Testing

Assessment 2 Team

Team 12

Richard Liiv

Umer Fakher

William Walton

James Frost

Olly Wortley

Joe Cambridge

a) Summary of Testing Methods

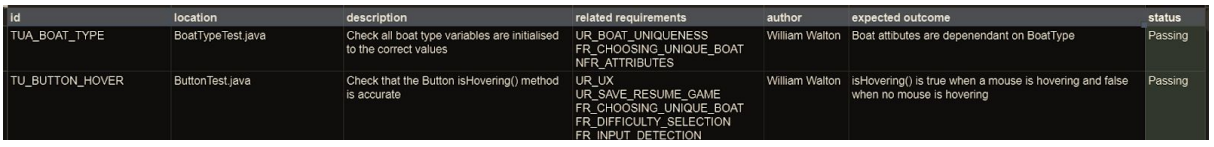
Whilst planning and designing tests, we decided that we should maximise the number of automatic tests, implementing unit tests for any components that could be tested individually. Integration tests were used heavily as a lot of the components were too reliant on each other to be tested in isolation. These allowed us to still run automatic tests for interaction between objects, for example Obstacles and Boats. (These integration tests required running the game in a headless environment, using a test runner from https://github.com/TomGrill/gdx-testing).
A large number of features could not be tested with automatic testing, for example, testing if the position of buttons were accurate or testing that the UI functioned. To test these, manual testing was employed. This testing methodology was appropriate for testing the project because, due to the nature of LibGDX, and by extension most graphical programs, the number of automated tests was limited. This was further compounded with the games heavy use of the freetype font library which we were unable to get working in a headless environment. To counter this, we tested around the library, disabling it in any automatic tests.

Tests were prioritised for core components such as the Boat and Obstacle classes. This was appropriate as it allowed us to be certain that any large or fundamental problems were caught in testing.

Black-box testing was used to allow all team members to contribute to testing without needing to learn the existing codebase. Because of this, the test cases were derived from the requirements of the project. This was appropriate as our team divided roles such that only around half of the members were active in development. The other members were assigned to planning and testing. This also helped eliminate the chance that a member may subconsciously only test cases they know work, as the person writing the test may not know the inner workings.

A small amount of white-box testing was used to test that core elements such as type enums were consistent throughout development. This was needed to ensure that these were not changed during testing by a developer then not returned to their original values.

We made a testing report Google Sheet which contained columns; test id, file location, description, related requirements, author, expected outcome and current status of the test in the master branch. A screenshot is shown below:



id	location	description	related requirements	author	expected outcome	status
TUA_BOAT_TYPE	BoatTypeTest.java	Check all boat type variables are initialised to the correct values	UR_BOAT_UNIQUENESS FR_CHOOSING_UNIQUE_BOAT NFR_ATTRIBUTES	William Walton	Boat attibutes are depenendant on BoatType	Passing
TU_BUTTON_HOVER	ButtonTest.java	Check that the Button isHovering() method is accurate	UR_UX UR_SAVE_RESUME_GAME FR_CHOOSING_UNIQUE_BOAT FR_DIFFICULTY_SELECTION FR_INPUT_DETECTION	William Walton	isHovering() is true when a mouse is hovering and false when no mouse is hovering	Passing

If you want to view on the website: https://umerfakher.github.io/ENG1Project/#testing

As an accompaniment to the test report, a naming scheme was developed for test IDs. This maintained traceability through uniform naming. Each test begins with a prefix; either TUA for simple unit tests, TU for more complex unit tests, TI for integration tests or TM for manual playtesting.

b) Test Report

Planning

Both a test report and test matrix were created in test development. The report was used to keep track of who was responsible for completing each test as well as the current status of the test. The test matrix was used to help create a test suite that tested all requirements with as little overlap as possible. An excerpt from the report and the matrix is shown below, with links to the full report available as a google sheet at (https://docs.google.com/spreadsheets/d/1X2zB9FlJgkkC5BU-5NkCIuKHkR51B-qbUA7MvZWE9mA/edit?usp=sharing), and the testing matrix available also as a google sheet, at (https://docs.google.com/spreadsheets/d/1r3KvjQNQ0kj7zwy5waTn-9Hq-sPPM9ROeYgUvQwiK44/edit?usp=sharing).

If you want to view on the website: https://umerfakher.github.io/ENG1Project/#testing

Report:

id	location	description	related requirements	author	expected outcome	status
TUA_BOAT_TYPE	BoatTypeTest.java	Check all boat type variables are initialised to the correct values.	UR_BOAT_UNIQUENESS FR_CHOOSING_UNIQUE_BOAT NFR_ATTRIBUTES	William Walton	Boat attibutes are depenendant on BoatType.	Passing
TU_BUTTON_HOVER	ButtonTest.java	Check that the Button isHovering() method is accurate.	UR_UX UR_SAVE_RESUME_GAME FR_CHOOSING_UNIQUE_BOAT FR_DIFFICULTY_SELECTION FR_INPUT_DETECTION NFR_POSITIVE_UX	William Walton	isHovering() is true when a mouse is hovering and false when no mouse is hovering.	Passing
TU_SAVING	RoundsScreenTest.java	Check the saveToFile function produces the expected output.	UR_UX UR_SAVE_RESUME_GAME NFR_POSITIVE_UX	William Walton	A text file called "testfile.txt" should be created with the given string as contents.	Passing
TU_HITBOX_POSITION	HitboxTest.java	Check that the Hitbox class reports the correct position.	UR_OBSTACLE_COLLISION UR_MOVEMENT FR_BOUNDARY_DETECTION FR_COLLISION_DETECTION	James Frost	A Hitbox created at (x,y) and moved (dx,dy) should report being at (x+dx,y+dy).	Passing
TU_HITBOX_COLLISION	HitboxTest.java	Check that the Hitbox class correctly reports collisions.	UR_OBSTACLE_COLLISION UR_MOVEMENT FR_BOUNDARY_DETECTION FR_COLLISION_DETECTION	James Frost	Collisions should be reported for Hitboxes that are overlapping, but not for Hitboxes that are not overlapping.	Passing
TU_HITBOX_LEAVE	HitboxTest.java	Check that the Hitbox class is able to determine if another Hitbox is colliding when moved.	UR_OBSTACLE_COLLISION UR_MOVEMENT FR_BOUNDARY_DETECTION FR_COLLISION_DETECTION	James Frost	A previously colliding Hitbox must report no collision if it moves out of the other Hitboxes bounds.	Passing
TUA_SCROLLINGBG	ScrollingBackgroundTest.java	Check ScrollingBackground is created with the correct values.	UR_UX FR_MOVEMENT NFR_POSITIVE_UX	James Frost	The attributes are initialised correctly.	Passing
TU_SCROLLINGBG_UPDATE	ScrollingBackgroundTest.java	Check the background moves on an update.	UR_UX FR_MOVEMENT NFR_POSITIVE_UX	James Frost	The backgroudn should move x amount when updated with a delta of x.	Passing
TUA_SETTINGS	SettingsTest.java	Check the default settings are correct.	UR_UX	James Frost	The default settings should be the same as in the design	Passing
TUA_ASSETS	AssetTest.java	Check all assets are in the correct place.	UR_UX	William Walton	All required assets should be in the assets directory and named acording ot the design.	Passing

Matrix:

Requirement Identifiers	Reqs Tested	UR_BOAT_UNIQUENESS	UR_DIFFICULTY_LEVEL	UR_PADDLERS_STAMINA_DECREASE	UR_PLAYER_PENALTY	UR_OBSTACLES	UR_OBSTACLE_COLLISION	UR_UX	UR_MOVEMENT	UR_POWERUPS	UR_RACE_TOTAL	UR_DIFFICULTY_BEFORE_GAME	UR_GAME_LENGTH	UR_GAME_END	UR_SAVE_RESUME_GAME	FR_CHOOSING_UNIQUE_BOAT	FR_DIFFICULTY_SELECTION	FR_OBSTACLE_RATE	FR_BOUNDARY_DETECTION	FR_AWARD_PENALTY	FR_OBSTACLE_SPAWN	FR_COLLISION_DETECTION	FR_HIT_DECREASED_BOAT_CONDITION	FR_INPUT_DETECTION	FR_MOVEMENT	FR_POWERUP_RATE	FR_QUALIFIER_RACES	FR_FINAL_RACE	FR_TIMED_DECREASED_BOAT_CONDITION	NFR_MOVEMENT_EXPLANATION	NFR_RULES_EXPLANATION	NFR_POSITIVE_UX	NFR_END_SCREEN	NFR_ATTRIBUTES	NFR_GAME_DURATION
Test Cases																																		
Tested Implicitly																																		
TUA_BOAT_TYPE		1														1																		1	
TU_BUTTON_HOVER								1							1	1	1							1								1		
TU_SAVING								1							1																	1		
TU_HITBOX_POSITION							1		1										1			1												
TU_HITBOX_COLLISION							1		1										1			1												
TU_HITBOX_LEAVE							1		1										1			1												
TUA_SCROLLINGBG								1																	1							1		
TU_SCROLLINGBG_UPDATE								1																	1							1		
TUA_SETTINGS								1																										
TUA_ASSETS								1																										
TUA_FRAMEWORK_TEST								1																										

Automatic Tests

If you want to view on the website: https://umerfakher.github.io/ENG1Project/#testing
Junit4 was used for creating automatic tests. All tests are located in the tests directory in both the team's github repo and the source zip file. These were run with the team member's IDE during development of each test, or in bulk with the bundled gradle files. The link to the test files is: (https://github.com/UmerFakher/ENG1Project/tree/master/tests). An example of one of the tests to test that moving reduces stamina, with it being run as shown below:

```
@Test
public void staminaUsageTest() {
    ComputerBoat boat = new ComputerBoat(BoatType.FAST, 1,  name: "__testing_boat__",  pickSpeedValue: 1);
    float oldStamina = boat.getStamina();
    for (int i = 0; i < 5; i++)
        // ComputerBoats always use stamina when they have it
        boat.update( deltaTime: 1);

    Assert.assertTrue( condition: oldStamina > boat.getStamina());
}
```

```
linux-terminal $ ./gradlew :tests:test --tests "com.dragonboatrace.entities.boats.BoatObst
acleTest.staminaUsageTest"

BUILD SUCCESSFUL in 1s
```

Completeness of Testing

All of our automatic tests pass, because of this we decided to check that our testing was sufficient and thorough enough. To do this, we used the inbuilt code coverage tool in IntelliJ IDEA while running tests. We did this to get an idea of how much of the codebase was being tested or ignored by our unit tests.

Element	Class, %	Method, %
entities	100% (10/10)	71% (60/84)
screens	54% (6/11)	21% (20/92)
tools	100% (5/5)	78% (29/37)
DragonBoatRace	100% (1/1)	46% (6/13)

The code coverage, shown above, shows that all the core elements of the game that were decided to have high priority (entities and tools) are tested with 70-80% code coverage. The team decided this was sufficient as, under further examination, the methods not being tested were for drawing textures to the screen. These methods were tested in manual testing thus no automatic testing was required, or possible in some cases.

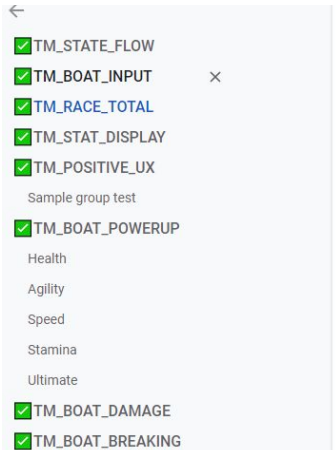
Proof of Testing

Each commit in the github repo has an associated test result indicating whether the commit passed or failed all the tests by a tick or cross, an example is shown here with the commit "boat tests" failing:

(https://github.com/UmerFakher/ENG1Project/commits/master)
An example of where testing found bugs would be when saving was added to the game. The test was designed to check for invalid and valid input, however, the implementation did not function for invalid inputs. Subsequently, the implementation for the save function needed to be updated to detect for invalid inputs.

Manual Playtesting

The team used manual playtesting as the method for testing anything too complex or anything where running the game was more appropriate. For example, TM_RENDER would be impossible to test in a headless environment as there is no screen to draw to. Because of this, a manual test was created.



To document manual tests, screenshots and screen recordings were used. The full report for all manual tests is found on a google doc linked both here and on the website: (https://docs.google.com/document/d/1OsZBECGvhG06pesvoOp2_sTIpsDf5wsfWKjsepcreEg/edit?usp=sharing). A screenshot from the document used for documenting the manual tests is shown above.

Problems Found During Playtesting

The textures and colour choices in the game were found to be hard to read in the manual test TM_POSITIVE_UX. To address this, the texture of the water and the colour of text was changed. The before and after textures are shown below:

(this screenshot may not be very representative of the actual appearance of the game, so please try the game to see)
A more detailed report of all manual testing, with evidence, is in the google doc found below: https://docs.google.com/document/d/1OsZBECGvhG06pesvoOp2_sTIpsDf5wsfWKjsepcreEg/edit?usp=sharing
If you want to view on the website: https://umerfakher.github.io/ENG1Project/#testing