SIMON FRASER UNIVERSITY

ENGAGING THE WORLD

CMPT 276: Phase 3

Nov.22, 2022

Interactions between components

We identified 4 important interactions between different components in our system. The first component with important interactions is the Board. It interacts with all of our logic for our game and is integral to make the game run. It interacts with most of our code like all the animate and inanimate objects, the state the game is in, and placing the objects on the board.

The second component is the factories that make our objects and the board. We have a factory for animate objects, inanimate objects, and the board. Each factory interacts with the board that was made from the board factory to create each and every object that is placed on the board. Things like the player, enemy, the board, rewards, blockers and more.

The third component is the map loader. The map loader makes the map from the specified text file and loads the data into the board. The map loader interacts with the board and text data file.

The last component is the animate objects like the player and enemy. It interacts with all parts of the board. Every time an animate object needs to move like the player and the enemy. It needs to check the board if it can move in the direction of its choice. In addition, whenever the player picks up an objective or collides with an enemy, the player score needs to update and the game state needs to change accordingly.

Test Case/Class Coverage

AbstractAnimateTest:

- Tests the position and facing direction of animate entities when the move method is called on them

PlayerTest:

- Tests the effect of regular/bonus rewards and punishments on the player's score.

HungryCowAnimateFactoryTest:

- Tests the creation of animate entities (player and enemies) in the game
- Tests their default score (player only), position and facing direction.

HungryCowInanimateFactoryTest:

- Tests the creation of inanimate entities (regular/bonus reward and punishments) in the game
- Tests if the entities are placed at intended position and if their respective values are as expected

HungryCowBoardFactoryTest:

- Tests the creation of a board instance from a 2 x 2 matrix containing board data.
- Tests if the dimensions of the board are as intended and if the player, start space and end space is placed at the position specified by the board data matrix.
- Tests if the lists/sets maintaining enemies, barriers, rewards (regular and bonus) and punishments are empty initially.

MapLoaderTest:

- Tests if the 2 x 2 matrix created by the loadBoard method of MapLoader is as expected.
- Tests if all animate/inanimate entities are placed exactly at their intended positions.

BoardTest:

- Tests all the logic pieces required to run the game.
- Tests if an animate entity can move to a particular position on every tick of the game.
- Tests if the player changes its position to the expected position after the move method is called on it.
- Tests if the enemies change their position to the expected position (based on the Manhattan distance between the enemy and the player) after the player moves.
- Tests if the manhattan distance generated is as expected for all the directions relative to the enemy.
- Tests if the game is won after the winning criteria is fulfilled and the player reaches the end space.
- Tests if the player's score goes negative as a result of stepping on punishments.
- Tests if the player encounters an enemy after moving on every tick of the game.
- Tests if the regular/bonus rewards disappear from the board after a player collects them
- Tests if the player score is reduced by the punishment value when the player encounters a punishment.
- Tests if the bonus rewards appear randomly on the board after a certain period of time.

Quality Assurance

To ensure our testing quality, we appropriately named each of our tests to clearly say what they are testing and what function they should be testing. We also used the method assertThat over assert to better convey what we are testing to ourselves in order to avoid confusion.

Line and Branch Coverage:

For the line coverage, we covered 100% of the lines in each component of the tests of our unit test. We also had 100% branch coverage for each component of the tests in our unit tests, and the integration tests.

Note:

The only thing that was not really covered was the UI state component, since in order to test that we would have to run the game while drawing the images, which is not really possible to test since repaint and drawing images cannot be tested.

Findings

While testing the Board class, we found that we wanted to test each small method within the Board that has private visibility. To test these methods, we added Google Guava library to use their VisibilityForTesting annotation to make these methods only visible for testing.

While testing the random position generation of the bonus rewards, we found that it was not testable with our current implementation.

Our current implementation uses the RandomUtils static class from Apache Commons library which uses a static random.

Although this did not affect the way our game is implemented, we changed it so that a new Random object is instantiated every time a Board is built.
We also created a method only visible for testing, to set the random of the Board class using any mock object.

With this change, we utilized Mockito to mock the Random object so that the values returned from the Random object are deterministic in the test. This way, we can assume that the implementation of Random works, while testing only the method within our Board class.