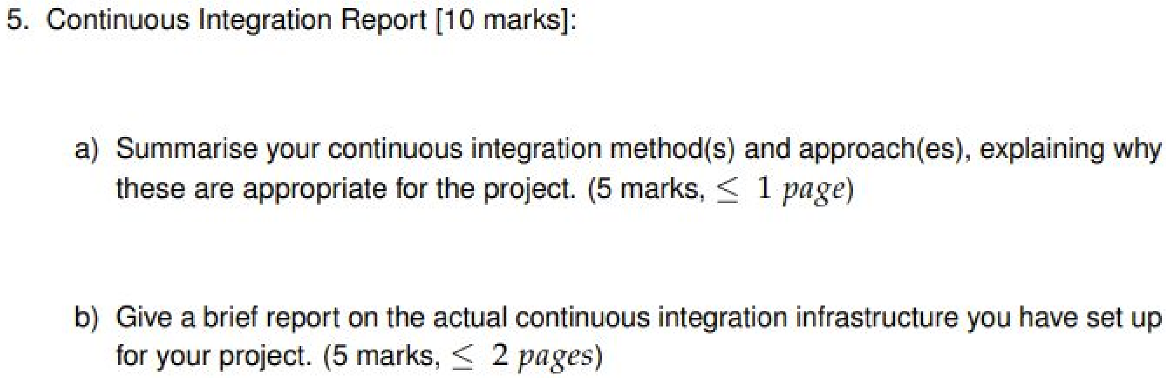
Module	Engineering1 (Eng1) - COM00019
Assessment Title	Assessment 2, Cohort 2
Team	Dragon Boat Z (Team 18)
Members	Robert Dalgleish, Benjamin Jenner, Joseph Lonsdale, Richard Upton, James Wilkinson, Xinyi Zhang
Deliverable	Continuous Integration



5. Continuous Integration Report [10 marks]:

a) Summarise your continuous integration method(s) and approach(es), explaining why these are appropriate for the project. (5 marks, $\leq\ 1\ page$)

b) Give a brief report on the actual continuous integration infrastructure you have set up for your project. (5 marks, $\leq\ 2\ pages$)

Methods and Approaches

Continuous Integration (CI) is the practice of integrating team members' work frequently into the main codebase. Automated tests are run on a temporary merge of the existing codebase and new changes, to make sure the integrated code doesn't cause any errors or break any existing functionality. CI ensures programmers produce non-conflicting code and reduces the likelihood of introducing new errors.

We will use several methods and approaches in our project.

- Maintain a single online repository with version control features that will store the code and any additional assets required for the build. This will enable team members to update their working local copies with the latest changes that have been submitted by other team members. Utilising this central repository will allow us to set up multiple branches of code that can be used to control when new code is integrated with the main codebase.
- Code must be automatically built at each integration. This includes compiling the code along with all other assets and libraries necessary for executing a program. An automated build, reduces the time taken to carry it out as multiple builds can be carried out concurrently and avoids risk of human errors. These errors are avoided by ensuring the same build commands are carried out every time.
- Automatic tests should be constructed that run on the built code to ensure functionality is intact. This reduces and should remove the risk of runtime errors when executing the final build. It also makes sure that the build meets requirements. This is an efficient way of ensuring functionality as they are automatically run every time new code is integrated. The output of these tests would be a pass or fail, and in the case of Unit testing, the failed tests and line number will be detailed in the log.
- Team members should commit their independent changes to the mainline frequently, to enable all team members to be made aware of integration errors. This will help to ensure that errors can be fixed at an earlier stage. This is to avoid the issue of having new code added later that is dependent on these broken changes.
- To further ease the process of integrating new code into the master branch, team members should use a constructed library of tests locally, to check for errors and ensure functionality is maintained. This helps developers identify errors in the features they are developing, before attempting to integrate their code.

Continuous Integration Infrastructure

Our chosen version control system is Git, which will be used alongside GitHub to host our centralised online repository. This allows us to have different branches of code, view changes made in commits and control access.

To enforce the use of CI, we set up branch protection on the master branch, this requires that a pull request from a non-protected branch must be made in order to add to the master branch. This ensures checks are carried out before any code is added to the master branch. These checks include GitHub automatically checking if the code can be merged without conflicts, automated actions to test the code and an approval from at least one assigned reviewer - who did not submit the pull request. If conflicts are found when trying to automatically merge the code, these must be solved manually by a team member. This will flag up files with conflicts and highlight where they are, a member must then edit the code and mark the conflict as solved. Reviewers are given a list of all the changes that they can read through before approving the request. They can either approve it or reject it and make comments with what changes need to be made.

Due to the team's lack of experience with these CI methods, branch protection was disabled for Administrators. This allows the Administrator to bypass pull request reviews as a backup. This was useful when dealing with merge conflicts that needed to be quickly fixed. This was only ever used in instances where the changes were minimal, such as a single file, and a reviewer was not currently available. Failure to pass automated tests was never ignored.

We decided that dedicated Branch protection should also not be included on other branches due to the lack of modularity and code interdependence. Small changes could affect the functionality of the entire code base. This made it necessary to work cooperatively on large sections of the codebase at a single time. This collaborative nature led to code changes that fixed some aspects, while breaking others. However, this was required to enable developers to then work on other sections. If branch protection and automated tests were applied to these branches, it would hinder development and not add to ensuring the integrity of the code.

Automated actions, to test the code, must run and pass before the user is able to merge their pull request. To carry out these automated tests we used GitHub Actions. YML files were created from templates to carry out specified Gradle commands. Each file represented a workflow that can be accessed through the Actions tab. For a given workflow, different triggers are set up. Below are the workflows we have set up.

- Gradle Unit Tests is run if the master branch is pushed to or a pull request is made into the master branch. It can also be run manually on defined branches, including master and testing (used for developing CI tests). This workflow runs the Gradle project named tests which carries out Junit tests on the codebase. It passes when all the unit tests pass.

- CodeQL is a workflow provided by GitHub that checks the quality of the code and for any security errors using Gradle. This is run on a push or pull request to the master branch and on a schedule using a cron job.
- Gradle Build builds the codebase and checks that there are no errors in compiling. This workflow is run on Windows, Ubuntu and MacOS to ensure compatibility with multiple operating systems. This is run on a push or pull request to the master branch and can be run manually on the master branch.

Our chosen build automation tool was Gradle as it comes with LibGDX, which is the game library we chose to use for development. This is a popular tool for Java development. Gradle allows tests to be run locally by developers, allowing for continuous testing. These tests can be run from the command line or integrated with the testing suites of some IDEs. A report is generated by the test project that details the status of all tests carried out and the errors causing failed tests. This is useful for developers in debugging errors.

Tests were written using JUnit 4 and Mockito 1.9. JUnit is a popular framework for testing the integrity and functionality of Java code. We chose to use version 4 for compatibility with our version of Gradle. Mockito was chosen as it allows us to ‘mock’ objects in tests. Version 1.9 is used for compatibility with Junit 4. This allows a single class to be tested that uses objects from other classes without depending on the functionality of those classes, ensuring an independent testing environment for each class.