Unit Features

Feature 1: Player character's movements

- Description: The player is able to try to move to one direction based on the input of the keyboard.
- Test: To test this feature, we construct a "PlayerCharacterTest" and use "setKeyState" to change the key state and simulate the input of the keyboard. We use "latch" to wait for one tick and use "getPosition" to check the position of the player after one tick.

Feature 2: Bonus objectives' disappearance

- Description: Bonus objectives should disappear after a few ticks.
- Test: We use "latch" to wait for a few ticks that the bonus objective should disappear. Then, we check the bonus objective in level to see whether it's been removed.

Feature 3: Moving enemies' movements

- Description: Moving enemies can move in some directions.
- Test: To test this feature, we construct a "EnemyTest" class and use "moveLeft", "moveUp", "moveRight", "moveDown" functions. We get the position of the moving enemies and check whether they are at the expected positions.

Feature 4: Resources test

- Description: Images in resources should work and not be damaged.
- Test: To test this feature, we construct a "ResourceTest" class. We write a function called "loadImage" and use "ImageIO" to load images from resources.We use this function to read the image that we want to test and check the result.

Feature 5: Screen test

- Description: The user is able to click all of the buttons in the game without error
- Test: To test this feature, we simulate a click for each of the buttons and ensure that no exception is thrown using "assertDoesNotThrow". The frame class is tested with "assertNotNull".

Feature 6: Level Plan Test

- Description: Level plan is created correctly
- Test: To test this feature, we construct a "LevelPlanTest" and create 2D arrays that are similar to our map and use "assertArrayEquals" to check.

Interactions

Interaction 1: Player and stationary entities: walls, stationary enemies, objectives and bonus objectives

- Description: The player should not go across walls and not be on walls. Player will lose score when moving over stationary enemies and gain score when moving over objectives and bonus objectives.
- Test: To test these interactions, we construct a class called “InteractionsTest”. In this class, we write “init()” to initialize levels and start the level before each test and we write “cancel” to cancel all timers in levels after each test. We use “setKeyState” to simulate the keyboard’s input and make the player try to go across walls. We use “latch” to wait for one tick and then get the position of the player to check whether the player is on the wall or is across the wall. For the player, stationary enemies, objectives and bonus objectives. We make the player be at positions of stationary enemies, objectives and bonus objectives and check the score to see whether the player loses or gains the expected score.

Interaction 2: Player and moving enemies.

- Description: Upon contacting the enemy, the player will lose when encountering moving enemies.
- Test: To test the interactions between the player and moving enemies, we position the player and moving enemy next to each other and start the game loop for one tick after which the “lose screen” should be visible.

Interaction 3: Moving enemies and wall

- Description: Moving enemies should not move through walls
- Test: To test the interaction between moving enemies and walls we placed the moving enemy on the other side of the wall from the player and let the game run for a tick. The moving enemy should be in the same position as before the tick.

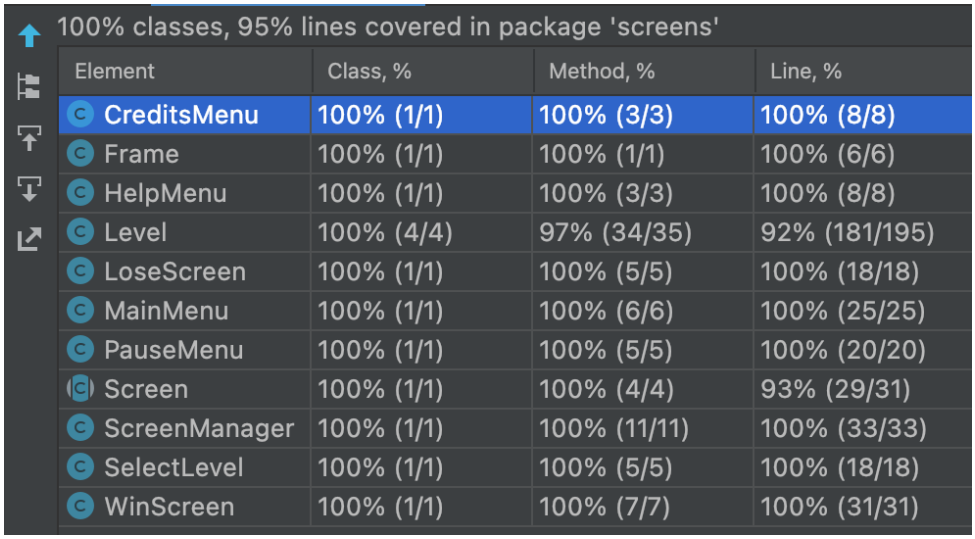
Measures taken to ensure quality of tests

- Our tests are independent of one another. The tests do not depend on one another to run. The required variables and methods are set up in a @BeforeEach method.
- Documentation: Included documentation for tests. This made our tests more readable and easier to understand.

Coverage

The IntelliJ coverage report revealed that for each of our packages we had 100% class coverage. For the “screens” package, a majority of the screen classes had 100% line coverage. The exceptions would be the “Level” class and the “Screen” class. For the “Screen” class we did not cover the catching of an IOException. For the “Level” class we were unable to test a few lines discussed below. We had 100% line coverage in the “misc” package. Lastly, we had nearly 100% line coverage in our “entities” package, the only exception is the same IOException that we were unable to cover.

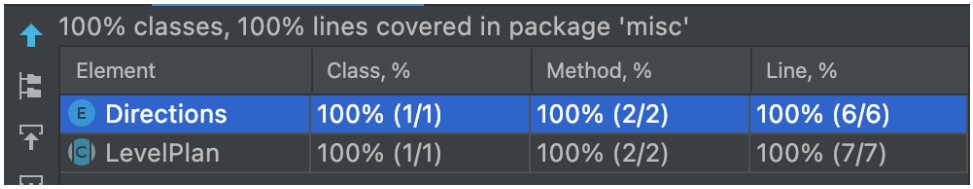
Screens Package:



100% classes, 95% lines covered in package 'screens'

Element	Class, %	Method, %	Line, %
CreditsMenu	100% (1/1)	100% (3/3)	100% (8/8)
Frame	100% (1/1)	100% (1/1)	100% (6/6)
HelpMenu	100% (1/1)	100% (3/3)	100% (8/8)
Level	100% (4/4)	97% (34/35)	92% (181/195)
LoseScreen	100% (1/1)	100% (5/5)	100% (18/18)
MainMenu	100% (1/1)	100% (6/6)	100% (25/25)
PauseMenu	100% (1/1)	100% (5/5)	100% (20/20)
Screen	100% (1/1)	100% (4/4)	93% (29/31)
ScreenManager	100% (1/1)	100% (11/11)	100% (33/33)
SelectLevel	100% (1/1)	100% (5/5)	100% (18/18)
WinScreen	100% (1/1)	100% (7/7)	100% (31/31)

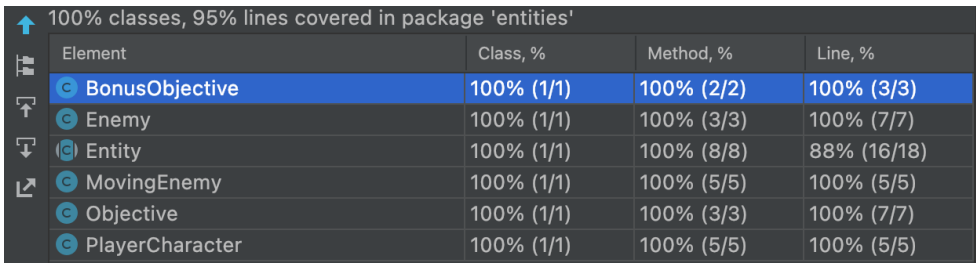
Misc. Package:



100% classes, 100% lines covered in package 'misc'

Element	Class, %	Method, %	Line, %
Directions	100% (1/1)	100% (2/2)	100% (6/6)
LevelPlan	100% (1/1)	100% (2/2)	100% (7/7)

Entities Package:



100% classes, 95% lines covered in package 'entities'

Element	Class, %	Method, %	Line, %
BonusObjective	100% (1/1)	100% (2/2)	100% (3/3)
Enemy	100% (1/1)	100% (3/3)	100% (7/7)
Entity	100% (1/1)	100% (8/8)	88% (16/18)
MovingEnemy	100% (1/1)	100% (5/5)	100% (5/5)
Objective	100% (1/1)	100% (3/3)	100% (7/7)
PlayerCharacter	100% (1/1)	100% (5/5)	100% (5/5)

Code not Covered

```
try {
    EntityImg = ImageIO.read(new File( pathname: path + ImgName));
} catch (IOException e) {
    e.printStackTrace();
}
```

1. The catch IOException: All images in resources work well, so there is no error thrown out.

```
@Override
public void keyPressed(KeyEvent e) {
    if (e.getKeyCode() == 37) {
        keyState = Directions.LEFT;
    }
    if (e.getKeyCode() == 38) {
        keyState = Directions.UP;
    }
    if (e.getKeyCode() == 39) {
        keyState = Directions.RIGHT;
    }
    if (e.getKeyCode() == 40) {
        keyState = Directions.DOWN;
    }
}
```

2. In the Level class, the overridden keyPressed method is not covered. This method reads in the user's key inputs and sets the KeyState variable according to the key pressed by the user. When testing the rest of the code, we used a "setKeyState" method to simulate the keyboard's input. We want to do automatic tests but not manually do keyboard's input by pressing some keys.

```
//right
if (enemyPos[0]+1 < map[0].length && map[enemyPos[1]][enemyPos[0]+1] != 1) {
    testDistance = Math.abs(enemyPos[0]+1 - playerPos[0]) + Math.abs(enemyPos[1] - playerPos[1]);
    if (testDistance < bestDistance) {
        bestMove = Directions.RIGHT;
        bestDistance = testDistance;
    }
}
```

3. In the moveEnemy method from the Level class, we were unable to test all of the branches. There is a branch for each possible move an enemy can make. Due to the positioning of the enemy of the map relative to the player's character, the moving enemy does not move right. However, since the behavior for each of the decision blocks is reused for each possible move, we decided that it was not essential to cover all movements in this function.

Things We Learned

- **Increased code testability:**
    - Adding getters and setters. In order to test one unit function at a time, some variables would have to be instantiated. For example, in order to simulate a key press without manually pressing a key, the resulting KeyState variable is set through a simulated “setKeyState” method.
- **Increased code readability**:
    - Replaced primitive integer values for directions with enum variables UP, DOWN, LEFT, RIGHT. Rather than relying on comments to understand the meaning of the primitive values, we could now simply read the variable name.
    - Addressed Long Method: extracted tasks into helper methods. In our code we had a start() method that controlled the game loop for each “tick” of the game. This method controlled several sub-tasks. The tasks included increasing the timer, moving the player/enemy, testing for collisions and testing for wins or loses. Rather than relying on comments to understand the purpose of the tasks, the tasks were extracted into helper methods that were named appropriately (increaseTimer, movePlayer, moveEnemy,etc…).
    - Replaced repeated code with calls to a new method. In our code we had ArrayLists of Entity objects (objectives, enemies, etc…). When we would apply any operation to these ArrayLists we would copy and paste the code, and we would only modify the specific ArrayLists object being used. For example, we would paint the Entity objects by applying the same operation to each of the ArrayLists. We reduced redundancy and increased readability by extracting the task into a method where there would be a parameter for the specific ArrayList being used.
- **Fixed Bugs**
    - We realized that the moving enemy was able to move into the border wall. We fixed this bug by changing the production code. The bug is due to forgetting to add “break;” in the switch case.