Engineering 1

Group Assessment 2

Continuous Integration Report

Cohort 2

Team 11

Continuous Integration Methods and Approaches

Our approach to Continuous Integration is to use two separate workflows for testing and building, both running whenever code enters the development or master branch of the project by merge or pull request.

This is well-suited to our project with regards to both our development methodology and the size of the project itself. As we are using the Scrum framework, changes to the codebase are small in size and rapidly produced, and should always be ready for a release build. Our CI methodology allows for this as any code entering the master or development branches will be automatically tested for both whether it compiles successfully and whether it passes on all unit tests.

By separating out the continuous integration into two discrete stages, we can assure that code entering the main branches of the project will always be buildable. If a commit causes the project to not compile, it should not be allowed into the main branches as we try to maintain a build that works. The second stage checks whether all unit tests are passing. This test will not stop code from entering a main branch as it's technically still a passable build, but a label on the repository should serve as a warning to any users trying to pull a specific build if it contains bugs.

Continuous Integration Infrastructure

Our approach to Continuous Integration utilises GitHub Actions with two Gradle workflows. Our project already uses Gradle as a build system, so it was the easiest to configure out of all available options. GitHub actions are configured using structured YAML files, which specify what Gradle actions should be run, when they should be run, what Java version they should run on, and in which order.

We use two separate GitHub actions as part of our approach to Continuous Integration which help to ensure code quality. Both run the code on a Linux virtual machine owned by GitHub, providing a development environment similar to what members of the team have so compatibility should not be an issue.

The first action compiles the code with Gradle, for any commits entering the development branch or pull requests into the master branch. This uses the exact same process that developers on the project should already be using to build and compile their copies of code, so there should be no cases of it working on one machine but not another. It ensures that code entering the two branches will always build for whoever is trying to clone it.

The second action compiles and runs a number of JUnit 5 unit tests using Gradle, for any commits or pull requests into the master branch. Like the compilation workflow, this should use the exact same approach that anyone else working on the project should be using to test the codebase. Unlike the compilation workflow, this should not stop code from entering

the branch, but will update a badge on the repository's README.md file which states whether or not the unit tests (and compilation workflow) are passing.

Using these two actions, we can ensure that all code in the master branch will compile to a stable and functional build, or that users are at least visually alerted when something is wrong with the current branch. Whilst not all builds will be significant enough to be classified as releases, all commits in the master branch should always represent an improvement.

Unfortunately, due to the fact that a LibGDX headless runtime has to be created using Mockito, not all unit tests are passing on GitHub Actions. Whilst all tests are passing when the source code is cloned from the repository and run on a dev PC, this is a bug that so far we have been unable to resolve. (My suspicion is that it's either Git LFS messing up some file pointers because it's isolated to a few tests)