UNIVERSITY OF YORK

DEPARTMENT OF COMPUTER SCIENCE

Continuous Integration Report

Group 20

Formerly Group 16

Group Members:

Group 16

Charlotte MacDonald
Hollie Shackley
Luis Benito
Kaustav Das
Sam Hartley
Owen Gilmore

Group 20

Leuay Dahhane
Max Irvine
Sam Butler
Flynn Gadsden
Jacob Wharton
Billy Moore

Our Processes

Our continuous integration (CI) approach follows the core principles outlined by Martin Fowler, emphasising frequent integration, automated builds, and comprehensive testing, to ensure a robust and reliable software development lifecycle.

Frequent Integration was employed by ensuring developers would commit and merge their code frequently; this helped expose integration issues early on and hence reduced the time and effort needed to resolve conflicts and bugs.

We made use of a Branching Strategy that included feature branches and pull requests, these feature branches were only merged into the main branch after passing CI checks. We also made use of Jira to keep track of features, bugs and other requirements for the project. This allowed us to have all branches link directly to a feature/bug within the KAN Board, where once a Feature/Bug was moved into ‘In Progress’ it would automatically generate a branch linked to the Feature/Bug, allowing for effective tracking on progress and enforcing ‘blocks’ on features that required another feature to be implemented.

We employed automated testing throughout the project, designing a comprehensive suite of unit tests to verify the functionality of individual components. These tests were run automatically within the CI pipeline, utilising Jacoco to supply reports on the tests and their coverage of the project. This allowed us to identify areas of the codebase that were untested and implement new tests to ensure a significant coverage of all areas of the codebase.

We also decided that static code analysis was extremely important to ensure the code was understandable and regularised. To achieve this, we decided to use Checkstyle automatically during the build process to report any violations. As opposed to designing our own Checkstyle template, we opted to use the widely popular Google Java Style, it was very feature rich and fitted our needs without any change needed.

To ensure the CI pipeline could stay informative, we opted to generate and store artifacts that we deemed would be useful feedback on a build. The JacocoTestReport was added to show test results and coverage; checkstyle report was added to display any areas of lacking code quality and the desktop Jar File so the build could be run swiftly on your own computer. We also implemented email notifications on the Github Actions in the case of a build or test failure, allowing a quick response and resolution.

We chose the CI methods and approaches to appropriately fit this project, as they ensure early detection of issues, maintainable code quality, and a smooth integration process. This all allowed us to focus more on feature development, and less on manual processes which would have impacted the efficiency of our development cycle.

The Configuration

We leveraged Github Actions for our continuous integration infrastructure, as it offers a powerful CI platform that integrates seamlessly within our Github Repository, which we all had significant prior knowledge of using.

The CI Pipeline Configuration:

- Workflow Definition – we defined our CI pipeline in the '.github/workflows/gradle.yml' file. The workflow was triggered on pushes and pull requests to the main branch, and tags following the semantic versioning pattern (e.g. v1.0.0).

- Build and Test Steps – The workflow included steps to check out the code, set up the JDK, build the project, run tests with JaCoCo coverage, and perform Checkstyle analysis.

- Artifact Upload – after the build and test steps were run, the artifacts generated, including JAR files and reports, were uploaded to the GitHub workflow for easy access and review.

Steps in the CI Pipeline:

- Checkout Code: Uses the 'actions/checkout@v4' action to clone the repository.
- Set up JDK: Use the 'actions/setup-java@v4' action to install JDK 11.
- Build and Test: Uses the 'gradle/actions/setup-gradle@v3' action with the argument 'build jacocoTestReport' to build the project, run tests, generate coverage reports and run checkstyle reports.
- Upload Artifacts: Uses the 'actions/upload-artifact@v4' action to upload JaCoCo coverage reports and Checkstyle Reports.
- Build for Distribution: Uses the 'gradle/actions/setup-gradle@v3' action with the argument 'dist' to build the project optimized for distribution, this was done so on release the downloaded file is optimized for release.
- Upload JAR File: Uses the 'actions/upload-artifact@v4' action to upload the JAR file.

Release Management:

- Automated Releases: There is a second job in the workflow specifically for creating releases, this job downloads the build artifacts, renames the JAR file to match the release tag, and publishes the release using the 'softprops/action-gh-release@v2' action.

Error Handling and Notifications:

- Error Reporting: Any errors in the build, test, or analysis steps are reported in the GitHub Actions interface, providing detailed logs to help diagnose issues.

- Notifications: The GitHub Project was configured to notify any contributors of any build failures or issues, ensuring a prompt resolution.

By using GitHub Actions, we benefitted from a robust and integrated CI environment that supports our development workflow. The pipeline ensured consistency in testing, providing confidence in the quality and stability of the game.