Phase 3 Report

Findings:

We started off by working as a team to divide up some unit and integration tests for the code that we had previously written in phase 2. In order to ensure complete coverage of all of our code with tests we went through and as a team looked at each of the classes within our code. Each member from our group split off and worked in pairs in order to make sure that we were writing code that not only worked but made sense. By having two members of our team working in tandem together this allowed for us to clear through any bugs we did come across much more quickly than had we been alone in our coding. Through the use of peer review we were able to detect code which no longer made sense. This also meant that we were able to quickly identify and change some portions of code which needed to be refactored. For instance our coins and keys were being spawned in separately which was basically just copied pasted code. Instead we replaced the code with for loops which simplified the code a lot and neatened it up.

Due to doing some refactoring and bug fixing we were able to actually significantly improve some sections of our code. In our partner teams we worked on creating some JUnit tests which helped with regression testing, these were done to test core features and interactions. We also ran some exploratory tests that were not just regression testing and fixed some bugs and issues which we hadn't bumped into when normally running through our code. An example of that was when you moved back and forth repeatedly into a wall you could slip between the two wall segments. Since then we were able to go back and fix them.

A non-exhaustive list of things that were updated and some JUnit tests were.

<u>Debugging:</u>

- whilst creating assert tests for the MainPlayer and Damage class I realised if a player hit back to back fire objects the player wouldn't lose a heart for the second fire object. The problem was the player was only losing 1 life value when being hit by fire instead of 2.

- Found a bug where if you hold "A" against a wall and then also press "D" the player will phase through the wall. Fixed the bug by adding a check to see if when the player is moving left the right key is not pressed and in the player is moving right the left key is not pressed in the method that checks for player wall collision.

- Damage cool down while using a shield against fire was 1 second instead of 2 which resulted in player not being able to push through fire as our game design intended.

- Found a bug where the coins were popping up on the win, lose, and home screen. Fixed the bug by adding an if statement in the draw method which checks if the current screen is not win, lose, or home and then allows the coins to print.

Refactoring:

- Added some more detailed comments and citations to the GamePanel class to get a better understanding of the functionality of the class.

- Added citations in the MainMap class and improved comments for better detailing of the class.

- Removed unused KeyHandler object from coin class and its constructors as well. This forced us to update all of the constructor calls to this class from different other classes.

- Removed unused KeyHandler field from Shield,Sword and Key class and updated its constructor and its calls.

- Changed grass, fire,and wall fields in the WorldMap class directly to BufferedImage objects. This enabled us to delete the IndividualWorldAsset class.

- Changed the name of the fire field in the WorldMap classroom pit as fire was a typo.

- Removed unused Soundclass.

- Changed the amount of how much a player can clip a wall in the collision class.

- Made some changes within the GamePanel class to loop through spawning instead of just spawning each thing individually.

TESTING:

Functional Testing:
- We are using JUnit for Automatic Testing of our code.
- Added Maven dependencies for JUnit class.
- We added getters and setters for many classes so that they can be put to testing.

Test Samples:
The code coverage was complete as far as we could tell as pretty much every method or class had a JUnit test for it or is already covered by a test which comes before it. These JUnit tests are properly documented in the code however here are some breakdowns of what each test covers.

Tests follow a similar pattern of Test ----- followed by whatever class, or function its testing some examples are
Game Panel Tests:
testGamePanelSetWindow(); - Tests that each level can be selected
testGameTotalGamePanels(); - Tests the total number of game panels initialized is correct

testGameScore(); - Tests if the game score is working as intended
testScoreReset(); - Tests to see if the game score is reset at level 0 / main menu
testResetTime(); - Tests to see if the game time is reset at level 0 / main menu

Collectibles tests:(Includes all collectibles like coins, keys etc)
All of the tests have the following tests:
testCollectibleXPos(); - checks x position of collectible
testCollectibleYPos(); - checks y position of collectible
testCollectibleVisibility(); - Checks to see if the visibility can be toggled
testCollectibleCollected(); - Checks to see if the toggle / amount collected
testCollectibleGetWindow(); - Checks to see if we can get what window the collectible is on
testMainPlayerCollisionWithCollectible(); - checks if the main player can collect the Collectible

MainPlayer Tests:
testMainPlayerIsHostile(): - main tests for the player, checks that it was initialized correctly
testPlayerXPos(); - checks x position of player
testPlayerYPos(); - checks y position of player
testMainPlayerIsProtected(); - Checks to make sure the player can be protected or not
testMainPlayerHorizontalSpeed(); - checks to make sure that the speed of the player is initialized properly
testMainPlayerVerticalSpeed(); - checks to make sure that the speed of the player is initialized properly
testMainPlayerLives(); -checks to make sure that the lives are initialized properly

Damage Tests:
testEnemyDamagesRegularPlayer(); - checks to see if the player takes proper damage from enemies
testEnemyDamagesPlayerWithSword(); - checks to make sure that the player doesn't take damage with a sword
testEnemyDamagesPlayerWithShield(); - checks to make sure that the player doesn't take damage with a shield
testFireDamagesRegularPlayer(); - checks to see if the player can take damage from fire
testFireDamagesPlayerWithShield(); - Does the user get damaged by fire when they have the shield

Collision Tests:

CollisionsTest();
testMainPlayerCollisionInCenterOfTileGoingUp(); - Tests the main players collisions with all objects

testMainPlayerCollisionInCenterOfTileGoingDown(); - Tests the main players collisions with all objects
testMainPlayerCollisionInCenterOfTileGoingLeft(); - Tests the main players collisions with all objects
testMainPlayerCollisionInCenterOfTileGoingRight(); - Tests the main players collisions with all objects

Door Test:
testDoorXPos(); - checks x position of Door
testDoorYPos(); - checks y position of Door
TestDoorGetWindow(); - makes sure door is on right window
testDoorGetDestinationWindow(); - tests that door takes you to the correct level
testMainPlayerCollisionWithDoor(); - tests player and door interaction.

Enemy Test:
testEnemyIsAlive(): tests if enemy is alive
testEnemyKill(): make sures that the enemy kills the player, lives get reduced by one.
testEnemyIsHostile(): tests if the enemy does not kill the player
testEnemyIsVisible(): tests if the enemy is visible on the map.
testEnemyMoving(): checks if the enemies is moving positions

Fire Test:
testFireXPos(): checks x position of Fire
testFireYPos(): checks y position of Fire
testFireGetWindow(): makes sure Fire is on the right window
testFireVisibility(): tests if the fire is visible on the map.

Non-Functional Testing:

- Data , code and methods have been checked for repetition.
- Our game has been checked for any dead codes as well.

Difficulties and Challenges:

- Team meetings were not long during this phase as everyone had multiple assignments, so it got a bit difficult communicating and dividing up tasks.

- We used class resources and some tutorials on youtube to write up our JUnit tests.

Coverage

We do not have 100% test coverage for both line coverage and branch coverage as we left out a few methods from our testing that we could not test or were redundant. For example, we did not add any tests for the MainMap class as it's simply read loaded in game textures, drew the game, and loaded in the map design. All these methods could be checked simply by looking at our game as it was running and they also didn't change from the beginning of the game's start. Another example of this is we did not add any tests for checking enemy wall collision detection as it is virtually the same as the collision tests for the player. We also did not test most constructors as the validity of these constructors would be tested when testing other parts of the code. For example, the constructor for a MainPlayer Character sets where the player is placed in the game. Instead of explicitly testing that the Character was in the right spot we could confirm it, using our wall collisions tests. For example, if the Character did not hit a wall where it was supposed to then that means the character was not in the right spot, to begin with. Besides these few reasons why our tests do not have 100% test coverage the rest of our tests does cover a very good range of our code. For example, for our MainCharacter wall collision detection tests we have several cases taken into account which covers every branch and line of the methods it's testing. We have included scenarios where the player is perfectly inside of 1 tile and moves either left, right, down, or up into a wall or a grass tile which means our tests must check if the Character made contact with exactly 1 wall or grass tile left, right, below and accordingly checks if the Character stops or moves. We also have taken into account scenarios where the user is in between 2 tiles and moves either left, right, up, or down into 2 walls or grass blocks. This means our test must also check if the user made contact with either 2 walls or 2 grass blocks either left, right, below, or above them and accordingly see if the Character stops or moves. We have several more examples of this rigorous testing such as in our Player enemy damage tests. We have several test cases for if a user hits fire, with a sword, with a shield, hits an enemy with nothing, etc. Ultimately, our code does not have 100% test coverage however, our code has still been rigorously tested.