Testing

Assessment 2

Team 10: Hard G For GIFs

Dragos Stoican

Rhys Milling

Samuel Plane

Quentin Rothman

Bowen Lyu

a)

Testing methods and justifications:

Black box testing:

Black box testing is done only through manual tests since automating these was beyond the scope of the ENG project. All the tests are created based on the requirements document, with the goal being to cover all the requirements if possible. Given the limited amount of time, we only test the user requirements and the functional requirements. Extra attention is put on testing elements of the game that require user input, like clicking buttons or keys to move the boat. This is the case because testing for input is not possible in libGDX testing with headless application. This also applies to UI elements that cannot be tested in the automated tests so we focus on them here. For us, the ultimate goal of black box testing is to find bugs and to check if requirements are met, or find new requirements through exploratory testing. Tests 1.x.x - 1.x.x cover invalid inputs, other tests cover inputs we thought are best for testing the requirements are met, but also boundary inputs that are more likely to identify bugs. Evidence of testing, as well as a traceability matrix are available on the website.

White box testing:

Using the GdxTestRunner class available at https://github.com/TomGrill/gdx-testing we can run libGDX in headless mode and test our code with automated JUnit tests. Tests are again created based on covering as many requirements as possible, but this time we also take into account getting as much code coverage as possible. Because in headless mode we can't test draw methods or anything that requires user input, like clicking buttons, the code coverage doesn't reach 100% but we try to cover as much as possible.

After we have written all the tests for the existing features made by the previous team, the workflow of the team consisted of developing a new feature, then creating the appropriate unit tests for it before moving on to the next one. Using this tactic allows us to make sure that once we add new changes to the code base, the old features remain unaffected. All white box tests are also executed on every push to the github repository, so if a bug is introduced the team is notified and a github issue can be opened and assigned to a member to fix the bug.

These tests are documented in the code with the appropriate id, description, input, expected outcome and author. The code coverage reached is 63% of classes and 55% of lines. The tests can be run all at once by executing the "test" gradle task, or separately by using the quick run option offered by most IDEs in each test class.

Justification:

We believe the approach adopted for the automated white box tests is appropriate for our project because throughout development we need to make sure that we are not unintentionally affecting the features implemented by the previous team, or the ones implemented by us. This also fits very well with our continuous integration, because we are able to run these tests on every push to the github repository.

The black box testing approach is the most suitable one for a project of this scale, and with a limited amount of time. We believe that through manual tests we can easily identify if all the user requirements are met by the game.

b)

Black box testing:

Link to manual tests table: https://hardgforgifs.github.io/assessment-2/tests/manual-table.html
Link to traceability matrix: https://hardgforgifs.github.io/assessment-2/tests/traceability-matrix.html

Statistics:

When covering manual tests we aimed to cover as many requests as possible. As we said we focus on covering all the user and functional requirements so the coverage here is also pretty good. The requirements that are not tested are omitted because they are either hard to test in a replicable way, or hard to test in a meaningful way.

User requirements tested: 18/19

Untested user requirements:
UR_PERFORMACE. Not tested because we haven't set a standard for what "good" performance is, so it is up to the user to decide if the game runs well or not on his/her computer.

Functional requirements tested: 28/32

The untested functional requirements are:
FR_INCREASE_OBSTACLES: It's hard to manual test this because the tester would have to count the number of obstacles in each leg. However this test is covered by our automated JUnit tests.
FR_DIFFICULTY_DISPLAY: This is not implemented so we cannot test for it
FR_NO_SAVE_FILE: As there is no way to remove a save file, this test can only be done before any save file is saved on a computer. This would be hard to replicate, so we decided not to cover it in our manual tests
FR_FULLSCREEN: This is not implemented so we cannot test for it

Failed tests:
The only test that fails is 1.0.11, because the feature is not fully implemented into the game.
The tests were designed prior/during the development process to show if any requirement isn't met. Most of the tests that were failing have been fixed by the end of the development period, so we are left with a table with mostly passed tests.

63% classes, 55% lines covered in package 'com.teamonehundred.pixelboat'

Element	Class, %	Method, %	Line, %
desktop	0% (0/1)	0% (0/1)	0% (0/6)
AIBoat	100% (1/1)	88% (8/9)	95% (62/65)
Boat	100% (1/1)	93% (43/46)	91% (175/191)
BoatRace	100% (1/1)	80% (17/21)	78% (143/183)
CollisionBounds	100% (1/1)	100% (10/10)	100% (34/34)
Effect	100% (1/1)	88% (8/9)	80% (21/26)
GameObject	100% (1/1)	90% (10/11)	95% (47/49)
InvulnerabilityEffect	100% (1/1)	66% (2/3)	70% (7/10)
ManeuverabilityEffect	100% (1/1)	66% (2/3)	70% (7/10)
MovableObject	100% (1/1)	100% (13/13)	100% (30/30)
Obstacle	100% (1/1)	100% (3/3)	100% (6/6)
ObstacleBranch	100% (1/1)	100% (2/2)	100% (15/15)
ObstacleDuck	100% (1/1)	100% (3/3)	100% (28/28)
ObstacleFloatingBranch	100% (1/1)	100% (1/1)	100% (5/5)
ObstacleLaneWall	100% (1/1)	80% (4/5)	81% (9/11)
PixelBoat	33% (1/3)	23% (3/13)	46% (104/225)
PlayerBoat	100% (1/1)	100% (8/8)	91% (56/61)
PowerUp	100% (1/1)	100% (4/4)	100% (27/27)
RepairEffect	100% (1/1)	66% (2/3)	72% (8/11)
SceneBoatSelection	0% (0/1)	0% (0/7)	0% (0/167)
SceneMainGame	100% (1/1)	71% (10/14)	66% (75/112)
SceneOptionsMenu	0% (0/5)	0% (0/14)	0% (0/95)
SceneResultsScreen	0% (0/1)	0% (0/7)	0% (0/76)
SceneStartScreen	0% (0/1)	0% (0/5)	0% (0/74)
SceneTutorial	0% (0/1)	0% (0/5)	0% (0/27)
SpeedEffect	100% (1/1)	66% (2/3)	75% (9/12)
StaminaEffect	100% (1/1)	66% (2/3)	72% (8/11)

Statistics:

This is the coverage report produced by the IDE when running the JUnit tests. Line coverage is 55%, but the percentage is heavily affected by some classes that only consist of untestable elements, like most of the Scene objects. LibGDX tests can't test for user input, so these classes that are heavily reliant on input can't be tested. In addition, we also cannot test what is displayed on the screen, which is a big part of the code base, so the percentage is influenced a lot by this.

The classes that are tested the most are the ones that have been modified the most. This is because we are aiming to produce tests as we added new features so the implementation is not affected by features we implement in the future.

None of the automated tests fail.

c) Evidence of testing:

Manual tests evidence link: https://hardgforgifs.github.io/assessment-2/tests/manual.html
Unit tests evidence link (online test report): https://hardgforgifs.github.io/assessment-2/tests/unit/