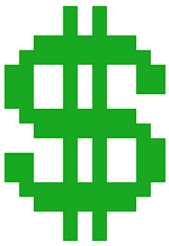


Currency Converter

SOFT2412: Agile Software Development Practices

Project 1 Technical Report

Contents

1 Executive summary

Currency Converter is a Windows XP-esk currency exchange coded in 100% Java. The project was developed for the unit SOFT2412: Agile Software Development Practices. This unit deals with the methods, principles and practices used by software developers to work and develop projects in Agile team environments. The *Currency Converter* project, created for the unit's first major work, is a reflection of our group's implementation of such practices towards a real software development task.

Though the final application itself is important, and insight on the functionality will be delivered where due, this report primarily aims to highlight the 'how' and 'why' behind the various technical tools used in the project such as automated testing, build automation, version control, continuous integration, and continuous deployment. Finally, this report will detail the social outcomes from this major work, including Agile principles followed, group communication and collaboration, individual contributions, and areas of improvement for future projects.

2 Project requirements

The project requirements entailed the creation of a manually-administered currency exchange software which allows users to view current and historical exchange rates as well as convert money between various currencies. The requirements of the application are relatively simple and can be divided into 4 use cases: three for basic users, and an additional use case only accessible to admins. These use cases have been transcribed into the use case diagram in Figure 1 and are elaborated on below.

The primary use case of the application is of course, to convert money from one currency to another. Users can also view historical exchange data: after selecting two currencies and a start and end date, the system must print a table of all exchange rates with these dates with the date and time they were added, alongside a 5-number summary of this selection. Finally, users can check the current exchange rates between "popular" currencies, a demarcation set by the admin role. If the exchange rate has increased since the last addition, it must appear with an (I) symbol next to it; alternatively, if the rate has since decreased, it is concatenated with a (D) symbol.

Besides setting the 4 "popular" currencies, the admin is in charge of maintaining and inputting new exchange rates and currencies into the database (hence why the exchange is deemed "manually-administered"). **There is no live connection between the database and a real-world exchange source** such as Xe; all rates and currencies are manually added.

Upon first load of the application, the database is programmed to initialise with 6 preset currencies: AUD, USD, NPR, INR, GBP, and SGD. These 6 currencies are loaded in alongside preset exchange rates between each of them [1] which are static and were sourced from `www.Xe.com` on the 20th of September, 2022. The admin's final privilege is to be able to reset the database to these original 6 currencies (its initial state).

The project's technical requirements constrained our development to a Java-based application with either a graphical or text-based UI. No restrictions were set on the data storage model used for the software, however all exchange data was required to persist over time.

[1] A total of $n(n-1)$ exchange rates where n is the number of currencies. In the initial state ($n = 6$) the database therefore has 30 exchange rates.

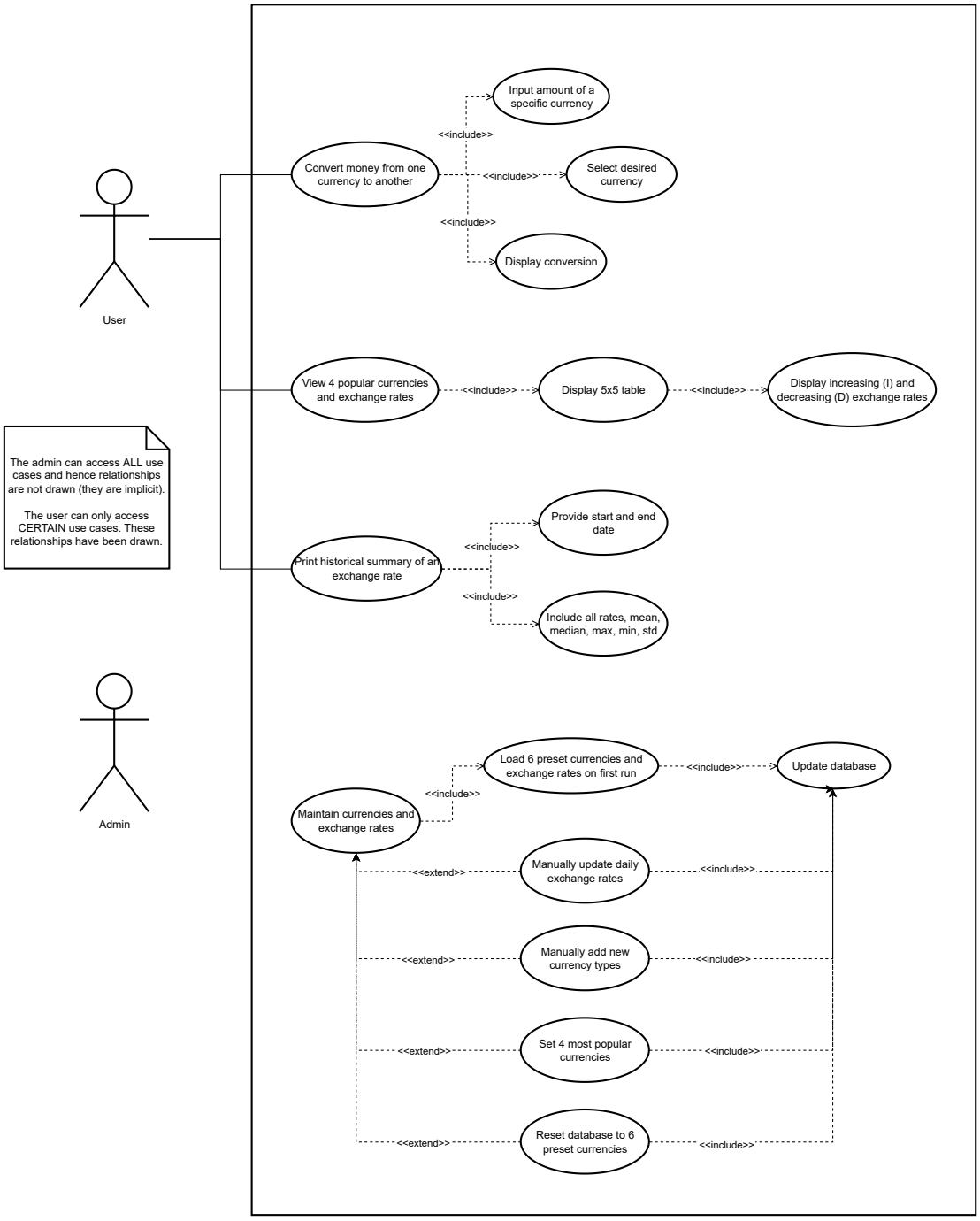


Figure 1: Use case diagram depicting the main project requirements

3 Application development: *Currency Converter*

Currency Converter is programmed in Java 17 with a GUI implemented using the Java Swing library. The data storage model used for the software is a local SQLite database. Thus, naturally, our code base structure was split up into frontend and backend files. This structure can be seen in Table 1 and is further elaborated on below.

Table 1: Codebase class distribution

Class	Description
Frontend	
AdminPortal	Screen to display admin-only actions
Converter	Satisfies the primary use case of converting between currencies
CurrencyExchange	The `JInternalFrame` which opens up in the `Desktop`
Desktop	Class which uses `JDesktopPane` to mimic a Windows XP operating system
DisplayPopular	Displays a table of 4 popular currencies and the exchange rates between them
History	Display a historical table of a specific exchange rate's value
SelectPopular	Admin-only panel to select 4 popular currencies
UpdateExchange	Admin-only panel to update an exchange rate
WelcomeScreen	The currency exchange's home page
Backend	
Admin	Methods used to fulfil admin use-cases
App	Class used to open a *Currency Converter* window
BasicUser	Methods used to fulfil user use-cases
CurrManager	Stores all database functionality

3.1 Frontend

Java Swing is a library built upon the now legacy Java AWT library, Java's first ever framework dedicated to graphical user interfaces. Swing is now gradually becoming defunct due to a lack of development and old UI standards, becoming replaced by the more modern Java FX. However, inspired by the old Look and Feel of Java Swing, as well as games such as Emily is Away, the team decided to base the GUI on an old Windows XP desktop for this project. The user would not immediately be presented with the currency exchange program, but rather an 8-bit rendition of the nostalgic *Bliss* wallpaper with a functioning menu bar at the bottom (all with official Windows XP icons of course). The user would then open up the exchange from the Start menu.

Implementing this in Java Swing requires two main classes: `Desktop` which subclasses `JDesktopPane`

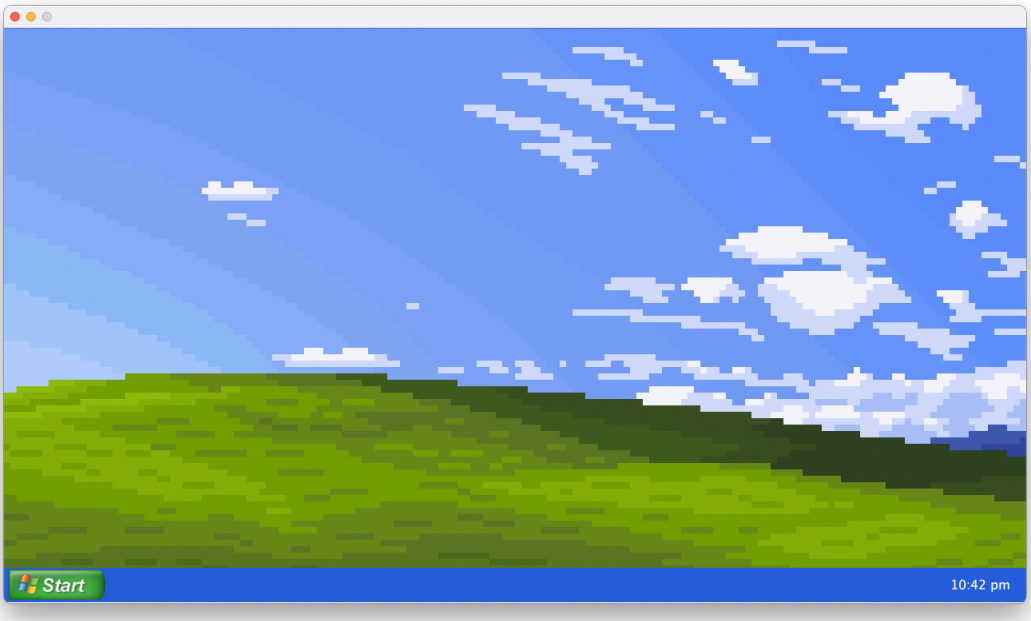


Figure 2: The noughties-inspired desktop of our *Currency Converter* application

and `CurrencyExchange` which subclasses `JInternalFrame`. The `Desktop` class is responsible for creating the actual Windows XP-esk application window which runs inside the users operating system. The `CurrencyExchange` sits *inside* the `Desktop` class and represents the window that pops up when one clicks on 'Currency Exchange' in the Start menu.

All other frontend classes in the codebase are different screens available to the user *inside* the currency exchange application, as Java Swing requires each pane displayed in the application to be its own class. These classes simply subclass `JFrame`, and are always placed inside the `JInternalFrame` panel created in the `CurrencyExchange` class.

The frontend works in an almost primitive way: each time a user clicks a button to navigate to a new screen, a new object of that screen is created. The panel which was previously visible is removed from the `CurrencyExchange` panel and the new panel is added. This is why all classes inside the currency exchange application have a constructor which contains a `CurrencyExchange` object.

3.2 Backend

The back end is quite well integrated with the Front end. So, it can be at times difficult to differentiate between the two in the code. The main two classes in the back end program is Admin and BasicUSer. The basic user has access to three basic functionalities of the program. A basic user can convert currencies, view the popular four currencies, and check a particular

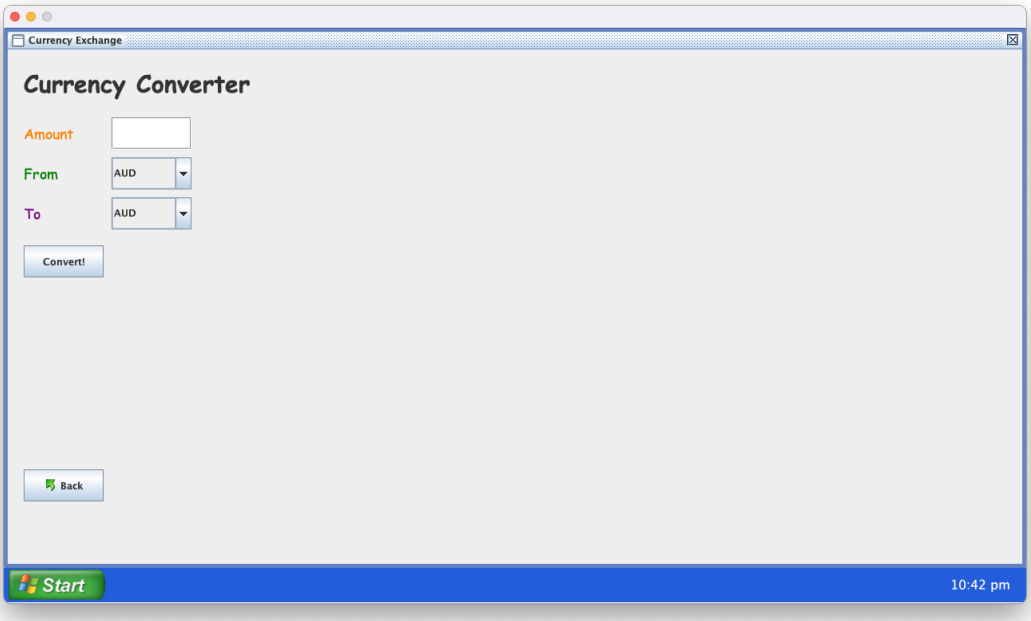


Figure 3: The primary use case of *Currency Converter* — a currency converter

exchange's history over a set of dates. Admin, quite intuitively, inherits from BasicUser. When a user has admin privileges, they should be able access basic functionality and hence the program has been structured as such. However, admin users also have access to an Admin portal where they can set the popular currencies, add a new currency or exchange, or reset the database with the orginal data it was populated with.

3.2.1 SQLite database

SQlite can be quite intimidating. "Why would a team go out of their way to store data on a local SQL server?", you may be wondering. However, the effort to gain ratio in favour of using SQL has proved to be very low. It has been very rewarding. Querying the most recent data has been not only less code, but also much more efficient than using an alternative like JSON.

What has been done with the SQL server? Firstly, the SQL Database has been on file locally along side the `build.gradle` file. The database is store in file called currency.db. This file acts like a typical SQL (MySQL, Oracle, PostGreSQL, ...) database with a few changes - datatypes aren't as strictly enforced. Everything is usually stored in a numeric or string format. Additionally, much of the functionality of major SQL servers such as PostGreSQL is missing. However, it is quite simple to query data from, and insert data into the database. This has been done in `CurrManager.java`.

`CurrManager` has been modularised to be able to be called from the backend application quite

easily. Before each query function is called `openConn` has to be called, and after every query function the database connection must be closed using `closeConn`. Every other function either inserts, queries, or resets data in/from the database.

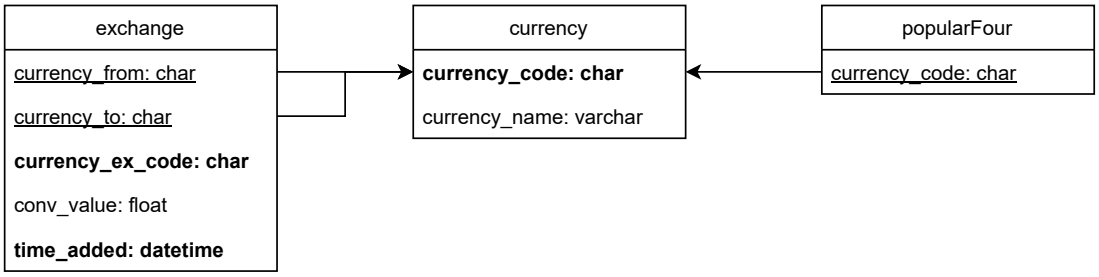


Figure 4: UML diagram of the SQLite database

3.2.2 Backend methods

In the back end there is are BasicUser and Admin classes where the BasicUser is responsible for normal users functionalities and the Admin is responsible of the admin functionalities. The Admin extends from the Basicuser as an Admin is a normal user who has the privilege of adding to and updating the database. All functions specified in the requirements have been implemented.

3.3 Integration

In order to link the frontend to the backend, each class requires either a `BasicUser` or an `Admin` object depending on which type of user should be allowed access to the given screen. Any links to the backend of the software are done via these classes.

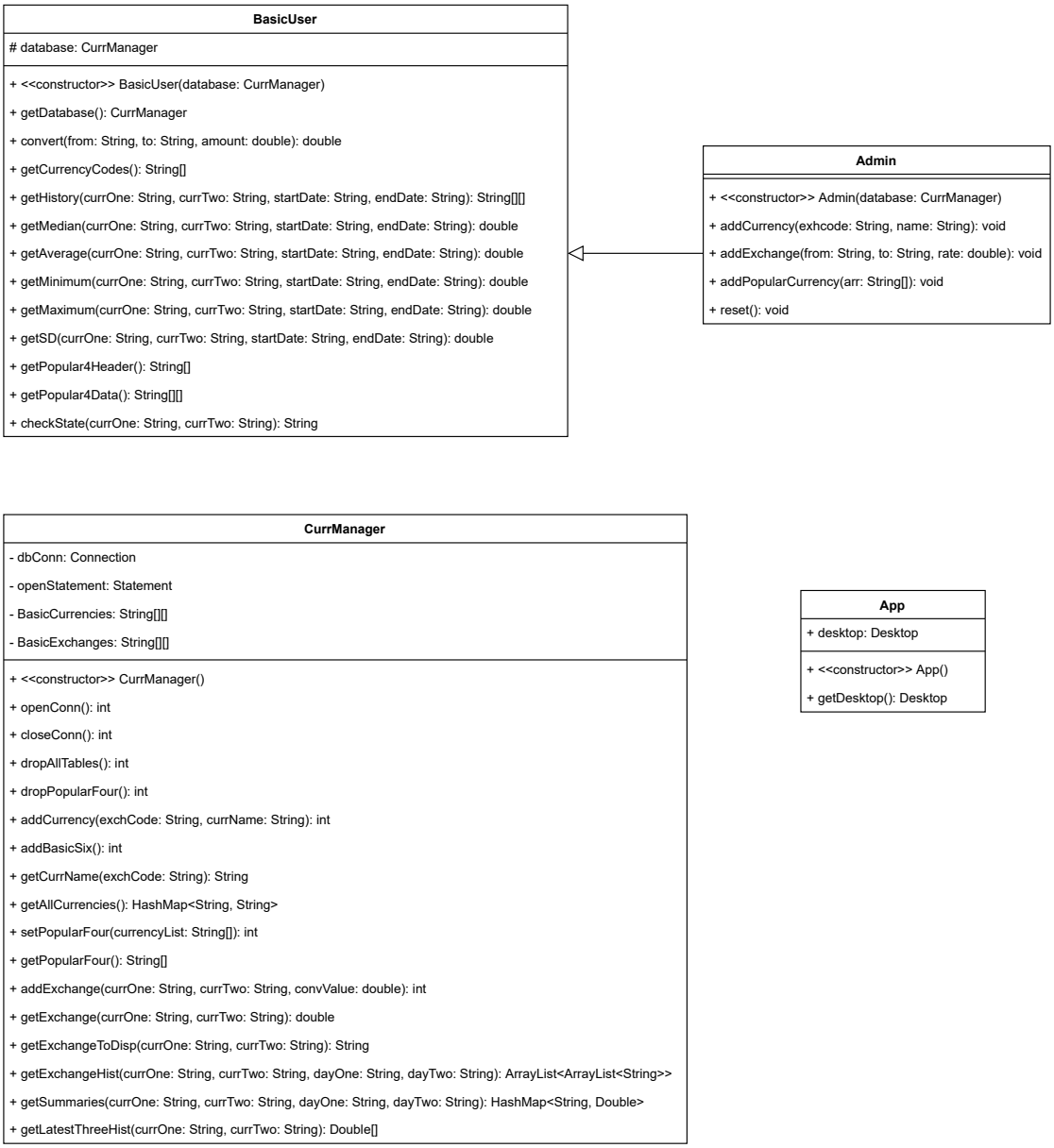


Figure 5: UML diagram of the backend code

4 Automated testing with JUnit

Software testing is an integral part of Agile software development and quality assurance. It is something that is conducted in each cycle of the development process to ensure that each deployment of the software performs the required functionalities correctly and securely. There are many way to conduct software testing and it is best done by a group of people including the developer themselves, other members of the development team, and real users of the software who is interested in using the software themselves.

When testing *Currency Converter*, the team to the best of their abilities attempted to adhere to Agile principles and conducted two major types of white-box testing: unit testing and integration testing.

Unit testing is a way of testing where a developer tests the smallest piece of code that can be logically tested [(What Is Unit Testing?, n.d.)]. Throughout the project the team conducted unit tests on the database API: this was done by the developer who wrote the API and created the database for the project. They tested each function in isolation.

Integration testing is defined as a way of testing where software modules are integrated logically and tested as a group ensuring that the modules are communicating as specified (Hamilton, 2022). Integration testing was used in *Currency Converter* to test the communication between the Database API (DAPI) and `Admin` and the DAPI and `BasicUser` individually. No tests were conducted between `Admin` and `BasicUser` classes because the only way they interact is through the database; hence the team decided that as long the database correctly interacts with the `Admin` and the `BasicUser` individually, interaction between the `Admin` and `BasicUser` was also correct. These tests were design and conducted by the developer of the and `Admin`.

All application testing was done in 4 test files:

1. `AppTest.java`: this file was used to test and ensure that the application ran properly and that a window did pop up when the program was run.

2. `CurrManagerTest.java`: Includes all the unit testing of the DAPI, ensuring all the functions perfrom the desired outcome. It tests that API has can access, change, and get data form the database correctly.

3. `AdminTesting.java`: Where all the communication between the admin and the DAPI was tested, checking that the database updated accordingly when Admin called, inserted or picked the popular 4 currencies.

4. `BasicUserTesting.java`: Includes all integration testing between the user and the DAPI, ensuring basic user had the access to the database in a way such that when user called a function the corrects data was returned.

Development of test were developed incrementally, so each time someone updates their file or added a functionality they would also write the unit test or the integration test depending on

the files, immediately after. Then all tests, old and new, were run using `gradle test`, and only once all the tests were successful were the changes pushed to the remote repository.

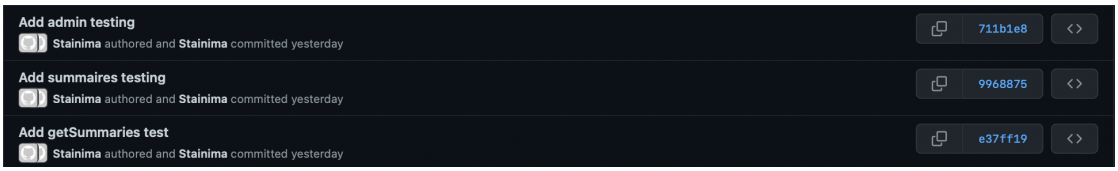


Figure 6: Evidence of incremental test development

The tests were conducted utilising the JUnit 5 testing framework. Each test was performed by calling the function for the functionality being tested, storing the result in a variable, and then using `assert` statements to check if the output was correct. Before each test, the environment was set up and each test was performed with a new instance of the object being tested. For example, when testing `convertCurrency()` for a basic user, 2 dummy currencies were added to the database and a exchange rate between the currencies and a new instance of basic user was instantiated. The function was run and the returned value was stored in a variable and tested using assert statements to ensure that the output was correct. Although the team wanted to compare to actual values, sometimes, due to the return result being a `double` for certain functions, comparing 2 values wouldn't always work because of precision error. Hence the team utilised the `delta` function of the tests to set a small margin of error. This would allow our tests to pass.

The team attempted to test all the test cases they could feasibly conduct in the time frame they had left themselves. Some of the cases were not considered and some did not need to be tested as the team limited some of the inputs using the GUI. For examples, a user can only convert currencies that are in the drop down menu of the GUI, showing only the currencies in the database, hence the team did not have to worry about testing of currencies that were not in the database.

Through this testing, at the end the team had a final test coverage of around 80% all together, excluding all the front end files (for evidence see Figure).

5 Continuous Integration with Jenkins

This section will detail a clear and sensible explanation of the work carried out by the team (what, how and why) to automate the CI pipeline.

5.1 Setting up the Jenkins server

During the first team meeting our group had decided to host Jenkins on a public server so that the (CI/CD) tests would run on the same machine. One of our team members purchased a Linode Server (4 Cores, 8GB RAM, Dedicated CPU). This server was configured to run the latest version of Ubuntu (A Linux Distro). Setting up Jenkins was then a matter of SSHing into the Server, installing Java, and Jenkins, and then configuring Jenkins settings.

The team used VS Code and IntelliJ terminals to SSH into the Server. SSH or Secure Shell is a method of secure network communication and authentication. "SSH uses the client-server model, connecting a Secure Shell client application, which is the end where the session is displayed, with an SSH server, which is the end where the session runs" - (Cobb, 2021). The team would use these SSH Sessions to install applications that were required on the Server which include but are not limited to - Jenkins, Java JDK, Xvbf.

5.2 Jenkins integration with GitHub

The team integrated GitHub with Jenkins using 'webhooks', one of two main methods of managing communication between two (or more) different services on a particular condition. The other method of doing this is with 'polling'. To explain the difference it is essential to understand that information travel in these communications is usually one way - from Service 1 to Service 2. Webhooking is creating a condition to communicate (from Service 1) to Service 2 on a condition. Polling is (Service 2) requesting information from Service 1 on a particular condition (usually certain times). As it was a requirement to test and deploy on every pushe to GitHub it was more appropriate to set up a Webhook from GitHub to Jenkins.

However this isn't truly integrating Jenkins with GitHub. Simply testing and deploying code on every push to GitHub does not actually help. We wanted to ensure that we were up to date with how our code was performing and what was happening on Jenkins. This could be quite difficult to do without any particular tool. On large project like these (even in small teams) each member pushes to main a few times a day. This can be quite difficult to keep track of. That is why we setup email notifications on the failure of a Jenkins Build, and a Discord webhook to notify us of any deployment to Jenkins and the build (and test) result (see Figure 7). This truly integrates GitHub (and Git) with Jenkins. Creating a feedback loop of pushing code, rectifying on failure, repushing, and continuing development.

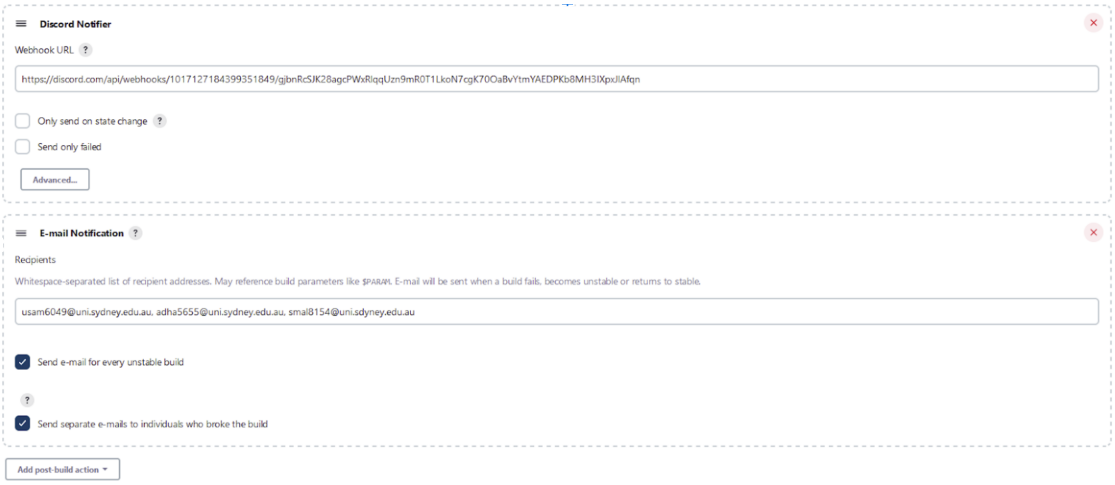


Figure 7: Discord webhook and email notifications on Jenkins

5.3 Setting up webhooks

We set our webhook by first specifying our GitHub repo while setting up our Jenkins Project. In this case, our Service 1 is Github, and our Service 2 is Jenkins. We have specified our condition for the Webhook to be any push to the main (previously called master). The URL the GitHub pushes (attempts to communicate using) is `http://45.79.239.48:8080/github-webhook/`. Jenkins is configured to listen (poll) at this URL for any events. So when one of the members of the group pushes to our team's GitHub repository, GitHub pushes the event (and some event metadata) to that URL.

As can be seen in Figure 8, we have specifically set up Jenkins to build only the main branch. This is because our main branch is the only branch that should be deployed. Hence, the only branch that should be tested on Jenkins before getting deployed.

5.4 Issues with Jenkins and GUI applications: Xvbf

This was a large enough issue to deserve its own section. For a large part of the development of *Currency Converter*, it was quite difficult to just check whether the application ran on the Jenkins Server. This was because our Jenkins server was a remote headless server, meaning that it did not have a Graphical User Interface, the drivers for it, or a screen that we could display the graphics. This may sound like a trivial issue, but when Java attempts to display graphical data where there is a lack of a driver, it runs into some issues - our code fails to build. This means we can't quite see the test coverage. The simple fix would be just not to test (or execute) any of the Graphical Aspects of the application on the Server. However, we were determined to fix this, in the name of continuous integration. It would not truly be continuous integration if the group had not checked if a major aspect of our application ran on our test server.

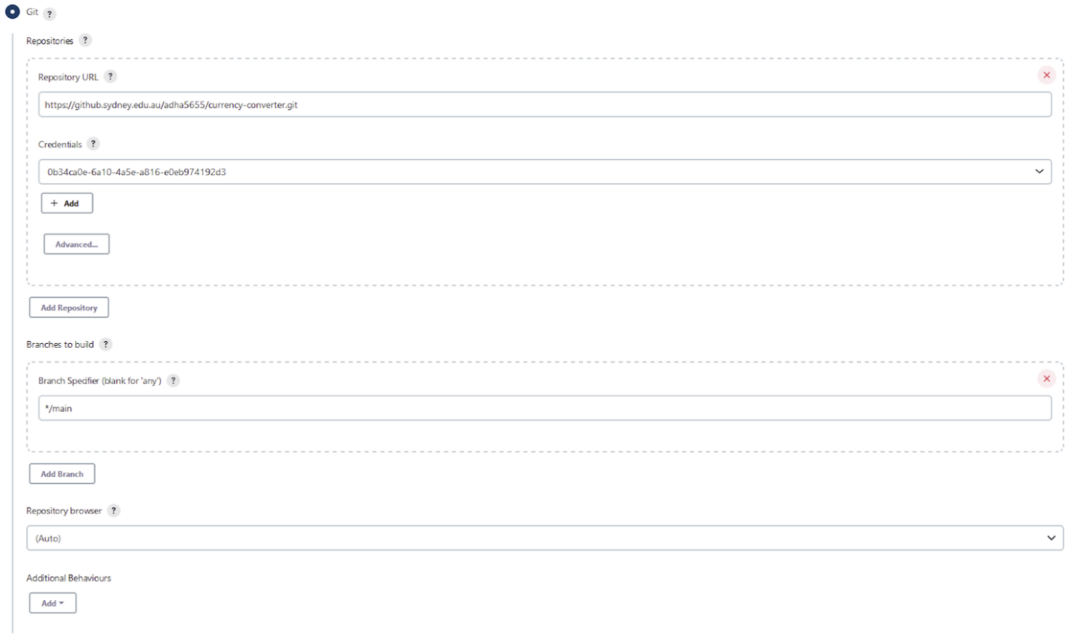


Figure 8: Specifying which branch to build on Jenkins

Members of our team who had experience coding on WSL 2 (before WSL g had come out) knew of solutions to fix this locally. WSL back a few months ago was not compatible with screens. The solution was to use a virtual framebuffer (How to Run Your Tests Headlessly With Xvfb, n.d.), a buffer that imitates a display. However, finding one compatible with Jenkins was quite difficult. After some, scouring on Google we came across the `Xvbf` plugin for Jenkins. The `Xvbf` plugin runs `Xvbf` as pre-build action and closes it at the end of the Jenkins build. It exports any GUI data to a virtual framebuffer for the period of Jenkins build. Setting this up was quite easy. It was simply installing `Xvbf` on the Jenkins server, and then setting up the plugin on Jenkins. You can see how we did this in Figure 9.

5.5 Artefacts

What are artefacts? Why are they important? How do we archive one? These were all questions that the group had difficulty understanding at the beginning. However, now it is something that all members have a clear understanding of. An artefact is simply just a by-product (usually a summarised version of a by-product) of the software development progress. This definition may seem quite vague, as it is meant to be. Many items may be considered artefacts. The value of a given artefact depends on the point in the timeline, that software in development is at. Depending on the value and the dependencies of an artefact it may provide useful to archive it.

An example of an artefact that we produced was the use case diagram that we produced

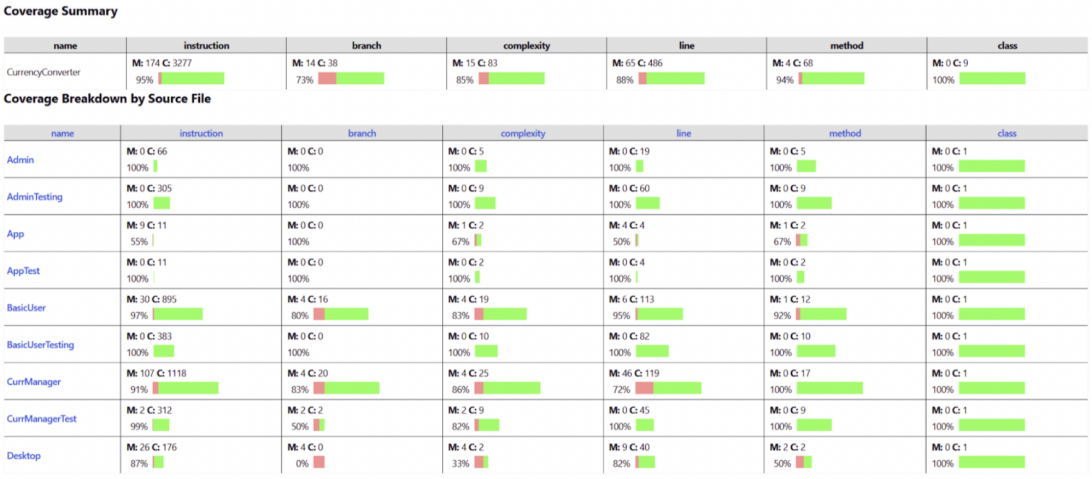
Figure 9: Installing the Xvbf plugin on Jenkins

during one of our first meetings (in the content requirement section). This was quite a valuable artefact while the group started to program. We checked of requirements one at a time from the diagram. What other artefacts did we produce? The main artefacts that we produced were the JUnit Test Report, JacocoTestReport, A Javadoc, and the Jar file.

Why are these especially important? The JUnit Test Report helped us understand which test failed on the Jenkins servers. On build failures, the JUnit Test Report gave a better understanding of which tests failed on the Server and what the standard outputs and standard errors were. It was quite useful to set up archiving for this test report, as many of the times, issues reappeared, and test cases failed in a manner similar to a previous build failure. On these reappearing issues, consulting archived Test Reports helped us debug the program at a much faster pace

The JacocoTestReport helped the members of the group understand what the current state of our testing was. It provided a concise summary of what was being tested and what was not. This gave the group clear goals to work towards to achieve a higher testing coverage. It was always ensured that each, NON-GUI, program had at least 70% test coverage as seen in Figure 10.

Jenkins was set up to also construct a Javadoc for every successful build. A Javadoc is a means of explaining the code and functionality of program users that have the requirement of reading through our code. It is essential that every build has an accompanying Javadoc to properly explain, not just for the benefit of outside parties, but also to help programmers



Coverage Summary

name	instruction	branch	complexity	line	method	class
CurrencyConverter	M: 174 C: 3277 95%	M: 14 C: 38 73%	M: 15 C: 83 85%	M: 65 C: 486 88%	M: 4 C: 68 94%	M: 0 C: 9 100%

Coverage Breakdown by Source File

name	instruction	branch	complexity	line	method	class
Admin	M: 0 C: 66 100%	M: 0 C: 0 100%	M: 0 C: 5 100%	M: 0 C: 19 100%	M: 0 C: 5 100%	M: 0 C: 1 100%
AdminTesting	M: 0 C: 305 100%	M: 0 C: 0 100%	M: 0 C: 9 100%	M: 0 C: 60 100%	M: 0 C: 9 100%	M: 0 C: 1 100%
App	M: 9 C: 11 55%	M: 0 C: 0 100%	M: 1 C: 2 67%	M: 4 C: 4 50%	M: 1 C: 2 67%	M: 0 C: 1 100%
AppTest	M: 0 C: 11 100%	M: 0 C: 0 100%	M: 0 C: 2 100%	M: 0 C: 4 100%	M: 0 C: 2 100%	M: 0 C: 1 100%
BasicUser	M: 30 C: 895 97%	M: 4 C: 16 80%	M: 4 C: 19 83%	M: 6 C: 113 95%	M: 1 C: 12 92%	M: 0 C: 1 100%
BasicUserTesting	M: 0 C: 383 100%	M: 0 C: 0 100%	M: 0 C: 10 100%	M: 0 C: 82 100%	M: 0 C: 10 100%	M: 0 C: 1 100%
CurrManager	M: 107 C: 1118 91%	M: 4 C: 20 83%	M: 4 C: 25 86%	M: 46 C: 119 72%	M: 0 C: 17 100%	M: 0 C: 1 100%
CurrManagerTest	M: 2 C: 312 99%	M: 2 C: 2 50%	M: 2 C: 9 82%	M: 0 C: 45 100%	M: 0 C: 9 100%	M: 0 C: 1 100%
Desktop	M: 26 C: 176 87%	M: 4 C: 0 0%	M: 4 C: 2 33%	M: 9 C: 40 82%	M: 2 C: 2 50%	M: 0 C: 1 100%

Figure 10: Each build of our program had at least 70% test coverage

understand each other's code while working in teams. This is what our team used the Javadoc for. Communicating what certain code does can be quite difficult and the Javadoc is a tool that helps bridge that gap.

The last artefact that the program produced was the `.jar` file. A `.jar` or Java Archive is simply as the name suggests is an archiving tool. "It's a file format based on the popular `.zip` file format and is used for aggregating many files into one" - (JAR File Overview, n.d.). Jenkins was setup to create a `.jar` file on every build. This was done by configuring an extra task in build - the `fatJar` task. This was an extension of the `.jar` task. It was quite important to archive these jar files as they provide a method for our clients and users to revert to a previous version for various reasons such as compatibility issues.

Configuring the collection of these artefacts was done as a post-build task. Specifying the files on Jenkins was quite simple with the use of wild cards (that allows Jenkins to match multiple files). As the reports were stored as HTML files, the archiving tool was specified to collect all HTML files in the reports folder. It was also tasked with collecting and archiving the entire java doc directory. Lastly, JacocoReports are also stored as an exec file from which data can be extracted and would be useful to have access to in the future.

5.6 Jenkins logs

Every Jenkins build has a respective console log - information regarding the processes and execution of the tasks that have been set up previously for builds. Below is the console output for the latest build on Jenkins as of 21/09/2022 10:31 pm. In the first portion of the Jenkins console output, Jenkins is attempting to pull changes from GitHub after being notified of a push. After which our Jenkins starts running `Xvbf` on the server and data starts to be sent to a virtual frame

buffer.

After the virtual framebuffer completes building (which can take quite a bit of time), Jenkins attempts to start building our Gradle and executing the other Gradle Tasks. This occurs in the order specified in the configuration of our project.

The last section is just Jenkins collecting all our artefacts. Something to note with the way we have configured JacocoTestReports is that Jenkins completely handles the creation of JacocoTestReports, and Gradle does not.

5.7 Adopted CI practices

We based our CI practices based on how we had structured our program. As we were aiming to complete a fully functionally GUI-based *Currency Converter* we felt as though working on different branches would prove to be a better practice (and it did!). Every few commits (sufficient for a minor change), the contributor would create a pull request on GitHub and notify the team (though our discord GitHub webhook did that as well). Then another team member would read through the code, and accept or request changes. Once the pull request was accepted the other team member would attempt to run the code, check if it functioned as expected, and cross-check this with Jenkins.

Why did we choose to work on different branches? We implemented very different types of code - the front-end used various Java Swing functions; for our database, we used SQLite3 and Java's SQL driver; for the backend, we communicated between the two. We expected that there would be times when the integration of these three categories would raise issues, even though each individual component was tested and working as expected. Should we have worked only on the main, it would have been much more difficult to track when integration of the three types of code we had failed. This happened during one of our integrations where the code ran as expected on two of our members' computers but not on the local machines of the other two. For our CI we had created a chain that had to be followed. We had an integration branch, a front-end branch, a back-end branch, and a database branch. When Udit (DB Manager) finished work (and tested code), our backend managers (Sulav, Nemo) would immediately merge from the database branch to the backend branch. Then whenever a change in, `front_end` or `back_end`, was committed, we would immediately merge that to the integration branch, and then if integration tests passed locally (on our integration manager's local machine), we would create a pull request to main, which two members would review. Once this was done, and conflicts were resolved we push to main wait for the Jenkins test report - rectifying code as required.

```
Started by user Udit Samant
Running as SYSTEM
Building in workspace /var/lib/jenkins/workspace/Jenkins GitHub
The recommended git tool is: NONE
using credential 0b34ca0e-6a10-4a5e-a816-e0eb974192d3
 > git rev-parse --resolve-git-dir /var/lib/jenkins/workspace/Jenkins GitHub/.git # timeout=10
Fetching changes from the remote Git repository
 > git config remote.origin.url https://github.sydney.edu.au/adha5655/currency-converter.git # timeout=10
Fetching upstream changes from https://github.sydney.edu.au/adha5655/currency-converter.git
 > git --version # timeout=10
 > git --version # 'git version 2.34.1'
using GIT_ASKPASS to set credentials
 > git fetch --tags --force --progress -- https://github.sydney.edu.au/adha5655/currency-converter.git +refs/heads/*:refs/remotes/origin/* # timeout=10
 > git rev-parse refs/remotes/origin/main^{commit} # timeout=10
Checking out Revision abe206163b644ba0152563d3b280b6bd86a83b0f (refs/remotes/origin/main)
 > git config core.sparsecheckout # timeout=10
 > git checkout -f abe206163b644ba0152563d3b280b6bd86a83b0f # timeout=10
Commit message: "Fix jenkins"
 > git rev-list --no-walk abe206163b644ba0152563d3b280b6bd86a83b0f # timeout=10
false
Xvfb starting$ Xvfb -displayfd 2 -screen 0 1024x768x24 -fbdir "/var/lib/jenkins/workspace/Jenkins GitHub/.xvfb-86-..fbdir288750298909784869"
_XSERVTransSocketUNIXCreateListener: ...SocketCreateListener() failed
_XSERVTransMakeAllCOTSServerListeners: server already running
1
[Gradle] - Launching build.
[app] $ /var/lib/jenkins/tools/hudson.plugins.gradle.GradleInstallation/Gradle_Jenkins/bin/gradle clean build test javadoc jacocoTestReport fatJar
Starting a Gradle Daemon (subsequent builds will be faster)
> Task :clean
> Task :compileJava
> Task :processResources
> Task :classes
> Task :jar
> Task :startScripts
> Task :distTar
> Task :distZip
> Task :assemble
> Task :compileTestJava
> Task :processTestResources NO-SOURCE
> Task :testClasses
> Task :test
> Task :check
> Task :build
> Task :javadoc
> Task :jacocoTestReport
> Task :fatJar

Deprecated Gradle features were used in this build, making it incompatible with Gradle 8.0.

You can use '--warning-mode all' to show the individual deprecation warnings and determine if they come from your own scripts or plugins.

See https://docs.gradle.org/7.4.2/userguide/command_line_interface.html#sec:command_line_warnings

BUILD SUCCESSFUL in 53s
12 actionable tasks: 12 executed
Build step 'Invoke Gradle script' changed build result to SUCCESS
Archiving artifacts
[JaCoCo plugin] Collecting JaCoCo coverage data...
[JaCoCo plugin] **/**.exec;**/classes;**/src/main/java/CurrencyConverter/; 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/app/build/jacoco/test.exec
[JaCoCo plugin] Saving matched class directories for class-pattern: **/classes:
[JaCoCo plugin]  - /var/lib/jenkins/workspace/Jenkins GitHub/app/build/classes 21 files
[JaCoCo plugin]  - /var/lib/jenkins/workspace/Jenkins GitHub/app/build/reports/tests/test/classes 0 files
[JaCoCo plugin] Saving matched source directories for source-pattern: **/src/main/java/CurrencyConverter/:
[JaCoCo plugin] Source Inclusions: **/*.java,**/*.groovy,**/*.kt,**/*.kts
[JaCoCo plugin] Source Exclusions: **/WelcomeScreen.java, **/SelectPopular.java, **/History.java, **/History$MyTableModel.java, **/Desktop.javaWelcomeScreen, **/Desktop$1.java, **/CurrencyExchange.java, **/AdminPortal.java,
**/Calculator.java, **/Converter.java, **/Converter$1.java, **/DisplayPopular.java, **/DisplayPopular$MyTableModel.java, **/UpdateExchange.java
[JaCoCo plugin] WARNING: You are using directory patterns with trailing /, /* or /** . This will most likely multiply the copied files in your build directory. Check the list below and ignore this warning if you know what you
are doing.
[JaCoCo plugin]  - /var/lib/jenkins/workspace/Jenkins GitHub/app/src/main/java/CurrencyConverter 5 files
[JaCoCo plugin] Loading inclusions files..
[JaCoCo plugin] inclusions: []
[JaCoCo plugin] exclusions: [**/WelcomeScreen.class, **/SelectPopular.class, **/History.class, **/History$MyTableModel.class, **/Desktop.classWelcomeScreen, **/Desktop$1.class, **/CurrencyExchange.class, **/AdminPortal.class,
**/Calculator.class, **/Converter.class, **/Converter$1.class, **/DisplayPopular.class, **/DisplayPopular$MyTableModel.class, **/UpdateExchange.class]
[JaCoCo plugin] Thresholds: JacocoHealthReportThresholds [minClass=0, maxClass=0, minMethod=0, maxMethod=0, minLine=0, maxLine=0, 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: 100.0, method: 94.44444, line: 88.20327, branch: 73.07692, instruction: 94.957985, complexity: 84.69388
Xvfb stopping
Sending notification to Discord.
Finished: SUCCESS
```

Figure 11: Jenkins logs

6 Build Automation using Gradle

In the development of *Currency Converter*, Java files needed to be compiled and run, automatic tests needed to be run and artefacts needed to be created. However this process could get time consuming and tedious especially when rapidly deploying and continuously integrating an application. Hence, the development team needed a solution, which was build automation.

Build automation is important because it speeds up the development process by automatically compiling the source code, running tests and creating artefacts. Historically, build automation was achieved in C applications with `makefiles`, but nowadays, software development teams use modern build automatic tools such as Gradle. Gradle was the preferred tool for this project and was heavily used throughout the development process of *Currency Converter*.

Gradle was used to automate building, running and testing of code. This section will detail the relevant `gradle` commands used, artefacts created and an explanation of the `build.gradle` file.

6.1 Relevant Gradle commands and artefacts

The development team used a variety of Gradle commands to automate builds and generate artefacts.

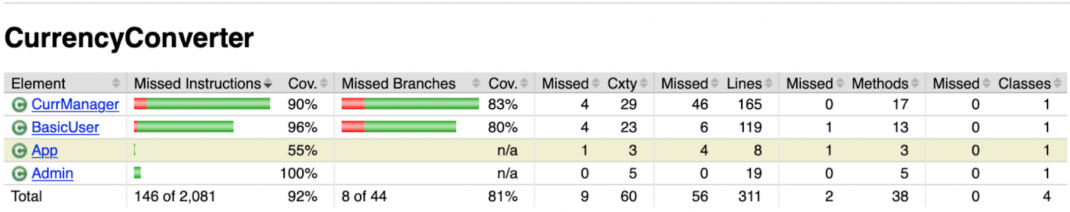
1. `gradle init`: This is the first command used and initialises the gradle build and the directories for source files, test files and resource files. The developer can follow the given prompts to set up their application.

2. `gradle build`: This command is used to compile all java classes and make it ready for gradle run. The output shows whether or not the build was successful or not and how long the automated build process took.

3. `gradle run`: The gradle run command is used to run the application after building. It can also be used to sequentially build and run the code if gradle build has not been run already. The output shows a progress bar of how far it is through the execution process. It also has a timer for how long the application has been running.

4. `gradle test`: Gradle test is used to run all Junit test cases and display test results. This is most useful when there are many test classes and it is time-consuming to run each class. Gradle's automatic testing makes the testing process much quicker. The output shows success if all test cases successfully passes and fail if some didn't pass.

5. `jtr`: The task is an alias for the JaccocoTestReport task that the team had created. It also depends on the build tasks, which ensures that the JacocoTestReport produced by jtr is based on the latest source code, and not the laste build files.The tasks generates a JacocoTestReport artefact at the following location: `currency-converter/app/build/reports/jacoco/test/html/index.html`.



CurrencyConverter

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
CurrManager		90%		83%	4	29	46	165	0	17	0	1
BasicUser		96%		80%	4	23	6	119	1	13	0	1
App		55%		n/a	1	3	4	8	1	3	0	1
Admin		100%		n/a	0	5	0	19	0	5	0	1
Total	146 of 2,081	92%	8 of 44	81%	9	60	56	311	2	38	0	4

Figure 12: The index file of the JacocoTestReport artefact

6. `fatJar`: This task aggregates all the files in the src folder and creates a jar file. The jar file produces can be compiled (more effeciently) and ran using the command `java -jar className.appName.jar`

   The JacocoTestReport `index.html` is shown in Figure 12.

   An explanation on how to read this will be detailed in the Testing (Junit) section of the report.

6.2 The Gradle build file

```
plugins {
    // Apply the java plugin to add basic functionality for Java Code.
    id 'java'

    // Apply the application plugin to add support for building a CLI application
    in Java.
    id 'application'

    // Apply the jacoco plugin to add support for test reports on test code
    coverage.
    id 'jacoco'
}

repositories {
    // Use Maven Central for resolving dependencies.
    mavenCentral()
}

dependencies {
    // Use JUnit test framework.
    // testImplementation 'org.junit.jupiter:junit-jupiter:5.8.2'

    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.8.1'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.8.1'

    // This dependency is used by the application.
    implementation 'com.google.guava:guava:30.1.1-jre'
```

```
26
27      // Adds sqlite3, and the java driver
28      implementation group: 'org.xerial', name: 'sqlite-jdbc', version: '3.36.0.3'
29  }
30
31  application {
32      // Define the main class for the application.
33      mainClass = 'CurrencyConverter.App'
34  }
35
36  test {
37      useJUnitPlatform()
38  }
39
40  task jtr {
41      dependsOn test
42      dependsOn jacocoTestReport
43  }
44
45  javadoc {
46      sourceSets {
47          build {
48              java.srcDir file('src/main/java/CurrencyConverter')
49
50          }
51      }
52  }
53
54  jacocoTestReport {
55      afterEvaluate {
56          classDirectories.setFrom(files(classDirectories.files.collect {
57              fileTree(dir: "build/classes/java/main/CurrencyConverter",
58                      exclude: ['WelcomeScreen.class',
59                                'SelectPopular.class',
60                                'History.class',
61                                'History$MyTableModel.class',
62                                'Desktop.class',
63                                'Desktop$1.class',
64                                'CurrencyExchange.class',
65                                'AdminPortal.class',
66                                'Calculator.class',
67                                'Converter.class',
68                                'Converter$1.class',
69                                'DisplayPopular.class',
70                                'DisplayPopular$MyTableModel.class',
71                                'UpdateExchange.class'])
72          }))
73      }
74  }
```

```
75
76 version = '1.1'
77
78 task fatJar(type: Jar) {
79
80     archiveBaseName = 'Currency-Converter'
81   manifest {
82         attributes 'Implementation-Title': 'Currency Converter',
83           'Implementation-Version': version,
84           'Main-Class': 'CurrencyConverter.App'
85     }
86     baseName = project.name + '-all'
87     from {
88         configurations.runtimeClasspath.files.collect {
89             it.isDirectory() ? it : zipTree(it)
90         }
91     }
92     with jar
93 }
```

Listing 1: build.gradle file

The `build.gradle` file also known as the build configuration file is used to adjust and configure the settings for the build automation process handled by Gradle. The file has main sections including:

1. Plugins - apply plugins like java to add basic functionality for Java Code.
2. Repositories
3. Dependencies - specifies JUnit version
4. Application - defines the main class as `CurrencyConverter.App`
5. Test
6. task jtr
7. Javadoc
8. JacocoTestReport
9. fatJar

The plugins are the modules that are added for basic functionality, for example the code used, in the case java. The repositories section is the source of the public libraries used in the application, for example, Maven Central which is a public repository of Java and other libraries. The dependencies section specifies what the application is dependent on, for example, the version of JUnit. The application section defines where the main application class can be

found, this allows Gradle to automatically run the program when the `gradle run` command is used. In this case the main class is `CurrencyConverter.App`. The Test section has the configuration for which testing platform should be used, which is JUnit. The `task jtr` section details what should happen when the `gradle jtr` task is run, in this case, Gradle will look for the jacocoTestReport section. The Javadoc section details which directory the Javadoc should be generated from. Finally the `jacocoTestReport` section has the configuration for the generation of a JacocoTestReport artefact.

7 Version Control of source code using GitHub

Link to repository: `https://github.sydney.edu.au/adha5655/currency-converter`

The group used all the GitHub features including pulling, pushing, pull requests, commenting and version numbers. Excluding merges, 5 authors have pushed 39 commits to main and 85 commits to all branches. On main, 25 files have changed and there have been 1,960 additions and 159 deletions.

The git commands used were:

1. git add - adds changed files in the working directory to the staging area
2. git commit - saves added changes to the local repository
3. git push - uploads local repository data to the remote repository
4. git fetch - download contents from a remote repository
5. git merge - puts a forked history back together again
6. git pull - is a combination of fetch and merge
7. git checkout - change branch

7.1 Examples of GitHub features

Pull requests
For Pull requests the steps are:

1. Get notification of Pull request on GitHub
2. Review code
3. Make comments
4. Approve or cancel pull request

Branches
Project boards

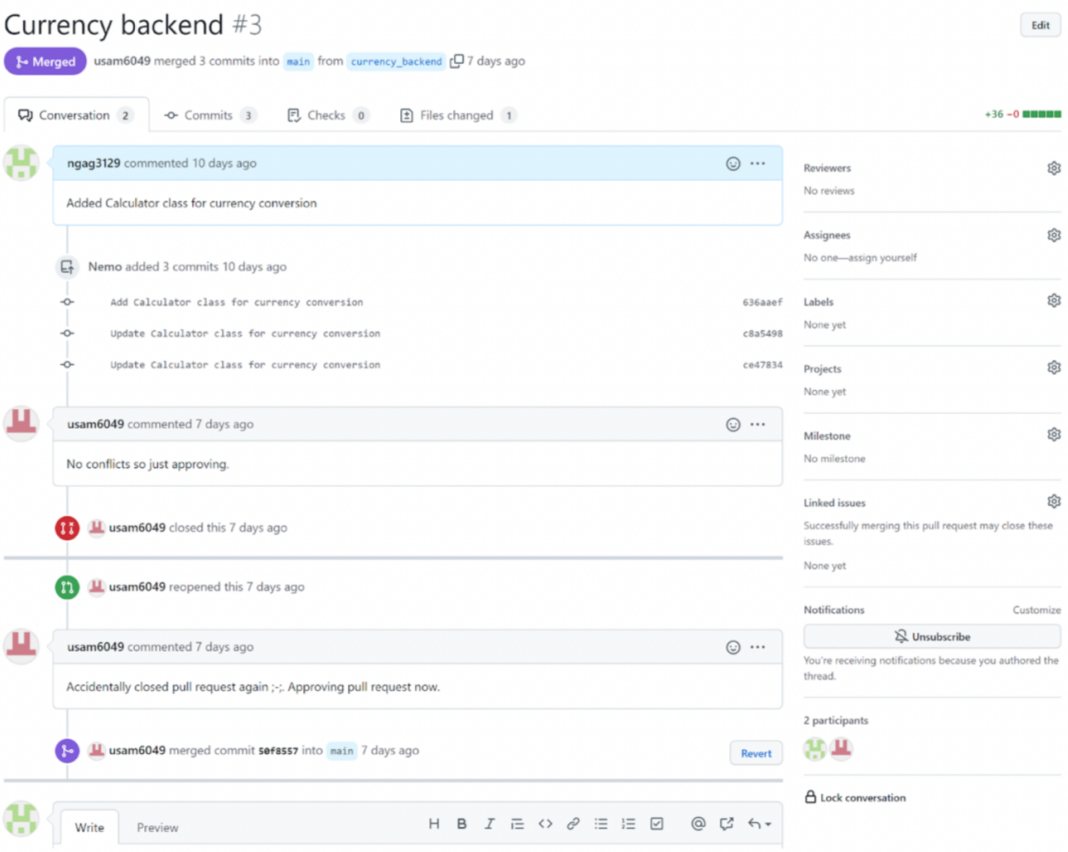


Figure 13: An example of a pull request on the *Currency Converter* repo

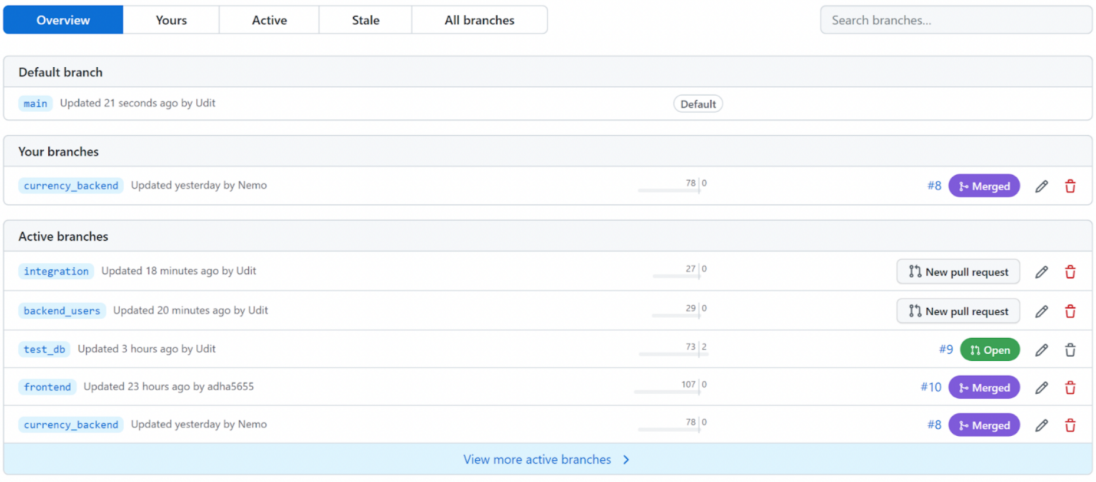


Figure 14: Branches on the *Currency Converter* repo

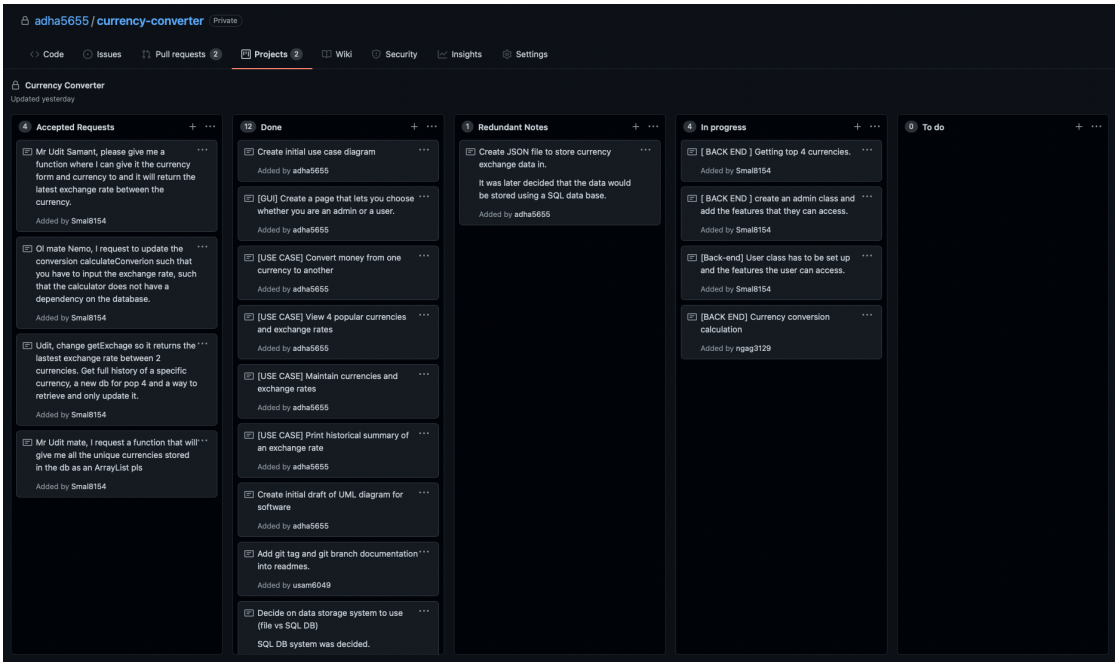


Figure 15: Project board used throughout development of *Currency Converter*

8 Social learnings and collaboration

8.1 Group meetings

The team used various communication methods including:

1. Facebook Messenger: Group members added each other on Facebook and Antriksh created a group chat and added everyone. The group chat was useful because issues could be discussed at home, uni or on transit by instantly notifying a group member's phone.

2. Discord: The team used voice channels for meetings and text channels for GitHub and Jenkins Bot notifications.

Meeting minutes can be found in Appendix

8.2 Individual contribution

The group contributed to the project by dividing up work fairly, creating branches and creating a pull request in GitHub when a feature was complete.

8.2.1 Antriksh Dhand (front end)

My primary role in this project was to develop a simple yet effective front end which met all the requirements of the project. Furthermore, once backend development was complete, I

was heavily involved in integrating the functions developed by other teammates to bring the static application to life. Front end development was done completely using `java.swing`, a library based on the now legacy `java.awt` (Abstract Window Toolkit) library. Having no prior experience in developing a GUI in Java, development of the front end came with a large learning curve in a relatively short period of time. One technique which I found helpful during this phase was to have a clear understanding of what the required use case was (which was aided by the use case diagram our team created in the first week), and to draw up the simplest implementation of a GUI for this use case on paper before even attempting to code. It is due to this aim of maintaining simplicity that some features such as the Admin login are so primitive. Files coded for the front end include AdminPortal.java, Converter.java, CurrencyExchange.java, Desktop.java, DisplayPopular.java, History.java, SelectPopular.java, UpdateExchange.java and WelcomeScreen.java, with development initially occurring on `origin/main` before moving to `origin/front end`.

The integration between the backend and front end occurred towards the tail-end of the development cycle. This process mainly occurred over set group meetings due to it requiring database knowledge from Udit, backend function development from Sulav, and my front end knowledge to tie these all together on the application itself. Integration was conducted on the `origin/integration` branch and once complete was merged with `origin/main`.

8.2.2 Sulav Malla (backend)

I was one of the back-end developers and was responsible for creating `Admin.java` and `User.java` and majority of the integration testing. I had to create functions such that the Admin and User classes could both access the SQL database, but only the Admin could make changes to it. As I relied heavily on the database, I had to use a lot of dummy values when creating the functions and toward the once the database and the API was completed had refactor code. As I was doing the interaction between the database and users I was also responsible for the integration testing between the database and the users. I personally did not experience a lot of issue and main was pushing to my own branch, but on the second last day there were some error in my branch that I could figure out how to debug, hence instead of spending more time on debugs, I abandoned my current branch and created a new branch from main and continued working. Hence, I had to redo a chunk of my code and continue on wards.

8.2.3 Nemo Gage (backend)

I worked on the backend development for *Currency Converter*. I was responsible for writing functions to connect the SQL database with the front-end application. For example in the `Calculator.java` class, the function `calculateConversion` parses the two currencies and the amount to be converted and returns the new calculated amount. I used this function to get the amount of currencyTo and feed it into the front end. I also worked on the group collaboration, by creating and reviewing pull requests, assisting team mates in group chats and Discord, and

creating shared Google documents and drives. I also set up this report, with numbered heading and designed the overall structure.

8.2.4 Udit Samant (Jenkins, database)

My contribution to this Project can be categorised into two Roles - A CI/CD Manager, and a Database Manager. I assisted the team by setting up the Continuous Intergration Pipline. This included purchasing and setting up the Linode Server; installing and configuring Jenkins on a headless Ubuntu Server, setting and artefact production and collection (JacocoTestCoverage, Junit5TestReport, and Javadoc), and lastly connecting version control, and communication with Webhooks. My contribution is explained in detail in the CD/CI Section of the report. However, you can also go to http://45.79.239.48:8080/, see our Jenkins in action.

The more fun role I had was as a database manager. I was tasked with creating the Application Program Interface to allow the back end method to communicate with the database. I did this by programming the entire `CurrManager` class. An instance of the `currManger` class can read, write, and edit the table in the database. For this program I completed a majority of the testing with great help from . Lastly, along side other members of the team I assisted setting up sprints, and assisted with version control issues.

8.3 Reflection

Throughout the project, although the team set out to adhere to the agile practice, the team majority of the times did not do so and learned the difficulty in amount of consistency, communication, and teamwork required to realistically follow these practices. Although there were multiple group chats and many meetings that took place, barely any communication was taking place outside the meeting. Not everyone had a clear direction on what they were doing, and that was not being voiced not understood very well by all members of the team. The team also did not have a clear and consistent work flow, majority of the coding and reporting writing was heavily skewed toward the of the project, including two days after the deadline. Hence, some of the functionalities were missed and other parts rushed or even incomplete. This is also due to the lack of communication between the team as the work pace and amount of work output dramatically when the team decided to code together and both the font and back end developers understood what the others needed. Therefore, in the future this team needs to step up their game. They need to communicate better, workout a better structure and work more consistently. Or else, their next assignment is gonna be 5 days late.

A Meeting minutes

Meeting minutes

Meeting 1 (30/08/22)

The team begins by briefly reading the specification together as a group and breaking down the specification and digesting the information. The team went around talking of what each person had understood from the assignment specifications and suggesting ideas on how to do certain aspects of the assignment. As the content and the ideas got too clouded, the team decided to create a simple use case diagram form the specification ensuring that everyone understood what had to be done. Then all 4 members came together to produce a simple use case diagram of the specification for the project (see Figure). Then each team member went away and were tasked to attempt to implement what they could in until the next meeting.

Meeting 2 (04/09/22)

Talked of the progress each person has made. Not much Technical coding had been done at this point so the group decided to issue each person a role, and what a person was in charge of. A remote GitHub repository was set in this meeting and shared with the team, so each person has access to the repository where they can contribute. A simple Kanban board on the GitHub repository page where anything that needs to be implemented, that is currently being implemented or has been implemented will be listed. Additionally, many discussions were had over the preferred data storage model for the software. Options explored included:

1. Retaining multiple csv files to keep track of each exchange rate's history
2. A JSON-based architecture with a JSON object for each exchange's history
3. Some form of SQL database

Option 1 was deemed inferior to both options 2 and 3 due to difficulty querying results from csv or text files in Java. As the team already had experience in SQL databases (gained through various other units at university) the team decided on using a local SQLite database as our data storage model. A database system was decided and a relational SQL database was chosen to store the information about the currencies and work was redelegated for each person. A picture of the Kanban board is below: (see Figure)

Meeting 3 (07/09/22)

Everyone went around showing what they had done since the last meeting. A simple GUI had been set up and a SQL database had been opened up. In this meeting, everyone talked about Jenkins and tried to understand what Jenkins is and how to set it up, then one person was tasked with setting the jenkins up for the team. The kanban board had been reviewed, and what was left to do. Attempted to plan out a small schedule for the

team. Created new branches on Github so now each person is working in 1 branch each and should only push to their own branch, then when they wanna push to main a pull request should be created which another person will review and decide if it will go through or not.

Meeting 4 (11/09/22)
Same as usually everyone went around talking of their progression of each person, the calculator had been done. Jenkins had been set up, and jenkins had been set up so discord could be used to receive updates whenever someone commits or makes a change. The team then talked about planning the report structure and what each person would do for each section of the report and who would do it. As each person required something from someone else regarding the code and communication became a little difficult so a new column was added in the Kanban board for anyone to put any request 1 person had for another in terms of the code, and the person responsible for the corresponding section would go to the request board and implement the functionalities required. The team talked of what needed to be done, and refined the kanban board more, and the meeting was concluded.

Meeting 5 (16/09/22)
Most of the database and simple database API had been created, now everyone could update the code so that they can now call functions for the database where they can access and modify the database as required. Requested accepted column for any request that has been completed and implemented. Each person who was handling the backend was requested to make a test for the functions and class they had created, and the responsibility of integration testing was given to one person. Each person who contributed to the backend development was tasked with something to write in the report. Everyone was to write a brief description of what they had done and any issues they encountered and how they solved it.

Meeting 6 (18/09/22)
Database functions had been completed. However, some issues were found with the implementation and new changes were requested through the kanban board. Each person had a lot to implement, hence a todo list was created so that each person. Brief discussion on what we were going to do on the final day. Everyone had other assignments due this night, report was sectioned off and each person was given a responsibility of a part of the report over the next day and before the final meeting.

Meeting 7 (20/09/22)
Each member has continuously worked on the assignment, the assignment is a day late. In this session members of the group came together, stayed in the call and sorted everything out, the integration and all unit testing was completed and done during this session achieve a

code coverage of roughly 80% . The integration between the front and the ack end was done and competed in this meeting. All full functionalities excluding the summaries on the front end was and completed, and the Kanban board was readjusted and cleaned (image of the final kanban board below) and the final sections of the report each member was responsible was completed.