Phase 3 Report

Set of Interactions and Tests

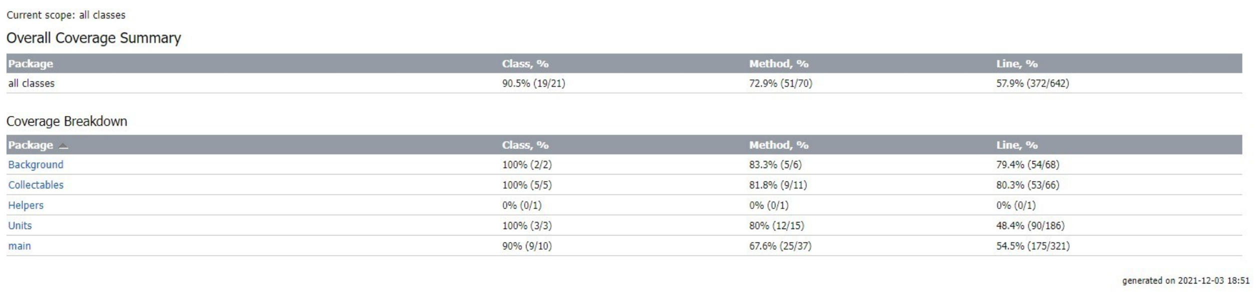
The overall game is run through *Main* class which instantiates a *Panel* that is responsible for controlling the game loop and instantiating all other logical aspects of the game.

- We have separate classes for Mouse and Keyboard event listeners that detect input from the user. These are represented by the *MouseListener* and *KeyListen* class.
    - There are no automation tests created for this as they are tested manually through the UI however there are tests created for validating and application of these inputs.
    - The *StartMenu* class manages the opening menu for starting the game.
    - *StartMenu.checkClick()* method is tested by *TestStartMenu_checkClick()* method and *render()* method is tested manually through the UI.
- The *BackgroundController* class is responsible for the managing and rendering the background resources of the map for the game.
    - *loadBackgroundTileImage* loads up all the images from the file system. The *tiletool* function is being used to read the image from the file system. This function is tested by the *TestBackgroundController_tiletool()* method.
    - *mapGen()* function reads the map matrix text file from the file system. This function is tested by the *TestBackgroundController_mapGen()* method
    - *render()* method renders the map and the background onto the UI using the map matrix and the loaded images. This has no automation tests and so is tested manually.
- The *HUD* class is responsible for keeping track of the player score., time and map level for the game. It has *render()* method which renders the information on to the screen such as showing reward collected, any collisions with player and triggers the game over or next map progression.

- The *Enemy* class represents moving enemies in the game and is responsible for loading the enemy images for each animation, updating the enemy positions and rendering the enemy on to the screen.
    - *loadEnemyImages()* loads all the enemy sprites from the file system. A similar test has been written for the background controller function, so no separate automation test was written for this.
    - *eucilideanDistance()* is used for checking how far the enemy is away from the player during spawning or collision checking. This method is tested by the *TestEnemy_euclideanDistance()* method.
    - *resetValues()* spawns the enemy in a random position on the map without being too close to the player. This method is tested by the *TestEnemy_resetValues()* method.
    - *update()* method updates the enemy position on the map relative to the player location as well as checking for collisions with obstacles on the map.. This function is tested manually through the UI rather than through automation as it was quite difficult to encompass all logic and setup.
    - *render()* displays the enemy on the screen and this is tested manually through the UI.
- The *Player* class is similar to the *Enemy* class and is also a child of the *Unit* class. It is responsible for managing the sprites, update and collision checking for the player.
    - *resetValues()* resets the player position on the map by placing the player on the top left corner of the map. This is tested by the *TestPlayer_resetValues()* method.
    - *loadPlayerImages()* accomplishes the same object as other similar functions and so does not have an automation test.
    - *update()* method checks for collisions with obstacles and other entities on the map as well as updates the player position given the input from the user. This method also triggers other parts of the system such as updating scores, map progression, etc. This function is tested by the *TestPlayer_update()* method.

    - *render()* method renders the player image on to the screen and hence is tested manually.
    - The getter methods for the *getWindowPositionX/Y* are tested by the *TestPlayer_getWindowPositionX* and *TestPlayer_getWindowPositionY* methods.
- The *CollectablePlacer* class places and manages all the Collectables. It places each object onto an appropriate random tile and keeps track of them throughout a game.
    - *isUsedPosition()* is a private method that determines whether a position on the board is being used by another *Collectable*. It is used in *setCollectables()*, and tested alongside it in the *TestCollectablePlacer_setCollectables()*
    - *setCollectables()* places normal and bonus rewards as well as traps on random grass tiles, and places an exit on the bottom edge of the map. This is tested by the *TestCollectablePlacer_setCollectables()* method.
    - *onCollide()* receives the position of a Collectable that has been collided with by the player, and triggers the Collectable's collect method, as well as removing the Collectable from the game when appropriate. This is tested by the *TestCollectablePlacer_onCollide* method for each type of collectable (*NormalReward, BonusReward, Trap and Exit)*
- The *Collectable* class is the parent class of all collectable objects. This includes *NormalReward, BonusReward, Trap, and Exit.* It implements the *render()* method and outlines the *collect()* method
    - The *render()* method, which displays a sprite onto the object's position on screen. This has no automated tests, and was tested manually.
    - *NormalReward* is a subclass of Collectable. It is responsible for the normal rewards that the player has to collect in order to progress through the game.
        - *collect()* increases the score in the *HUD* by a set amount, updates the reward count on the UI, and tells *CollectablePlacer* to remove this instance.
    - *BonusReward* is a subclass of *Collectable*. It is responsible for the bonus rewards that increase the player's score but are not required to progress through the game.

        - *collect()* increases the score in the *HUD*, and tells *CollectablePlacer* to remove this instance.
    - *Trap* is a subclass of *Collectable*. It is responsible for the traps that decrease the player's score.
        - *collect()* decreases the score in the *HUD*, and tells *CollectablePlacer* to remove this instance.
    - *Exit* is a subclass of Collectable. It is responsible for the exit that appears after collecting all of the rewards.
        - *collect()* displays a message on the UI, and tells the *HUD* that the level is over.
    - Every *collect()* method is tested by the *TestCollectablePlacer_onCollide()* method
- The *checkCollision* class is responsible for checking if a collision is going to occur between the player and a collectible, the player and a tile, the enemy, and a tile, and between the player and the enemy.
    - The *tileCheck* class checks whether the unit is about to move into a tile with a collidable object. It has been tested in *TestCheckCollision_tileCheck*.
    - The *collectableCheck* class checks collisions between *Player* and every *Collectable* and sends a message to *CollectablePlacer* when collision is found. It has been tested in *TestCheckCollision_collectableCheck()*
    - The *enemyCheck* class checks collisions between the player and a moving enemy. Causes the game to stop. It has been tested. *TestCheckCollision_enemyCheck***()**
    - The *isCollidableTile* function checks whether the tiles at the given location in the matrix are hard obstacles. No automation test as it is a private function but it has been tested through the *TestCheckCollision_tileCheck*.



Current scope: all classes

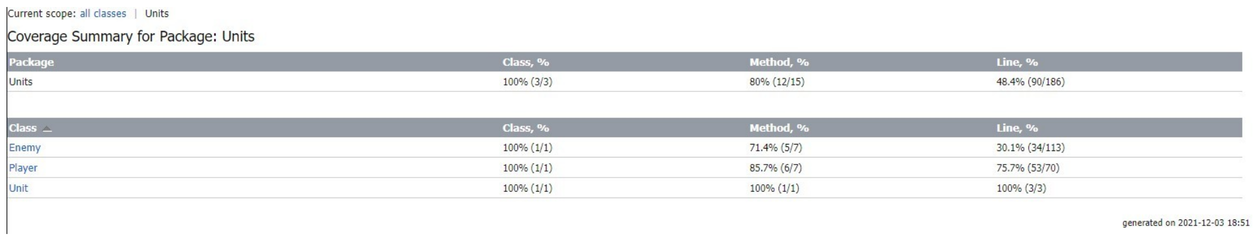
Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	90.5% (19/21)	72.9% (51/70)	57.9% (372/642)

Coverage Breakdown

Package ▲	Class, %	Method, %	Line, %
Background	100% (2/2)	83.3% (5/6)	79.4% (54/68)
Collectables	100% (5/5)	81.8% (9/11)	80.3% (53/66)
Helpers	0% (0/1)	0% (0/1)	0% (0/1)
Units	100% (3/3)	80% (12/15)	48.4% (90/186)
main	90% (9/10)	67.6% (25/37)	54.5% (175/321)

generated on 2021-12-03 18:51



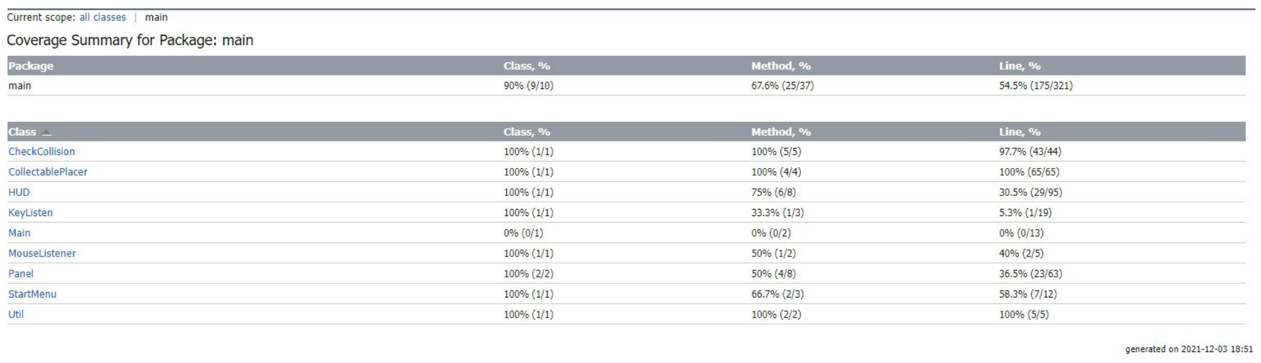
Current scope: all classes | Units

Coverage Summary for Package: Units

Package	Class, %	Method, %	Line, %
Units	100% (3/3)	80% (12/15)	48.4% (90/186)

Class ▲	Class, %	Method, %	Line, %
Enemy	100% (1/1)	71.4% (5/7)	30.1% (34/113)
Player	100% (1/1)	85.7% (6/7)	75.7% (53/70)
Unit	100% (1/1)	100% (1/1)	100% (3/3)

generated on 2021-12-03 18:51



Current scope: all classes | main

Coverage Summary for Package: main

Package	Class, %	Method, %	Line, %
main	90% (9/10)	67.6% (25/37)	54.5% (175/321)

Class ▲	Class, %	Method, %	Line, %
CheckCollision	100% (1/1)	100% (5/5)	97.7% (43/44)
CollectablePlacer	100% (1/1)	100% (4/4)	100% (65/65)
HUD	100% (1/1)	75% (6/8)	30.5% (29/95)
KeyListen	100% (1/1)	33.3% (1/3)	5.3% (1/19)
Main	0% (0/1)	0% (0/2)	0% (0/13)
MouseListener	100% (1/1)	50% (1/2)	40% (2/5)
Panel	100% (2/2)	50% (4/8)	36.5% (23/63)
StartMenu	100% (1/1)	66.7% (2/3)	58.3% (7/12)
Util	100% (1/1)	100% (2/2)	100% (5/5)

generated on 2021-12-03 18:51

Test Report

The above Figures summarizes the test coverage for our project so far. In total we have 57.9% line coverage which covers about 72.9% of our methods and 90.5% of the classes. Upon further inspection, it can be noticed that we have about 80% coverage for the *Background* and *Collectables packages.* Most of the uncovered lines were part of integration between the entities and the UI whose logic was mostly covered by *render()* methods within the classes or other method calls dealing with the UI. Other than that, most of the methods that are dealing with the logical aspects of the game are covered by automated tests – observe that the method coverage is nearly 80% for most of our packages. The *Panel* class managed the game loop and overall flow of the game. However, the logic for interacting with game was separated into the packages that were tested. Hence the *Panel* was not tested.

Findings

During this phase we began where we left off completed the remainder of our project. Hence several changes were made to the production code to integrate the features that could not be added in Phase 2. The testing phase exposed several quality issues within our code such use of same hard-coded values, repeated logical lines, tightly coupled programming and bugs. Refactoring was required to not only heighten the code quality but also to mitigate writing automated tests and make our program more scalable. A new class called *DefaultProperties* was created to record a list of constant values that are used throughout the program; large methods were refactored to remove repeated logic and redundant lines such as in the *CheckCollision.tileCheck()* method; and some *switch* statement logic was simplified through use of *polymorphism*. The tests also revealed some minor bugs which were immediately fixed. The testing phase has improved our program through enhancing quality, readability, scalability, and reliability.