Assessment: 2

Deliverable: Continuous Integration

Team Name: Team 8

Team members: Charlie Hayes, Matilda Garcia, Joshua Stafford, David Kayode, Ionut Manasia, Matthew Tomlinson

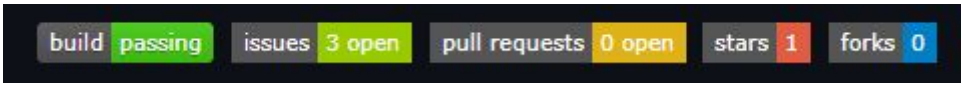
Method and Approach

The most critical aspect of our approach to continuous integration was the employment of a version control system. Following on from our previous experience, we chose to keep the source code on GitHub[1]. The use of git and GitHub allowed our team to separate change requests from earlier in the process into feature branches. These branches would be worked on synchronously with other branches, making minor code changes compared to the main code base, meaning they all shared a large portion of common code. These branches were then frequently integrated into the main code base via pull requests. This approach was appropriate for this project as each change could be worked on independently of another (the code from one feature was not needed by another due to the already large amount of common code). Also the frequent integration meant team members were always working on an up-to-date base allowing new branches to be created from the main branch at any time.

On each pull request and push to the repository, we wanted an automated build to take place. This automated build process' job was to verify that the code being integrated could be compiled to ensure that the code base remained valid and functional. We specifically wanted it to be automated on a server as it was feasible that team members may not have time to build and test the code for each integration and certainly wouldn't have the means to do so on multiple targets (this will be covered in our infrastructure later). The targeting of 3 different operating systems seemed appropriate for our project as we cannot guarantee that the customer will be using a machine similar to those we were developing on. In a similar vein, we felt it would be appropriate to cover both the oldest version of java we plan to support and the current lts version (8 and 11 respectively).

However, the building of source code was not enough on its own as the compilation of the code does not guarantee that it will run correctly (or indeed at all), only that the syntax is valid. Therefore, the automated build process was extended to also automatically run our JUnit white box tests. This was especially appropriate for this project as testing was already needed and laid out so it was only a simple (but very powerful) extension to enable automatic testing to be run on every pull request and push. Additionally, the build and testing was made available for manual execution (on GitHub) so the process could be cancelled and restarted later at any time if needed.

If builds were to fail it was important that the system we used strongly suggested not merging a pull request (but still allowed it if forced for situations where a failing build may be expected) Similarly, it was also important that all team members could see that the current build was passing or failing, so badges were deployed so that this important information was communicated every time the repository was visited.



Finally, consideration was given to automating deployment of JAR releases of the game. However, we decided that, as a distributable version of the game was not necessary until the final build, that continuous deployment was redundant for this project. Whilst it would have served a purpose, the effort necessary to implement it would outweigh its utility.

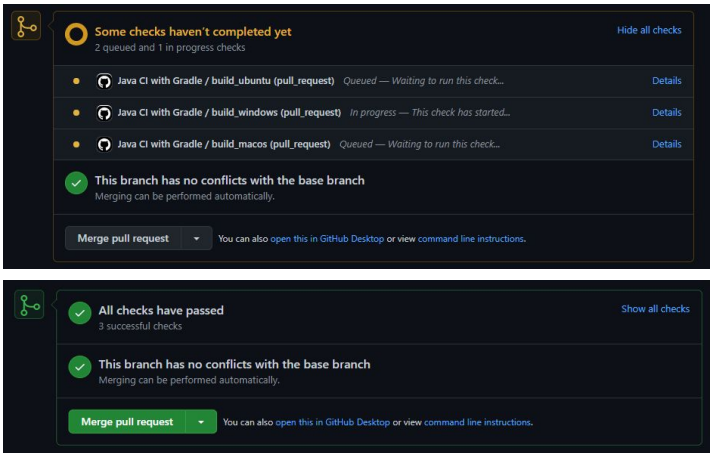
CI Infrastructure

As we were already using GitHub for our version control, it made sense to use GitHub actions[2] for continuous integration. GitHub actions allow you to specify automated tasks to run on certain events within GitHub. We chose to use GitHub actions specifically as it provided all the tools we needed for our CI approach, seemed simple to set up (as it already had a template for Gradle, the build tool we use) and had great integration with the system we were already using.

```yaml
name: Java CI with Gradle

on:
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]
  workflow_dispatch:
    branches: [ master ]

jobs:
  build_jdk8:
    name: Java 8 build on
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macOS-latest]

    steps:
    - uses: actions/checkout@v2
    - name: Set up JDK 1.8
      uses: actions/setup-java@v1
      with:
        java-version: 1.8
    - name: Grant execute permission for gradlew
      run: chmod +x gradlew
    - name: Build with Gradle
      run: ./gradlew build

  build_jdk11:
    name: Java 11 build on
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macOS-latest]

    steps:
    - uses: actions/checkout@v2
    - name: Set up JDK 1.11
      uses: actions/setup-java@v1
      with:
        java-version: 1.11
    - name: Grant execute permission for gradlew
      run: chmod +x gradlew
    - name: Build with Gradle
      run: ./gradlew build
```

For our project we configured it, as shown to the right, to set up jdk8 (the oldest version recommended to support) for each operating system, checkout the main branch and run the `gradlew build` command. The Gradle build tool was a key component of our CI infrastructure as it handles much of the automation. This specific Gradle command compiles all the source code. As our tests are set up as a project in gradle, they are also run during the build process, failing the whole build if any tests fail (and outputting which ones), or creating a successful build if all tests pass. The main stages of the build process involve compiling the core classes and building the desktop project into a JAR, then doing the same for the ‘tests’ project and running the command ‘tests:test’ (which is the same command used for running the tests locally. Therefore, our infrastructure fulfills the key aspects of continuous integration. This process is then repeated for jdk11 (the current LTS release) to ensure that it is likely to run on any kind of target machine.

Specified at the top of the configuration is that the CI action will run on each push to master, pull request into master and on manual execution. This worked together with our use of feature branches by allowing small, frequent (and perhaps incomplete) changes to be pushed to the other branches whilst requiring pull requests from the feature branches into the main branch to be valid and tested.



Each pull request will inform the developer of the build status (known as checks) while they are run. We were especially happy with our choice of infrastructure for its clarity of build status. For example, seen to the left is the process of making a pull request. Each check being made is shown and highlighted in an appropriate colour to indicate whether it is in progress/queued (orange) or has failed (red). If a check fails, the person responsible gets sent an email of the failed builds.

If the checks pass, a large green tick is shown, the pull request icon turns green and the ‘Merge pull request’ button lights up green, clearly indicating that the code is valid and merging is now recommended.