Continuous Integration 2

Cohort 1, Team 4:

Robert Watts
Adam Wiegand
Josh Hall
Travis Gaffney
Xiaoyu Zou
Bogdan Lescinschi

a)

For our continuous integration, we wanted the system to run automated tests on the code to ensure we caught any bugs early. If the tests were passed, the source code would be compiled and packaged into a runnable we made available to the public.

The first test that was run was a build test. This ran when there was either a pull request or a push on the master on our GitHub. This CI runs the gradle test command which builds the entire project, and then runs any generic tests (for the overall project) that have been set up. We chose not to write any unit tests for this although future maintainers may choose to do so. The output of the workflow is just a simple pass-fail as to whether it was built correctly or not. We felt it was important to check that for any changes on the master branch, the project is built. This proved useful to us as on several occasions the build failed which prevented the other CI components from running, thus inhibiting a faulty build.

Furthermore, with our GitHub accounts we were limited to 3000 minutes per month of run time. This meant we wanted a quick and simple way to determine that there were no syntax errors in the code before spending time doing longer workflows, such as unit tests and burelases.

The next phase of the continuous integration that was executed was the unit tests. This workflow ran all of the unit tests as part of the “core” code through gradle. If any tests failed, it did not carry on to the next workflow to create a release. This workflow ran when there was a pull request or push on the master branch. The purpose of this workflow was to make sure that we did not push code to the master (and therefore subsequently the release) that had bugs or did not work as intended. This cycled through all the tests in the core project and ran them. Assuming they passed it outputted a pass-fail for the overall project.

The final workflow that ran was a release workflow. This workflow first generated a build for the desktop using *gradlew desktop:dist*. Using the Github API, it then creates a release incrementing the version number each time. It is worth noting that all of these version numbers are in the 2.x range, as we felt it necessary to distinguish the Assessment 1 builds from the Assessment 2 builds. The workflow then finally uploads the release (again using the Github API) uploading the output .jar file to the repository (and renaming it to include the version number). At the same time it includes the source code (compressed), so that the source code for our individual previous builds are available to clients. Both the executable file and the source code for each release are available on our git repository.

This final workflow on runs when there is a push request to the final branch. This gives individual branches the ability to alter and update their code without having to wait for the testing to complete on every update, while still ensuring that all code is tested before it is released. We chose to release it automatically to give both the customer and users instant access to the most recent features, which from our team customer meeting we understood was important. We did consider to run the release workflow as a “nightly” build, however we felt it was not appropriate, as our code did not necessarily change on a daily basis, leading to redundant builds.

b)

While researching the continuous integration we determined that there were two main contenders for continuous integration: GitHub Actions and Circle CI. We liked the way the workflows of Circle CI were set up, however you could only run one action at a time and we had to set up new accounts with billing information adding complexity (and potentially cost if we went over the "free" limit). In comparison we all already had premium GitHub accounts and as such 3000 hours of run time was included. On top of this we already had GitHub accounts for the code anyway so we decided to keep it simpler and use GitHub Action. This also has the advantage for future maintainers that everything is hosted on GitHubwithout the need for multiple cloud services.

GitHActions makes it simple to add an action; all you need to do is put a YAML file in a folder /.github/workflows/ for each workflow you want to run. When pushed to GitHub the action is then automatically detected and run. We determined that we wanted 3 workflows (as described above) and therefore would need three YAML files.

For the first workflow, to test build the project, we run it on a Ubuntu VM. The environment is first of all set up, and the JDK is then installed. We opted to use a slightly older version of the JDK. This helps to make sure that our project is backwards compatible for older machines. The workflow then sets up the file directory, as gradle needs permission to run. Once all this is complete the command *gradlew build* is run. If it is completed without error the workflow passes otherwise it fails.

The sound workflow (running the core unit tests) is much the same as the first workflow. First we set up an Ubuntu VM environment, with the older JDK. Then we give gradle permission to run finally running *gradlew :core:test* to run the unit tests. This then runs each test in turn. Each test can either pass or fail. If all the tests pass the workflow passes otherwise the workflow fails, returning the error message.

The final test, building and releasing the product is more complex than the others although it starts out like the others. First the Ubuntu VM environment is set up, with the older JDK and giving gradle permissions to run. The next step is to build the .jar file and we do this with the command *gradlew desktop:dist.* Once the command has run we then need to register the release with GitHub. This is done using the *actions/create-release* addon. This creates the release with the name, tags it with the version number and packages the source code. This all uses the GitHub API to achieve this. Once the release is registered with GitHubwe need to upload the .jar file. This is done with the *actions/upload-release-asset* addon. We then get the file from the build director on the Ubuntu VM and use the upload url for the release upload. Once uploaded it renames the .jar file from *desktop-1.0.jar* to *Auber_Assessment_2_Version_2.X.jar* where X is the version number. This all again uses the GitHub API. Once complete this workflow returns as a pass. It's worth noting if the .jar file fails to build or the GitHub API fails the workflow also fails.