UNIVERSITY OF YORK

DEPARTMENT OF COMPUTER SCIENCE

ENG1 Assessment 2
Group 18 - Octodecimal

Continuous Integration

Group Members:

Izz Abd Aziz

Phil Suwanpimolkul

Tom Loomes

Owen Kilpatrick

Zachary Vickers

Michael Ballantyne

6a) During our project development, we followed several practices to align with Continuous Integration (CI) principles of integrating often, with each integration being verified by an automated build:

- We maintained a single source repository including all of our project files (code, build scripts, assets, etc.). We also made sure our branches were short-lived to avoid hard-to-integrate forks.
    - All group members committed any changes they had to the mainline every day. This helped us stay up to date with what others were doing, and allowed us to regularly verify that the program is building and passing automated tests.
- We created an automated build that is self-testing. This means a single command can be activated to compile, run automated tests, package into JAR, etc. Every commit also runs this command to build the mainline on an Integration Machine.
    - We kept the build fast, it consistently completes in under 5 minutes.
    - Broken builds were generally fixed as soon as possible. We'd allocate someone to fix a broken build quickly, or revert the changes until the problem can be resolved.
    - Automated tests are run on builds of all platforms we intended to create JARs for (Windows, Linux, MacOS) in parallel. This way we test a clone of the production environment.
- Pushes to tags create a release under Releases in our GitHub repository. This makes it easy for users to get the latest executable (download and run the latest build).
    - Users can easily try the latest versions of the game, as they are in an obvious location.
- We made it easy for everyone to see the state of the project by having automated builds upload their results to our shared GitHub Actions in our repository.
    - GitHub Actions marks builds with a green tick if they succeed and a red cross if they fail. This gives a clear and efficient indicator as to the state of the project.

Our project has one pipeline. It's triggered automatically on any push or pull request to the 'main' branch of our repository, allowing us to see the state of the build after every major push/pull. Since our releases used the standard software versioning format "major.minor.patch", the workflow is also triggered on any push to a tag named "v#.#.#" where each # is an integer, allowing a release to be automatically created when needed.

- The pipeline's inputs are the source code (including tests), and the branch/tag name of the push/pull
- When triggered, its outputs always include one JAR, testing report and testing coverage report for each OS (Windows, MacOS, Ubuntu) and two checkstyle reports (one for core, one for tests). It will also output a dependency graph, which shows all ecosystems and packages the code depends on.
- When triggered by a tag push, the pipeline outputs everything above, and also outputs a JAR for each OS and a zip and tar.gz of the GitHub repository. These outputs go to a separate location to the ones mentioned in the previous bullet point (specifically our releases page).

JARs and testing reports were created for all OSes, even though requirement FR-Device only requires us to make the game run on Windows. This is because, based on the brief's context, this game may eventually be used at virtual open days, or students with varying OSes may wish to play it at home after open days. Our pipeline allows us to ensure the game is working cross-platform, in case of these circumstances.

We wanted testing reports to make sure our tests passed on all OSes, and help review our test coverage as the project progressed. Checkstyle reports don't require the code to run, so these didn't require different OS variants, this report allowed us to ensure we were following Google style guidance, aiding communication between our developers.

The pipeline outputs zip and tar.gz files for tag pushes, as releases should contain the source code, keeping the game open-source.

6b) Our CI pipeline was implemented using a GitHub Actions (GHA) workflow. We chose GHA as it's an appropriate CI platform for our purposes, allowing us to automate our build, test and deployment pipeline. It was preferred to use this integrated CI as our repository is already hosted by GitHub, so it would help our consistency to keep everything in one place.

The workflow was written in YAML, a human-readable data serialisation language which GHA requires to be used to define workflows. This workflow can be seen in the file .github/workflows/gradle.yml in our repository (https://github.com/eng1team18/HesHustle2/blob/main/.github/workflows/gradle.yml). After the workflow has been triggered as described in 6a, the results can be found under the Actions tab of our GitHub repository (https://github.com/eng1team18/HesHustle2/actions). This includes an overall tick or cross for every run, showing clearly whether the automated build failed or not.

Our workflow's first job ('build') uses GHA's matrix strategy functionality to run the job in parallel on the latest version of each major OS. A matrix strategy is a GHA feature that lets you use variables in a single job definition to automatically create multiple job runs, such as testing code on multiple OSes at once. This was a useful feature as it helped us keep the build fast, and avoided having large amounts of duplicated YAML code.

The 'build' job first checks out the repository and sets up JDK11, then it sets up and builds the Gradle accordingly with the build.gradle settings in the repository. If this is successful, it will upload the executable JAR as a workflow artifact, which allowed us to easily play the latest version of the game at any time during development (without manually building the Gradle). It will then upload the JaCoCo test coverage report as a workflow artifact, when downloaded and unzipped, this meant we could see a breakdown of our test coverage after every push/pull. The JaCoCo report shows coverage values (numerical and percentage) for every class, including showing our number of missed instructions, branches, methods, lines, etc. This was regularly used throughout the project to help us write tests where they were most needed. The workflow then adds the test report to GitHub using dorny's test reporter, this report can be found in the jobs list of this workflow run's summary and provides a quick, easily accessible overview of how many tests passed and failed [https://github.com/eng1team18/HesHustle2/actions/runs/9199812993/job/25305292778]. Only on ubuntu (As there's no need for duplicates for each OS) the workflow will also automatically upload a Google checkstyle report for each of the tests folder and the core folder. This allowed us to keep track of our code style violations, and fix these to create consistency across our codebase

The workflow's 2nd job 'dependency-submission' runs in parallel with 'build', generating and submitting a new version of the dependency graph to our repository's insights page, allowing us to track all packages the codebase is reliant on (https://github.com/eng1team18/HesHustle2/network/dependencies). This was useful for keeping track of our licensing and helped us keep an eye on when packages were updated during the project.

The final job in the workflow, 'release', only runs when the pipeline was triggered by a tag push of name 'v#.#.#', as explained in 6a. It creates a new release in our repository's Releases page, including the additional output artifacts (JARs, ZIP, Tar.gz) mentioned in 6a (https://github.com/eng1team18/HesHustle2/releases).

Any workflow steps that only needed to be completed on one OS were carried out using MacOS. This is because macos-latest is the OS which builds our Gradle fastest (similar speed to ubuntu-latest, much faster than windows-latest). Also, the macos-latest has the highest specifications in private repositories,with 3 CPUs instead of 2. Information on the specifications can be found here:
https://docs.github.com/en/actions/using-github-hosted-runners/about-github-hosted-runners/about-github-hosted-runners#standard-github-hosted-runners-for-private-repositories