Engineering 1
Group Assessment 2
Continuous Integration Report Document

Cohort 2 Team 17
Gnocchi Games

Summary

Our continuous integration approach consists of multiple processes.

Firstly, we have a series of checks that are run on the code base. These checks include building the code, running tests against the code, and checking the conformity towards the Google Java Style Guide.

In addition, our repository is set up with a branch protection rule on its main branch. To be able to push changes to the main branch, a team member must open a pull request, from another branch, into the main branch. Then, the pull request must pass all the checks mentioned above. Finally, the changes can only be merged to the main branch once at least one other team member has reviewed and approved the changes made. These protections help ensure that the main branch only contains working code, which others will be able to safely develop upon.

Once the changes have been merged, they undergo the same checks once again for good measure. Additionally, another CI process is started: JavaDocs, UML diagrams and test reports are generated in this process, and pushed to a separate repository. This repository holds our team's updated version of the repository, and I will go further into detail in the next section as to why we have set up a separate repository.

As mentioned above, one of our CI checks consists of building the code. However, this check actually does a little bit more than that. Once it has built a JAR file containing the executable game, it uploads it as an artefact. We have chosen for multiple reasons. Firstly, it gives us some sort of "history" of how the game evolved: a new executable is uploaded on every change to the main branch. It also enables us to review pull requests. As this check and upload is performed on every push to a pull request, a reviewer can just download the executable, test it out, view the file changes on the pull request page, and from there give his review or approval. This saves time and effort involved in pulling the changes from the development branch, building them and testing the changes out. It also eliminates the risk of untracked files or human errors using git influencing the resulting executable: the JAR file uploaded by the CI process is built in the same conditions as the JAR file to be delivered to the client.

When it comes to the Unit Test check, we decided to add that check to make sure that if a test fails, the branch in which the unit test fails is only merged once either the test (if it is broken), or the unit it is testing are fixed. This helps keep the number of bugs relatively low. Additionally, as we have added unit tests towards the end of the project, this check helped us make sure that we didn't just add tests: we had to fix the errors we had made. An example of this would be in the implementation of saving the game. A unit test helped identify a bug which saved all floating branches as simple branches. Another identified that the "stamina usage" statistic was saved in the "stamina regain" statistic field.

Finally, the Checkstyle CI process is responsible for checking that the code complies with our chosen code style standard. This was decided to ensure that the code base had a consistent style throughout. Adding this check was actually a lot of work, as once this check was properly configured, the whole code base had to be updated and refactored to conform with the Google Java Style Guide. The code base, as it was given to us at the start of the project, did not follow any code standard and the style was not consistent throughout.

Report

For our continuous integration infrastructure we have chosen to use GitHub, and its GitHub Actions. This choice seemed logical as the previous team already used GitHub to host their repository, and GitHub is a popular tool that our team had also used in our previous project. Furthermore, GitHub Actions are accessible from the same interface as the repository, helping our team focus their attention on other aspects of the project, rather than having to check the CI runs on another provider (Circle CI, Travis CI, etc.) in the case of failed runs.

To configure the actions, YAML configuration files were written inside the ".github/workflows/" directory of our repository, with a one-to-one correspondence between *Workflow* and YAML file. These YAML configuration files contained the name of the *Workflow*, the events that trigger its execution, and the steps to follow when executing.

All of our CI processes use these configurations. More precisely, they are configured to run in GitHub runners, hosted by GitHub. All *Workflows* were configured to run on "ubuntu-latest", which at the time of our project is Ubuntu 18.04. This seemed the most appropriate as our project runs on Java, and has very little OS-specifics: it runs on the Java VM. Additionally, we have a quota of runs we cannot overrun in our free tier of GitHub services. Running our *Workflows* on Windows or macOS would have made less economical sense, as these are respectively 2x and 4x more expensive to run, without any major benefit.

Three of our *Workflows* were configured to run on each pull request onto the main branch, and again on each push to main. These same three *Workflows* were also set as required checks in the branch protection rules for main. The three are as mentioned in the previous section those that: build the project, run unit tests, and check style conformity.

Our project uses the Gradle build system. Gradle helps us manage the dependencies of our project. It however also helps us in our continuous integration process. Gradle has many plugins and tasks that it can run. Firstly, it makes it very easy to produce an executable JAR file. Secondly, it has a plugin which allows us to very easily run JUnit unit tests as a simple Gradle task. Similarly, there is a Checkstyle plugin, which uses the Checkstyle tool to check the conformity of the code base towards the configured code standard.

Another reason for which Gradle is a good tool is that there are ready-made GitHub Actions that help improve the build times. These Actions cache the dependencies that Gradle needs to build our project, preventing the need to re-download them on every *Workflow* execution. This saves time, and allows the cache to be shared between different *Workflow* runs and tasks.

Finally, the last *Workflow* used on our repository is the one used to update the documentation on our website. It was eventually decided that the website was to be hosted on a separate repository on GitHub, as branch protection rules made it complicated to automatically update the documentation on the main repository. Using a separate one made it trivial. This *Workflow,* in addition of using a YAML file, executes a series of Bash scripts to get through the steps for updating the repository. This approach was chosen to minimise the efforts involved in changing CI providers, as Bash is likely to be available on any CI runner. Before running the Bash scripts, the environment is set up to include the needed dependencies: a JDK and Graphviz libraries (to generate UML diagrams). The script then runs the JavaDoc Gradle task, copies them to the checked-out website repository. Next, UML diagrams are generated using a PlantUML executable JAR, and similarly copied. Test reports are then generated using the Gradle plugins for JUnit and JaCoCo, to provide test results and test coverage respectively. These are also copied to the website. Finally, the changes are committed and pushed to the website repository.

Source code and jar file can be found through this link:
https://github.com/Frinksy/PixelBoat/releases/tag/v1.2