CI 2

Group 2 - Vikingz

Damian Boruch

Tommy Burholt

Elliott Bryce

James Butterfield

Ren Herring

Zhenggao Zhang

Sharlotte Koren

Continuous Integration

Methods and Approaches

Our goals for using continuous integration in our project were

- To ensure our program can work on all operating systems, (Macos,Ubuntu,Windows)
- To make it easy to check if new code is going to create issues
- Ability to more easily release and check code coverage reports
- Checking for dependencies vulnerabilities in the code base

Our approach was to have a matrix continuous action (CI) for testing and building jars:
With pipelines of windows, macos, linux based runners ensure all operating systems would be able build the CampusTycoon jar file. Which includes checks for if our code passes tests and builds correctly. Ensuring our game will run on any machines without difficulties. This generated outputs of successful game jars for each operating system which can be used to test each commit.

This can be accomplished by inputs using gradle and github actions to make every runner build and upload a jar that can be used to verify each commit passes both unit tests and jar bundling.
This would be triggered every time someone pushed to the main git branch either directly pushing or via a push request, This CI could also be triggered if someone made a new release of the game to allow jars to be built and ready for upload to a tagged release.

To make it easier for us to create releases we choose to use a simple linux runner that can then be used to upload one of the jars created by the matrix runners to our github release page making it easier to create releases that are tested completely.
This can be done by inputs to tell CI to download the latest jar created by one of the matrix runners build jobs and upload it to github releases.

To allow us to more easily check code coverage reports we added a job linux runner with inputs using gradle to output a newly created jacoco report which would then can be uploaded using github actions to our latest commit. Allowing us to have the latest update on our current code coverage. This would be done in gradle by adding a jacoco report generation task to our build gradle file.

For our goal of using CI to check for dependencies vulnerabilities we used github's dependabot to check if our dependencies had vulnerabilities. This was done by outputting a dependency-submission file which is created by input to gradle build tools, dependency-submission then can be uploaded to github dependabot which could then check if the listed dependencies had vulnerabilities outputting all our current dependencies along with any issues with them.

Our CI Infrastructure

Our implementation of our approach to continuous integration (CI), was to use github actions with a single yaml file with jobs for building and testing jar files, uploading releases, code coverage report upload, dependency submission checks. This GitHub Actions YAML is triggered on push, pull requests and on newly created release tags.

Building and testing jar files:
We used a matrix runner approach for this to allow us to test our code being built into jars on multiple platforms, for which we choose github latest non beta runners which at the time was windows 2022 (testing for window), Ubuntu 24.04 (Testing for Linux) and Macos 14 (Testing for mac). This gave us a range of platforms to ensure users could run our game.

For the actual build process we used steps of setting up java 17 (version used by libgdx and our project) and gradle 7.5 (the version we use in development). After ensuring that gradle could be used by making it executable the build job would run a test build on the project before producing the final release jar by that specific platform runner.

Then the last of part build job is to upload each runners newly created jar assuming they didn't fail previous test to github actions, allow us to to see the changes in our game for each commit making it easier to track down bugs

Uploading release:
This action is only done if the github action workflow was started by a new tagged release before running. Which then the action is handled by waiting for all runners to successfully complete building and uploading a jar to github action's artifacts. Which then a simple ubuntu 24.04 runner is then used to download and upload one of the created jars (we used the linux runner) to github this is because all successfully passed built jars function the same across different operating systems, The jar is then uploaded to the tagged release.

Code coverage reports:
Code coverage uses similar initial steps as building and testing of the jars but since we don't need to actually build or test the game here. We use a single simple linux runner (ubuntu 24.04) to run java 17 and gradle which runs our coverage report generation task, in which the report is then uploaded to github action's artifact store, to allow us to check per commit on our current progress in testing coverage.

Dependabot scans:
Since we are using external java libraries import using gradle it's a good idea to have dependencies scanning, that's we have dependabot enabled on our organization which works by us running a simple linux runner job that runs a gradle task to produce a list of our current dependencies which is then uploaded to dependabot which checks if the list has any vulnerabilities.