() PASSED
Agile_47.AppTest > testFileWriteNoFilePresent() PASSED
Agile_47.CalculationTest > convertCurrency() PASSED
Agile_47.CalculationTest > calculateConversionRate() PASSED

BUILD SUCCESSFUL in 10s
9 actionable tasks: 9 executed

```

The above screenshot shows that the build was successful and that all tests were passed. Below, we can see that **gradle build** also creates a JAR file in `/build/libs` that contains all the project dependencies.

```

/mnt/c/Data/Documents/Uni/SOFT2412/Project/Agile_47/build/libs master*
» jar tf Agile_47.jar
META-INF/
META-INF/MANIFEST.MF
Agile_47/
Agile_47/App.class

```

The command used to clean files was: **gradle clean**

The task “clean” was used to ensure that any leftovers from the last build are removed. It does this by removing the “build” folder. This ensures that no remnants of previous builds are affecting the current build.

```
[Lizs-MacBook-Pro:Agile_47 lizandrews$ gradle clean
```

```
BUILD SUCCESSFUL in 2s
1 actionable task: 1 executed
```

```
[Lizs-MacBook-Pro:Agile_47 lizandrews$ gradle test

> Task :test

Agile_47.AppTest > testFileWriteThreeRates() FAILED
    org.opentest4j.AssertionFailedError at AppTest.java:96

Agile_47.AppTest > testFileWriteInvalidCurrency() PASSED

Agile_47.AppTest > testPromptUserInvalidSelection() PASSED

Agile_47.AppTest > testFileWriteOneRateLastLine() FAILED
    org.opentest4j.AssertionFailedError at AppTest.java:48

Agile_47.AppTest > testUserUpdateRatesInvalidCurrencyInput() PASSED

Agile_47.AppTest > testGradlePrintToStandardOut() STANDARD_OUT
    Gradle is capturing System.out

Agile_47.AppTest > testGradlePrintToStandardOut() PASSED

Agile_47.AppTest > testUserUpdateRatesInvalidRateInput() PASSED

Agile_47.AppTest > testFileWriteOneRateFirstLine() FAILED
    org.opentest4j.AssertionFailedError at AppTest.java:27

Agile_47.AppTest > testPromptUserToChangeRates() PASSED

Agile_47.AppTest > testUserUpdateRatesValidInput() PASSED

Agile_47.AppTest > testUserUpdateRatesInvalidNumberInput() PASSED

Agile_47.AppTest > testWriteSameCurrencyTwice() FAILED
    org.opentest4j.AssertionFailedError at AppTest.java:69

Agile_47.AppTest > testFileWriteNoFilePresent() PASSED

Agile_47.CalculationTest > convertCurrency() PASSED

Agile_47.CalculationTest > calculateConversionRate() PASSED

15 tests completed, 4 failed

> Task :test FAILED
```

The command used to run tests was: **gradle test**

This command automatically ran the JUnit tests. As shown above, the code passes the tests. However, earlier versions of the code did not as shown below. The tests that failed were used to improve the code and ultimately pass all test cases created.

Gradle also created a report which showed which tests passed and which failed, as well as the overall rate of success of the code. The content of this report for the failed tests demonstrated above is:

Test Summary

15	4	0	0.172s
tests	failures	ignored	duration

73%
successful

Failed tests

Packages

Classes

AppTest. testFileWriteOneRateFirstLine()
AppTest. testFileWriteOneRateLastLine()
AppTest. testFileWriteThreeRates()
AppTest. testWriteSameCurrencyTwice()

The report for the final code is shown below:

Test Summary

33	0	0	0.334s
tests	failures	ignored	duration

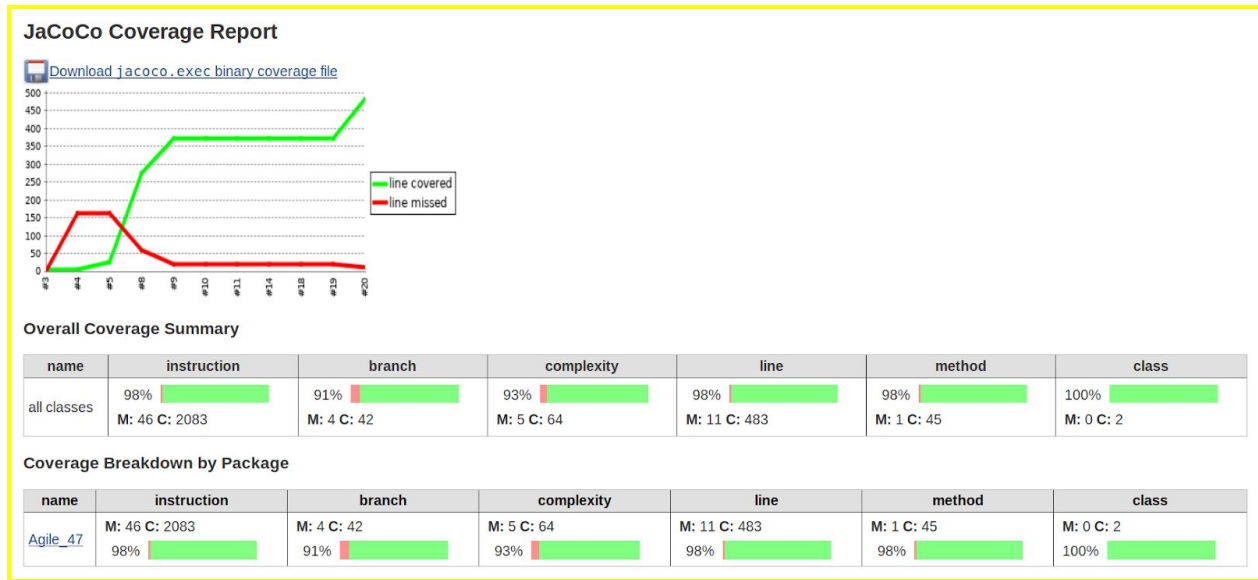
100%
successful

Packages

Classes

Package	Tests	Failures	Ignored	Duration	Success rate
<u>Agile_47</u>	33	0	0	0.334s	100%

Jacoco was also integrated with Gradle into order to produce code coverage reports. The reports included a breakdown of code coverage for every class, method and line. The report for the final code is shown below:



The command to run the program is: **gradle run**

The output of “gradle run” when run on the finished program is shown below:

```
[Lizs-MacBook-Pro:Agile_47 lizandrews$ gradle run

> Task :run
Enter 1 for converting currency or 2 for editing rates
<=====75% EXECUTING [5s]
How many currencies do you want to convert?
<=====75% EXECUTING [7s]
Enter the 1st amount (with a currency symbol e.g. 3.6USD)
<<=====75% EXECUTING [14s]
Enter the 2nd amount (with a currency symbol e.g. 3.6USD)
<<=====75% EXECUTING [18s]
Enter the 3rd amount (with a currency symbol e.g. 3.6USD)
<<<=====75% EXECUTING [23s]
Which currency do you want the result displayed in?
<=====75% EXECUTING [26s]
Your result is 45.04 EUR
```

2. Explanation of the build.gradle file

When the gradle command is called, it looks for a build.gradle file in the project directory. It is a build script and defines a project and its tasks. Our build.gradle file is as follows:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'org.junit.platform:junit-platform-gradle-plugin:1.1.0'
    }
}
/**
```


The dependencies in the 'buildscript' block controls the dependencies specific to the Gradle build process, not for the application code. An example of this being gradle-lint-plugin for linting, which are found from build script repos, and would not be referenced as dependencies for the application code.

```
*/
```

```
}
```

```
apply plugin: 'java'
```

//The above line was needed as java was the language used for the assignment

```
apply plugin: 'application'
```

```
/**
```

The above line was needed as we wanted to run a Java application. It was also needed as it comes with JUnit4 for testing, and creates the directory structure for source code and test files.

```
*/
```

```
apply plugin: 'jacoco'
```

//The above line was used to integrate Jacoco with gradle in order to create reports for code coverage

```
mainClassName = 'Agile_47.App'
```

//This is the name of the main class used

```
repositories {
```

```
    mavenCentral()
```

//This was needed as all dependencies are supposed to be looked up on the Maven Central Repository

```
}
```

```
dependencies {
```

```
    testCompile 'org.junit.jupiter:junit-jupiter-api:5.1.0'
```

```
    testRuntime 'org.junit.jupiter:junit-jupiter-engine:5.1.0',
```

```
        'org.junit.vintage:junit-vintage-engine:5.1.0',
```

```
        'org.junit.platform:junit-platform-launcher:1.1.0',
```

```
        'org.junit.platform:junit-platform-runner:1.1.0'
```

```
    /**
```

The above dependencies are needed as they allow Gradle to integrate with JUnit for testing. junit-vintage-engine is for JUnit 4 and 3, junit-platform-launcher provides an API for launching and configuring tests when using it with IDEs or other build tools, and junit-platform-runner allows the tests to be executed in a JUnit4 environment.

```
    */
```

```

}
test {

    useJUnitPlatform()
    // This project will use JUnit for testing

    test.finalizedBy jacocoTestReport
    // A Jacoco report will be generated after the tests are run

    testLogging {

        showStandardStreams = true
        // This will allow standard out streams to be printed to the console when called during tests. This allows for debugging of test cases.

        events "passed", "skipped", "failed", "standardOut", "standardError"
        // This will print the test results next to each test case for quick and easy debugging during development
    }
}
jacocoTestReport {
    reports {
        html.enabled = true
        // This will generate a Jacoco report in HTML format

        csv.enabled = true
        // This will generate a Jacoco report in CSV format
    }
}
run {
    standardInput = System.in
    //This line allows gradle to capture standard input when the command “gradle run” is used. This is because our program gets user input from stdin. This allows us to run our program and give it standard input in order to check that it works.
}

```

3.Jenkins CI

Jenkins was used in the project to automate building and testing at every commit to master. It also provided code coverage reports and reporting of test results, as well as archiving the project output for each build so that they can be referred to as required.

Github Integration

We created a new job on Jenkins that would trigger a build when a git event is performed, in our case a push to the master branch. Then, we added the Github Integration Plugin on Jenkins.

To automate the CI pipeline, we hooked Jenkins to every Github commit to the master branch, which ran gradle and JUnit automatically. We created an ngrok instance using **ngrok http 8080** on our Ubuntu virtual machine to obtain a public IP address, then we updated the IP address in our Github repository so that each commit can communicate with our Jenkins instance.

```
soft2412@soft2412-VirtualBox: ~
File Edit View Search Terminal Help
ngrok by @Inconshreveable (Ctrl+C to quit)

Session Status      online
Session Expires     7 hours, 38 minutes
Version             2.3.34
Region              United States (us)
Web Interface        http://127.0.0.1:4040
Forwarding           http://e6359bba.ngrok.io -> http://localhost:8080
Forwarding           https://e6359bba.ngrok.io -> http://localhost:8080

Connections          ttl    opn    rt1    rt5    p50    p90
                    5      0      0.00   0.00   0.75   0.92

HTTP Requests
-----
POST /github-webhook/ 200 OK
POST /github-webhook/ 200 OK
POST /github-webhook/ 200 OK
POST /github-webhook/ 200 OK
POST /github-webhook/ 200 OK
```

Webhooks

Add webhook

Webhooks allow external services to be notified when certain events happen. When the specified events happen, we'll send a POST request to each of the URLs you provide. Learn more in our [Webhooks Guide](#).

We will also send events from this repository to your [organization webhooks](#).

✓ <http://e6359bba.ngrok.io/github-webhook/> (push)

Edit

Delete

On commits to master, we also set up Jenkins to invoke the gradle wrapper and run gradle clean build on our project, which builds and tests our program.

Build

Invoke Gradle script

X

?

☐ Invoke Gradle

☒ Use Gradle Wrapper

Make gradlew executable

☒

Wrapper location

?

Tasks

clean build

▼

?

Advanced...

Add build step ▼

After the build, the project JAR file as well as JUnit and Jacoco HTML and XML reports are archived and can be downloaded as a .zip file. This is repeated for each build so that test reports can be retrieved for previous builds easily. This was done by configuring post-build actions in Jenkins.

Post-build Actions

Archive the artifacts

X

?

Files to archive

build/test-results/test/*.xml, build/reports/tests/test/index.html, build/reports/jacoco/test/html/index.html, build/libs/*.jar

?

Advanced...

Publish JUnit test result report

X

?

Test report XMLs

/test-results//*.xml

?

[Fileset 'includes'](#) setting that specifies the generated raw XML report files, such as 'myproject/target/test-reports/*.xml'. Basedir of the fileset is [the workspace root](#).

☐ Retain long standard output/error

?

Health report amplification factor

1

?

1% failing tests scores as 99% health. 5% failing tests scores as 95% health

Allow empty results

☐ Do not fail the build on empty test results

?

The trends for JUnit test results are also shown after each build based on the .xml file and provides easy visualisation of passed/failed test cases. A Jacoco report is also shown to keep track of code coverage.

Project jenkins-github

Jenkins Github Integration

[edit description](#)

[Disable Project](#)

Workspace

Last Successful Artifacts

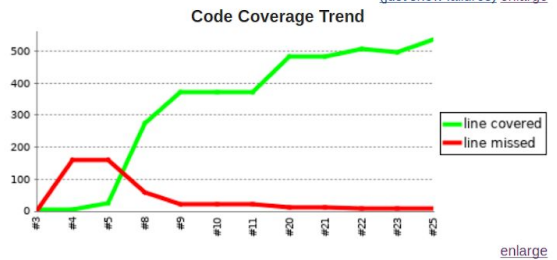
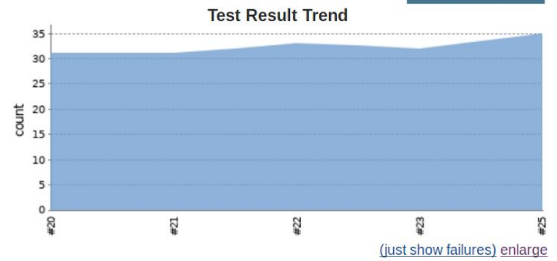
 Agile_47.jar	4.40 KB view
 html/index.html	3.03 KB view
 test/index.html	2.27 KB view
 TEST-Agile_47.AppTest.xml	3.41 KB view

Recent Changes

Latest Test Result (no failures)

Permalinks

- [Last build \(#25\), 1 min 54 sec ago](#)
- [Last stable build \(#25\), 1 min 54 sec ago](#)
- [Last successful build \(#25\), 1 min 54 sec ago](#)
- [Last failed build \(#24\), 8 min 10 sec ago](#)
- [Last unsuccessful build \(#24\), 8 min 10 sec ago](#)
- [Last completed build \(#25\), 1 min 54 sec ago](#)



As shown above, jenkins automates the code coverage reporting, as well as archiving the project outputs. Below we have evidence of automated build and test coverage for a successful build.

Console Output

```
Started by GitHub push by bphu6538
Running as SYSTEM
Building in workspace /var/lib/jenkins/workspace/jenkins-github
using credential 9b2b02d9-9c9e-4c6b-943a-9806645e966e
> git rev-parse --is-inside-work-tree # timeout=10
Fetching changes from the remote Git repository
> git config remote.origin.url https://github.sydney.edu.au/50FT2412-201952/Agile_47.git # timeout=10
Fetching upstream changes from https://github.sydney.edu.au/50FT2412-201952/Agile_47.git
> git --version # timeout=10
using GIT_ASKPASS to set credentials
> git fetch --tags --progress https://github.sydney.edu.au/50FT2412-201952/Agile_47.git +refs/heads/*:refs/remotes/origin/*
> git rev-parse refs/remotes/origin/master^{commit} # timeout=10
> git rev-parse refs/remotes/origin/origin/master^{commit} # timeout=10
Checking out Revision b211201b97b4ab799738fa7986f43157f380cda9 (refs/remotes/origin/master)
> git config core.sparsecheckout # timeout=10
> git checkout -f b211201b97b4ab799738fa7986f43157f380cda9
Commit message: "Merge pull request #9 from 50FT2412-201952/readFileUnitTests"
> git rev-list --no-walk 868d7885a380a2a62b9ed041e0221a97f3b4f362 # timeout=10
[Gradle] - Launching build.
[jenkins-github] $ /var/lib/jenkins/workspace/jenkins-github/gradlew clean build
> Task :clean
> Task :compileJava
> Task :processResources NO-SOURCE
> Task :classes
> Task :jar
> Task :startScripts
> Task :distTar
> Task :distZip
> Task :assemble
> Task :compileTestJava
> Task :processTestResources NO-SOURCE
> Task :testClasses
> Task :test
Agile_47.AppTest > testFileWriteThreeRates() PASSED
Agile_47.AppTest > testFileWriteInvalidCurrency() PASSED
Agile_47.AppTest > testPromptUserInvalidSelection() PASSED
Agile_47.AppTest > testFileWriteOneRateLastLine() PASSED
```

```

Agile_47.AppTest > testUserUpdateRatesInvalidNumberInput() PASSED

Agile_47.AppTest > testnotValidCurrency() PASSED

Agile_47.AppTest > testWriteSameCurrencyTwice() PASSED

Agile_47.AppTest > testFileWriteNoFilePresent() PASSED

> Task :jacocoTestReport
> Task :check
> Task :build

BUILD SUCCESSFUL in 4s
9 actionable tasks: 9 executed
Build step 'Invoke Gradle script' changed build result to SUCCESS
Archiving artifacts
Recording test results
[JaCoCo plugin] Collecting JaCoCo coverage data...
[JaCoCo plugin] **/*.exec:**/classes:**/src/main/java; locations are configured
[JaCoCo plugin] Number of found exec files for pattern **/*.exec: 1
[JaCoCo plugin] Saving matched execfiles: /var/lib/jenkins/workspace/jenkins-github/build/jacoco/test.exec
[JaCoCo plugin] Saving matched class directories for class-pattern: **/classes:
[JaCoCo plugin] - /var/lib/jenkins/workspace/jenkins-github/build/classes 2 files
[JaCoCo plugin] - /var/lib/jenkins/workspace/jenkins-github/build/reports/tests/test/classes 0 files
[JaCoCo plugin] Saving matched source directories for source-pattern: **/src/main/java:
[JaCoCo plugin] Source Inclusions: **/*.java
[JaCoCo plugin] Source Exclusions:
[JaCoCo plugin] - /var/lib/jenkins/workspace/jenkins-github/src/main/java 1 files
[JaCoCo plugin] Loading inclusions files..
[JaCoCo plugin] inclusions: []
[JaCoCo plugin] exclusions: []
[JaCoCo plugin] Thresholds: JacocoHealthReportThresholds [minClass=0, maxClass=0, minMethod=0, maxMethod=0, minL
minBranch=0, maxBranch=0, minInstruction=0, maxInstruction=0, minComplexity=0, maxComplexity=0]
[JaCoCo plugin] Publishing the results..
[JaCoCo plugin] Loading packages..
[JaCoCo plugin] Done.
[JaCoCo plugin] Overall coverage: class: 100, method: 98, line: 98, branch: 91, instruction: 98
Finished: SUCCESS

```

An explanation of the output above is as follows: Jenkins fetches the latest commit from our Github repository, then it runs “gradlew clean build” to build and test the commit. After building and testing without errors, it then runs the post build actions in the pipeline, which are archiving the artifacts, recording the test results, and then calling the jacoco plugin for code coverage report.

4.Application Development

Work Iterations and Group Communication

We went through weekly work iterations after each week's action items were set. In between meetings, any problems we encountered would be communicated and discussed on an online group chat. This was set up at the beginning of the project. The minutes for each of our meetings are included below:

Minutes for Agile_47

Meeting 1

- Location: Madsen Computer Lab 211
- Time: 11:30am - 12:00pm
- Date: 03 September 2019

Attendance: Sarah, Darby, Liz

Agenda:

- Discuss how to implement the application
- Allocate specific components of the currency conversion to each group member, and deadline for each task

Action Items:

- Darby: main interface in retrieving input from user
- Liz: read the rates from file, write the updated rates to a file
- Sarah: write the conversion function (i.e. convert 100 AUD to USD)
- Ben: Gradle and Jenkins on Virtual Machine

Meeting 2

- Location: Madsen Computer Lab 211
- Time: 11:30am - 12:00pm
- Date: 10 September 2019

Attendance: Sarah, Darby, Liz, Ben

Agenda:

- Discuss how to integrate Jenkins, Github and Jacoco coverage reports

- How to integrate the individual components from each group member into one coherent class

Action Items:

- Write up simple test cases to test if the test coverage reports show up by next week
- Place all small individual class files into one main App class
-

Meeting 3

- Location: Madsen Computer Lab 211
- Time: 11:30am - 12:00pm
- Date: 17 September 2019

Attendance: Sarah, Darby, Liz, Ben

Agenda:

- Discuss how many and who writes the test cases for each function
- Allocating which section of the technical report shall be assigned to which member

Action Items:

- Ben: report for Jenkins CI, unit tests for userUpdateRates function
- Liz: report for Gradle Integration, unit tests for writeToFile
- Sarah: report for Github Integration, unit tests for currConvert
- Darby: report for Junit Testing, unit tests for readInRatesFile
- Everyone: review each other's code and performing integration/acceptance tests

Application Structure / Design

The application was designed in the first meeting. It was decided to create a command line application which would allow the user to choose between either 1) converting currencies or 2) updating rates. It was decided to store the rates in a separate text file which would be read in at the start of the application. This was done so that any changes to the rates would be permanent and would not be lost when the program ended.

If the user chose the converting currency option, they would be asked how many currencies they wanted to convert. They could then enter the symbol and amount for each currency (e.g. 3USD). They would then be prompted for the output currency (e.g. AUD). In this way, they could sum together any number of different currencies and get the output in any other desired currency.

If the user chose the updating rates function, they would be asked which rates they wished to change. They would then be able to input new values for these rates, which would be written to the file.

Explanation of how the currency converter produces correct output

The converter reads in the rates of a currency for 1 AUD. For example, "USD 0.7" means that 1AUD = 0.6803USD, while "HKD 5.318" means that 1AUD = 5.318. If the user wants to convert

20USD to HKD, the application firstly converts 20USD to AUD. It does this by inverting the rate i.e. $20 \text{ USD} = 20 * 1 / 0.6803 \text{ AUD} = 29.40 \text{ AUD}$. It then converts it from AUD to HKD. It does this by multiplying by the rate i.e. $29.40 \text{ AUD} = 29.40 * 5.318 \text{ HKD} = 156.34 \text{ HKD}$. In this manner, any currency can be converted to any other currency included in the rates file.

Agile Tool Setup

Agile tools such as Github (version control), Jenkins, Gradle (build and test automation), JUnit, and Jacoco (code coverage reporting) were set up at the start of the project, and were integrated to provide an agile development environment for the entirety of the project.

5.JUnit Testing

One of the most important pieces in the development of the teams currency converter, “App.java”, was the use of the Junit Framework to assist in identifying errors and potential flaws in the program. It was paramount that the changes to code tested regularly and consistently in order to ensure the best functionality possible.

JUnit, as has been described in both the course’s lectures and tutorial sheet in week 5, is designed to write and run tests written in the Java programming language. During the report the team particularly made use of both *unit testing* and *integration tests*. Both important methods the difference between the two is that in *unit testing*, the written test focuses on a particular method, ensuring that it works in the way that the programmer expects, giving clarity over whether the given method is functional or ‘bugged’. The alternative, *integration testing* is essentially the opposite of this, opting to test the interactions between the various methods and classes used in the development and ensuring that the code remains functional when passing through multiple sections of code. As will be shown below, the team wrote various test cases to cover the majority of the code, in both a unit test and integration tests written using JUnit.

In order to write these Junit tests effectively in conjunction with the teams, test automaker/builder, Gradle, a few prerequisites were needed in order to allow for the use of this method of testing at all. As described in the section on Gradle, when setting these up on our computers it was necessary to add various dependencies and plug in to the build.gradle file of the teams repository. This then allowed to team to have access to the Junit classes after inputting:

```
import static org.junit.jupiter.api.Assertions.*;  
import org.junit.jupiter.api.AfterEach;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;
```

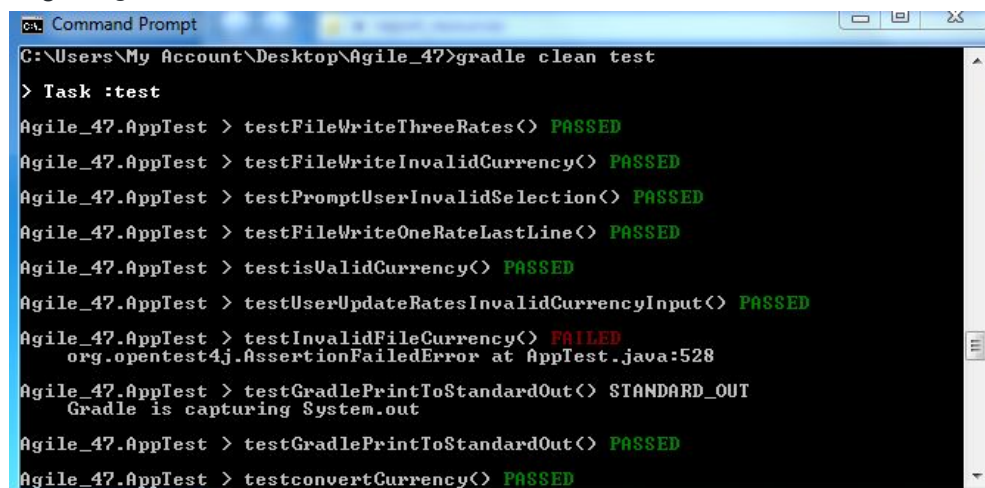
At the start of the testing classes. These dependencies in the build.gradle file can be seen above in the Gradle section.

Each member of the team was then provided with access to the Junit tests and was free to write relevant tests for assigned sections of code, being, Liz - writeToFile, Ben - updateUserRates, isValidCurrency, Sarah - currConvert, Darby - readInRatesFile. (these were assigned via an issue on the team GitHub).

In order to write a test, once the code for the group member was written they would need to place these tests into a specific area of the teams github gradle branch, into a folder called `.../src/test/java/Agile_47`, different to the folder kept for the main source code for the application `.../src/main/java/Agile_47`. This is to ensure that the tests are able to be executed by gradle once they were written.

To create one of these testing files, the team members created their own java testing class on their branch, allowing for the next step to take place, the writing of the Junit tests which were later merged into the same file. Some of the syntax used in these junit tests include, the import junit modules as mentioned above, the `@Test` in order to mark that the preceding method is infact a test, `@BeforeEach` and `@AfterEach` to mark that the following method should be executed before every test or after every test respectively along with many versions of the `Assert` function, such as `assertTrue`, `assertFalse`, `assertEqual` etc, a decisive part of each test to determine whether the actual output of the program is what the developer expects or something else.

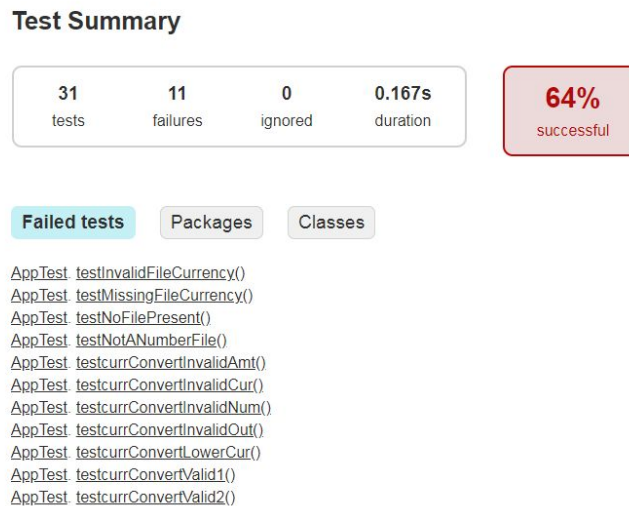
Once a team member had written a valid test, providing that gradle was running correctly with files in the correct places, these tests were able to be run through the “gradle clean test” or “gradle clean build” commands from the root of the setup gradle file. Once the tests were written correctly, the build would either fail or pass depending on whether the test cases produced their expected outputs defined in the `assert` commands. The screenshot below shows the output when running the `gradle clean test` command on an older version of the code.



```
C:\Users\My Account\Desktop\Agile_47>gradle clean test
> Task :test
Agile_47.AppTest > testFileWriteThreeRates() PASSED
Agile_47.AppTest > testFileWriteInvalidCurrency() PASSED
Agile_47.AppTest > testPromptUserInvalidSelection() PASSED
Agile_47.AppTest > testFileWriteOneRateLastLine() PASSED
Agile_47.AppTest > testIsValidCurrency() PASSED
Agile_47.AppTest > testUserUpdateRatesInvalidCurrencyInput() PASSED
Agile_47.AppTest > testInvalidFileCurrency() FAILED
    org.opentest4j.AssertionFailedError at AppTest.java:528
Agile_47.AppTest > testGradlePrintToStandardOut() STANDARD_OUT
    Gradle is capturing System.out
Agile_47.AppTest > testGradlePrintToStandardOut() PASSED
Agile_47.AppTest > testconvertCurrency() PASSED
```

As gradle had also been integrated with jacoco (defined in the build.gradle folder) along with the build file, we are able to have a closer look at the report using the gradle and jacoco files in

order to see what was going on. This was done by going from the root of the gradle repository going to `>build>reports>tests>test>index.html` which when opened in a browser was able to give more feedback on the code an example is shown below, which will be explained in detail below.



The displayed HTML document was able to give a summary of all the tests that were failing and why they were failing and when they were failing. In the case of these test cases above the expected output string matches the actual output however due to the way that the assertions are formatted (as seen later on), don't match in expected and given output. Each team member following this process, was able to produce a satisfactory amount of test cases for their assigned piece of source code on their branch.

Code coverage is another important part of testing as it has the effect of maximizing the chance of finding a bug the higher the coverage of the code, lowering the chance of having an unexpected, damaging and potentially dangerous flaw in your code. According to Atlassian; *"Code coverage is the percentage of code which is covered by automated tests. Code coverage measurement simply determines which statements in a body of code have been executed through a test run, and which statements have not"*. The assignment specs made it clear that groups should be aiming for code coverage greater than or equal to 75% of the code. In order to meet this requirement, the *jacoco report* is necessary in order to get a better idea of the code coverage. This was able to be added into the gradle project through again editing the build.gradle file and adding the lines of code,

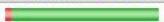

apply plugin: 'jacoco'

```
jacocoTestReport {  
    reports {  
        html.enabled = true  
        csv.enabled = true  
    }  
}
```

These have the effect of making an automatic jacoco report available every time the gradle build is executed. After the final round of tests, the report gives the following report:

Agile_47


















Agile_47

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Agile_47		95%		91%	5	35	11	152	1	12	0	1
Total	25 of 543	95%	4 of 46	91%	5	35	11	152	1	12	0	1

The general report at a glance tells us that we have a total code coverage of 95%, missing only 4/46 branches, in order to get a closer look at this we can access the further breakdown into methods of our main file, "App.class".

Agile_47 > Agile_47 > App

App

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
readInRatesFile(String)		89%		80%	2	6	4	33	0	1
main(String[])		0%		n/a	1	1	3	3	1	1
writeToFile(String, double, String)		95%		91%	1	7	3	33	0	1
promptUser(App)		94%		75%	1	3	1	11	0	1
currConvert()		100%		100%	0	4	0	36	0	1
userUpdateRates(Scanner, String)		100%		100%	0	3	0	24	0	1
isValidCurrency(String)		100%		100%	0	6	0	4	0	1
calculateConversionRate(double, double)		100%		n/a	0	1	0	2	0	1
convertCurrency(double, double)		100%		n/a	0	1	0	2	0	1
static {...}		100%		n/a	0	1	0	1	0	1
gradlePrintToStandardOut()		100%		n/a	0	1	0	2	0	1
App()		100%		n/a	0	1	0	1	0	1
Total	25 of 543	95%	4 of 46	91%	5	35	11	152	1	12

This tells us which of the methods in the main file are being covered by the tests that have been written. As a whole it can be seen that the coverage of the code is high and there are some tests that are failing (indicated by the red), and the majority of the branches for the team applications code is high, hopefully minimising the chance of having an unknown bug in the code.

A further graphic provided by jacoco to assist in the analysis of code coverage is shown below.

App.java

```
1. package Agile_47;
2.
3. import java.util.*;
4. import java.io.*;
5.
6. public class App{
7.     public static HashMap<String, Double> rates = new HashMap<String, Double>();
8.
9.     //Check that the currency input is valid
10.    public static boolean isValidCurrency(String currency){
11.        if (currency.equals("AUD") || currency.equals("USD") || currency.equals("RMB") || currency.equals("HKD")
12.        || currency.equals("EUR")){
13.            return true;
14.        }
15.        return false;
16.    }
17.
18.    //Read in the rates stored in the file
19.    public static void readInRatesFile(String filename){
20.        //String filename = System.getProperty("user.dir") + "/rates.txt";
21.        rates = new HashMap<String, Double>();
22.        File f = new File(filename);
23.        try{
24.            BufferedReader br = new BufferedReader(new FileReader(f), 100);
25.            String s;
26.            int lineNumber = 0;
27.            while ((s = br.readLine()) != null){
28.                lineNumber++;
29.                //Check the line is long enough
30.                if (s.length() < 5){
31.                    System.out.println("Line " + lineNumber + " has the incorrect format");
32.                    continue;
33.                }
34.
35.                //Get the currency (3 letters)
36.                String currency = s.substring(0, 3).toUpperCase();
37.                if (isValidCurrency(currency)!=true){
38.                    System.out.println("Line " + lineNumber+ " is not a valid currency");
39.                    continue;
40.                }
41.            }
42.        }
43.    }
44. }
```

Above shows the code branches of the tree that have been covered in green highlight, whilst the lines of code that have not been reached during testing are highlighted in red. The team made great use of all of the outlined features of jacoco and gradle, particularly the reports, to provide a greater quality of testing, allowing each of us to see where in the code has been effectively checked and places that could potentially have issues or might not have been tested thoroughly enough.

After continual consultation of the reports in making the test cases, they were eventually finalised, and each of us created a GitHub pull request to the repository, ensuring that the tests were working before applying them to the master working branch, although there were some initial problems (as will be explained below), the tests that were ultimately decided on were those that correctly ran, provided useful and practical examples of the code in use, the final tests will be described below, with an explanation of what that code is testing for, and why it's testing for that outcome. Each of the following tests are displayed in bold, with explanations shown below.

@BeforeEach

```
void setUp() {
    test = new App();
    test.rates = new HashMap<String, Double>();
}
```

```
}
```

The above is the first line of testing in the teams finalised testing file of the currency converter (App.java), which has been called "AppTest.java" and of course has the relevant Junit and other classes defined before this first test, however for this particular test, the type defined is a *BeforeEach* meaning that this scope of code, 'BeforeEach', is to be executed before each of the tests, in this case an instance of the App class is initialised along with an instance of the accompanying HashMap that stores the conversion rates of the currency converter, this is done here to save us from writing similar lines of code at the start of every individual test.

```
@AfterEach
void tearDown() {
    test = null;
}
```

Similar to the code above, this method in the "AppTest.java" file, named "tearDown" is executed at the end of every test, essentially the code sets the value of the variable "test", to null, meaning that there is no memory associated with that variable any longer. This is done at the end of every test to complement the setUp() method. Essentially by having this method at the end of every test case, it stops the occurrence of errors, like saying that the value for 'test' has been already defined, and this error is avoided by resetting the area of memory associated with it to null each time so that it can be again setUp() effectively at the start of the next test case without errors. Having these two functions above has the added benefit of being able to reuse the same variable name for the instance of the App class which can prevent confusion later on.

```
@Test //RegularCase
void testIsValidCurrency() {
    String currency = "AUD";
    assertTrue(test.isValidCurrency(currency));

    currency = "USD";
    assertTrue(test.isValidCurrency(currency));

    currency = "RMB";
    assertTrue(test.isValidCurrency(currency));

    currency = "HKD";
    assertTrue(test.isValidCurrency(currency));

    currency = "EUR";
    assertTrue(test.isValidCurrency(currency));
}
```

The above test is the first Junit test marked with the @Test meaning that this going to be a proper test of the code with assertions. In the test case above the code is determining whether the isValidCurrency() function will return the desired output. In our application, the 5 currencies that the user is able to convert between includes AUD, USD, RMB, HKD, and EUR. This test runs the instance of the App defined in the setUp() method previously through the "isValidCurrency" method with the string holding the currency code, named "currency" as a parameter, the method its being passed to outputs a boolean result true if the parameter string is the code of a valid (one of the 5 currencies) or not. This test has been decided on to test the basic functionality of this important function of the code, because if the assertTrues were to fail

this would mean that when the user is inputting a valid currency to be converted, then for some reason our code would not be seeing it as valid. This is an important part of the program, so it is vital that this test case is passed.

```
@Test //EdgeCase
void testNotValidCurrency() {
    String currency = "INR";
    assertFalse(test.isValidCurrency(currency));
}
```

Similar to the explanation given to the code above this code also determines whether the important method in the App.java file, isValidCurrency() is working. The only difference is this time we test the alternative outcome, we now make sure that when a string of a currency that is not accepted by our applications. INR, in this example, returns false and not true. As can be seen by the assert false method shown. If this assertFalse returns true, the test passes and it can be said that the function performs its role effectively determining when both a valid and invalid currency are fed to the isValidCurrency() function through the parameter.

```
@Test //RegularCase
//Tests writing a currency on the first line of the file
public void testFileWriteOneRateFirstLine() {
    String filename = System.getProperty("user.dir") + "/rates.txt";
    test.writeToFile("AUD", 22.8, filename);

    File f = new File(filename);
    String s = "";
    try {
        BufferedReader br = new BufferedReader(new FileReader(f), 100);
        s = br.readLine();
    } catch (Exception e) {}
    assertEquals("AUD 22.8", s);
}
```

The function above tests a different function this time within the teams application source code, this time it tests that the writeToFile method of the App class performs its role effectively when given a valid currency, valid number and a valid filename (in this case the rates.txt file located in the users current directory path. The try and catch make a *file reader* object from the *file* object made using the *rates.txt* file path as a name, once this file has been read in it checks the first line of the file reader and assigns it to a previously defined string variable. This string variable representing the first line of the edited file, is then able to be compared to the expected output, "AUD 22.8" shown in the parameters of the assertEquals function at the bottom. This test is important as it tests the basic functionality of the writeToFile method of the app, the team values this test result as writing to files is very important as it is the way in which the team has chosen to implement the editing rates feature. If this test were to fail, this would mean that the outcome of adding a valid file and valid rate to the file would be unsuccessful, resulting in a failure of the changing rates feature.

```
@Test //RegularCase
public void testFileWriteOneRateLastLine() {
    String filename = System.getProperty("user.dir") + "/rates.txt";
    test.writeToFile("EUR", 1.89, filename);
}
```

```

File f = new File(filename);
String s = "";
try {
    BufferedReader br = new BufferedReader(new FileReader(f), 100);
    s = br.readLine();
    s = br.readLine();
    s = br.readLine();
    s = br.readLine();
    s = br.readLine();
} catch (Exception e) {}
assertEquals("EUR 1.89", s);
}

```

In the code above, a very similar functionality is being tested. The difference is that this time it is the value for another valid currency code, EUR, that is being altered rather than the AUD as before. In order to analyse whether this `writeToFile()` function was performed as expected, we again use a `BufferedReader` type to read the file and then proceed to read the lines in the file until the fifth line which is the line on which the changed rate is expected to be and by storing this line as a string, we can again pass it to the `assertEqual` function to compare it to the expected output. This function is also important because it tests to see whether the `writeToFile` function is able to change not just one currency and is able to correctly write to the given file for any valid rate, despite being in different lines of the text file on which the rates are stored.

```

@Test //RegularCase
//Tests writing to the same currency twice
public void testWriteSameCurrencyTwice() {
    String filename = System.getProperty("user.dir") + "/rates.txt";
    test.writeToFile("HKD", 11.11, filename);
    test.writeToFile("HKD", 22.22, filename);

    File f = new File(filename);
    String s = "";
    try {
        BufferedReader br = new BufferedReader(new FileReader(f), 100);
        s = br.readLine();
        s = br.readLine();
        s = br.readLine();
        s = br.readLine();
    } catch (Exception e) {}
    assertEquals("HKD 22.22", s);
}

```

This time the code continues to test the `writeToFile` method of the `App.java` file, but this time it is given a little trickier input. In this example rather than just giving a single valid rate to change, we test that when the method is called twice both with valid rates, that the most recent (i.e. second) rate is the one that is stored in the file. This is done in a similar way to the previous test cases however the main difference is that before the file is analysed again using a `BufferedReader` to compare the expected to the output value, it is given two lots of the `test.writeToFileMethod`. This test is important as it again determines whether an important feature of the program is performing as expected, it is easily possible that the second outcome could be ignored, added to a different line, or handled in a different way to expected, but with this test case we are able to determine that the method when in this situation performs as expected.

```

@Test //RegularCase
public void testFileWriteThreeRates() {
    String filename = System.getProperty("user.dir") + "/rates.txt";
    test.writeToFile("EUR", 16.2, filename);
    test.writeToFile("HKD", 3.5, filename);
    test.writeToFile("AUD", 109.788, filename);

    File f = new File(filename);
    String s1 = "";
    String s4 = "";
    String s5 = "";
    try {
        BufferedReader br = new BufferedReader(new FileReader(f), 100);
        s1 = br.readLine();
        br.readLine();
        br.readLine();
        s4 = br.readLine();
        s5 = br.readLine();
    } catch (Exception e) {}
    assertEquals("AUD 109.788", s1);
    assertEquals("HKD 3.5", s4);
    assertEquals("EUR 16.2", s5);
}

```

Again testing the `writetoFile` method this test makes sure that when given some valid inputs, in this case three different ones rather than the same as shown before that the output is as expected. This test obtains the needed strings in the same way as described in the last two test cases. This test is important in the same way the others are as it gives the developers the assurance that despite adding multiple different valid currencies that the code is still running effectively, and that the text file called upon with the method actually changes the way in which the user expects.

```

@Test //EdgeCase
public void testFileWriteNoFilePresent() {
    PrintStream orig = System.out;
    String filename = System.getProperty("user.dir") + "/notAFile.txt";
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setOut(new PrintStream(out));
    test.writeToFile("USD", 200.01, filename);
    String res = out.toString();
    assertEquals(res, "Error reading file\n");
    System.setOut(orig);
}

```

This code is a little different to the previous ones, although testing the same method of the source code, `writeToFile`, this time it is given an invalid filename, one that does not exist and it has to test that the stdout given to the user matches the error message that is expected. In order to do this test a filename for the rates is constructed which doesn't exist in the users directory. Although given valid input as parameters the result is expected to come up with the error message "Error reading in file". This expected output is saved to a string and is compared to the standard out in an `assertEquals()` function. The hardest part of the implementation of this method is the capturing of the standard output of the program. This is why a `ByteArrayOutputStream` type is assigned to the `System.out` using the line, `System.setOut(new PrintStream(out));` This is then converted to a String which is thrown into the `assertEquals` function. Once

the assertion is done, the standard output is set back to the original which was saved on the very first line of the test. This test is important as it tests whether the error message when an invalid or non existing file is passed to the method that it provides the user with the appropriate error message and doesn't continue with any unexpected output or break the program all together.

@Test //EdgeCase

```
public void testFileWriteInvalidCurrency() {
    PrintStream orig = System.out;
    String filename = System.getProperty("user.dir") + "/rates.txt";
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setOut(new PrintStream(out));
    test.writeToFile("ABC", 200.01, filename);
    String res = out.toString();
    assertEquals(res, "ABC is not found in the file - cannot edit it\n");
    System.setOut(orig);
}
```

The next function now again continues to check the standard output of the program to see whether the appropriate error message is displayed when an invalid code is given to the writeToFile() method in the parameter. This test captures the standard output in the same way as above making use of the .setOut() method in the System class, and the ByteArrayOutputStream Type to store this output in temporarily which can be later converted to a string for comparison using the .toString() method. This file follows the same procedure as the one above, trying to use the method to set invalid input to the chosen valid file, but checks to see whether the appropriate error message and action are shown. This is again important as it ensures that invalid input from this method is not able to break our code or cause unexpected output.

@Test //RegularCase

// Tests that a valid user input will make the correct changes in rates.txt

```
public void testUserUpdateRatesValidInput() {
    PrintStream orig = System.out;
    ByteArrayInputStream in = new
    ByteArrayInputStream("1\nUSD\n1.5".getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setOut(new PrintStream(out));
    System.setIn( in );
    String filename = System.getProperty("user.dir") + "/rates.txt";
    Scanner sc = new Scanner(System.in);
    test.userUpdateRates(sc, filename);

    File f = new File(filename);
    String s = "";
    try {
        BufferedReader br = new BufferedReader(new FileReader(f), 100);
        br.readLine();
        s = br.readLine();
    } catch (Exception e) {}
    System.setOut(orig);
    // System.out.print(out.toString());
    assertEquals("USD 1.5", s);
}
```

The next test tests a different method in the class, the userUpdateRates() method, which calls upon the same function as above but this time obtains the parameters through interactions with the user. This test makes sure that when the user inputs valid input, the expected display is shown to them on the screen. In

order to make this one happen, it is important to not only capture the standard out but also predefining some standard input to simulate the user interaction with the program.

Although the standard output of the program is captured in the same way as the test cases above the standard input is set to the program in a similar way. A new `ByteArrayInputStream` type is set with the expected outputs of the user, in this case, `"1\nUSD\n1.5".getBytes()`, 1 being the answer to the `userUpdateRates()` methods first question of "How many rates do you want to edit?\n", USD being the users input for the next question "What is the 1st currency you want to edit?\n" (in this case), and 1.5 being the answer to the final user input responding to "What is the new rate for USD?\n". Then using the same method as above we test that the appropriate line in the file has been changed accordingly. This test is important because it tests that not only the function of writing to the test file is correct but it remains correct when called through the `userUpdateRates()` function as well. This test is important because it tests integration and also removes the chance of code breaking or unexpected output for user inputted values, and the output they are displayed.

@Test //EdgeCase

```
public void testUserUpdateRatesInvalidNumberInput() {
    PrintStream orig = System.out;
    ByteArrayInputStream in = new ByteArrayInputStream("NotANumber\nUSD\n1.5".getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setIn( in );
    System.setOut(new PrintStream(out));
    String filename = System.getProperty("user.dir") + "/rates.txt";
    Scanner sc = new Scanner(System.in);
    test.userUpdateRates(sc, filename);
    System.setOut(orig);
    assertEquals("How many rates do you want to edit?\nNot a number\n", out.toString());
}
```

The following code is similar to the code above, except this time rather than the user giving valid entries into the program, an invalid entry is given. It is done using the same method as above. This function is important as it checks that when not given an expected value that the program is able to handle it as well.

@Test //EdgeCase

```
public void testUserUpdateRatesInvalidRateInput() {
    PrintStream orig = System.out;
    ByteArrayInputStream in = new ByteArrayInputStream("1\nUSD\nNotANumber".getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setIn( in );
    System.setOut(new PrintStream(out));
    String filename = System.getProperty("user.dir") + "/rates.txt";
    Scanner sc = new Scanner(System.in);
    test.userUpdateRates(sc, filename);
    System.setOut(orig);
    // System.out.print(out.toString());
    assertEquals("How many rates do you want to edit?\nWhat is the 1st currency you want to edit?\nWhat is the new rate for USD?\nNot a number\n", out.toString());
}
```

This test again continues to focus on the same `userUpdateRates` method using the same method for analysis, although this test rather than when given an invalid input for how many rates they'd like to change, an invalid new rate is given. This case is chosen because it continues to make sure that all the

different branches of code that can be reached through errors like this one are able to be reached and again don't throw unexpected errors or behave in ways that are not expected.

```
@Test //EdgeCase
public void testUserUpdateRatesInvalidCurrencyInput() {
    PrintStream orig = System.out;
    ByteArrayInputStream in = new ByteArrayInputStream("1\nAAA\n1.5".getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setIn( in );
    System.setOut(new PrintStream(out));
    String filename = System.getProperty("user.dir") + "/rates.txt";
    Scanner sc = new Scanner(System.in);
    test.userUpdateRates(sc, filename);
    System.setOut(orig);
    assertEquals("How many rates do you want to edit?\nWhat is the 1st currency you want to edit?\nInvalid currency\n",
out.toString());
}
```

Using the same method as stated above, this testcase checks whether the expected output is shown to the user when an invalid currency code is given from user input. This test again continues to check all branches making sure that all potential areas of code that could be errored or produce unexpected output are tested.

```
@Test //Regular case
public void testPromptUserToChangeRates() {
    PrintStream orig = System.out;
    ByteArrayInputStream in = new ByteArrayInputStream("2\n1\nUSD\n1.5\n".getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setIn( in );
    System.setOut(new PrintStream(out));
    String filename = System.getProperty("user.dir") + "/rates.txt";
    test.promptUser(test);
    System.setOut(orig);
    assertEquals("Enter 1 for converting currency or 2 for editing rates\nHow many rates do you want to edit?\nWhat is the 1st currency you want to edit?\nWhat is the new rate for USD?\n", out.toString());
}
```

This function uses the method for checking the system output and providing user input, to take the integration testing a step further. In this test case rather than just skipping straight to the userUpdateRates() function, it opts to start the test at the promptUser() function which is the function called from main when properly executing this function. This test checks that when given valid input, the correct output is shown to the user. This test is an important one as it gives further evidence that the methods are able to work with each other as well as when called individually, this shows that when a valid input is given to change the rates table using the users input, the program is able to produce the expected result without bugs.

```
@Test //Edge case
public void testPromptUserInvalidSelection() {
    PrintStream orig = System.out;
    ByteArrayInputStream in = new ByteArrayInputStream("INVALID\n1\nUSD\n1.5\n".getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setIn( in );
```

```

    System.setOut(new PrintStream(out));
    String filename = System.getProperty("user.dir") + "/rates.txt";
    test.promptUser(test);
    System.setOut(orig);
    assertEquals("Enter 1 for converting currency or 2 for editing rates\nInvalid input\n", out.toString());
    //System.out.print(out.toString());
}

```

This time the same way as above is implemented to determine that the function `promptUser()` produces the correct error message when an invalid input is given for the first part of the conversion process. The test simulates the user entering 'INVALID' in response to the first question printed to the standard output "Enter 1 for converting currency or 2 for editing rates", it then uses the same comparison method as described in the above test cases to compare the standard output to what it's expected to be. This test was chosen because it's another unexpected input that the program should be able to handle properly without crashing and this test cases provides evidence that it can in fact do this.

```

@Test //Regular case
public void testcurrConvertValid1(){
    ByteArrayInputStream in = new ByteArrayInputStream("1\n5.0RMB\nUSD\n".getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setIn(in);
    System.setOut(new PrintStream(out));
    test.rates = new HashMap<String, Double>();
    test.rates.put("USD", 1.5);
    test.rates.put("AUD", 22.8);
    test.rates.put("RMB", 4.84489);
    test.rates.put("EUR", 1.89);
    test.rates.put("HKD", 22.22);

    test.currConvert();
    String expected = "How many currencies do you want to convert?\n" +
        "Enter the 1st amount (with a currency symbol e.g. 3.6USD)\n" +
        "Which currency do you want the result displayed in?\n" +
        "Your result is 1.55 USD\n";
    assertEquals(expected, out.toString());
}

```

The test above tests the `currConvert` method of the program, which does the calculations of the program in getting the rates. Similar to above it pre defines a series of user inputs in a string, which take the program to convert currencies in this case RMB to USD (both valid currencies). In this testcase the rates are predefined by calling `put` into the `HashMap` that holds all the conversion rates. Once this is done a string of expected outputs are put together which are then compared to the actual system output obtained using the `ByteArrayOutputStream` type. This test is important to test as it is probably one of if not the most critical parts of the program to get right, by having this test case passing it indicates that when valid user input is given to the function, the program is able to produce the expected correct result.

```

@Test //Regular case
void testcurrConvertValid2() {
    PrintStream orig = System.out;
    String input = "3\n" +

```



```

        "99USD\n100AUD\n105EUR\n" +
        "USD\n";
        ByteArrayInputStream in = new ByteArrayInputStream(input.getBytes());
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        System.setIn(in);
        System.setOut(new PrintStream(out));

        //Set up the rates
        test.rates = new HashMap<String, Double>();
        test.rates.put("USD", 0.67922473);
        test.rates.put("AUD", 1.0);
        test.rates.put("RMB", 4.81940382);
        test.rates.put("EUR", 0.61379632);
        test.rates.put("HKD", 5.31817394 );

        test.currConvert();
        String expected = "How many currencies do you want to convert?\n" +
            "Enter the 1st amount (with a currency symbol e.g. 3.6USD)\n" +
            "Enter the 2nd amount (with a currency symbol e.g. 3.6USD)\n" +
            "Enter the 3rd amount (with a currency symbol e.g. 3.6USD)\n" +
            "Which currency do you want the result displayed in?\n" +
            "Your result is 283.12 USD\n";
        assertEquals(expected, out.toString());
    }
}

```

This testcase is another that tests that when a user inputs correct values into the program, the program is able to produce the correct results. The only difference this time is that the user inputs 3 different currencies rather than 1, which are a requirement for the assignment.

```

@Test //Edge case
public void testcurrConvertInvalidNum() {
    String input = "0.1\n";
    ByteArrayInputStream in = new ByteArrayInputStream(input.getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setIn(in);
    System.setOut(new PrintStream(out));
    test.currConvert();
    String expected = "How many currencies do you want to convert?\n" +
        "Not a number\n";
    assertEquals(expected, out.toString());
}

```

The test above tests another kind of edge case, a currency that is between 0 and 3 the accepted inputs, yet is none of them, but is still a number. This is done using the same procedure defined above. The expected output of this should still be Not a number, as it is not an accepted one of the numbers that can be inputted here.

```

@Test //Regular case
public void testcurrConvertLowerCur() {
    PrintStream orig = System.out;
    String input = "1\n" +
        "99 usd\neur\n";
    ByteArrayInputStream in = new ByteArrayInputStream(input.getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();

```

```

System.setIn(in);
System.setOut(new PrintStream(out));

//Set up the rates
test.rates = new HashMap<String, Double>();
test.rates.put("USD", 0.67922473);
test.rates.put("AUD", 1.0);
test.rates.put("RMB", 4.81940382);
test.rates.put("EUR", 0.61379632);
test.rates.put("HKD", 5.31817394 );

test.currConvert();
String expected = "How many currencies do you want to convert?\n" +
    "Enter the 1st amount (with a currency symbol e.g. 3.6USD)\n" +
    "Which currency do you want the result displayed in?\n" +
    "Your result is 89.46 EUR\n";

System.setOut(orig);
assertEquals(expected, out.toString());

}

```

This test is another edge case in which the user should expect to see a valid output for the currencies they have inputted, but the codes inputted by the user are in lower case. The program is expected to still carry out the relevant calculation despite the codes not being in upper case as they are in the file or in the HashMap. This testcase again is important as it makes sure that the calculation is able to be carried out and displayed as expected to use even though the input is not in its expected form, an important part of the program.

```

@Test //Edge case
public void testcurrConvertInvalidCur() {
    String input = "1\n" +
        "99 nzd\n";
    ByteArrayInputStream in = new ByteArrayInputStream(input.getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setIn(in);
    System.setOut(new PrintStream(out));
    test.currConvert();
    String expected = "How many currencies do you want to convert?\n" +
        "Enter the 1st amount (with a currency symbol e.g. 3.6USD)\n" +
        "Invalid currency\n";
    assertEquals(expected, out.toString());
}

```

This case is when the user inputs into the calculation something with a valid layout but the currency code is not valid, this time the message slightly differs from the test cases above, in order to give the user a better idea of what exactly is going wrong. This again is another important branch of the code that needs to also be run in a test case to make sure expected output is given to the user.

```

@Test //Edge case
public void testcurrConvertInvalidAmt() {
    String input = "1\n" +
        "fiveaud\n";
    ByteArrayInputStream in = new ByteArrayInputStream(input.getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();

```

```

System.setIn(in);
System.setOut(new PrintStream(out));

//Set up the rates
test.rates = new HashMap<String, Double>();
test.rates.put("USD", 0.67922473);
test.rates.put("AUD", 1.0);
test.rates.put("RMB", 4.81940382);
test.rates.put("EUR", 0.61379632);
test.rates.put("HKD", 5.31817394 );

test.currConvert();
String expected = "How many currencies do you want to convert?\n" +
    "Enter the 1st amount (with a currency symbol e.g. 3.6USD)\n" +
    "Not a number\n";
assertEquals(expected, out.toString());
}

```

This rate again is testing for a different output message that should happen when the user inputs a value in the wrong format.

```

@Test //Edge case
public void testcurrConvertInvalidOut() {
    String input = "1\n" +
        "99 aud\nzasxusd\n";
    ByteArrayInputStream in = new ByteArrayInputStream(input.getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setIn(in);
    System.setOut(new PrintStream(out));

    //Set up the rates
    test.rates = new HashMap<String, Double>();
    test.rates.put("USD", 0.67922473);
    test.rates.put("AUD", 1.0);
    test.rates.put("RMB", 4.81940382);
    test.rates.put("EUR", 0.61379632);
    test.rates.put("HKD", 5.31817394 );

    test.currConvert();
    String expected = "How many currencies do you want to convert?\n" +
        "Enter the 1st amount (with a currency symbol e.g. 3.6USD)\n" +
        "Which currency do you want the result displayed in?\n" +
        "Invalid output currency\n";
    assertEquals(expected, out.toString());
}

```

This again has the same idea however, this time the invalid currency is put in on the second question where the program asks for the currency that it would like displayed in. This allows for another branch of code to be executed and tested for unexpected output.

```

@Test //Regular case
void testcalculateConversionRate() {
    double source_rate = 0.68747;
    double target_rate = 1.07654;
}

```

```

    double expected = 1.566;
    double actual = test.calculateConversionRate(source_rate, target_rate);

    assertEquals(expected, actual, 0.001);
}

```

The above test, is a unit test for the specific calculation method in the application, calculateConversionRate(), testing that the output is the same as expected when putting in hardcoded values rather than passing them from the user input, to make sure that the calculation itself is producing the correct output. This is an important test as it is a specific unit test, which are just as important as integration testing as shown above which are calling this method, as this calculation is the main piece in the conversion needed it is vital that the team has this method working when given valid numbers (as shown in the methods above, invalid input is never passed into this function as it is caught in the currConvert() function before it can be passed on)

```

@Test //Regular case
void testconvertCurrency() {
    double amount = 50;
    double conversion_rate = 1.566;
    double expected = 78.3;
    double actual = test.convertCurrency(amount, conversion_rate);

    assertEquals(expected, actual, 0.01);
}

```

This again like the test above tests the same method the only difference being the assert is being this time compared to 2 decimal places rather than three as shown in the above test.

```

@Test //Regular case
public void testReadInValidFile1(){
    String filename = System.getProperty("user.dir") + "/ValidInputTestOne.txt";
    test.readInRatesFile(filename);

    //Check if it produces the correct hashmap
    HashMap ans = test.rates;

    assertEquals(ans.size(), 5);
    assertEquals(ans.get("USD"), 33.0);
    assertEquals(ans.get("AUD"), 15.0);
    assertEquals(ans.get("RMB"), 1.0);
    assertEquals(ans.get("HKD"), 56.7);
    assertEquals(ans.get("EUR"), 0.98);
}

```

The test above tests a different function, readInRatesFile(), which is called directly after main in the promptUser() function. The function has the goal of loading all the rates from the file specified in the filepath and putting them into a static HashMap for use throughout the rest of the program. In order to test this function a few variations of the file were made, in which the layout and content of the file varies. In this test, the function is given a valid file and is expected to load the respective rates into the file without additional problems. In order to assert this a newFileName variable is defined pointing to the file to be used in this particular test case, and after calling the readInRatesFile() on the instance of App, 'test', the instance's hashmap is gathered through the HashMap ans = test.rates; line. After we have this we are

then able to call multiple `assertEquals` calls with the expected values making sure that the hashmap has in fact loaded the correct rates in. This test is very important as it tests the general functionality of reading from the specified file information that is vital in the running of the App. If this were to fail or have any issues it would mean that all other calculations could be wrong or not be calculated at all. The function also extends the amount of code coverage over the whole program.

@Test //Regular case

```
public void testReadInValidFile2(){
    String filename = System.getProperty("user.dir") + "/ValidInputTestTwo.txt";
    test.readInRatesFile(filename);
    HashMap ans = test.rates;

    assertEquals(ans.size(), 5);
    assertEquals(ans.get("USD"), 10009.0);
    assertEquals(ans.get("AUD"), 33.0);
    assertEquals(ans.get("RMB"), 6.0);
    assertEquals(ans.get("HKD"), 15.09);
    assertEquals(ans.get("EUR"), 0.56);
}
```

The test above also continues to test the `readInRatesFile()` function, although this time the filename points to another valid file that is found in the root of the groups gradle system, called `ValidInputTestTwo`. This file is different from the first in that the order is mixed up and the rates are larger. The same method as the first is then applied in order to assert that the values of the rates in the static hashmap are as expected when obtained from the file specified. This is an important test chosen to again make sure that there is no unexpected error that arises when calling the function on a function with a different format.

@Test //Regular case

```
public void testLowerCaseFile(){
    String filename = System.getProperty("user.dir") + "/TestLowerCaseFile.txt";
    test.readInRatesFile(filename);

    HashMap ans = test.rates;

    assertEquals(ans.size(), 5);
    assertEquals(ans.get("USD"), 1.5);
    assertEquals(ans.get("AUD"), 22.8);
    assertEquals(ans.get("RMB"), 4.84489);
    assertEquals(ans.get("HKD"), 22.22);
    assertEquals(ans.get("EUR"), 1.89);
}
```

The above test is very similar to the above two, with the values inside the file this time being in lower case, the hashmap is again compared to the expected value and the rates are all expected to be in there despite the lower case.

@Test //Edge case

```
//Tests when there is an invalid currency (e.g. "ABC") in the rates file
public void testInvalidFileCurrency(){
    PrintStream orig = System.out;
    String filename = System.getProperty("user.dir") + "/TestInvalidFileCurrency.txt";
    ByteArrayOutputStream out = new ByteArrayOutputStream();
```

```

System.setOut(new PrintStream(out));

test.readInRatesFile(filename);

String expected = "Line 4 is not a valid currency\nRates file missing rates \n";
System.setOut(orig);
assertEquals(expected, out.toString());
}

```

This time there is an invalid currency (e.g. "ABC") in the rates file, using the same method to test the programs standardoutput as the tests earlier, this makes sure that the appropriate error is given when attempting to read in a file. In this case the error message will be printed, as well as "Rates file missing rates", because there are not enough valid rates to fill the HashMap.

@Test //Edge case

```

public void testMissingFileCurrency(){
    PrintStream orig = System.out;
    String filename = System.getProperty("user.dir") + "/TestMissingFileCurrency.txt";
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setOut(new PrintStream(out));

    test.readInRatesFile(filename);

    String expected = "Rates file missing rates\n";
    System.setOut(orig);
    assertTrue(out.toString().equals(expected));
}

```

This time there are less than the minimum amount of valid currencies in the specified file, so this test makes sure that the relevant error message is printed to the standard output to let the user know.

@Test //Edge case

```

public void testNoFilePresent(){
    PrintStream orig = System.out;
    String filename = System.getProperty("user.dir") + "/NotAFileName.txt";
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setOut(new PrintStream(out));

    test.readInRatesFile(filename);

    String expected = "Error reading file\n";
    System.setOut(orig);
    assertEquals(expected, out.toString());
}

```

This time is slightly different in that the file path given does not actually have a file corresponding to it. This time when the test runs the output should be only Error Reading File, which is compared using the ByteArrayOutputStream way shown in previous test cases

@Test //edge case

```

public void testNotANumberFile(){
    PrintStream orig = System.out;
    String filename = System.getProperty("user.dir") + "/TestNotANumberFile.txt";

```

```

ByteArrayOutputStream out = new ByteArrayOutputStream();
System.setOut(new PrintStream(out));

test.readInRatesFile(filename);

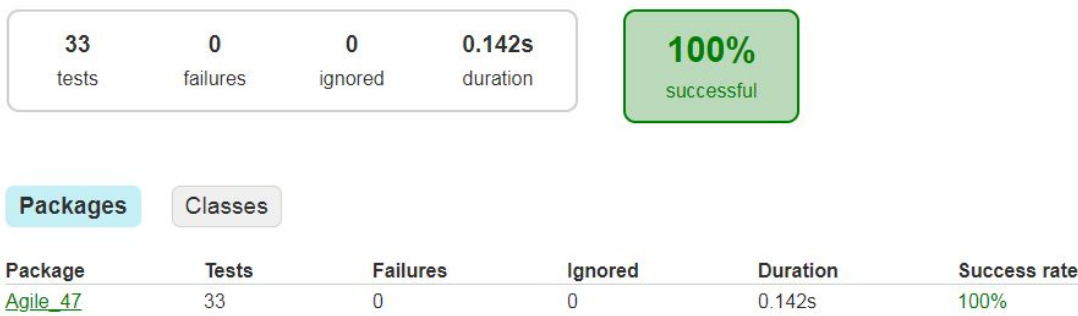
String expected = "Not a valid rate on line 3\n"+
"Rates file missing rates\n";
System.setOut(orig);
assertEquals(expected, out.toString());
}

```

In this final test, it again makes sure that the user is able to see that there is a rate in the file, that is not a number. This should be outputted to the user via the error messages: "Not a valid rate on line 3\n" and "Rates file missing rates\n". These are compared to the expected using the assertEquals() shown down the bottom of the function.

The results of these Junit tests after some minor tinkering with the code was all successful as we could see from the provided reports section on gradle. The image below shows the final version of the test cases all passing, meaning that they were able to meet the expected output that they were compared to in the assert functions.

Test Summary





Generated by [Gradle 5.6](#) at 20 Sep 2019, 03:24:54

This result was initially providing problems for the group with Darby having issues in testing the readInRatesFile() method, as the way in which the java os modules operated on gradle were different to the testing environment he was using. In order to fix this, Liz was able to help in suggesting to include a filename as a parameter for the function, allowing for the function to be tested without having to rename and edit the file, instead allowing other 'test' files to be initialized and used for a different test rather than having to continually try to reset and set the content of one rates.txt file.

























Once these tests were added to the source code of AppTest.java and the gradle was again built and tested, it could be seen that this had increased the code coverage and the image above was now highlighted green to indicate that it had been covered.

Agile_47

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Agile_47		97%		95%	3	38	7	157	1	15	0	2
Total	15 of 560	97%	2 of 46	95%	3	38	7	157	1	15	0	2

We can see from the report that all the methods within the source code, App.java, have been tested for errors.

App

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
main(String[])		0%		n/a	1	1	3	3	1	1
writeToFile(String, double, String)		95%		91%	1	7	3	33	0	1
promptUser(App)		94%		75%	1	3	1	11	0	1
currConvert()		100%		100%	0	4	0	36	0	1
readInRatesFile(String)		100%		100%	0	6	0	33	0	1
userUpdateRates(Scanner, String)		100%		100%	0	3	0	24	0	1
isValidCurrency(String)		100%		100%	0	6	0	4	0	1
calculateConversionRate(double, double)		100%		n/a	0	1	0	2	0	1
convertCurrency(double, double)		100%		n/a	0	1	0	2	0	1
static {...}		100%		n/a	0	1	0	1	0	1
gradlePrintToStandardOut()		100%		n/a	0	1	0	2	0	1
App()		100%		n/a	0	1	0	1	0	1
Total	15 of 543	97%	2 of 46	95%	3	35	7	152	1	12

These methods outlined in the above image, each correlate to a correlation of the application.

main(), this method as shown above was not tested but does not have any branches, this was not tested because all that was done was declaring an instance of the class that was then passed to the other methods for the functionalities. **writeToFile()** method was tested with a coverage of 95%, this is the method that is called when the user wishes to update rates in the apps additional text file, rates.txt. The tests were described earlier, but in essence it checked to make sure when the user wanted to update an exchange rate, providing it was valid, he could do so. **promptUser()** this method is called immediately after main and loading in the text file, essentially it asks the user what they'd like to do on the app and redirects the program to the necessary location. **currConvert()** this was the method that the user was passed to from prompt user if they indicated they wanted to convert a currency rather than change a rate.

readInRatesFile() this was the method that was called immediately from main before prompting to both ensure the rates were in the file, and were valid before putting the rates in a HashMap.

userUpdateRates() was the method the program was passed to if the user indicated they wanted to change rates in the rates.txt file. This would call the writeToFile method providing the users input was valid. **IsValidCurrency()** was a method used throughout different parts of the program to check that a user input was one of the accept codes like AUD USD etc.

CalculateConversionRate() was the mathematical function that took in the two rates given in the file and calculated the final rate for the convertCurrency() function **convertCurrency()** this was the function that found the final amount the person ended up with after giving the necessary input.

Each of the above methods were testing using Junit tests, with the help of the gradle, github for collaboration, and jacoco. Evidence for this and the explanations of why and how the group did this are all outlined above. Further source code can be seen at the end of the report.

6.The entire source code in text format

There were numerous files and pieces of code used throughout the assessment, however, as specified in the assignment brief, the source code for the application and the tests must be provided, thus the following code is all the code that has been taken from the *src* folder of the groups gradle repository. It contains both the source code for the tests and the application in the most recent commit on the online repository.

Source 1: App.java source code (main file of the application)

```
package Agile_47;

import java.util.*;
import java.io.*;

public class App{
    public static HashMap<String, Double> rates = new HashMap<String, Double>();

    //Check that the currency input is valid
    public static boolean isValidCurrency(String currency){
        if (currency.equals("AUD") || currency.equals("USD") || currency.equals("RMB") || currency.equals("HKD")
            || currency.equals("EUR")){
            return true;
        }
        return false;
    }

    //Read in the rates stored in the file
    public static void readInRatesFile(String filename){
        //String filename = System.getProperty("user.dir") + "/rates.txt";
        rates = new HashMap<String, Double>();
        File f = new File(filename);
        try{
            BufferedReader br = new BufferedReader(new FileReader(f), 100);
            String s;
            int lineNumber = 0;
            while ((s = br.readLine()) != null){
                lineNumber++;
                //Check the line is long enough
                if (s.length() < 5){
                    System.out.print("Line " + lineNumber + " has the incorrect format\n");
                    continue;
                }

                //Get the currency (3 letters)
                String currency = s.substring(0, 3).toUpperCase();
                if (isValidCurrency(currency)!=true){
                    System.out.print("Line " + lineNumber+" is not a valid currency\n");
                    continue;
                }

                //get the exchange rate
```

```

String rateString = s.substring(4, s.length());
double rate = -1;
try{
    rate = Double.parseDouble(rateString);
}
catch (Exception e){
    System.out.print("Not a valid rate on line " + lineNumber+"\n");
    continue;
}
if (rate <=0){
    System.out.print("Rate on line " + lineNumber + " is not positive\n");
    continue;
}

//Save the currency exchange rate to the dictionary
rates.put(currency, rate);
}
}
catch (Exception e){
    System.out.print("Error reading file\n");
    return;
}

//Check that all 5 rates have been read in correctly
if (rates.size() != 5){
    System.out.print("Rates file missing rates\n");
    return;
}
}

//Update the rates in the file
public static void writeToFile(String myCurrency, double myRate, String filename){
    //String filename =
"C:\\Data\\Documents\\Uni\\SOFT2412\\Project\\Agile_47\\src\\main\\java\\Agile_47\\rates.txt";
    //String filename = "../././././rates.txt";
    File f = new File(filename);
    ArrayList<String> lines = new ArrayList<String>();
    int lineToEdit = -1;
    try{
        BufferedReader br = new BufferedReader(new FileReader(f), 100);
        String s;
        int lineNumber=-1;

        //Find the line you want to edit
        while ((s = br.readLine()) != null){
            lineNumber++;
            lines.add(s);

            //Check if this is the line you want to edit
            if (s.length() < 5){
                continue;
            }
            String currency = s.substring(0, 3).toUpperCase();
            if (currency.equals(myCurrency)){
                lineToEdit=lineNumber;
            }
        }
    }
    catch (Exception e){
        System.out.print("Error reading file\n");
        return;
    }

    if (lineToEdit ==-1){
        System.out.print(myCurrency + " is not found in the file - cannot edit it\n");
        return;
    }
}

```

```

    }

    //Write the new rate to a file
    try{
        FileWriter fw=new FileWriter(filename);
        for (int i=0; i<lines.size(); i++){
            if (i==lineToEdit){
                //write new line
                String newLine = myCurrency + " " + Double.toString(myRate);
                fw.write(newLine+"\n");
            }
            else{
                //write old line
                fw.write(lines.get(i)+"\n");
            }
        }
        fw.close();
    }
    catch (Exception e){
        System.out.print("Issue writing to file\n");
    }
}

//Get the currencies the user wants to convert and perform the conversion
public static void currConvert(){
    String nth[] = {"1st", "2nd", "3rd", "4th", "5th"};
    Scanner sc = new Scanner(System.in);
    System.out.print("How many currencies do you want to convert?\n");
    String s = sc.nextLine();
    int nc = -1;
    try{
        nc = Integer.parseInt(s);
    }
    catch (Exception e){
        System.out.print("Not a number\n");
        return;
    }

    //Get the currencies to convert
    double resultInAUD = 0;
    for (int i=0; i<nc; i++){
        System.out.print("Enter the " + nth[i] + " amount (with a currency symbol e.g. 3.6USD)\n");
        s = sc.nextLine();
        //1. Get the currency symbol
        String sym = s.substring(s.length()-3, s.length()).toUpperCase();

        //2. Get the amount
        Double amount = -1.0;
        try{
            amount = Double.parseDouble(s.substring(0, s.length()-3));
        }
        catch (Exception e){
            System.out.print("Not a number\n");
            return;
        }

        //3. Make sure the amount and currency are valid
        if (isValidCurrency(sym)==false){
            System.out.print("Invalid currency\n");
            return;
        }

        //4. Make the conversion
        double rate = rates.get(sym);
        double res = convertCurrency(amount, calculateConversionRate(rate, 1));
        resultInAUD += res;
    }
}

```

```

    }

    //Get the currency to convert it to
    System.out.print("Which currency do you want the result displayed in?\n");
    String outCurrency = sc.nextLine().toUpperCase();
    if (isValidCurrency(outCurrency)==false){
        System.out.print("Invalid output currency\n");
        return;
    }

    //Convert from AUD to outCurrency
    double rate = rates.get(outCurrency);
    double convertedResult = convertCurrency(resultInAUD, calculateConversionRate(1, rate));
    System.out.printf("Your result is %.2f %s\n", convertedResult, outCurrency);
}
/**
 *
 * @param source_currency_rate: double containing the conversion rate of AUD to the source currency (e.g.
1AUD:0.69USD)
 * @param target_currency_rate: double containing rate of AUD to target currency (e.g. 1AUD:49INR)
 * @return conversion_rate: double containing conversion of 1 source currency unit in term of the target currency
unit (e.g. USD:INR)
 */
public static double calculateConversionRate(double source_currency_rate, double target_currency_rate) {
    double conversion_rate = (1/source_currency_rate)*target_currency_rate;
    return conversion_rate;
}

/**
 *
 * @param amount: double, amount of currency needed to convert
 * @param conversion_rate: double, conversion rate between source and target currency
 * @return converted_amount: amount in terms of the target currency
 */
public static double convertCurrency(double amount, double conversion_rate){
    double converted_amount = amount*conversion_rate;
    return converted_amount;
}

//Get the rates the user wants to update and update them
public void userUpdateRates(Scanner sc, String filename){
    String nth[] = {"1st", "2nd", "3rd", "4th", "5th"};
    // Scanner sc = new Scanner(System.in);
    System.out.print("How many rates do you want to edit?\n");
    String s = sc.nextLine();
    int numEdits = -1;
    try{
        numEdits = Integer.parseInt(s);
    }
    catch (NumberFormatException e){
        System.out.print("Not a number\n");
        return;
    }

    for (int i=0; i<numEdits; i++){
        System.out.print("What is the " + nth[i]+ " currency you want to edit?\n");
        String curr = sc.nextLine();

        //Check this is a valid currency
        if (isValidCurrency(curr)==false){
            System.out.print("Invalid currency\n");
            return;
        }

        System.out.print("What is the new rate for " + curr + "?\n");
        Double newRate = -1.0;
        try{

```

```

        newRate = Double.parseDouble(sc.nextLine());
    }
    catch (Exception e){
        System.out.print("Not a number\n");
        return;
    }

    //Write this new rate to the file
    writeToFile(curr, newRate, filename);
}

}

public void promptUser(App instance) {
    String filename = System.getProperty("user.dir") + "/rates.txt";
    // Read in the rates file, making sure it is valid
    readInRatesFile(filename);

    //The user inputs whether to convert currency or edit the rates
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter 1 for converting currency or 2 for editing rates\n");
    String type = sc.nextLine();

    if (type.equals("1")){
        currConvert();
    } else if (type.equals("2")){
        instance.userUpdateRates(sc, filename);
    } else {
        System.out.print("Invalid input\n");
    }
}

public static void main(String[] args){
    App instance = new App();
    instance.promptUser(instance);
}

// Easy way to test if Gradle is capturing System.out.print()
// Delete before final submission
public void gradlePrintToStandardOut() {
    System.out.print("Gradle is capturing System.out\n");
}
}

```

Source 2: source code for AppTest.java, the main testing file used in the assessment.

```

package Agile_47;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.io.*;
import java.util.*;

public class AppTest {
    private App test;

    @BeforeEach
    void setUp() {
        test = new App();
    }
}

```

```

    test.rates = new HashMap<String, Double>();
}

@AfterEach
void tearDown() {
    test = null;
}

//UNIT TESTS FOR isValidCurrency(String currency)
//Check that valid currencies return true for the function isValidCurrency
@Test
void testIsValidCurrency() {
    String currency = "AUD";
    assertTrue(test.isValidCurrency(currency));

    currency = "USD";
    assertTrue(test.isValidCurrency(currency));

    currency = "RMB";
    assertTrue(test.isValidCurrency(currency));

    currency = "HKD";
    assertTrue(test.isValidCurrency(currency));

    currency = "EUR";
    assertTrue(test.isValidCurrency(currency));
}

//Check that invalid currencies return false for the function isValidCurrency
@Test
void testNotValidCurrency() {
    String currency = "INR";
    assertFalse(test.isValidCurrency(currency));
}

//UNIT TESTS FOR FUNCTION writeToFile(String myCurrency, double myRate, String filename)
@Test
//Test writing a currency which should appear on the first line of the file
public void testFileWriteOneRateFirstLine() {
    String filename = System.getProperty("user.dir") + "/rates.txt";
    test.writeToFile("AUD", 22.8, filename);

    File f = new File(filename);
    String s = "";
    try {
        BufferedReader br = new BufferedReader(new FileReader(f), 100);
        s = br.readLine();
    } catch (Exception e) {}
    assertEquals("AUD 22.8", s);
}

@Test
//Test writing a currency which should appear on the final line of the file
public void testFileWriteOneRateLastLine() {
    String filename = System.getProperty("user.dir") + "/rates.txt";
    test.writeToFile("EUR", 1.89, filename);
}

```

```

File f = new File(filename);
String s = "";
try {
    BufferedReader br = new BufferedReader(new FileReader(f), 100);
    s = br.readLine();
    s = br.readLine();
    s = br.readLine();
    s = br.readLine();
    s = br.readLine();
} catch (Exception e) {}
assertEquals("EUR 1.89", s);
}

```

```

@Test
//Tests writing to the same currency twice
//The file should only contain the results of the second write as the second
//write should overwrite the first write
public void testWriteSameCurrencyTwice() {
    String filename = System.getProperty("user.dir") + "/rates.txt";
    test.writeToFile("HKD", 11.11, filename);
    test.writeToFile("HKD", 22.22, filename);

    File f = new File(filename);
    String s = "";
    try {
        BufferedReader br = new BufferedReader(new FileReader(f), 100);
        s = br.readLine();
        s = br.readLine();
        s = br.readLine();
        s = br.readLine();
    } catch (Exception e) {}
    assertEquals("HKD 22.22", s);
}

```

```

@Test
//Tests writing 3 different currencies to the file
//Should correctly write all 3 to the file
public void testFileWriteThreeRates() {
    String filename = System.getProperty("user.dir") + "/rates.txt";
    test.writeToFile("EUR", 16.2, filename);
    test.writeToFile("HKD", 3.5, filename);
    test.writeToFile("AUD", 109.788, filename);

    File f = new File(filename);
    String s1 = "";
    String s4 = "";
    String s5 = "";
    try {
        BufferedReader br = new BufferedReader(new FileReader(f), 100);
        s1 = br.readLine();
        br.readLine();
        br.readLine();
        s4 = br.readLine();
        s5 = br.readLine();
    } catch (Exception e) {}
    assertEquals("AUD 109.788", s1);
}

```



```

    assertEquals("HKD 3.5", s4);
    assertEquals("EUR 16.2", s5);
}

```

@Test

//Tests writing to a file which doesn't exist
 //Should print "error reading file" and return

```

public void testFileWriteNoFilePresent() {
    PrintStream orig = System.out;
    String filename = System.getProperty("user.dir") + "/notAFile.txt";
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setOut(new PrintStream(out));
    test.writeToFile("USD", 200.01, filename);
    String res = out.toString();
    assertEquals(res, "Error reading file\n");
    System.setOut(orig);
}

```

@Test

//Tests writing a currency which doesn't exist
 //Should print "currency not found" and return

```

public void testFileWriteInvalidCurrency() {
    PrintStream orig = System.out;
    String filename = System.getProperty("user.dir") + "/rates.txt";
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setOut(new PrintStream(out));
    test.writeToFile("ABC", 200.01, filename);
    String res = out.toString();
    assertEquals(res, "ABC is not found in the file - cannot edit it\n");
    System.setOut(orig);
}

```

//UNIT TESTS FOR FUNCTION userUpdateRates()

// These also call writeToFile() so it is integration testing with writeToFile()

@Test

// Tests that a valid user input will make the correct changes in rates.txt
 // This function will also call writeToFile() so it is integration testing with writeToFile()

```

public void testUserUpdateRatesValidInput() {
    PrintStream orig = System.out;
    ByteArrayInputStream in = new ByteArrayInputStream("1\nUSD\n1.5".getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setOut(new PrintStream(out));
    System.setIn(in);
    String filename = System.getProperty("user.dir") + "/rates.txt";
    Scanner sc = new Scanner(System.in);
    test.userUpdateRates(sc, filename);
}

```

File f = new File(filename);

String s = "";

try {

BufferedReader br = new BufferedReader(new FileReader(f), 100);

br.readLine();

s = br.readLine();

} catch (Exception e) {}

System.setOut(orig);

assertEquals("USD 1.5", s);

}

```

@Test
// This function will also call writeToFile() so it is integration testing with writeToFile()
// Tests if user does not enter a valid number for the amount of currencies they want to convert
// Should return "Not a number"
public void testUserUpdateRatesInvalidNumberInput() {
    PrintStream orig = System.out;
    //Enter "NotANumber" instead of an actual number of currencies you want to convert
    ByteArrayInputStream in = new ByteArrayInputStream("NotANumber\nUSD\n1.5".getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setIn( in );
    System.setOut(new PrintStream(out));
    String filename = System.getProperty("user.dir") + "/rates.txt";
    Scanner sc = new Scanner(System.in);
    test.userUpdateRates(sc, filename);
    System.setOut(orig);
    assertEquals("How many rates do you want to edit?\nNot a number\n", out.toString());
}

@Test
// This function will also call writeToFile() so it is integration testing with writeToFile()
// Tests if user does not enter a valid currency rate for the specific currency
// Should return "Not a number"
public void testUserUpdateRatesInvalidRateInput() {
    PrintStream orig = System.out;
    //Enter "NotANumber" instead of an actual number for the rate
    ByteArrayInputStream in = new ByteArrayInputStream("1\nUSD\nNotANumber".getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setIn( in );
    System.setOut(new PrintStream(out));
    String filename = System.getProperty("user.dir") + "/rates.txt";
    Scanner sc = new Scanner(System.in);
    test.userUpdateRates(sc, filename);
    System.setOut(orig);
    assertEquals("How many rates do you want to edit?\nWhat is the 1st currency you want to edit?\nWhat is the new rate for USD?\nNot a number\n", out.toString());
}

@Test
// This function will also call writeToFile() so it is integration testing with writeToFile()
// Tests if user does not enter a valid currency
// Should return "invalid currency"
public void testUserUpdateRatesInvalidCurrencyInput() {
    PrintStream orig = System.out;
    //Enter invalid currency "AAA" instead of a valid currency
    ByteArrayInputStream in = new ByteArrayInputStream("1\nAAA\n1.5".getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setIn( in );
    System.setOut(new PrintStream(out));
    String filename = System.getProperty("user.dir") + "/rates.txt";
    Scanner sc = new Scanner(System.in);
    test.userUpdateRates(sc, filename);
    System.setOut(orig);
    assertEquals("How many rates do you want to edit?\nWhat is the 1st currency you want to edit?\nInvalid currency\n", out.toString());
}

```

//INTEGRATION TESTING

@Test

//Integration testing for option 1- currency conversion

//This test tests whether the user can enter valid input and be given the correct conversion

```
public void testPromptUserCurrencyConversion() {
    PrintStream orig = System.out;
    ByteArrayInputStream in = new ByteArrayInputStream("1\n1\n5.0RMB\nUSD\n".getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setIn( in );
    System.setOut(new PrintStream(out));
    String filename = System.getProperty("user.dir") + "/ActualRates.txt";
    test.promptUser(test, filename);
    System.setOut(orig);
    String expected = "Enter 1 for converting currency or 2 for editing rates\n" +
        "How many currencies do you want to convert?\n" +
        "Enter the 1st amount (with a currency symbol e.g. 3.6USD)\n" +
        "Which currency do you want the result displayed in?\n" +
        "Your result is 0.70 USD\n";
    assertEquals(expected, out.toString());
}
```

@Test

//Integration testing for option 1- currency conversion, with 2 currencies

//This test tests whether the user can enter valid input and be given the correct conversion

```
public void testPromptUserCurrencyConversion2() {
    PrintStream orig = System.out;
    ByteArrayInputStream in = new ByteArrayInputStream("1\n2\n5.0RMB\n3.0AUD\nUSD\n".getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setIn( in );
    System.setOut(new PrintStream(out));
    String filename = System.getProperty("user.dir") + "/ActualRates.txt";
    test.promptUser(test, filename);
    System.setOut(orig);
    String expected = "Enter 1 for converting currency or 2 for editing rates\n" +
        "How many currencies do you want to convert?\n" +
        "Enter the 1st amount (with a currency symbol e.g. 3.6USD)\n" +
        "Enter the 2nd amount (with a currency symbol e.g. 3.6USD)\n" +
        "Which currency do you want the result displayed in?\n" +
        "Your result is 2.74 USD\n";
    assertEquals(expected, out.toString());
}
```

@Test

//Integration testing for option 1- currency conversion, with invalid input

//This test tests whether the user can enter valid input and be given the correct conversion

```
public void testPromptUserCurrencyConversionInvalid() {
    PrintStream orig = System.out;
    ByteArrayInputStream in = new ByteArrayInputStream("1\n1\n99 nzd\n".getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setIn( in );
    System.setOut(new PrintStream(out));
    String filename = System.getProperty("user.dir") + "/ActualRates.txt";
    test.promptUser(test, filename);
    System.setOut(orig);

    String expected = "Enter 1 for converting currency or 2 for editing rates\n" +
```

```

        "How many currencies do you want to convert?\n" +
        "Enter the 1st amount (with a currency symbol e.g. 3.6USD)\n" +
        "Invalid currency\n";
    assertEquals(expected, out.toString());
}

@Test
//Integration testing for option 2- updating rates
//This test tests whether the user can enter valid input and have the rates file
//updated accordingly
public void testPromptUserToChangeRates() {
    PrintStream orig = System.out;
    ByteArrayInputStream in = new ByteArrayInputStream("2\n1\nUSD\n1.5\n".getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setIn( in );
    System.setOut(new PrintStream(out));
    String filename = System.getProperty("user.dir") + "/rates.txt";
    test.promptUser(test, filename);
    System.setOut(orig);
    assertEquals("Enter 1 for converting currency or 2 for editing rates\nHow many rates do you want to edit?\nWhat is
the 1st currency you want to edit?\nWhat is the new rate for USD?\n", out.toString());
}

@Test
//Integration testing for option 2- updating rates
//This tests whether the program will display "Invalid input" when the user gives invalid input for writing to a file
public void testPromptUserInvalidSelection() {
    PrintStream orig = System.out;
    ByteArrayInputStream in = new ByteArrayInputStream("INVALID\n1\nUSD\n1.5\n".getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setIn( in );
    System.setOut(new PrintStream(out));
    String filename = System.getProperty("user.dir") + "/rates.txt";
    test.promptUser(test, filename);
    System.setOut(orig);
    assertEquals("Enter 1 for converting currency or 2 for editing rates\nInvalid input\n", out.toString());
    //System.out.print(out.toString());
}

// UNIT TESTING for currConvert() function
// Tests whether the program has the correct output when given one valid currency to convert
@Test
public void testcurrConvertValid1(){
    //Input: convert 5RMB to USD
    ByteArrayInputStream in = new ByteArrayInputStream("1\n5.0RMB\nUSD\n".getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setIn(in);
    System.setOut(new PrintStream(out));
    Scanner sc = new Scanner(System.in);
    //Set up the rates
    test.rates = new HashMap<String, Double>();
    test.rates.put("USD", 1.5);
    test.rates.put("AUD", 22.8);
    test.rates.put("RMB", 4.84489);
    test.rates.put("EUR", 1.89);
    test.rates.put("HKD", 22.22);
}

```

```

test.currConvert(sc);
String expected = "How many currencies do you want to convert?\n" +
    "Enter the 1st amount (with a currency symbol e.g. 3.6USD)\n" +
    "Which currency do you want the result displayed in?\n" +
    "Your result is 1.55 USD\n";
assertEquals(expected, out.toString());
}

//Test converting three currencies
@Test
void testcurrConvertValid2() {
    //Input: 99USD + 100AUD + 105EUR into USD
    PrintStream orig = System.out;
    String input = "3\n" +
        "99USD\n100AUD\n105EUR\n" +
        "USD\n";
    ByteArrayInputStream in = new ByteArrayInputStream(input.getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setIn(in);
    System.setOut(new PrintStream(out));
    Scanner sc = new Scanner(System.in);

    //Set up the rates
    test.rates = new HashMap<String, Double>();
    test.rates.put("USD", 0.67922473);
    test.rates.put("AUD", 1.0);
    test.rates.put("RMB", 4.81940382);
    test.rates.put("EUR", 0.61379632);
    test.rates.put("HKD", 5.31817394 );

    test.currConvert(sc);
    String expected = "How many currencies do you want to convert?\n" +
        "Enter the 1st amount (with a currency symbol e.g. 3.6USD)\n" +
        "Enter the 2nd amount (with a currency symbol e.g. 3.6USD)\n" +
        "Enter the 3rd amount (with a currency symbol e.g. 3.6USD)\n" +
        "Which currency do you want the result displayed in?\n" +
        "Your result is 283.12 USD\n";
    assertEquals(expected, out.toString());
}

//Tests whether the program displays "Not a number" when it is given
//a non-integer number of currencies to convert e.g. 0.1
@Test
public void testcurrConvertInvalidNum() {
    String input = "0.1\n";
    ByteArrayInputStream in = new ByteArrayInputStream(input.getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setIn(in);
    System.setOut(new PrintStream(out));
    Scanner sc = new Scanner(System.in);
    test.currConvert(sc);
    String expected = "How many currencies do you want to convert?\n" +
        "Not a number\n";
    assertEquals(expected, out.toString());
}

```

//Tests whether the correct conversion output is displayed when the currencies
//are entered in lower case characters with a space

@Test

```
public void testcurrConvertLowerCur() {
    PrintStream orig = System.out;
    //Input= 99 usd into eur
    String input = "1\n" +
        "99 usd\neur\n";
    ByteArrayInputStream in = new ByteArrayInputStream(input.getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setIn(in);
    System.setOut(new PrintStream(out));
    Scanner sc = new Scanner(System.in);

    //Set up the rates
    test.rates = new HashMap<String, Double>();
    test.rates.put("USD", 0.67922473);
    test.rates.put("AUD", 1.0);
    test.rates.put("RMB", 4.81940382);
    test.rates.put("EUR", 0.61379632);
    test.rates.put("HKD", 5.31817394 );

    test.currConvert(sc);
    String expected = "How many currencies do you want to convert?\n" +
        "Enter the 1st amount (with a currency symbol e.g. 3.6USD)\n" +
        "Which currency do you want the result displayed in?\n" +
        "Your result is 89.46 EUR\n";

    System.setOut(orig);
    assertEquals(expected, out.toString());
}
```

//Tests whether the program displays "Invalid currency" when a non-supported
//currency is input e.g. 99NZD

@Test

```
public void testcurrConvertInvalidCur() {
    String input = "1\n" +
        "99 nzd\n";
    ByteArrayInputStream in = new ByteArrayInputStream(input.getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setIn(in);
    System.setOut(new PrintStream(out));
    Scanner sc = new Scanner(System.in);
    test.currConvert(sc);
    String expected = "How many currencies do you want to convert?\n" +
        "Enter the 1st amount (with a currency symbol e.g. 3.6USD)\n" +
        "Invalid currency\n";
    assertEquals(expected, out.toString());
}
```

//Tests whether "Not a number" is printed when an input is given which is
//not a number e.g. "fiveaud" instead of "5AUD"

@Test

```
public void testcurrConvertInvalidAmt() {
    String input = "1\n" +
        "fiveaud\n";
```

```

    ByteArrayInputStream in = new ByteArrayInputStream(input.getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setIn(in);
    System.setOut(new PrintStream(out));
    Scanner sc = new Scanner(System.in);

    //Set up the rates
    test.rates = new HashMap<String, Double>();
    test.rates.put("USD", 0.67922473);
    test.rates.put("AUD", 1.0);
    test.rates.put("RMB", 4.81940382);
    test.rates.put("EUR", 0.61379632);
    test.rates.put("HKD", 5.31817394 );

    test.currConvert(sc);
    String expected = "How many currencies do you want to convert?\n" +
        "Enter the 1st amount (with a currency symbol e.g. 3.6USD)\n" +
        "Not a number\n";
    assertEquals(expected, out.toString());
}

//Tests whether "Invalid output currency" is displayed when an invalid
//output currency is given i.e. "zasxusd"
@Test
public void testcurrConvertInvalidOut() {
    String input = "1\n" +
        "99 aud\nzasxusd\n";
    ByteArrayInputStream in = new ByteArrayInputStream(input.getBytes());
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setIn(in);
    System.setOut(new PrintStream(out));
    Scanner sc = new Scanner(System.in);

    //Set up the rates
    test.rates = new HashMap<String, Double>();
    test.rates.put("USD", 0.67922473);
    test.rates.put("AUD", 1.0);
    test.rates.put("RMB", 4.81940382);
    test.rates.put("EUR", 0.61379632);
    test.rates.put("HKD", 5.31817394 );

    test.currConvert(sc);
    String expected = "How many currencies do you want to convert?\n" +
        "Enter the 1st amount (with a currency symbol e.g. 3.6USD)\n" +
        "Which currency do you want the result displayed in?\n" +
        "Invalid output currency\n";
    assertEquals(expected, out.toString());
}

@Test
//Tests whether the correct conversion rate is returned
void testcalculateConversionRate() {
    double source_rate = 0.68747;
    double target_rate = 1.07654;
    double expected = 1.566;
    double actual = test.calculateConversionRate(source_rate, target_rate);
}

```

```
    assertEquals(expected, actual, 0.001);
}
```

@Test

//Tests whether the returned amount is correct when given a conversion rate and amount to convert

```
void testconvertCurrency() {
    double amount = 50;
    double conversion_rate = 1.566;
    double expected = 78.3;
    double actual = test.convertCurrency(amount, conversion_rate);
```

```
    assertEquals(expected, actual, 0.01);
}
```

//UNIT TESTS FOR readInRatesFile

@Test

//Tests whether a valid file is read in correctly and stored in the HashMap rates

```
public void testReadInValidFile1(){
    String filename = System.getProperty("user.dir") + "/ValidInputTestOne.txt";
    test.readInRatesFile(filename);
```

//Check if it produces the correct hashmap

HashMap ans = test.rates;

```
    assertEquals(ans.size(), 5);
    assertEquals(ans.get("USD"), 33.0);
    assertEquals(ans.get("AUD"), 15.0);
    assertEquals(ans.get("RMB"), 1.0);
    assertEquals(ans.get("HKD"), 56.7);
    assertEquals(ans.get("EUR"), 0.98);
}
```

@Test

//Tests whether another valid file is read in correctly and stored in the HashMap rates

```
public void testReadInValidFile2(){
    String filename = System.getProperty("user.dir") + "/ValidInputTestTwo.txt";
    test.readInRatesFile(filename);
```

//Check if it produces the correct hashmap

HashMap ans = test.rates;

```
    assertEquals(ans.size(), 5);
    assertEquals(ans.get("USD"), 10009.0);
    assertEquals(ans.get("AUD"), 33.0);
    assertEquals(ans.get("RMB"), 6.0);
    assertEquals(ans.get("HKD"), 15.09);
    assertEquals(ans.get("EUR"), 0.56);
```

```
}
```

@Test

//Tests whether the file is read in correctly when given in a mixture of lower and upper case

```
public void testLowerCaseFile(){
    String filename = System.getProperty("user.dir") + "/TestLowerCaseFile.txt";
    test.readInRatesFile(filename);
```



```

//Check if it produces the correct hashmap
HashMap ans = test.rates;

assertEquals(ans.size(), 5);
assertEquals(ans.get("USD"), 1.5);
assertEquals(ans.get("AUD"), 22.8);
assertEquals(ans.get("RMB"), 4.84489);
assertEquals(ans.get("HKD"), 22.22);
assertEquals(ans.get("EUR"), 1.89);

}

@Test
//Tests when there is an invalid currency (e.g. "ABC") in the rates file
//Should print "Line x not a valid currency" and
// "rates file missing rates" (as the expected number of valid currencies is 5)
public void testInvalidFileCurrency(){
    PrintStream orig = System.out;
    String filename = System.getProperty("user.dir") + "/TestInvalidFileCurrency.txt";
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setOut(new PrintStream(out));

    test.readInRatesFile(filename);

    String expected = "Line 4 is not a valid currency\n" +
        "Rates file missing rates\n";
    System.setOut(orig);
    assertEquals(expected, out.toString());
}

@Test
//Tests when there are less than 5 currencies in the rates file
//Expected output is "Rates file missing rates"
public void testMissingFileCurrency(){
    PrintStream orig = System.out;
    String filename = System.getProperty("user.dir") + "/TestMissingFileCurrency.txt";
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setOut(new PrintStream(out));

    test.readInRatesFile(filename);

    String expected = "Rates file missing rates\n";
    System.setOut(orig);
    assertEquals(expected, out.toString());
}

@Test
//Tests when the file given to read rates from does not exist
//Expected output "Error reading file"
public void testNoFilePresent(){
    PrintStream orig = System.out;
    String filename = System.getProperty("user.dir") + "/NotAFileName.txt";
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setOut(new PrintStream(out));

    test.readInRatesFile(filename);

```

```

    String expected = "Error reading file\n";
    System.setOut(orig);
    assertEquals(expected, out.toString());
}

@Test
//Test for an invalid rate (e.g. something that is not a number like "abc")
//Input: file with "abc" as one of the rates
//Expected output: "Not a valid rate on line x" and "Rates file missing rates"
public void testNotANumberFile(){
    PrintStream orig = System.out;
    String filename = System.getProperty("user.dir") + "/TestNotANumberFile.txt";
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setOut(new PrintStream(out));

    test.readInRatesFile(filename);

    String expected = "Not a valid rate on line 3\n"+
        "Rates file missing rates\n";
    System.setOut(orig);
    assertEquals(expected, out.toString());
}

@Test
public void negativeRateFile(){
    PrintStream orig = System.out;
    String filename = System.getProperty("user.dir") + "/NegativeRateFile.txt";
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setOut(new PrintStream(out));
    test.readInRatesFile(filename);
    String expected = "Rate on line 1 is not positive\n"+"Rates file missing rates\n";
    System.setOut(orig);
    assertEquals(expected, out.toString());
}

@Test
public void smallLineFile(){
    PrintStream orig = System.out;
    String filename = System.getProperty("user.dir") + "/SmallLineFile.txt";
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    System.setOut(new PrintStream(out));
    test.readInRatesFile(filename);
    String expected = "Line 1 has the incorrect format\n"+"Rates file missing rates\n";
    System.setOut(orig);
    assertEquals(expected, out.toString());
}
}

```

Source 3: rates.txt file, hold the initial values for our currency exchanges

AUD 1
USD 1.5

RMB 4.84489
HKD 5.34745
EUR 0.620881

Source 4: negativeRateFile.txt, used in a Test case

EUR -0.56
USD 10009
HKD 15.09
RMB 6.0
AUD 33.0

Source 5: SmallLineFile.txt, used in a Test case

EUR
USD 10009
HKD 15.09
RMB 6.0
AUD 33.0

Source 6: TestInvalidFileCurrency.txt, used in a Test case

AUD 22.8
USD 1.5
RMB 4.84489
ABC 22.22
EUR 1.89

Source 7: TestLowerCaseFile.txt, used in a Test case

auD 22.8
USD 1.5
rmb 4.84489
HKD 22.22
eur 1.89

Source 8: TestMissingFileCurrency.txt, used in a Test case

AUD 22.8
USD 1.5
RMB 4.84489
EUR 1.89

Source 9: TestNotANumberFile.txt, used in a Test case

AUD 22.8
USD 1.5
HKD abcd
RMB 4.84489
EUR 1.89

Source 10: ValidInputTestOne.txt, used in a Test case

USD 33
HKD 56.7
EUR 0.98
AUD 15

RMB 1

Source 11: ValidInputTestTwo.txt, used in a Test Case

EUR 0.56

USD 10009

HKD 15.09

RMB 6.0

AUD 33.0