Continuous Integration Report

Cohort 2 Team 4 (THADJAM)
Abdul Fofanah
Matthew Holleran
Toby Watchorn
Daneena Roydean
Arwen Minton
Jess McKerill
Henry Bambrough

Methods + Approaches:

- **Daily Integration** - For our project, we integrated regularly every day that the code was updated, sometimes even multiple times a day. This meant that the code was integrated in small chunks. This reduces the risk of any issues in the code when it is integrated, and if there are any problems, then keeping the integrations small means that these issues will be easily resolved when compared to larger, more difficult chunks, which are avoided by keeping the integrations small.

- **Automated Build** - Another thing our project used was automated builds that ran automated tests to catch any issues in the software. These include the following:
    1. **Unit tests** - Fast and small tests in our code to verify the small section works.
    2. **Integration tests** - tests that ensured separate parts of code worked together.
    3. **System-level and gameplay tests** - Larger tests that ensure the entire project worked as expected. We used this much more towards the end of the project. These tests ensure that the whole project works as expected, and the smaller parts of the unit tests also work well integrated.
- The build we used caught bugs in the project early on, which prevented them from having a large effect later on. Firstly, the unit tests caught small bugs, and any issues in the system-level and gameplay tests were not major, as all of the smaller builds ran fine, so it was simple integration issues.

- **Built the application using Gradle command "./gradlew build" -** the same command can be later used to automate compilation by a CI tool whenever the code changes. This also runs the automated tests, which can run as part of the build. Stops us from releasing the buggy build to the user by mistake.

- **Maintain the functionality of the build -** we prevented allocating time and resources to fix broken builds as soon as possible, and to also figure out why something broke as soon as possible, to stop it from happening in the future. This prevents broken builds from developing and stops them just in case they occur.

- **Keep building fast** - only run the faster tests, such as unit and integration tests, and leave the slower builds for later. This means the fixes can be faster as well as stop broken code sections from being in the branch for too long.

In our project, GitHub Actions acted as the designated integration machine. Each workflow run performs a clean checkout of the repository and executes the build and test process in a consistent Linux environment. This ensures that the GitHub Actions workflow is the reference for whether the codebase builds and functions correctly.

Continuous Integration Set Up in Project:

As we used GitHub to write the code for the project, continuous integration was added using GitHub actions. The continuous integration setup runs every time code is pushed into our main function and is accessed through Engineering-1/.github/workflows/main.yml at main · MattHoll26/Engineering-1.

We **wrote a file in YAML** (yet another markup language) that defines:

- When the code should run (when code is pushed into main)
- Which automated jobs should be running
- The job details - such as what operating system it runs on, the security permissions granted to the job and the jobs actions

- **The Job Steps:**
- **uses**: (downloads the repository's code)
- **java-version and distribution:** both install the right Java version
- **Name, uses, with, arguments, build-root-directory:** all for Gradle to set it up + run tests

Then we **added test coverage** to keep test results so they can be downloaded to examine results: (added JaCoCo + attached it to the report for our build results as a workflow artifact):

- Extended the plugins section by adding:
- Name, uses, with, name + path for the JaCoCo coverage report
- This ensures the workflow artefact is created as a report (to be downloaded + examined by programmers)
- Can help us debug the program for our game by examining and inspecting the report

Then we added **binary artifacts** to produce an output during the build process:

- Added JAR upload to GHA workflow by adding:
- The JAR uploads the name + uses.
- This ensures that deployable output is produced, which ensures our game can be tested by actually playing our game itself

Then we added an **automated code style check,** which makes sure the build abides by the rules of Java. We integrated Checkstyle into the Gradle build to do so:

- Added checkstyle plugins:
- Did this using the id 'checkstyle' and the checkstyleVersion 10.4
- Then used Google style rules to which were fetched automatically in the configurations + dependencies
- This integration ensured that code style violations were automatically detected during the CI build.

Then added automated release tags to ensure we were all programming on the same version. Did this by pushing the new tags into GitHub whenever they were updated.

The workflow was designed to run in a consistent Linux-based environment to ensure reproducibility in the builds across all integrations.