CMPT 276 Group 15 Phase 3 Document

Test Quality and Coverage

For our testing we mainly would like to test the interaction between the main player, enemies, items, and obstacles. In this document the coverage for these tests are organized by the java classes which they offer test coverage for. For a list of tests, please see the Test Automation section. In the Integration Test Reasoning section a brief justification for the integration tests is given.

Line Coverage

For our testing we chose to go with line coverage due to the large amount of possible branches. In the Test Automation section, each test will say the estimated line coverage next to its header. We know that line coverage = lines_covered / total_lines. Our goal was to achieve a line coverage of about 50% in our tests, but sometimes this was not possible. We will cover this more in the Test Quality Assurance section.

Branch Coverage

Branch coverage for our project is very difficult to test because the player has almost limitless choices for movement. Because of this it is not possible to have an accurate estimate of the branch coverage. However in our Test Automation section a “Test Coverage” percentage can be seen for every test that we completed.

Test Quality Assurance

There are certain interactions in our code which have not been fully covered in our tests. For example there are no specific tests for our keyboards IO due to the fact that we could not input keys into our Unit Tests automatically. There also could be more testing of the GUI if we had access to tools such as Selenium. This would be an excellent integration for the future, but is outside the scope of this course. Lastly a large spot of missing coverage would be in our Level.java class where the level is generated. It is difficult to create specific tests for this due to the infinite possibilities that can happen during a game.

To gain as much coverage as possible we decided to create Unit Tests for each classes’ methods when it was appropriate. In doing so we achieved a good amount of line coverage in our program.

Test Automation: Integration and Unit Tests

Character

PlayerActor | Line Coverage: 56% | Test Coverage: 74%

Unit Tests

1. testSetCharacterType()
    a. Tests if the Player’s CharacterType enum is changeable.
2. testSetHealth()
    a. Tests if the Player’s health can be changed within its bounds [0,100].
3. testSetHealthException()
    a. Tests if the Player’s health fails outside of its bounds [0,100].
4. testSetSpeed()
    a. Tests if the Player’s speed can be properly set.
5. testSetWalking()
    a. Tests that the Player correctly flags if they are walking.

6. testSetStepsAllowed()
    a. Tests that the player's steps allowed are properly changed.
7. testSetStepsAllowedExeption()
    a. Tests that the player cannot have negative steps allowed.
8. testSetPosition()
    a. Tests the setting of a Player's position.
9. testSetStartState()
    a. Creates a new PlayerActor and tests that the start state is working correctly.
10. testSetDirectionFacing()
    a. Tests the Player's Direction is correctly set.
11. testSetNextMovement()
    a. Tests the Player's Next Movement is correctly set.
12. testMoveCharacter()
    a. Tests a Player's ability to move north, south, east, and west.
13. testAddToHealth()
    a. Tests that the player's health changes when picking up different types of items.

Skeleton | Line Coverage: 56% | Test Coverage: 72%

Unit Tests

1. testMoveCharacter()
    a. Tests a skeleton's ability to move north, south, east, and west.
2. testSetPosition()
    a. Tests the setting of a skeleton's position.
3. testheroKill()
    a. Tests to see if skeleton kills hero
4. testNOTheroKill()
    a. Tests to see if a skeleton does not kill a hero when not needed.
5. FollowPlayer()
    a. Tests to see if skeleton follows the player

Zombie | Line Coverage: 60% | Test Coverage: 77%

Unit Tests

1. testheroKill()
    a. Tests to see if zombie kills hero
2. testNOTheroKill()
    a. Tests to see if a zombie does not kill a hero when not needed.
3. rushTest()
    a. Tests to see if zombie switches rushing direction upon hitting wall
4. testMoveCharacter()
    a. Tests a zombie's ability to move north, south, east, and west.
5. testSetPosition()
    a. Tests the setting of a zombie's position.

Clock

BonusTreasureClock | Line Coverage: 18% | Test Coverage: 29%

Unit Tests

1. testGetTime()
    a. Tests that the timer is correctly counting.
2. testIsVisible()
    a. Tests that it is possible for the bonus reward to become visible.

Timer | Line Coverage: 8% | Test Coverage: 17%

Unit Tests

1. testSetStartTimer()
    a. Tests if the timer is properly stopping and starting.
    b. TrapClock /src/test/Clock/Timer

Trap Clock | Line Coverage: 4% | Test Coverage: 17%

Unit Tests

1. testSetIsHurting()
    a. Tests if the player is properly toggling isHurting.

GUI

Boot | Line Coverage: 73% | Test Coverage: 85%

Integration Tests

1. bootTest()
    a. Tests if the system will properly load the GUI.
2. bootStateTest()
    a. Tests if the system is able to boot the title, game, and end screen.

GameState | Line Coverage: 8% | Test Coverage: 50%

Integration Tests

1. testGameStateWin()
    a. Test that if the player wins the game.
2. testGameStateLose()
    a. Test that if the player loses the game.
3. testGameStatePLAY()
    a. Test that if the player is playing the game.
4. testGameStateTitle
    a. Test that if the player is on the title page.

Items

BonusTreasure | Line Coverage: 22% | Test Coverage: 37%

Unit Tests

1. getImageTest()
    a. Tests if the system will properly load the image of BonusTreasure.

2. testBonusTreasurePoint()
    a. Test that the point is 1

3. testUpdate()
    a. Test that the update() is working

ExitCell | Line Coverage: 15% | Test Coverage:33%

Unit Tests

1. testExitCellPoint()
    a. Test that the point is Constants.EXIT_CELL
2. getImageTest()
    a. Test that getImage() is working
3. testUpdate()
    a. Test that the update() is working

ItemDetection | Line Coverage: 40% | Test Coverage: 59%

Unit Tests

1. testDetectionWithTrap()
    a. Test that the player intersects with the trap in position 1, 1
2. testDetectionWithTreasure()
    a. Test that the player intersects with the treasure
3. testDetectionWithBonusTreasure()
    a. Test that the player intersects with the BonusTreasure
4. testDetectionWithExitCell()
    a. Test that the player intersects with the ExitCell

Trap | Line Coverage: 18% | Test Coverage: 37%

Unit Tests

1. testTrapPointl()
    a. Tests if the player will lose one point

Treasure | Line Coverage: 18% | Test Coverage: 74%

Integration Tests

1. getImageTest()
    i. Tests the image loaded by a treasure.

Map

Grid | Line Coverage: 66% | Test Coverage: 44%

Unit Tests

1. testGetLevel()
    a. Tests that the level created by Grid is not null.

Level | Line Coverage: 48% | Test Coverage: 29%

Unit Tests

1. testCollisionCheck()
    a. Tests if a character will collide with a wall.

Obstacle | Line Coverage: 1% | Test Coverage: 8%

Unit Tests

1. testGetPosition()
    a. Tests if the obstacle's position has been properly placed.

Sound | Line Coverage: 7% | Test Coverage: 20%

Integration Tests

1. testPlay()
    a. Tests that all sound files load.
2. bootStateTest()
    a. Tests that a sound file can loop and stop the loop.

Integration Test Reasoning

For our integration tests we focused on testing the interaction between the GUI, Sound, Images, and Game States. Specific tests can be found in the above "Integration and Unit Tests" section of this document. It was a struggle to perform integration tests due to the lack of tangible feedback from the system, so we focused on making sure that the integration tests simply did not crash when they ran. Out of the four elements listed above, all of them rely on loading external elements (GUI, images, sounds).

For the GUI integration tests we tested that the GUI would launch and would be able to cycle through the different screens associated with game states. For example a test we wrote will cycle through the title screen, main player screen, and end screen.

For the Sound integration tests we simply focused on loading every sound available to the player to make sure that each file loads properly.

For the Game State integration tests, we tested the results that will happen when the player wins, plays, and loses the game.

For the Image integration tests we tested that image assets would properly load for the user of the game.

Findings

High Coupling and Low Cohesion

Writing tests for the project required a complete refactor of the project's structure. In Phase 2 we had a high amount of coupling between our classes. Our Characters, Items, and Level classes all relied on our Grid class, which was really just our class which painted our UI. This coupling was due to the fact that we had constants inside of the Grid class which required other classes to use it as a dependency.

The Grid class having two different classes inside of it is an example of low cohesion. It was really trying to be two classes, so it needed to be broken up. As a solution, we moved the constants out of the Grid class and into the Constants class which made it so other classes would no longer need to have Grid as a dependency.

Further refactoring needs to occur in our Level class, which we have plans to tackle in Phase 4 of the project.

Bugs Found

While making the tests we encountered some bugs. In particular there were bugs that may have been created from multiple people working on a class at the same time. For example, the playerActor was

given a _health attribute, but also inherited a health attribute from the Character class. This resulted in the character never being able to adjust their health.

There is also a current bug in our integration test where the background sound cannot be stopped as it is a very long file that gets trapped in a background thread. This will be addressed in Phase 4 of the project.

Takeaways

It would have been much easier if the team had made Unit Tests while in Phase 2. Obviously we had not learned about these yet, but it would have been helpful to encourage tight coupling and better cohesion. These two things would probably have resulted in a refactoring of the system earlier in our development cycle.