Team 4 - “Pirate Hygiene”

Team members:

Alex Angelidis

Ivan Barinskiy

Sophie Foster

Daynyus Halinauskas

Jakub Mlodzinski

John Scheland

Section - 5 Continuous Integration

5.a Our team’s continuous integration methods will include using tools such as GitHub actions to produce a build of the game every time a change is committed to the project. We will configure it so that it automatically runs tests we have produced every time someone pushes to the main branch of the project. We decided on this method as the group is most familiar with GitHub and we have been using it to code collaboratively throughout the project. The frequent testing will also allow for the group to ensure that no new bugs are created by adding new features or updating old code. The team's use of GitHub actions also will allow members to perform manual tests of the project, allowing the team to evaluate the game's playability as new content is added. This is due to being able to easily access the most recent build of the game as GitHub actions produces a new release of the project each time a change is made.

5.b The team used GitHub actions to automate our continuous integration activities. GitHub actions is a tool provided by GitHub that allows for the automation of continuous integration. It does this by running workflows that can be triggered by an event in the repository, manually run or set to a defined schedule. You can choose to have multiple workflows within a repository, but we only needed to use one to implement our continuous integration methods. Our workflow was triggered by pushing onto the main branch of the repository which allowed for the team to easily access and test a build of the game with all the latest changes implemented. This can be seen in ‘Figure 1’:

```
3  on:
4    push:
5      branches:
6        - "main"
```

Figure 1

The ‘on’ key is used to trigger the workflow, ‘push’ is the event, ‘branches’ allows you to specify which branch the push needs to be on to trigger the workflow and “main” is referring to the main branch of the project.

We configured our workflow to automatically run our unit tests. We used the Ashley-Taylor Junit plugin to produce annotations detailing whether our tests passed or failed. This can be seen in ‘Figure 2’:

```
26      - name: Junit Report to Annotations
27        # You may pin to the exact commit or the version.
28        # uses: ashley-taylor/junit-report-annotations-action@562e0277515cae408f30ad1ea2d6dea44fc1df87
29        uses: ashley-taylor/junit-report-annotations-action@1.3
30        if: success() || failure()    # run this step even if previous step failed
31        with:
32          # github token
33          access-token: "${{ secrets.GITHUB_TOKEN }}"
34          # glob to junit xml files
35          path: "**/TEST-*.xml" # default is **/TEST-*.xml
36          # include summary annotation
37          includeSummary: true # default is true
38          # max number of failed tests to include
39          numFailures: 100 # optional, default is 10
```

Figure 2

GitHub actions also automatically runs a compile test upon building the project file. This can be seen in ‘Figure 3’:

```
41      - name: "Build with Gradle"
42        uses: gradle/gradle-build-action@937999e9cc2425eddc7fd62d1053baf041147db7
43        if: success() || failure()    # run this step even if previous step failed
44        with:
45          arguments: desktop:dist
```

Figure 3

This builds the desktop-1.0.jar file and ensures that it compiles correctly. If this part of the workflow fails, the new will not be published.

If all the tests pass GitHub actions produces a badge stating that the pre-release is currently passing all the tests that have been run on it.