Testing

a)

Automated Testing

Before adding any new features, we first tested the base game by writing Unit Tests to ensure that the original (base) game covered all the initial requirements and had these tests in our Continuous Integration. From here, the team decided to implement an iterative development method where the team would plan and write write code for the new feature/requirement and then work on writing tests for the same, and would only move forward to another feature once all the tests for the previous feature had passed. By using this tactic, we ensured that the old features remained unaffected when new features were added to the code base. All the tests were created based on the requirements documentation and were aimed to cover most of the requirements, and we also considered the idea of code coverage and getting as much of it as possible as in headless mode, we can't test any elements which required user input (such as moving the ship by using keys), and therefore the code coverage does not reach 100%. All the automated tests were run when we push to the repository as well as on a pull request. This ensured that if any bug was found, the team would be notified and a member from the team could work on fixing this bug. All our tests are documented with relevant id, description, input, expected outcome and author. The code coverage reached is XX% of classes and XX% of lines. All our tests can be executed together by using the "test" Gradle task, or by using the run option which is offered by most IDEs for each test class. As described by Ian Sommerville [1] tests where created in such a way that individual object classes were tested, for example `collegeExists()` checks that the colleges are created alive. This would be unit testing. Other tests were created to check the components of a function, for example `collegeFires()`. Others test how some or all components integrate and test the system as a whole, for example `powerUpPickup()`.

Manual Testing

We implemented Black Box and white box testing during manual tests which we developed based on our tracebility matrix (features which we could not test automatically) as automating these was beyond the scope of the ENG-1 project. To create the tests, we referred to the requirements documentation and aimed to test as many requirements as possible. Our main aim was to first test the user requirements followed by testing the functional requirements. As we were slightly ahead of schedule, we decided to test the non-functional requirements as well. It was very important for us to test elements that utilized user input such as pressing keys/using the mouse in order to move the ship/attack other ships or colleges, as testing for user input with headless application is not possible in gdx-testing. We also manually tested the UI elements of the game as it wasn't possible to test them automatically. The teams' main goal of using this testing method was to ensure that the requirements were met, and any bugs were found and rectified. The white box element of the testing involved the implementation team suggesting ways that the tests could expose bugs in the game due to their knowledge of the code base. Majority of our test cases covered inputs which we thought were the best to test and ensure that all the requirements were met, with some additional tests covering invalid inputs, and some covering boundary inputs which would help in identifying any bugs. The traceability matrix as well as further evidence of testing can be found on our website.

Why?

By using the approach outlined for our automated tests, we believed that it was appropriate for our project as this ensured that throughout the development of the project, any features which were implemented in the previous (base) game as well as any new features which we were implementing were not unintentionally affected. Moreover, this approach satisfied our continuous integration, as the tests were run on every push to the repository as well as on any pull requests. The focus on black box approach for manual testing was suitable for such a project with a big codebase and with a certain amount of time. Using manual testing enabled us to easily keep track of the user requirements that were met by the game.

[1]I. Sommerville and M. Paul, Software engineering--ESEC '93 : 4th European Software Engineering Conference, Garmisch-Partenkirchen, Germany, September 13-17, 1993 : proceedings. Berlin ; New York: Springer-Verlag, 1993.

b)

Upon inheriting the codebase for the assessment, a series of tests were created to cover the initial requirements already implemented.

Uttility classes

QueueFIFO(), Utilities(). Due to the simple nature of the methods in these classes, they could be tested with more straightforward unit tests, so a series of unit tests were created to measure the methods against both expected inputs and extreme inputs to verify the robustness of the methods. All implemented Utilities() and QueueFIFO() methods have been tested. The results of all the tests are successful. Asset tests were carried out to check each asset including images and JSON files are present, these were all successful too.

Game Code

In order to aid with testing the game itself, we implemented Headless Testing and used the LibGDX Testrunner, which allows the game to be simulated without the UI and mocks the rendering classes.

The game code was split into sections which we tackled individually
AITests - tested as unit-style tests for paths and nodes to verify they worked as intended. These all passed as expected.
Components - we determined that it would be best to test Components in tandem with the objects they were designed to assist with, as otherwise the results of the test may not be an accurate representation of the functionality of the project.
Entities - On similar principles, since some Entities present in the project are codependent these were grouped for testing into: CollegeTests, PlayerTests, ProjectileTests and ShipTests (originally the Ships were tested together with the Player but we determined there were too many unique aspects to the Player that needed to be tested independently such as the PlayerController() component).

CollegeTests - The College() tests were paired with the Building() tests due to the fact that as part of the construction of a College, Building objects are also constructed and their references are kept within the College structure. The condition for a College being captured is that all the buildings within it have been destroyed, meaning that while testing for successful college capture, ensuring that Buildings were destroyed accordingly was an integral part of the tests. The only part of the College() class that wasn't able to be tested was the update() method.

PlayerTests - While testing the Player(), it became apparent that in order to test facets of the Player such as plunder and points, then the Quest() class and its subclasses would have to be tested too. All of the implemented methods within Player() and Quests (in addition to the QuestManager) were tested since they had no dependencies that couldn't be instantiated, however there were aspects of the PlayerController() that were not able to be tested.

ProjectileTests - Initially there were queries regarding the ability to test projectiles, seeing as the projectiles needed to be rendered for them to move over time. However, since those aspects of the projectile are handled by the Transform() component, which could be considered separately, we were able to test all facets of the Cannonball() without issue.

ShipTests - ShipTests were tested along with the RigidBody() component. The applyForce() method in the Rigidbody couldn't be tested however the majority of other methods were implemented. The tests to verify that ships could move in every direction were very extensive to the point they initially took up the majority of the runtime of the testing process before they were redone to be more efficient. The remaining features of Ships such as whether they could fire projectiles and whether they could die were also tested successfully. Finally there were tests regarding specifically NPCShips, ensuring that the time period of their attacks was set correctly and that they behaved according to the behaviours set in the EnemyState() class. The EnemyState() class was tested by creating an NPCShip and a target player then simulating different scenarios to verify states were changed correctly.

GameStateTests With all the entities tested, the final block of the initial codeblock we were able to test was the Game State. These tests covered the GameManager and the functionality it provided in the running of the game. The most expansive test is the gameStart() test which covers each asset that the game requires to run and ensures it is present and accounted for. The other faculty of these tests is testing the success and failure states of the game.

Once all the tests of the existing codebase were dealt with, we added tests for each feature that we implemented if possible. These usually took the form of a series of dedicated tests covering the entire span of functionality introduced by the feature, such as is the case for

PowerUpTests, SavingTests and ObstacleTests. When instead a previous feature was being introduced, such as Game Difficulty, tests were added to the existing block to cover the addition.

Due to the limitations presented during testing, such as issues where testing couldn't be done if it required testing, and there being pre-existing code that isn't implemented and couldn't be removed (such as methods within classes which implement interfaces) we were only able to achieve **63% code coverage**. However, the portion of code which is untested we have verified the functionality of through manual testing, and the portion of the code which is unimplemented doesn't need to be tested.

All of our tests pass in the current iteration of the project, and our tests are designed to test the intended functionality and extremes of the codebase, ensuring that the project can handle both the expected and unexpected.

So to conclude, the parts of the codebase that we ended up not implementing tests for were:
UI
The UI section, containing the Game/Menu/Pause/End Screens and the Page superclass all couldn't be tested due to the fact that
TileMapGraph() / TileMap()
This is due to the fact that in order to properly test it one would have to instantiate a TiledMap object which was believed to be an issue without rendering the game.
RenderingManager() / Renderable()
The Rendering Manager couldn't be tested since it relies on rendering.
PirateGame()
A lot of the functionality of PirateGame() is circumvented in setting up Headless testing
College.update()
This is due to the fact that the update class relies on the game runtime, which isn't accessible without rendering the game.
PlayerController.getDIRFromWASDInput()
The getDIRFromWASDInput method relies on receiving user input, which we were unable to implement considering the restrictions of the testing method.
PlayerController.update()
The update function relies on output from the above function and also relies on facets of the project being rendered which we could not simulate within tests.
Rigidbody.applyForce()
The applyForce function could not be tested since the results of applying force rely on the object being rendered.
GameManager.createWorldMap()
This couldn't be tested since the World Map requires rendering to test.
EntityManager.raiseEvents()
Requires rendering
PhysicsManager.createMapCollision()
Requires tileMap

Other
Other functions that weren't tested were not included as the functions were not actually implemented, such as QueueFIFO.offer() or because they were getters/setters.

Manual Tests

After completing the unit tests, we went through the Requirements of the project and constructed a Traceability Matrix to document which requirements were left untested and which had been covered throughout. The requirements which were left untested and the functionality of the methods above which couldn't be tested due to restrictions were recorded and tested manually. We decided to also manually test all User Requirements to have a two-point guarantee of their functionality. This means that every user, functional and nonfunctional requirement has been tested.