Assessment 2: Brownfield Development

Continuous Integration

Cohort 3 Group 9

Oliver Barden

Connor Burns

Sam Goff

Milap Keshwala

Lewis Mitchell

Jeevan Singh Kang

Harry Thomas

Titas Vaidila

Approach

Our team used an agile, scrum-inspired methodology which necessitated a robust continuous integration system to manage frequent code merges from eight different developers.

We implemented GitHub Actions because it integrates directly with our version control repository and provides immediate feedback on the health of the codebase. This approach was chosen to mitigate the risk of build failure ensuring that broken code can be identified before it can be merged into the main branch.

We are also pushing changes to the repository as often as possible to reduce issues such as difficulties merging. Frequent pushing with small changes will also leverage the automated testing system to quickly highlight errors, which will reduce the amount of time spent hunting for issues in large chunks of added code.

Additionally we are communicating when making changes to the codebase to ensure they do not overlap, reducing the risk of two developers wasting time trying to implement the same features at once, or making changes to the same area of code at the same time which could lead to problems with merging.

Infrastructure

The CI infrastructure consists of two primary pipelines defined in a YAML configuration file called gradle.yml.

As mentioned, one of the tools we are using to help with continuous integration is GitHub Actions, specifically the “Java CI with Gradle” premade action. This builds the project each time a change is pushed to the repository, and lets us know if the change introduced an issue (the build failed).

To test the newly added action, we ran it on a version of the project we knew should build properly. The action consists of two jobs - the first one, build, worked out of the box, but the second, dependency-submission, did not. The error it threw related to dependency graphs which were disabled for our repository. To fix this a setting was edited to enable automatic dependency submission, this created another action to carry this out, and the dependency-submission job now passes.

Our infrastructure also utilises a dedicated headless backend subproject. This is so the CI server can execute tests and compile code without the need for a graphical user interface which would otherwise cause the builds to fail.

The process also automatically triggers our custom generateAssetList task. This ensures the assets.txt manifest is programmatically updated during the build, preventing errors related to missing or unlinked game resources.

The build and test pipeline works as follows:

- Trigger: Automatically triggered on every push or pull request to the main branch
- Input: Java source code, Gradle wrapper, and project dependencies such as assets
- Process: The pipeline executes ./gradlew build which compiles the code and runs the JUnit test suite.
- Output: A build status (pass/fail) and a JaCoCo XML code coverage report which allows us to monitor testing progress through GitHub.