Continuous Integration Report

Group 14

Tecch Titans:

Bradley Mitchell

Daniz Hajizada

Ellie Gent

Joel Crann

Keela Ta

Leo Crawford

Lukas Angelidis

A summary of the Continuous Integration methods and approaches implemented and why they are appropriate for the project.

Continuous Integration is an essential development tool for projects spanning with multiple developers working on a single collaborative code unit or project development space. It helps to manage and integrate code with each action or integration within a collaborative work space verified by automated building and testing.

Firstly, as one of the main reasons for implementing continuous integration within our project, we established an automated build process to try and make the testing of the project as it progressed a much easier feat. This was made possible through the generation of both test reports and code coverage reports for analysing the validity, results and range of tests covering our code vs the amount of possible tests we could have implemented.

This was ever more useful as it allowed the reports to be cached and compared as the project developed, so we could clearly and frequently monitor our progress. We utilised automated testing by having workflow actions inbuilt into our GitHub repository. After each commit, the CI pipeline runs an automated set of tasks to check that the code and tests still run as correctly intended.

There is also an automated building task to generate both the executable and any version releases, given by a created version tag, that are made for the project. Once again, this is so that progress and issues that could have risen along with the project could be easily tracked and backdated if we ever needed to roll back to a prior successful implementation.

The executables and reports were stored as artifacts on GitHub to further increase the accessibility with tracking the history of changes and the accessibility of current versions that can be easily used for user evaluation or the accessibility to those outside of the project.

It is also to be noted that there is some commented out code towards the top of the document. This could be used to trigger actions when a commit is made with a version tag. This was to separate version-specific actions from branch specific commits. We decided against using this in our project however. Instead, we preferred having actions on every branch so that all team members could see what others were working on and be reminded to pull regularly.

Some of these methods we implemented came from premade code structure packages provided under GitHub workflows. These then required delving into the documentation of the packages and then understanding and learning how to tailor them best to our project. For example, we were then able to tailor when the workflow tasks were run and actions were produced. Once again, for example, here we chose to run them for all commits and pull requests under every branch.

Overall, our continuous integration methods of automated building, testing, and version tagging have allowed us to easily maintain code quality and both track the progress of our project and effectively manage our errors.

A brief report on the actual Continuous Integration infrastructure set up for our project.

The Continuous Integration system we set up uses Java CI with Gradle that had some initial generation from a GitHub workflow example. This generated the GitHub workflow Gradle yml file under the main code for the repository. The infrastructure was designed with the overarching condition that every GitHub action made was built for every push on every branch or every pull request, to merge to master, that takes place amongst the repository. Smaller merges from master into sub branches are still picked up by the GitHub actions as a push will still need to be made for everything pulled into that branch.

Following this, actions were made sure to be built under the correct Linux and Java versions and gradlew (Gradle for windows) was set up for actions using the most appropriate version. The Gradle wrapper under gradlew was set up to invoke its declared version while running all tests following that specified by the overarching build.gradle file alongside JaCoCo. Each package for the project was made sure to be able to access its directorial requirements and a task was built specifically to generate only the relevant JaCoCo report files.

Gradle invokes and runs all tests on a branch with each push or pull request to check whether they all pass and JaCoCo automatically generates both a test report and coverage report for the code alongside this. These are produced in the local build files and are also created as artifacts on Github. Gradle also invokes the build of the desktop distributable executable jar file to be built to the local Git repository and also to be stored as an artifact package on GitHub. Much like the JaCoCo reports, the distributable file is generated for each action made which for both makes it very easy to find and verify the status of the tests implemented and the executable file that is generated at every committed stage.

The artifact generation has been very useful for the debugging of our project as it allowed us to find out which version of the executable, built and committed has not broken and both runs and functions as intended for any given action. We were also able to trace the refactoring of code that initially broke our JaCoCo reports and their generation.

If all tests do not pass under a branch or an executable file cannot be generated with the given code, the workflow action will fail updating the status visible under the GitHub actions tab (and the status badge mentioned later below) and this also sends an email to the user that committed the action as soon as all workflow tasks have been run for the given commit.

Penultimately, an automatically generated versioning system was created with Gradle such that workflow actions made with a version tag are then set up to make a GitHub release with the code base and the generated distributable jar file attached as packages.

And, finally, The CI workflow also has an auto badge status implemented and linked to the main code repository’s README file which displays the workflow task status of the last action committed displaying whether all tests pass and a distributable can be generated.