

Code coverage report for the final commit, including JUnit results for the commits. All JUnit test cases that are written must be documented and explained (i.e. why the unit test was chosen and what does it test (regular input, edge cases etc). Document any tests that may have failed. You will also need to explain what the test coverage report is displaying

- JUnit must be integrated with Gradle
- Jacoco must be integrated with Gradle to get code coverage
- JUnit test files with correct test cases to test the program
- Explanation about the test cases including why they were chosen
- Brief explanation about the results/output obtained for JUnit tests
- Explanation about the results/output from coverage test, all functionalities tested
- The above must be su

JUnit Testing

What is Junit testing, how did the team do it and why did they do it.

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, using both unit tests and integration tests written with 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 below.

```
// }  
  
buildscript {  
    repositories {  
        mavenCentral()  
    }  
    dependencies {  
        classpath 'org.junit.platform:junit-platform-gradle-plugin:1.1.0'  
    }  
}  
apply plugin: 'java'  
apply plugin: 'application'  
apply plugin: 'jacoco'  
mainClassName = 'Agile_47.App'  
  
repositories {  
    mavenCentral()  
}  
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'  
}  
test {  
    useJUnitPlatform()  
    //testLogging.showStandardStreams = true  
    test.finalizedBy jacocoTestReport  
  
    testLogging {  
        showStandardStreams = true  
        events "passed", "skipped", "failed", "standardOut", "standardError"  
    }  
}
```

Adding each of these lines of code to our project, we were then able to effectively integrate out testing with Junit. After successfully integrating, each member of the team was then provided with access to the Junt tests and was free to write relevant tests for assigned sections of code, being, Liz - writeToFile, Ben - userUpdateRates, 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 was to ensure that the tests were 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. Although examples will be run through in detail below, the following screenshot shows the results of a merged “AppTest” class, containing more than 30 tests:

```
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;
    }

    //positive test for valid currency
    @Test
    void testIsValidCurrency() {
        String currency = "AUD";
        assertTrue(test.isValidCurrency(currency));

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

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

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.

```
Command Prompt
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 can be seen in the image above, the test cases written that produced their expected output were shown with a green “passed” whilst the test cases that did not were shown with a red “failed” next to it. 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.

Test Summary

31	11	0	0.167s	64% successful
tests	failures	ignored	duration	

Failed tests

Packages

Classes



[AppTest](#) [testInvalidFileCurrency\(\)](#)
[AppTest](#) [testMissingFileCurrency\(\)](#)
[AppTest](#) [testNoFilePresent\(\)](#)
[AppTest](#) [testNotANumberFile\(\)](#)
[AppTest](#) [testcurrConvertInvalidAmt\(\)](#)
[AppTest](#) [testcurrConvertInvalidCur\(\)](#)
[AppTest](#) [testcurrConvertInvalidNum\(\)](#)
[AppTest](#) [testcurrConvertInvalidOut\(\)](#)
[AppTest](#) [testcurrConvertLowerCur\(\)](#)
[AppTest](#) [testcurrConvertValid1\(\)](#)
[AppTest](#) [testcurrConvertValid2\(\)](#)

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".

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 a failing (indicated by the red), and the majority of branch coverage for the team's code is high, 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.                }

```

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
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
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
//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 user's 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
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 `assertEquals` 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
//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

```

//Tests writing 3 different currencies to a 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);
}

```

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 testcases. 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

```

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
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
// 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

```

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

```

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
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
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
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
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
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
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
public void testcurrConvertLowerCur() {
    PrintStream orig = System.out;
```

```

String input = "1\n" +
    "99 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" +
    "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
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
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
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
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
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
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
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
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
//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

```
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

```
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.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

33	0	0	0.142s	100%
tests	failures	ignored	duration	successful

Packages		Classes			
Package	Tests	Failures	Ignored	Duration	Success rate
Agile_47	33	0	0	0.142s	100%

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 this hurdle was overcome, the second hurdle was that after all of the respective tests had been merged into the one `AppTest` class, some group members when running 'gradle clean build' and 'gradle clean test' passed all of the test cases that had been built whilst others were having issues such as the one seen below.



testInvalidFileCurrency()

```
org.opentest4j.AssertionFailedError: expected: <Line 4 is not a valid currency
Rates file missing rates
> but was: <Line 4 is not a valid currency
Rates file missing rates
>
    at org.junit.jupiter.api.AssertionUtils.fail(AssertionUtils.java:52)
    at org.junit.jupiter.api.AssertEquals.failNotEqual(AssertEquals.java:197)
    at org.junit.jupiter.api.AssertEquals.assertEquals(AssertEquals.java:186)
    at org.junit.jupiter.api.AssertEquals.assertEquals(AssertEquals.java:181)
    at org.junit.jupiter.api.Assertions.assertEquals(Assertions.java:486)
    at Agile_47.AppTest.testInvalidFileCurrency(AppTest.java:528)
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.base/java.lang.reflect.Method.invoke(Method.java:564)
    at org.junit.platform.commons.util.ReflectionUtils.invokeMethod(ReflectionUtils.java:436)
    at org.junit.jupiter.engine.execution.ExecutableInvoker.invoke(ExecutableInvoker.java:115)
    at org.junit.jupiter.engine.descriptor.TestMethodTestDescriptor.lambda$invokeTestMethod$6(TestMethodTestDescriptor.java:170)
    at org.junit.jupiter.engine.execution.ThrowableCollector.execute(ThrowableCollector.java:40)
    at org.junit.jupiter.engine.descriptor.TestMethodTestDescriptor.invokeTestMethod(TestMethodTestDescriptor.java:166)
    at org.junit.jupiter.engine.descriptor.TestMethodTestDescriptor.execute(TestMethodTestDescriptor.java:113)
    at org.junit.jupiter.engine.descriptor.TestMethodTestDescriptor.execute(TestMethodTestDescriptor.java:58)
    at org.junit.platform.engine.support.hierarchical.HierarchicalTestExecutor$NodeExecutor.lambda$executeRecursively$3(HierarchicalTestExecutor.java:108)
    at org.junit.platform.engine.support.hierarchical.SingleTestExecutor.executeSafely(SingleTestExecutor.java:66)
    at org.junit.platform.engine.support.hierarchical.HierarchicalTestExecutor$NodeExecutor.executeRecursively(HierarchicalTestExecutor.java:108)
    at org.junit.platform.engine.support.hierarchical.HierarchicalTestExecutor$NodeExecutor.execute(HierarchicalTestExecutor.java:79)
    at org.junit.platform.engine.support.hierarchical.HierarchicalTestExecutor.lambda$executeRecursively$2(HierarchicalTestExecutor.java:183)
    at java.base/java.util.stream.ForEachOps$ForEachOp$OfRef.accept(ForEachOps.java:183)
    at java.base/java.util.stream.ReferencePipeline$2$1.accept(ReferencePipeline.java:177)
    at java.base/java.util.Iterator.forEachRemaining(Iterator.java:133)
    at java.base/java.util.Spliterators$IteratorSpliterator.forEachRemaining(Spliterators.java:1801)
    at java.base/java.util.stream.AbstractPipeline.copyInto(AbstractPipeline.java:484)
    at java.base/java.util.stream.AbstractPipeline.wrapAndCopyInto(AbstractPipeline.java:474)
    at java.base/java.util.stream.ForEachOps$ForEachOp.evaluateSequential(ForEachOps.java:150)
    at java.base/java.util.stream.ForEachOps$ForEachOp$OfRef.evaluateSequential(ForEachOps.java:173)
```

With the report stating that the test had failed despite the test having the same output as expected. This was eventually fixed by changing all of the ‘System.out.println’ lines to ‘System.out.print(...\n)’ which resolved the problem. With these tests producing the passed, output shown earlier, this meant that the following tests:

```
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() PASSED
Agile_47.AppTest > testGradlePrintToStandardOut() STANDARD_OUT
    Gradle is capturing System.out
Agile_47.AppTest > testGradlePrintToStandardOut() PASSED
Agile_47.AppTest > smallLineFile() PASSED
Agile_47.AppTest > testconvertCurrency() PASSED
Agile_47.AppTest > testcurrConvertLowerCur() PASSED
Agile_47.AppTest > testNotANumberFile() PASSED
Agile_47.AppTest > testUserUpdateRatesInvalidRateInput() PASSED
Agile_47.AppTest > testFileWriteOneRateFirstLine() PASSED
Agile_47.AppTest > testcalculateConversionRate() PASSED
Agile_47.AppTest > testNoFilePresent() PASSED
Agile_47.AppTest > testReadInValidFile1() PASSED
Agile_47.AppTest > testReadInValidFile2() PASSED
Agile_47.AppTest > testLowerCaseFile() PASSED
Agile_47.AppTest > testcurrConvertValid1() PASSED
Agile_47.AppTest > testcurrConvertValid2() PASSED
Agile_47.AppTest > testMissingFileCurrency() PASSED
Agile_47.AppTest > testcurrConvertInvalidAmt() PASSED
Agile_47.AppTest > testcurrConvertInvalidCur() PASSED
Agile_47.AppTest > testcurrConvertInvalidNum() PASSED
Agile_47.AppTest > testcurrConvertInvalidOut() PASSED
Agile_47.AppTest > testPromptUserToChangeRates() PASSED
Agile_47.AppTest > testUserUpdateRatesValidInput() PASSED
Agile_47.AppTest > testUserUpdateRatesInvalidNumberInput() PASSED
```

Had successfully tested for all the output that was explained in the description of each test. Once all of the group members were able to pass the test cases the focus moved onto obtaining a sufficient percentage of code coverage. In side the gradle file structure accessing the jacoco report could be done in the folder:...\Agile_47\build\reports\jacoco\test\html\index.html, which showed the following result initially;

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

This showed that from the tests that the group had written they had managed to cover 95% of all the code and 91% of all branches (code scopes). In order to improve this clicking through the links we were able to access the jacoco highlighted code, as shown earlier, however two scopes caught the eye that looked particularly easy to cover:

```

//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;
            }
        }
    }
}
//Use the currency exchange rate to the destination

```

These scopes in red looked fairly simple to write a testcase to cover for, seeing this was the function allocated to Darby, he was able to quickly create 2 additional test cases:

@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());
}

```

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

Agile_47

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Agile_47	<div><div></div></div>	97%	<div><div></div></div>	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

```

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;
        }
    }
}

```

These then showed a 2% increase from 95% coverage to 97%, further increasing the likelihood that we would find any potential hidden bugs. 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.