Continuous Integration

Cohort 4 Group 6

Javengers

Braithwaite, Max
Faruque, Amber
Fu, Zhuoran
Kocaman, Melike
McDermott, John
Rissen, James
Scott, Charlotte

Continuous Integration Methods / Approaches:

For our project, we have adopted a Single Source Repository approach for continuous integration (CI), where all source code, build scripts, configuration files, and assets are versioned in a single GitHub repository. This ensures that all team members have access to the latest versions of the project files, reducing confusion, merge conflicts, and the risk of overwriting changes. Developers primarily work on short-lived feature branches under main to simplify integration and reduce the complexity of merging.

To make CI more manageable, we have broken tasks into smaller, focused units of work. This encourages frequent commits, allowing team members to stay up to date and resolve conflicts early. Smaller tasks are easier to test and integrate, improving overall development efficiency.

We have implemented automated builds using Gradle. Whenever code is pushed or a pull request is opened against the main branch, the CI pipeline is triggered automatically. The build pipeline compiles the project, runs automated unit tests, and packages the application into a JAR file.

The inputs include Source code changes, Gradle build scripts, configuration files, and unit tests. The Outputs include Built JAR files, test reports, code coverage reports (JaCoCo), and style reports (Checkstyle). Finally, the triggers of these events include Pushing to main, pull requests, and creation of version tags.

The automated pipeline also includes self-testing. Unit tests are executed on every build, and test results are reported in real-time, allowing the team to quickly identify and fix failures. Code quality and coverage are also measured using Checkstyle and JaCoCo, ensuring adherence to coding standards and comprehensive test coverage.

Our team uses GitHub Actions as our designated integration tool, automatically running test builds from our code. This approach ensures that every change is tested in a consistent and isolated environment, reducing the chances of a local machine configuration masking errors. GitHub Actions displays the results of the builds, such as whether they passed or failed and the time it took, making it easy for team members to quickly see the current state of the project.

Our team also aimed to keep the build as fast as possible. To do so, we implemented Parrelisation which allows us to build and test Linux, Windows, and Mac distributions simultaneously. By running these builds in parallel, we significantly reduced the overall build time, ensuring quicker feedback for developers and allowing faster integration of changes across different platforms, and cross-platform validation.

Finally, we automated the release process. Once a version tag is pushed, the CI pipeline collects the built artifacts from all operating systems and publishes them to the GitHub Releases page. This ensures that users and team members always have access to the latest stable version of the project with minimal manual intervention.

Continuous Integration Infrastructure Report:

The project implements a fully automated continuous integration (CI) pipeline using GitHub Actions, defined in the workflow file gradle.yml. GitHub Actions was selected due to its native integration with GitHub, cross-platform runner support, and built-in capabilities for parallelisation, artifact management, and automated release deployment, eliminating the need for external CI services.

Workflow Triggers: The pipeline is configured to trigger automatically on push events to the main branch, pull request events targeting the main branch and semantic version tag creation (for example v1.0.0). This ensures that any changes to our code are validated and that official releases can be created automatically to the releases page.

Build and Test Automation: The CI pipeline uses gradle as the build automation tool. All tasks are executed via the Gradle Wrapper (gradlew), guaranteeing a consistent Gradle version without requiring preinstallation on runners. Gradle is used to compile the project, execute automated tests and build the application JAR file. Tests run automatically as part of the pipeline, and the workflow is configured so that failed tests do not block subsequent steps, ensuring complete feedback and generation of all artifacts for each CI run. Key build tasks include:

- **./gradlew :lwjgl3:jar** → builds the JAR artifact for distribution
- **./gradlew :tests:test** → executes all unit tests with.
- **./gradlew :tests:jacocoTestReport** → generates JaCoCo code coverage report
- **./gradlew :core:checkstyleMain :core:checkstyleTest** → performs Checkstyle analysis on main and test code

Tests are configured with **(continue-on-error: true**) for the testing step and (**-- continue**) for Gradle tasks, ensuring that subsequent reporting and artifact upload steps execute regardless of test outcomes.

Parallel Multi-Platform Builds: To improve build efficiency as well as ensure cross-platform compatibility, the CI pipeline uses a matrix strategy to run builds in parallel across **{Ubuntu-22.04, Windows-2022, macOS-15}**. Each platform executes the same build and tests steps independently, significantly reducing overall build time. Furthermore, platform specific issues can be identified, for example if tests are passing on one OS, but failing on another.

Artifact Management and Automated Releases: All generated artifacts are uploaded using **actions/upload-artifact@v4.** These include:

- JAR files (**My-Maze-Game-${{ matrix.os }}**)
- JaCoCo coverage reports (**jacoco-report-${{ matrix.os }})**
- Checkstyle reports (**checkstyle-core-${{ matrix.os }}**)
- Test result reports (**test-report-${{ matrix.os }}**)

Artifacts are preserved for subsequent jobs or downloads, ensuring traceability and reproducibility of builds.

A dedicated release job is triggered only when a version tag is pushed (**startsWith(github.ref, 'refs/tags/')**). This job downloads the platform-specific JAR artifacts from the build job using **actions/download-artifact@v4**, then renames each artifact consistently, for example for Linux (**maze-ubuntu.jar**). Finally it publishes all artifacts to the GitHub Releases page using **softprops/action-gh-release@v1.** This fully automates release generation, providing reliable, versioned distribution files with minimal manual intervention.