UNIVERSITY OF YORK

ENG1 — TEAM 18

Testing

ROB ANDERSON
MORGAN ELKINS
ZIAD ISMAILI ALAOUI
SAM KNIGHT
SCOTT REDSHAW
GEORGE TONNERO

4.a - Testing Summary of Methods

We are using 3 main testing techniques: whitebox, blackbox and unit testing. In this section we will be identifying and explaining our reasons for using each testing technique and how it relates to the project . Most of these techniques work in unison to create a fully fault proof system. We wrote each of the unit tests for the whole project in the test folder which is adjacent to the main inside the src folder.

Unit testing is an automated testing technique where it runs a script and creates tests for certain instances inside the game. We use unit tests to initialize the different screens to see if they build and see if entities are created and interact with the game as a whole. These unit tests test if the screens can be edited with different inputs from the user and test the functionality of the entities to see if they work in the desired way. Unit tests should be run after each individual entity or screen is written to make sure they work in the way that they are intended to. We are using Junit tests with the libgdx gamestate and entities to create the unit tests. Through using GitHub we have added continuous integration which checks and runs tests on the commits to the project to make sure they are correct and won't throw any errors.

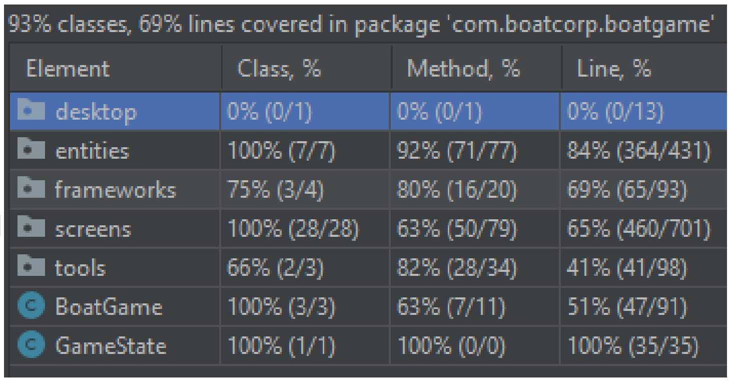
Unit testing is a large part of whitebox testing. Whitebox testing checks if the internal functions of the system works and creates the desired output. Whitebox makes sure all components, classes, entities and screens work cohesively through testing the scripts for each of these values. Whitebox testing will occur once all the unit tests for each entity and scene are written and passed. We test each function inside that entity class, which differs for each entity as they have different functionality, but for most of the classes we test the creation, disposing and the update of the game state. Some additional functions may include: testShoot, testIsHit, testTakeDamage, testGetPosition, testCombat and many more. For each screen we test the creation and disposing of the test as well as some other functionality like: setMode, upgradePlayer, screenCreated, freeze and many more.

Blacbox testing is used to test the behavior of the system as a whole through the use of manual testing. Blackbox testing is done without the knowledge of the system and how it interacts with each other, so it tests what the customers and the users will interact with and tests how the whole game play will work as a whole. Blackbox testing will occur once all the whitebox and unit tests are created as the whitebox tests are done to make sure all the components are working and provide the correct output and blackbox will test how that can be represented as a fully functioning game. We will be doing manual tests for the main potential scenarios: player collisions with other entities in the game and world border, player combat with colleges and enemy ships, player interaction with powerups and the shop and many more.

4.b - Testing Report

Unit testing:

For unit testing we used JUnit4 and mockito to create tests to evaluate our classes and methods. JUnit was used as it is an easy way to write and run these tests automatically when a test command is run. Mockito was used to mock components which can't be typically tested, such as sprite batches, and allows us to get a much larger line coverage.



93% classes, 69% lines covered in package 'com.boatcorp.boatgame'

Element	Class, %	Method, %	Line, %
desktop	0% (0/1)	0% (0/1)	0% (0/13)
entities	100% (7/7)	92% (71/77)	84% (364/431)
frameworks	75% (3/4)	80% (16/20)	69% (65/93)
screens	100% (28/28)	63% (50/79)	65% (460/701)
tools	66% (2/3)	82% (28/34)	41% (41/98)
BoatGame	100% (3/3)	63% (7/11)	51% (47/91)
GameState	100% (1/1)	100% (0/0)	100% (35/35)

The code for the unit tests can be found here and the coverage report is linked on the website and here.

Another advantage of doing unit testing with JUnit is we can use it for automated testing with CI by using it in conjunction with github actions. This allows us to automatically test each push of the java files to make sure that nothing has broken during the development of a new or improved feature. This is talked about in more detail in the CI report document.

In total we have 99 tests which cover 93% of classes and 69% of line coverage across the game. The reason why the class coverage is not 100% is because the DesktopLauncher cannot be unit tested as it is entirely graphical due to its nature. However, the desktop launcher was extensively tested in our manual testing section so it is still included in our testing report.

A ~70% line coverage for the project is a good result as it means that the vast majority of the code which was written has been tested. The majority of the lines which have not been tested are due to them being GUI elements, so cannot be easily tested in headless mode, or they are Libgdx scene2D Stage which also cannot be tested for similar reasons. Similarly to the desktop launcher, these are all tested during the manual testing phase

None of the unit tests failed, which means that all of the functions and classes, which were tested, work as intended. This means that bugs/errors which may occur during the game are caused by the functions which were not tested. Or it could mean that both the logic for the tests and the function are flawed in some way, leading to bugs in the final program.

Manual testing:

We decided to use manual testing to cover sections of the project and requirements that could not be tested through the unit tests, as stated in the prior section. Each individual manual test was carried a minimum of 5 times to ensure that the respective tests were consistent.

Six main sections were covered by these manual tests, these include :
1. Collision System,
2. Combat System,
3. Power-Ups,
4. UI-Interactions,
5. Plunder System,
6. Points System.

Within these, things such as manual user inputs, like movement and menu controls, that couldn't be tested automatically by Unit Tests, as well as collision with world borders and entities, were tested to ensure that they were running as intended.

There were two manual tests which failed, both of which stemming from the same overall issue. These were Test 6.3, Points are reset upon starting a new game and Test 5.4, Plunder is reset upon starting a new game, both of which are in regards to resetting scores at the end of a game. This was initially a problem earlier on in the project by having the values initialized to 0 upon starting a new game, however at some point a lot later on in development this issue reappeared during the final testing stage. The issue may possibly have occurred due to the introduction of multiple difficulties or save games, however it only occurs when a new game is made in a pre-existing window. If a new window of the game is opened all values will return back to their original values and a new game will be produced as intended. Overall this is a very minor error, and does not majorly affect the game at all, only really introducing a slight inconvenience by having to restart the window when wanting to play again with fresh values, however it should also be a fairly easy issue to resolve in future.