Continuous Integration Report

Group 1
Assessment 2

Emma Hogg
Isaac Rhodes
Isioma Offokansi
Toby Meehan
Seungyoon Lee
Ali Yildirim

(a) Throughout our project, each member of the team is working on different tasks with the ultimate aim of bringing these together to form a successful project. This can have many issues, however, as if each individual brings all their work together at the last minute there could be problems with compatibility, even though we agreed previously how to complete each task. In order to prevent this issue, we implemented multiple continuous integration pipelines.

We have created three different pipelines for continuous integration, each of which have their individual task. Each pipeline is triggered only under specified conditions so that they only run when required. The first pipeline is the pipeline to build the executable; next we have the pipeline for the running of the automated tests and checkstyle tests. Finally, the last pipeline relates to a release job. Each of these pipelines have their own YAML file to specify the context under which they are run and the steps they must follow. These different conditions have been described and explained below.

Firstly, the build stage is triggered by any push to the master branch. This is important as we want a new version of the code to be built after any change made to the code. This ensures that the executable file associated with the project is up to date at all times. This means the executable file within GitHub Actions matches the code in the repository confirming there are no inconsistencies in the repository. Next, the repository must have access to the most recent repository in order to create a build that matches this code. Therefore, as an input, it retrieves the most recent version of the repository. Finally, this pipeline must output the executable files using the code within the repository for each required operating system. These files are placed in a location that is known by the users of the repository so it can be easily found. This ensures that the building of the executable file occurs completely automatically to save time for the developers.

The next pipeline covers ensuring that all the unit tests that we have built pass and that the checkstyle is consistent. This is an important pipeline to include as we must ensure that the changes made have not caused any failures and all requirements are still being met. It also ensures that the style of the coding is consistent between members of the team which is vital for readability. This means that this pipeline should be triggered whenever a user pushes or pulls from the “master” branch or pushes to the “ci” branch. This ensures when the repository has been changed it will cause the pipeline to run to ensure that none of the tests now fail. This is vital to implement as we want to catch any errors early so we can fix them immediately. In order to run the tests that we have written, it must have an input of the most recent repository. Finally, as developers it is important that we can see the results of the tests that have been run so we know if anything has failed. This means that this pipeline will output the results of the tests and the checkstyle report in a known location for the developers to access and use.

The final pipeline is a pipeline used for releasing code given a tag. This pipeline is essential for users who want to download the game so they can trust the download is working reliably. They do not want to retrieve a version of the game that does not work as expected therefore this pipeline can be used by developers to create releases, of working executables, under a certain tag. This pipeline is therefore triggered when the developer pushes to the “master” branch with a tag. This, therefore, will only produce releases when developers are ready rather than after every push to the branch. In order to create the release, the pipeline needs an input of the most recent version of the repository. This will be used to create the executable files to be used in the release. The output of this pipeline is to release the two built versions of the code, covering the two operating systems specified in the requirements. Therefore, under the tag name given, the pipeline will output the built files, with their desired names, for users to find in the releases section of the repository.

These three pipelines cover what we require to be automated in the project to improve efficiency. It will fail if either the project has errors while building or any of the automated tests do not pass. This is important to include to ensure that the project has no issues while in the development stage and everything compiles as expected.

(b) For the continuous integration section of the project, we chose to use the platform GitHub Actions. We decided this platform was the most suitable as we are using GitHub as our version control software. GitHub Actions is integrated into GitHub, hence we can implement the pipelines so they work right alongside our code.

Our first step was to automate the building of our code. We first created a new YAML file named “build-jar.yml” which is used to tell GitHub Actions how to build the gradle. We first gave it a name at the beginning of the file to specify the purpose of the YAML file. Next, we needed to specify when the workflow is to be run and in this case we chose when there is a push by any user to the “master” branch. GitHub Actions must know what operating systems the built .jar file will be able to run on. We therefore specified Ubuntu and Windows in order to follow our user requirements. We also do not want this workflow to alter any of the code in the repository therefore we gave it read only permissions. Next, we must tell GitHub Actions what specific actions it should take in order to complete the build. The first step is to checkout the most recent version of the repository under the GitHub workspace so that our workflow can access and use it. Next, we set up the Java development environment (JDK 11) to build the application and set the distribution to ‘Temurin’ which is an open source implementation of the Java platform required to build the application. Then, the executables must be built which is completed using the gradle file found within the repository in the location we specified. Finally, the result of the build will be placed back into a folder in the repository. We used the action upload-artifact in order to place the newly built executable file into the repository for us to find and use. We specified the name of these files and the location that they should be placed.

Next, we needed to implement the automation of tests within GitHub Actions using a new YAML file. We named this new YAML file “ci.ml” and titled it “CI Tests and Reports” so it was clear what the purpose of this pipeline is. We specified when to run this workflow, the two operating systems to use and the file’s permissions; which were all the same as the build file. After this we moved onto the actions the workflow must take to automate the tests. The start of this set of actions is similar to those needed for the build. Firstly, we must checkout the repository so that the workflow can access it and set up the Java development environment so that it is able to run the unit tests within the code. Next, we need to run the tests. To do this, we first decided to use JaCoCo which is a test coverage tool to help us to quickly and easily see how good our tests are. Next, the gradle must be run to complete the tests. The argument of ‘build jacocoTestReport’ must be included to incorporate the JaCoCo tool within the tests. Finally, we need to upload the newly produced JaCoCo test result file that was produced from running the tests. We place this under the name of jacoco-reports under a specified file location so it is easy to find. For the checkstyle report, to ensure formatting is consistent, it is very similar. We upload both the checkstyle report for the core of the repository and the desktop to the repository for the developers to find and use.

Finally, we implemented the release of the builds using a final new YAML file named “release.yml” and titled “Automated Releases by Tag” to clarify the purpose of this pipeline. We specified to run this workflow only when a push is made with a tag so it is not after every push and instead only when the developers plan to make a release. It consists of two different jobs. The first job is the building of the executable files which is the same as the first pipeline. Provided this is completed without errors, the next job will run. This next job also has a requirement of the push being a tag to be certain that the pipeline should create a release. We specified for it to use the two required operating systems but this part has permission to write so it can release the files that it has just built. The steps involved in this job is to download the two executable files that have been created in the build job and rename them to suitable names. Finally, it releases these two files together in the release page of the repository, under the given tag. The users can find these to download the most recent working version of the game. It releases both the Ubuntu executable file and the Windows executable file together however these can be downloaded separately by the user depending on the operating system they have.