Software Testing Report

Group Name: Team siKz
Group Number: 6

Ryan Bulman
Frederick Clarke
Jack Ellis
Yuhao Hu
Thomas Nicholson
James Pursglove

4a)
Due to the nature of the original project, almost all of the classes and functions we would want to test were reliant on LibGdx, and its assets in order to be instantiated. Therefore, testing would require rewrites of every single one of the classes affected. Instead, a testing environment was created that allowed the assets to be loaded. It is important to note that neither the assets or their associated render functions were to be tested. The assets were loaded simply to enable the classes to be created and the functions to be called which would enable the testing of the functionality.

A variety of testing methods were used during this process. These included:

- White box testing
- Black box testing
- Integration testing

White box testing involves running a function and requiring the implementation of this function to be accurate to pass the test. An example of where this is necessary is testing the move function in the Boat class. Here, the testing output is valid if the boat correctly reacts to a simulated player's movements - with what is considered valid changing dynamically throughout the test. In a game with many moving parts, tests like this are important to ensure the systems can react correctly.

The other main method used was black box testing. In contrast to white box testing, the actual implementation of the function is largely irrelevant to the validity of the test. What matters is that the output is correct based on a preset, expected result. One example of the use of this style of testing is in the Weather class, to test which weather effect is being felt. A scenario is given, with the correct output being known in advance, and the test passes if the correct effect is returned. This is more lightweight than white box testing, and for functions where only the output is required to be correct, it is suitable to test this way.

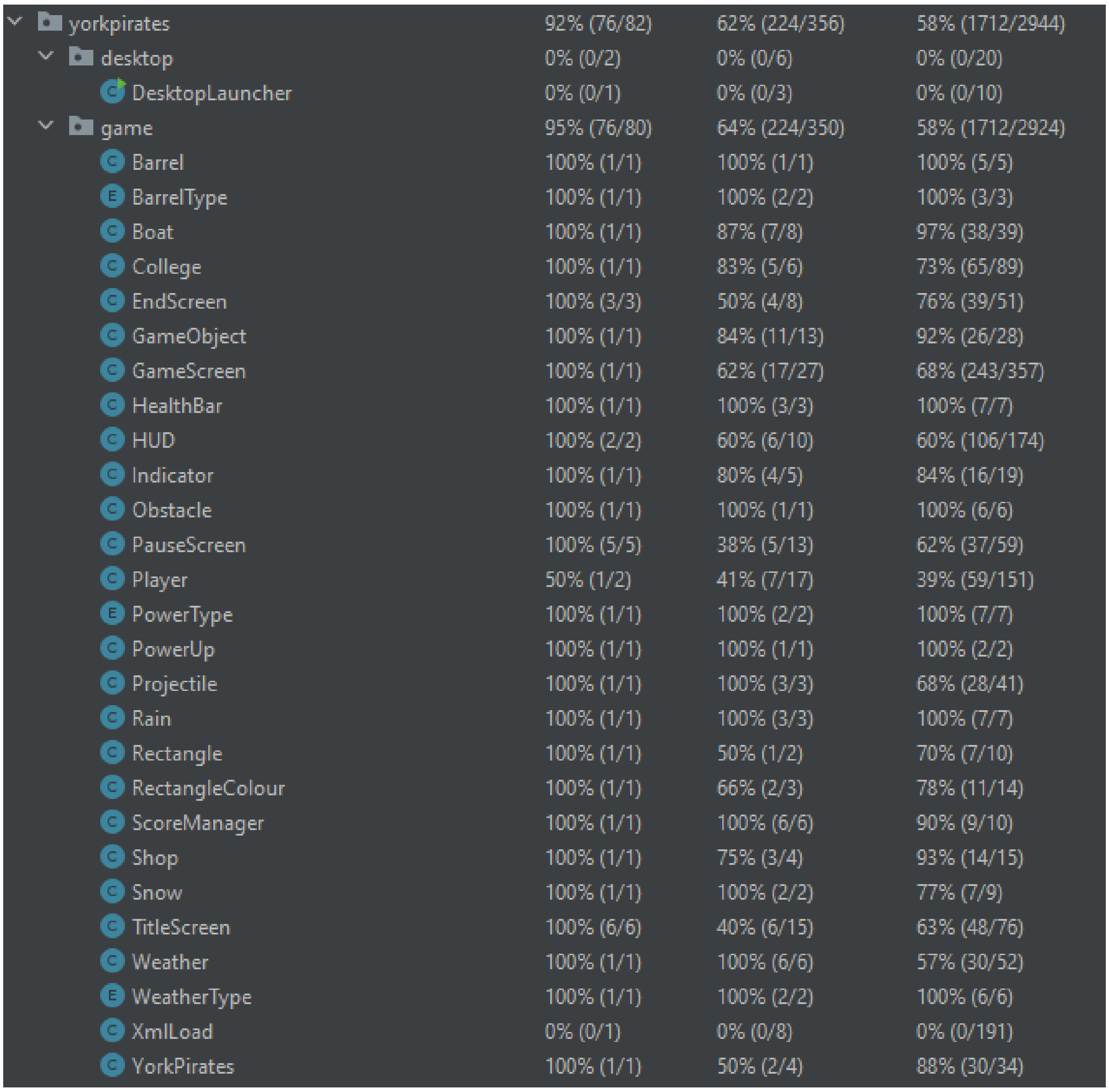
The final testing method used is integration testing. It requires actual values to be passed into the functions in order to properly confirm that they work together properly. This is not as common as the other two methods as there are not many functions that have these requirements. One example in the code is testing the upgrade function in the Shop class. This requires both a Player and the loot ScoreManager. In order to confirm the function is working correctly all parts must be correctly changed: the loot subtracted, the correct upgrade being selected and then applied to the player.

4b)
The majority of testing done within this project ended up being white box unit testing. Black box testing became quite obsolete due to certain changes from our further implementation. However, some initial black box tests were carried out when play testing the final release of part 1. Showing some understanding towards certain perks, behaviours, and problems from the project. The game was visually very pleasing but was littered with multiple graphical and memory related problems. One such issue was a significant memory leak when restarting the game either after dying or restarting from the menu. Another was its initial high memory usage by an abundant use of unneeded Arrays generated every frame. These highlighted issues lead to our team quite quickly reading straight into the implementation of the code with the ambition of understanding and solving them.

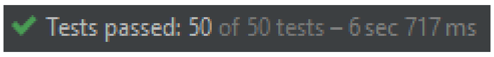
Considering the introductory interest into the code of the project most of our unit testing developed had pre-existing knowledge about the implementation of the classes we were working on. This in turn led to a majority of the unit tests written using white box testing. Mainly using this method enabled our group to write specific unit tests targeted at certain branches within functions and write the expected outcomes based on our understanding of how we thought the function worked, which was successful in effectively producing valid tests. Furthermore, when a result was unexpectedly outputted it allowed us to deepen our knowledge of the codebase by highlighting the actual functionality of the targeted code. An example of this was our testing of the Player class, which contained a very daunting function within the update loop that enabled player collision with the tiled map. Testing this function allowed our group to modify and improve it for usage within the newly added Boat class to enable AI collision detection with the tiled world.

Within most of these unit tests an integrated approach was taken, this was partially forced by the structure and relationships of the code base. The code passed to us doesn’t directly follow most object oriented programming approaches, most classes have references to parents or other classes with unrelated relationships. This was also reinforced by an abundant use of static functions nested within the program. From these points mentioned the integrated testing approach helped test correct areas as a selection of unit tests required an interaction between two classes. A good example of this is from the College class. The College class constructor requires a player instance to be passed as a parameter to set up the indicator class between the two instances. Now an integrated test is required to correctly test the College class, otherwise instantiating a College without the Player will lead to no functional indicator.

However this webbed code structure led to a much more prolonged implementation of the testing environment, testing very connected classes required an understanding of most of the codebase. Moreover setting up these tests required the instantiation of multiple specific objects to allow the correct code to be targeted. In combination with initial issues highlighted via black box testing, meant classes needed to be refactored and retested further lengthening the process.



yorkpirates	92% (76/82)	62% (224/356)	58% (1712/2944)
desktop	0% (0/2)	0% (0/6)	0% (0/20)
DesktopLauncher	0% (0/1)	0% (0/3)	0% (0/10)
game	95% (76/80)	64% (224/350)	58% (1712/2924)
Barrel	100% (1/1)	100% (1/1)	100% (5/5)
BarrelType	100% (1/1)	100% (2/2)	100% (3/3)
Boat	100% (1/1)	87% (7/8)	97% (38/39)
College	100% (1/1)	83% (5/6)	73% (65/89)
EndScreen	100% (3/3)	50% (4/8)	76% (39/51)
GameObject	100% (1/1)	84% (11/13)	92% (26/28)
GameScreen	100% (1/1)	62% (17/27)	68% (243/357)
HealthBar	100% (1/1)	100% (3/3)	100% (7/7)
HUD	100% (2/2)	60% (6/10)	60% (106/174)
Indicator	100% (1/1)	80% (4/5)	84% (16/19)
Obstacle	100% (1/1)	100% (1/1)	100% (6/6)
PauseScreen	100% (5/5)	38% (5/13)	62% (37/59)
Player	50% (1/2)	41% (7/17)	39% (59/151)
PowerType	100% (1/1)	100% (2/2)	100% (7/7)
PowerUp	100% (1/1)	100% (1/1)	100% (2/2)
Projectile	100% (1/1)	100% (3/3)	68% (28/41)
Rain	100% (1/1)	100% (3/3)	100% (7/7)
Rectangle	100% (1/1)	50% (1/2)	70% (7/10)
RectangleColour	100% (1/1)	66% (2/3)	78% (11/14)
ScoreManager	100% (1/1)	100% (6/6)	90% (9/10)
Shop	100% (1/1)	75% (3/4)	93% (14/15)
Snow	100% (1/1)	100% (2/2)	77% (7/9)
TitleScreen	100% (6/6)	40% (6/15)	63% (48/76)
Weather	100% (1/1)	100% (6/6)	57% (30/52)
WeatherType	100% (1/1)	100% (2/2)	100% (6/6)
XmlLoad	0% (0/1)	0% (0/8)	0% (0/191)
YorkPirates	100% (1/1)	50% (2/4)	88% (30/34)



✓ Tests passed: 50 of 50 tests – 6 sec 717 ms

As visible from the intellij code coverage overview of our testing includes **50** unit tests across **23** test files.The results were: **92%** of the classes were tested **(76/82)**; **62%** of the methods were tested **(224/356)**; **58%** of the lines were tested **(1710/2944)**. All **50** of these tests passed with a relatively quick compilation rate which varied between **5** and **10** seconds. The reason for such a lower method and line coverage compared to class coverage is due to the structure and functionality of how libgdx works. Since all of these tests are purely functional tests they do not require the usage of the libgdx graphics libraries. Mainly noting the **com.badlogic.gdx.scences.scene2d** and the **com.badlogic.gdx.graphics.g2d** packages. Within these packages contains sprite batches and stages designed to render components and user interface elements onto the screen. The purely logic unit tests are not required to test the render functions of each of the components as all the program functionality besides rendering is done elsewhere. Furthermore this also excludes the need for testing the desktop section of the project. The purely native classes within the subproject are entirely related to the native graphics and can't be tested as it would be unique to most devices, this is also the case for other classes within the core project causing this decrease in code coverage.

The 50 tests can be split up into three different cases of testing:
- GameObject tests
- Screen tests
- Other tests

GameObject testing being the most prominent out of all the test files refers to any instance within the project in which it is a subclass of the superclass GameObject which includes Player, College, Boat etc. The purpose of these tests was to test each class's update functionality, does the object change correctly each frame based on certain input parameters. Also to test the movement and damage functions which remained constant for all GameObjects but were implemented in slightly different ways. Finally the death test which tested correct functionality when a GameObject is destroyed. For example when a college is killed it switches to an ally, or when a boat is destroyed its asset must be removed from the world.

Screen tests involve testing the main game loops of the program for each screen. In particular GameScreen which manages most of the assets and instances of the program. These tests were to check the initialisation and update functions of the screens and check that screens are switched correctly by involving the YorkPirates class. Testing the screens requires some processing time compared to the other unit tests due to the loading of assets. Moreover, to set up the tests our team had to integrate junit with a libgdx testing application manager. This was done within the GdxTestRunner class in combination with **junit-vintage** engine, the GdxTestRunner code is based on a project by TomGrill on Github[1] which develops a basic libgdx test environment allowing access to the projects assets. Using this approach to the testing environment allowed a much larger code coverage without the need of refactoring as much code. However there was a limitation with this system, coinciding the test environment doesn't render any of the instances to the screen the graphics libraries are not loaded. This meant that all classes that interacted with previously mentioned user interface and graphic packages had to not run under a test case. This was quite easily overcome by checking if the **Gdx.gl20** graphics library had been instantiated on runtime. Once this issue was overcome, almost the whole project could be covered within the unit tests.

Lastly, other tests involved interactions with unique classes without a given superclass ScoreManager or Shop classes as examples. These classes are usually not related to graphics at all, generally the unit tests would cover up to 100% of the file. In addition most of these classes are independent and have very few relations meaning they were usually easier to test and didn't require integrated testing. An example, ScoreManager, simply contains the loot and score of the player within GameScreen. It is a purely logic class so the unit tests for it covered 100% of the files methods.

As previously mentioned 50 out of 50 of our unit tests passed, this was ensured by our group as our build scripts require no failed tests to build a release jar file. The reasoning behind this is that it prevents any problems that may be picked up by our testing environment to be highlighted and displayed to our group before a release. It also confirms that all our requirements implemented within the code work as intended based on these unit tests, and if an undesirable outcome were to appear it can be swiftly fixed and maintained.

4c)
https://tomnicho.github.io/yorkpirates/test/index.html

Bibliography

[1] https://github.com/TomGrill/gdx-testing