

4. Testing

Team 10 | YorKorsairs

Abdullah

Tom Burniston

Omer Gelli

Adam Kirby-Stewart

Sam Mabbott

Benjamin Stevenson

A. Methods

The main purpose of testing is to establish to stakeholders and customers that the software meets the original requirements specification (**validation testing**, which is less focused on the possibility of faults) and to systematically detect bugs and faults within the software using a diverse range of inputs/input groups representative of how the software may be used by a user (**verification testing**, which is usually applied with more obscure data). The methods we use will be heavily focused on these 2 types of testing.

A large majority of the tests will be **unit tests** on the methods and classes within the system. By using unit testing, we can test for the presence of mistakes within individual units of the game making it a useful method for verification testing, while still testing that the game meets specific requirements (validation testing). Unit tests are easy to write, since they are only used to test a relatively small portion of code. Because of the nature of unit tests, many of them can be automated and are less reliant on user input so they end up being faster than most other tests and can be controlled due to the scale of the code. Unfortunately, unit tests are not realistic and are not representative of a genuine user input, especially since they are usually only applied to a singular method/class when in reality, many different units will interact with each other, giving potentially different results. This may also lead to other unidentified bugs caused by the interactions between methods, which means that unit tests can not be used by themselves.

When **designing unit tests**, tests will either be used to test 4 main scenarios (valid values with no errors, valid values at the fringe of a function, invalid errors at the fringe of a function and invalid values guaranteed to give errors). By testing these specific cases, we can gather enough data to verify that the function is working as expected and that it meets all of our set requirements.

In order to further test the software and the integration between units, we will use **integration testing**. The focus of these tests will be more on the interactions between different methods and classes rather than the classes themselves. These tests will be harder to write since they will have to account for the complexity of the interactions between units; however, less integration tests are needed to get more coverage over errors. These tests will have a similar design process to the unit tests with heavy focus on the extreme/fringe cases to root out as many bugs in the integration process as possible.

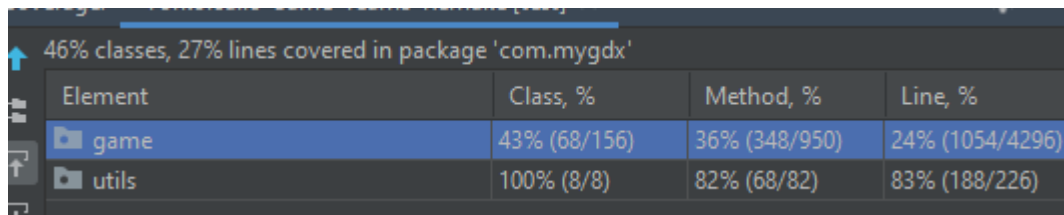
The final level of testing would be **software testing** which would test the system as a whole. Software tests help provide a realistic test comparable to how a user may use the system which is helpful for finding errors found on a consumer level. Software testing focuses less on individual functions and so they also focus less on verification testing since the majority of the errors will be found in the unit and integration testing phases, which means that these tests are more likely to be focused on making sure that our requirements have been met.

<https://aj141299.github.io/Kroojel.github.io/test.html>

B. Tests

When testing the game, we came across a number of difficulties and obstacles due to the architecture of the code. Numerous classes relied on other classes and managers. This meant that the number of unit tests that we could conduct was significantly reduced. We had to rely mostly on integrated tests when testing individual classes.

We were able to cover in total 46% of the classes within the project as a whole, with 40% of coverage of its methods and 27% of lines covered.



A screenshot of a code coverage tool's output. At the top, a summary bar states '46% classes, 27% lines covered in package 'com.mygdx''. Below this is a table with four columns: 'Element', 'Class, %', 'Method, %', and 'Line, %'. The table lists two elements: 'game' and 'utils'. The 'game' element shows 43% class coverage (68/156), 36% method coverage (348/950), and 24% line coverage (1054/4296). The 'utils' element shows 100% class coverage (8/8), 82% method coverage (68/82), and 83% line coverage (188/226).

Element	Class, %	Method, %	Line, %
game	43% (68/156)	36% (348/950)	24% (1054/4296)
utils	100% (8/8)	82% (68/82)	83% (188/226)

The project has multiple utilities that weren't class dependent. These utilities are used throughout the project and many other classes and functions are reliant on them, so testing them is evidently important because of the foundations they provide. These classes are all self-contained and were tested as individual units.

The Constants and TileMapCells classes only held constants and did not require rigorous tests. The QueueFIFO class however, requires more tests since it is building a data structure, making sure that it adds, removes and orders objects correctly so that it can be used reliably in later tests. The Utilities class falls under the same requirement of thorough tests to make sure that the Utilities' functions used in other classes are able to convert values and easily access essential functions. We were able to achieve full coverage of the utilities' functions with as many cases covered as possible.

The main objects within the game are the quests from which the players can gain ammunition, gold and XP which are used by the player to progress through the game. The quests were tested as integrated tests that used information from the Player class (position) and the Pirate class (isAlive) to determine whether the testing functions. We tested the Quest class' ability to track progress and complete and its ability to reward the player. These tests were completed for the separate subclasses of quests, locate and kill quests where locate quests progress used the player's location to test fringe parameters and range of the quest and the kill quest tested whether the targeted pirate is alive or not. The quests also have different rewards depending on the type of quest that is completed so the reward output was also tested for each subclass. The tests displayed that the quest's main functions were working properly and that the player's progress was being tracked adequately while integrating other classes as detailed by the requirements below.

All of the smaller individual AI classes with no/little integration have been fully tested as unit tests. These classes, including the individual nodes, paths and the node heuristic are the smaller parts that end up making up the AI within the proposed tilemap graph so testing these components is important to making sure that the game AI is working. The AI tests all passed, at which point we considered it fair to test the entire tilemap graph.

The main functions of the tilemap graph has been created with tiles within the integrated test implementing other AI classes (Node, NodeHeuristic, Path) world map being abstracted into nodes with connecting paths; however, tests revealed that the paths between the nodes were not initialised properly leading to the class being unable to find optimal paths between nodes. Since AI is not a specified requirement and the TileMapGraph class has not been properly implemented within the project, we decided that it was not essential for us to fix this issue and focussed on other parts of the code.

The ships (Ship.Class) is one of the main entities within the game, as ships are a major component controlled by the player and NPCs. The class takes many of the functions from the Pirate class including getting an individual ship's health and taking damage from other ships. We tested the pirates' takeDamage() function as a unit test to show that the ships are taking damage appropriately when hit and making sure that the ship is killed when its health reaches 0 or lower. We tested fringe cases within the test to make sure that the ship only died when the ship's health was 0 and not at an earlier value.

The Pirate class also controlled how the function player levels up when they gain XP, incrementing the player's level when their XP value exceeds 100. Using unit tests, we accurately tested that the addXp() and updateXp() functions can add to the player's XP value while remaining within the 100XP range and increasing the pirate's level correctly. The function was able to adequately do this when the amount of the XP increased by smaller values (0XP to 199XP). However, if the amount of XP being added is large enough to increase the level of the pirate by more than one (101XP or higher), the function will only increment the pirate's level by one and its XP will be greater than 100 which is outside of the required range. This failed test is due to a lack of consideration for large XP values that could potentially increase the players level by more than 1. We used this as an opportunity to redesign the updateXp() function so that it accommodates larger XP values and levels up by multiple levels.

The Pirate class also controls plunder and ammo being given to the player ship. These are simpler unit tests that check that initial plunder and ammo values are instantiated correctly and that their values increase accordingly when addAmmo() and addPlunder() are called. The tests passed with 100% success rate, meaning that the player will be able to gain with no bugs/errors when playing the game.

Another important feature of the Pirate class is keeping a track of targeted ships using a FIFO Queue. This is used by NPC ships to keep track of ships that come within range of the ship, giving them the opportunity to aggro onto these ships and attack them. The targeted ships are added to the beginning of a queue, with the target at the front of the queue being the target of the main ship until it is removed from the front of the queue. The addTarget() function has been tested with several mock target ships to test that it adds targets in the correct order within the queue and the queue returns the correct values when other pirate functions are called on it (e.g. getTarget(), targetCount(), etc.). All the functions directly related to the target queues passed.

The target function is also used to find out whether the targeted ship is within the pirate's range. The pirate has 2 separate ranges, a larger range being the pirate's agro range and

the smaller one being the pirate's attack range. The pirate class calculates whether the target ship at the front of the target queue is within these ranges using the `canAttack()` and `isAgro()` functions using the distance between the target ship and the pirate. These tests implement mocks of target ships to get positions of the ships in relation to the pirate, testing fringes to make sure that the targeted ships are no longer aggroed or attacked when they exit the function's ranges.

The Transform class is commonly used in the renderables, rigid bodies and entities that are used throughout the project, allowing objects to easily change the orientation and location of objects. The tests for this class have been made to verify that individual transformations (e.g. changing position, scale, rotation, orientation) are implemented correctly and act appropriately on objects. In addition to these transformations, the class also has conversion functions which convert a vector to an angle and vice versa. These tests were carried out successfully and display that the Transform class is fit for use in other classes.

A Text class has been implemented to help format the position and colour of text throughout the project. Formatting the text requires simple unit tests to confirm that the text is instantiated correctly and check that changes can be correctly made to format the text. The tests also have to be able to update in relation to parent entities using the `update()` function. These functions have been tested and tested with a 100% success rate.

In addition to automated tests, we conducted a small handful of manual tests where the whole system is implemented. We used these tests to identify possible issues with functions that were not possible to test using automated tests whether it be due to the functions not having any outputs or the tests being specifically related to graphics.

In our manual test we tested the function we tested shooting to test the different ways that shooting was implemented. We attempted shooting using the mouse/trackpad, with the cannonballs following the player's cursor which worked correctly for the player/tester. We also tested to see that the player was also capable of shooting using the spacebar, with the cannonballs shooting in the direction that the ship was facing. We showed the inputs being used in the keyboard/mouse input feed so that the inputs being used was obvious.

In addition to shooting, we tested the 2 newly implemented entities Monster and Storm. Testing the Storm class with automation was not possible due to a lack of available public variables and the random nature of the Storm events (randomly changing the direction and velocity of the player), meaning that concrete automated tests were not an option. When testing the Storm class, our objective was to test that the storm caused an obstacle by randomising the players movement and making the game harder. This was effectively shown in the footage as the player's ship moves around randomly within the storm despite receiving no input from the user.

The Monster class had many closed off/private functions, changing sprites, and overlapping functions that made testing the class hard automatically. With manual tests, we were able to test the behaviour of the monster (dormant when the player is out of range, following the player when in range) with the monster's sprite updating depending on the monster's behaviour. This is much easier to test without unit and integrated tests and instead with footage showing the class working clearly.