4.a

We began by designing and writing unit tests using Junit to test individual components and entities within the project. We would then perform these tests as well as a compile test every time a new change was implemented into the project The unit tests would be run prior to pushing the new change while the compile test would run automatically through GitHub actions after the change had been pushed. We would then test the game manually by playing through the game to ensure that it was still playable with the new functionality implemented.

These methods were appropriate for the project because unit tests are easy to write and control meaning that the team could write tests that were specific to our project. They are also fast, allowing us to quickly run them after each update to the project, saving time on an otherwise tedious and elaborate testing routine. The unit tests were designed to cover most of the features of the product and allowed for the team to ensure everything was working as it should be. Manual tests were also appropriate as they allowed the team to perform various non-functional tests and evaluate how the game will be played and how enjoyable it will be for the customer, as well as ensuring that the game's visuals look proportional and are placed correctly. The autonomous testing was appropriate as it allowed for the team to ensure that everything was still functioning correctly after every push

4.b

To test each upgrade to the product, a series of JUnit tests were employed. These tests aimed to simulate scenarios in the game the player is likely to come across as well as to test whether or not each entity class correctly performs its tasks.
Overall, the testing routine consisted of the following:

Asset tests

- Every file the game used was verified to be present in the game's directory

Movement tests

- Movement tests were run to check whether rigid body positions are updated when a velocity is applied.
    - A test for each of 8 directions permitted by keyboard inputs was ran, with the pass criterion being the change in the ship's position representing a movement in the direction specified e.g to pass a test for moving north, the ship's Y coordinate must have increased after applying a velocity and updating the physics manager and entity.

Sea monster tests

- For the sea monster entity, which remains stationary in combat with the player but turns to face them, a series of tests was run where an instance of Player class was positioned in different directions relative to the sea monster. Each test then consisted of checking whether the direction of the monster (returned as a string via a method in the sea monster class corresponded to the position of the player.)
- To test the sea monster's poison aura was tested by placing a player entity within the "poison" range and ensuring that the monster was decrementing its countdown timer to apply damage to the player.
- To test the monster's shooting capabilities, a cannonball was fired via the GameManager's shoot method targeting the player and another cannonball targeting a non-player ship. The criterion for passing was the player ship receiving damage after the shot, and the non-player ship remaining undamaged. This scenario corresponded to the way the team implemented the sea monster - only engaging with the player and only hurting the player ship with its poison aura and projectiles.

Combat tests

This test class was testing combat scenarios the player could encounter. The tests included the following:

- A cannonball instance was fired in 4 different directions, with the test criterion being that the cannonball's velocity vector corresponded to the direction it was fired at.
- A player instance was damaged, with the pass condition being that the player's health was updated accordingly. Similarly, in another test, a player instance was drained of all its health, with the pass condition being that the player has been registered as killed.
- A test covering the "player kills NPC ship and receives plunder" scenario has been implemented, where an instance of Player class and an instance of NPCShip class belonging to a non-player faction have been spawned. The level of health of the NPCShip instance was set such that the next shot from the player would kill it. Then, the player instance fired a projectile in the npc's direction. To pass the test, the following conditions had to be satisfied:
    - The NPC had to be killed and registered as destroyed
    - The player must have received plunder for killing the NPC (verified via Pirate component's getPlunder method)
    - The NPC had to be moved offscreen according to its update method.

Powerup tests

This test class covered the Powerup class and its effects on the player.

The testing done consisted of the following:

- For each of the 5 powerup types, an instance of that powerup was spawned and its effect applied to the player. The pass conditions for the test varied for the different powerup types:
    - Instant powerups (ammo refill, health refill) were tested by verifying that the player instance's fields were reset back to starting values after experiencing the effect.
    - Time - based powerups (invincibility, weather resistance, speed boost) were tested by verifying that the respective boolean field was set to true (isWeatherResistant and isInvincible in the player class) and that the speed value of the player's playerController component increased upon collecting a boost powerup.
    - For each powerup, it was also verified that the player received 10 units of plunder for collecting them.
- To verify that powerup effects do not compound i.e only one timed powerup can be applied at a time, two instances of timed powerups of different types were spawned. The first powerup applied itself to the player, then the player instance was updated, and the second powerup applied. The pass condition was that the player was under the effect of the first powerup only, receiving double the plunder but no effect for collecting the second one.

College/Building tests

This test class covered the combat between the player and enemy college buildings, including flags and defender buildings, which are capable of shooting back at the player. The testing consisted of the following:

- A College instance of a non-player faction has been spawned.
- It was verified that one and only one building belonging to that college was a defender to match with the desired implementation.
- A player instance was spawned and shot a projectile at a non-defender building, with the pass criterion being that the building was marked as destroyed upon first hit.
- Similarly, a player instance shot a projectile at a flag building, with the pass condition being that the flag took no damage according to the building class implementation.
- Finally, a player instance shot a projectile at a defender building, with the pass condition being that the defender building remained alive but its health has been reduced by the player's damage level.

Save/Load tests

This test class covered saving the game's state and verifying the correctness of the save. The following tests were conducted:

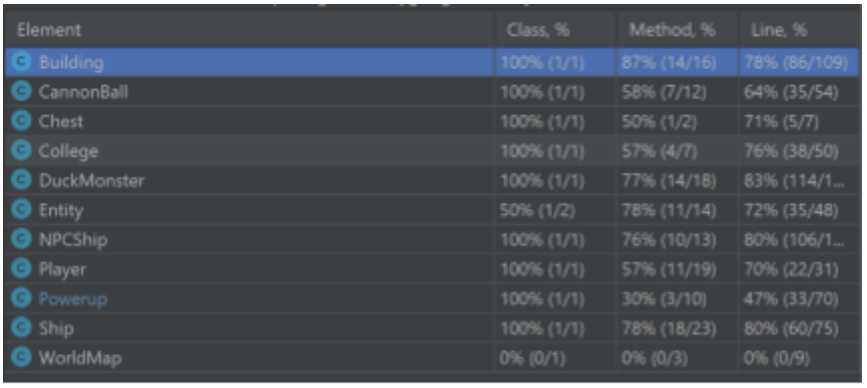
- A "mock" game was spawned, with a player instance, a npc belonging to the player's faction and another npc belonging to a different faction. A College instance was also spawned for each of the game's 5 factions. A list of quests was randomly generated via the QuestManager class. Then, this gamestate was recorded via a SaveManager instance. To pass the first test, the SaveManager had to produce and store a file in the path that is used to retrieve a save file upon game reload.
- In another test, a file generated from the mock scenario was retrieved and analyzed. To pass that test, the following was required to happen:
    - Stored ship position had to match each of the ship's current position as they were not moved between the tests.
    - All quests must have been present and recorded as not completed
    - All colleges must have been present and recorded as not destroyed
    - One instance of KillDuckQuest must have been present among the quests.

Results

For the latest version of the game, all tests passed within 20 seconds as some of them depended on waiting for events to occur such as cannonball collisions.

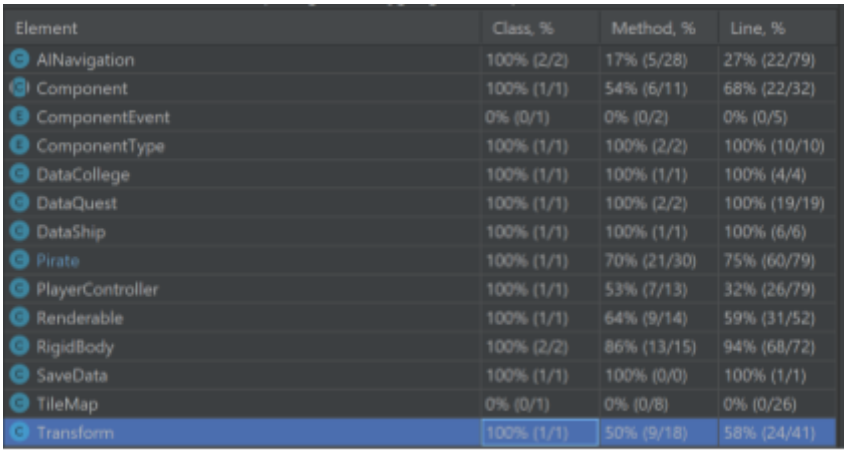
Discussion

The tests implemented covered a large number of scenarios and automated a large amount of otherwise tedious manual testing. Most of the functionality of each entity has been covered by the tests, as evidenced by the coverage report seen on figure 1. Some entities and components depended on features that are difficult to automatically test, such as keyboard inputs and rendering (seen in figure 2, where the PlayerController and Renderable class are covered less than other components), so these features were tested manually after big updates. A manual testing routine involved making sure that the player is unable to go over land, that the shop screen was sized correctly and that its visual components (progress bars for each purchase) updated on purchase. Furthermore, since the re-spawning of powerups was impossible to do in a headless testing environment, that part of the Powerup class was verified manually by zooming out on the map and verifying that no powerup spawned on land. Overall, the automated tests combined with the manual testing routine allowed the team to have a high degree of confidence in the fact that the game was working correctly and behaved according to the desired implementation.



Element	Class, %	Method, %	Line, %
Building	100% (1/1)	87% (14/16)	78% (86/109)
CannonBall	100% (1/1)	58% (7/12)	64% (35/54)
Chest	100% (1/1)	50% (1/2)	71% (5/7)
College	100% (1/1)	57% (4/7)	76% (38/50)
DuckMonster	100% (1/1)	77% (14/18)	83% (114/1...
Entity	50% (1/2)	78% (11/14)	72% (35/48)
NPCShip	100% (1/1)	76% (10/13)	80% (106/1...
Player	100% (1/1)	57% (11/19)	70% (22/31)
Powerup	100% (1/1)	30% (3/10)	47% (33/70)
Ship	100% (1/1)	78% (18/23)	80% (60/75)
WorldMap	0% (0/1)	0% (0/3)	0% (0/9)

Figure 1. Entity package test coverage



Element	Class, %	Method, %	Line, %
AINavigation	100% (2/2)	17% (5/28)	27% (22/79)
Component	100% (1/1)	54% (6/11)	68% (22/32)
ComponentEvent	0% (0/1)	0% (0/2)	0% (0/5)
ComponentType	100% (1/1)	100% (2/2)	100% (10/10)
DataCollege	100% (1/1)	100% (1/1)	100% (4/4)
DataQuest	100% (1/1)	100% (2/2)	100% (19/19)
DataShip	100% (1/1)	100% (1/1)	100% (6/6)
Pirate	100% (1/1)	70% (21/30)	75% (60/79)
PlayerController	100% (1/1)	53% (7/13)	32% (26/79)
Renderable	100% (1/1)	64% (9/14)	59% (31/52)
RigidBody	100% (2/2)	86% (13/15)	94% (68/72)
SaveData	100% (1/1)	100% (0/0)	100% (1/1)
TileMap	0% (0/1)	0% (0/8)	0% (0/26)
Transform	100% (1/1)	50% (9/18)	58% (24/41)

Figure 2. Testing coverage report for the Components package.