ENG1 Group 2 Marlin Studios
Part 2

Continuous Integration Report

SKYE FULLER

JONATHAN HAYTER

ALEXANDER MIKHEEV

MARKS POLAKOVS

FREDERICK SAVILL

HAOWEN ZHENG

Summarise your continuous integration method(s) and approach(es), explaining why these are appropriate for the project.

In our CI system, team members work in Git branches, with the expectation that a branch should not diverge too far from the current “master” or mainline branch. The developers run the test suite locally before pushing, to ensure the correctness of the (tested) code. When a branch is ready for merging, our test suite gets run automatically and results are reported back. In parallel, another team member reviews the code, leaves comments inline, and gives a “ready for merge / changes required” signal.

Once both the automated testing and manual review have approved the code, it is merged into the mainline “master” branch. CI automation then builds the game into an executable JAR, as well as automatically updating the Javadocs on the website. It also does another run of the tests, to prevent the (unlikely) event where the new changes passed in isolation but broke the build when merged onto the mainline (although this is made less likely by keeping branches short and merging often). We aim for an “evergreen” mainline, where the tests always pass.

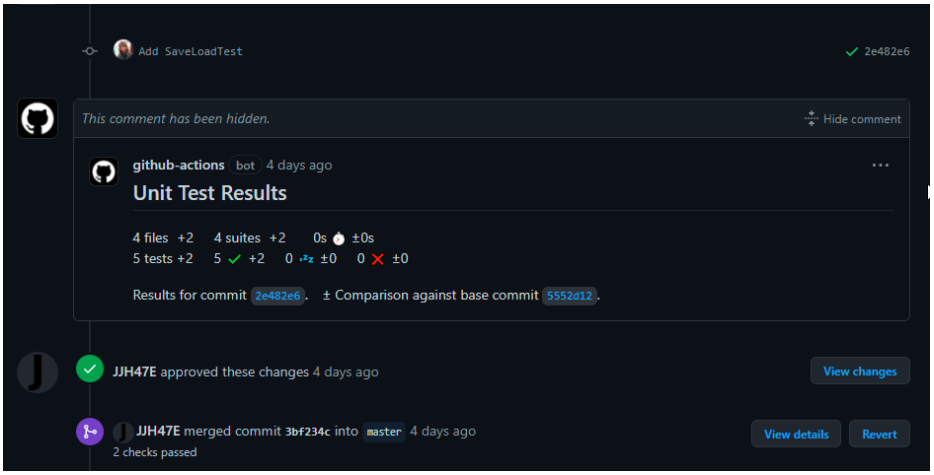
We settled on this approach as it is a relatively low-friction approach - all a developer needs to do is open a pull request, and the infrastructure does the rest, giving a clear “ready / not ready” signal, as well as doing sanity checks once the merge is done. Artifacts (executable JAR and Javadocs) are also built automatically once the code is landed on the mainline, and uploaded to the repository’s artifact storage.

Give a brief report on the actual continuous integration infrastructure you have set up for your project.

Our CI infrastructure is based around GitHub, GitHub Actions, JUnit, JaCoCo, and Coveralls.

When code is ready, it is pushed to GitHub and a pull request opened, with a review request to another team member. This kicks off a GitHub Actions run, which compiles the code, runs the unit test suite (discussed in Testing2), and calculates the code coverage using JaCoCo, reporting it to Coveralls. It then reports this back in a comment on the pull request, along with a green/red “status check” of whether the tests passed. Meanwhile, another team member can review the code, leaving comments inline, and submitting a final “approve / request changes” response. Once both of the above are green (passing tests, and approving review by another team member), the code is ready to be merged into the mainline. This kicks off another set of GitHub Actions runs, which runs the tests on top of current master (as there may have been changes on master in the meantime, although we aim for an “evergreen” master i.e. one where the tests always pass), as well as building an executable JAR and updating the online Javadoc (which is hosted on GitHub Pages and thus can be automatically updated).

We settled on GitHub Actions as it is free, with a generous quota of runner minutes, and integrated into GitHub which we were already using (placing a .yaml file in a special directory is sufficient to enable Actions). The peer review system is a relatively low-friction way of ensuring that at least two people are familiar with each area of the codebase, as well as a simple way of checking for both “howlers” (obvious mistakes) as well as logic errors that a second pair of eyes can spot. JaCoCo works natively (with a little configuration) with our test suites. We also used Coveralls (a free online service for tracking code coverage). Unfortunately Coveralls expects a report in *.lcov* format, which JaCoCo cannot natively emit, so we had to use a Gradle plugin to report results to Coveralls, which was somewhat flakey and difficult to troubleshoot. Given more time we may have used an alternative service (such as Codecov) or written our own action to publish coverage results, but owing to time constraints we stuck with Coveralls.



An example of automated and manual feedback on a pull request.