Phase 3 Report

Findings

After writing tests for the program, some changes that needed to be made were adding getters and setters, to help verify results, in Animate, Player, and UiButtons classes. To the Animate class getters for isMoving was added to see if the animate is moving in a specific direction. For the Player class a setter was added to change the “canMove” variable. In the UiButtons class getters for getButtonGameState and getButtonSpriteRowNumber were added to verify the correct game state was being set by the button as well as if the correct button was being set to use for a certain menu. Bugs that were found and fixed was the set() method in the Pathfinding class that did not have checks for out of bounds values, so we would get out of bounds errors if invalid values were passed into the method. This was fixed by adding checks to verify the given values are valid. Another bug that was in the Pathfinding class that was corrected was the setPlayer() method had the same bug as the set() method. It was also fixed by adding a check for invalid values.

Tests

For the Entities there are animate (player and enemy) as well as inanimate (key, egg, potion, and trap) objects that will spawn in game and interact with each other. Therefore unit tests need to be written to check if they spawn in the correct tile location and if they spawn in the center of a tile. For the animate class both player and enemy can also be set to move in a certain direction, the setting of the correct direction in addition to movement in the correct direction can also be unit tested for. The player can interact with all other entities in the game. When the player collects a key, their key count should increase, and after all the keys have been collected the current level should end. When the player collects an egg their egg count should increase. There are also two types of potions a player can collect. A player can have at most 100 health points. Thus, when the player collects a health potion it should give them an additional 25 health points if their health is under 100 points, bring them back to full health 100 points if their health is under 100 but with the health bonus would go over 100 points. If the player is at full health, collecting a potion should have no effect. The player can also collect a speed potion, which permanently boosts their speed for the round, they can obtain a maximum of two speed boosts, additional speed potion collected should do nothing. When a player runs into a trap their health should take a 50 point hit, and the game should end when the player hits 0 health points. All inanimate objects should disappear after they have been interacted with by the player. If an enemy tries to interact with any of the inanimate objects, nothing should occur. Also when an enemy interacts with a player the player's health should be decimated and the current round should end. Some integration tests that can be considered between player, enemy, and pathfinding classes is to check if Player and Enemy classes call methods in the Pathfinding class to verify if areas are valid to be moved in. Furthermore, a test can be made to check if enemies track players using the Pathfinding class. Between player, enemy, and level classes, integration tests can be made to check if interactions can occur between the Player and other entities (Taps, Enemy, Potion, Egg, Keys) and if the player stats increase or decrease accordingly.

The gameStates package identifies what part of the game the player is in, for example if the player is in the main menu loading screen or actively playing a level, and renders the correct graphics for that part of the game. The MainMenuState loads the correct background graphics for the game lobby. LevelSelectionState loads a screen that lets the player choose what level they want to start playing, from the three levels available and should loop back to level one when you reach level three in the available selections. PlayerSelectionState lets you choose the color of the dinosaur you want to play, and should loop around to the first color selection when you hit the end of the different available colors. InLevelState is how the game moves from one stage to another and determines what menu or level needs to be displayed. An integration test that can be run is to test if the correct game state is being entered when moving from one part of the game to another.

The Graphics package renders the screen the game is displayed on. GraphicsPanel class renders the area that displays the maps, menus, and various other game objects. The GraphicWindow class renders the frame that the GraphicPanel will be displayed in. GraphicsGrid class maps out a tileset grid from the dimensions of the graphic panel, to make it easier to set where to render entities on the map.

The helperClasses package helps load the various correct graphic files for the backgrounds, maps, and entities. The assetLoader class loads the correct entities .png files from resources into the game to be used and returns a null pointer if the asset being requested doesn't exist or is not defined, this can be unit tested to check for correct behaviour. BackgroundMovingSpeed controls the speed of the parallax background images from the main game lobby as well as the background image speed in the gameplay top menu bar.

The levels package contains various classes that control what level is being loaded and what entities and where they should spawn in each level depending on the gameState set and the level selected by the player from the Game class. Unit tests can be written to check if the correct level state is entered when the player has 0 health versus 100 health, in addition to a boundary check, to test what happens when you try to set up a level that doesn't exist. The Level class works in conjunction with assetLoader and Entity classes to load the correct levels and spawn the correct entities on the map. The Levels class also determines if a player interacts with other entities and keeps track of the various player stats like health or eggs and keys collected. The LevelManager class determines which level is the correct level to load depending on the player's choices at the beginning of the game from character or level selection. The Pathfinding class helps determine where an animated entity (player or enemy) can and cannot move, for example, the borders of the game screen and walls that the player and enemy can not walk through. The Pathfinding class also works in conjunction with Enemies class to determine the shortest possible path to take from its current position to the player's position if they become alerted to the player's existence. Classes Level one through three sets where on the map the various entities should spawn for the corresponding level.

The userInterface package works in conjunction with InLevelState class to determine what menu should be loaded and the correct format for each type of in-game menu, in addition to setting the correct parallax background for the different stages of the game. Unit tests can be written to determine if the correct number of buttons and button type are loaded and what happens if you try to generate a menu with an invalid level. UiButtons class loads the correct button sprite .png assets into the game. The UiDeathScreen class sets the correct menu to be displayed upon a player's death. The UiFinishedGameMenu class loads and sets the correct format stats to be displayed when a player successfully passes a level. UiMenu initializes the correct parallax background as well as formats the displays for the main game lobby menu. The UiPauseMenu class generates a menu when you pause the game while playing a level. The UiTopMenuBar class loads and scales the correct parallax background asset for a level, in addition to displaying the current statistics of the player, such as player health, keys collected, eggs collected, and time played.

Test Quality and Coverage

Measures Taken to Insure Quality

- Generated code line and branch coverage report using Jacoco to determine where more tests needed to be written and where.

Line and Branch Coverage

- For the abstract Entity class there is 57% line coverage and 70% branch coverage.
- For Animated Entities there is 84% line coverage and 56% branch coverage.
    - Where Enemy class has 80% line and 39% branch coverage,
    - and Player class has 90% line and 59% branch coverage.
- For Inanimate Entities there is 89% line coverage and 62% branch coverage.
    - Where Key has 100% line coverage,
    - Egg has 100% line and 100% branch coverage,
    - Potion has 100% line and 100% branch coverage,
    - and Trap has 91% line and 50% branch coverage.
- For gameStates there is 39% line coverage and 7% branch coverage.
    - Where gameStates has 100% line coverage,
    - States has 12% line coverage,
    - InLevelStates has 3% line coverage,
    - LevelSelectionState has 74% line and 60% branch coverage,
    - and PlayerSelectionState has 78% line and 50% branch coverage.
- For Graphics there is 10% line coverage.
    - Where GraphicsGrid has 23% line coverage.
- For helperClasses there is 84% line coverage and 50% branch coverage.
    - Where Direction has 100% line coverage,
    - AsserLoader has 48% line coverage,

    - and BackgroundMovingSpeed has 94% line and 50% branch coverage.
- For levels there is 66% line coverage and 71% branch coverage.
    - Where LevelState has 100% line coverage,
    - Level has 68% line and 50% branch coverage,
    - And Pathfinding has 94% line and 86% branch coverage.
        - Pathnode in pathfinding has 92% line coverage.
- For userInterface there is 38% line coverage and 24% branch coverage.
    - Where UiPauseMenu has 100% line,
    - UiButtons has 65% line and 33% branch coverage,
    - UiDeathScreen has 55% line and 100% branch coverage,
    - UiFinishedGameMenu has 58% line and 100% branch coverage,
    - UiMenu has 41% line and 9% branch coverage,
    - and UiParallelBackground has 30% line and 25% branch coverage.

Code Segments Not Covered

- Keyboard inputs
    - We didn't know what we could do to test these classes. We thought about creating fake keyboard inputs but didnt know how something like that could be implemented.
- Mouse inputs
    - Similar to the keyboard inputs, we didn't know how we could go about creating tests for these inputs.
- GraphicsPanel
- GraphicsWindow
    - GraphicsPanel and GraphicsWindow were really small and simple classes that called a few methods from Jframe and JPanel that we did not know how to test.
- All render methods
    - Trying to debug render methods and classes resulted in odd behaviour. We would have the game window popup for a brief second and our tests would then proceed to fail. We couldn't figure out the cause, so we decided to not test these methods.

In conclusion, while unit tests can be useful for a lot of the lower level libraries and classes that were used in the game code, it was not of much use for the higher level interconnected classes. Since the game is a simulation and relies on large amounts of shared and changing states that can not be meaningfully replicated and tested in an isolated way. A fair amount of the functions in the game do not return any sort of value that can be checked instantaneously, instead, the methods set in motion a process that should terminate at some future point in time.