Continuous Integration Report

Team 4 - Undercooked

Fin Cochrane
Sehran Ahmed
Sam Davis
Hamza Salman
Owen Thomas
Zhenyi Xu

A)
Continuous integration is the process of testing every successful build at every interval in a project so any fault can be traced back and different parts of the project can fit in together. We made sure to set up a branch with access to all participants where they could commit any changes made and made tests to make sure that all the requirements were being met. This allowed us to keep track of the progress being made and see which stage everyone was working on. Using continuous integration allowed our project to flow smoothly without major setbacks. We were able to work on different tasks and requirements at the same time without the code conflicting since we were making commits after every change and testing the project at every build. We were able to keep track of every single commit via Github which meant if a requirement or a test had failed we could trace it back and see the changes made at every step.
The pipelines used in our projects were as followed:

- We had a build pipeline which made sure that every time a commit was made Java would build properly with no issues, any issues would be displayed on the console in Github actions. When a commit was made Github would run the test provided with Java CI Gradle to give us a successful build outcome. This pipeline would be triggered by pushing a change on the Github repository.
- We also implemented a Jacoco test coverage pipeline which would attach a test coverage to the build on every successful run which meant that we didn’t have to use external tools to get test coverage. This report came attached with the build on Github and was accessible for everyone to download as a zip file. This was also generated automatically when a change or a commit was pushed.
- For testing and efficiency reasons we added binary artifacts to the build which made a lib.jar executable file which was also accessible to all members of the group. This executable file would download and run which meant after every successful build we had a working product ready for testing, and if the build was unsuccessful it would list the errors in the console instead of building the lib.jar executable which saved us time in the long run.
- For consistency purposes we also added automated code style checks which made our code reliable and consistent throughout. After every successful commit the code would get tested for Google style coding and comments which made sense since we were working in a large group meaning everyone had an easier time reading and editing the code. This was also done at every commit and change being pushed onto Github.

This allowed us to make the coding more efficient and optimal. We were also able to save a lot of time as we were able to have more than one person on the coding. By having continuous integration we were able to keep the code at a very high working standard without worrying about conflicting code. Therefore by having multiple people on different tasks we were able to combine the code at regular intervals making sure the all the requirements were met without having to worry about integrating two sections of the game at the end where it could cause multiple problems which would be hard to debug as there would be no testing or record of where the problem(s) occurred and who was responsible for it. This meant that we could have a better grasp of the progress made and we could clearly see which areas needed work so everything was being updated accordingly.

B)
For the projects' continuous integration infrastructure we chose Java CI with Gradle as it provides some basic files and tests to start off with, which really helped us get started. This tool also provides us with test pipelines after every successful test or build. We used Github Actions to present the pipelines which helped us visualise the progress and obstacles of the project. This meant that anyone could download the latest version from Github. Having test coverage and builds meant that we could attach reports and binaries to builds as well which allowed us to track every change.
To implement the pipelines the group used Github Actions with Java Gradle CI to build and attach artifacts to the build. Each pipeline was set up as followed:

- The build pipeline was implemented as a simple Java Gradle YAML file which would use build with Gradle and Java to test the build, we used JDK 11 as it seemed most fit for our needs. This was the most basic pipeline to set up as it just required a Gradle YAML file in the workflows folder.
- The Jacoco test coverage pipeline was implemented by adding Jacoco as a plugin in the build.gradle file and by attaching the file as an artifact to the build by using steps in the workflow YAML file. The path of the file was directed to the tests folder which meant that after every successful build, Github Actions would allow the user to download the test folder and the test coverage as an artifact.
- The binary pipeline and the executable lib.jar file was also implemented in the same way, but since Gradle already made executable files we just had to attach it to the build as an artifact using the workflow YAML file. We had to set the path in the gradle.yml file where the executable was being made and Github Actions would attach it to the build ready for download. The attachment would be implemented in the file by just adding another step to the jobs section and the step would specify the name, detail, and the path of the file.
- For the Automated code style check pipeline, we were required to implement a few more steps such as importing Checkstyle: a tool provided by Java, as a plugin in the build.gradle file and importing the rules as well so they could be followed by the code. All of this was done by importing the plugin, defining the version; we used 10.4 as it was the latest option at the time; and then applying the rules using the 'com.puppycrawls.tool' plugin. After all of this was implemented we had to attach it to the build just like before, this was done in the same way by declaring the name, description and the path in the gradle.yml file in the workflows folder.

After all of the implementation, we would get a build, after a successful commit, with test coverage, an executable JAR build file and automated code style checks attached as artifacts. This allowed us to have everything ready whenever a successful change was made which helped the group stay on track and we had a ready product at hand all the time. This method helped the group stay professional and organised.