Parth Kapur, Thomas Gorney, Saren Yavuz
Prof. Zheng
Software Engineering
4/25/2019

Assignment 4 - JUnit Tests

-public void updateGame()

- **gameSpeedShouldBe1PT035()** - Tested the game speed after a single updateGame(). Resetting the game sets the gameSpeed to 1.0, and after an update should increase it to .035 + initial gameSpeed. We check this using the assertEquals method, and checking gameSpeed against 1.035f after a single updateGame.(**PASSED**)
- **dropCooldownShouldBe25()** - Tested that the drop cooldown for the blocks is set to 25 after a single updateGame. We do this with the assertEquals method, checking the dropCooldown after a single updateGame against an integer of 25. (**PASSED**)
- **newScoreShouldBeZero() -** Testing to see if the score is resetting when playing a new game. The score should be set to zero when a new game is started. We use the assertEquals method to compare score against zero. (**PASSED**)
- **FirstPieceLandsShouldBeZero() -** Testing to see if the score remains zero when the first block lands at the bottom. It should equal zero if a player can only score with lines. First we generate a block and set its current row to 20 so that it has landed at the bottom, we then updateGame and see if the score remains zero by using assertEqual method to check score against zero. (**FAILED**)
    - This test does not pass and we can see that the score is being updated as soon as a block lands at the bottom of the board and not just scoring a line. This tells us that there is an issue with the scoring functionality of the game.
- **updateScoreShouldBe800() -** Testing to see if a user scores 800 points when scoring 4 lines at the same time. This test should pass if the player scores 4 lines at the same time. We check this by adding pieces, sending them to row 20 which is at the bottom and making a total of 4 lines for scoring. We then updateGame and see if we get a tscore of 800. We do this using assertEquals to compare the score against 800. (**FAILED**)
    - This test does not pass and we can see that the score does not get the right value after scoring 4 lines. This tells us that there is an issue with the scoring functionality of the program.

- **leveShouldBe() -** Tested to see if the level gets the correct value after an updateGame has been called. We spawn in a block and set its currentRow to 20 so that it is already at the bottom. Then we create a value called expectedLevel and set set it to the gameSpeed * level. We know that the expected level should be based on the gameSpeed after each update, which adds 0.035 each time updateGame is called. We check this using the assertEquals method to check the level against expectedLevel. (**PASSED**)
- **logicTimerCyclesShouldBe() -** Test if the timer cycles is as expected after a single updateGame is called. We test this by spawning in a block and setting its currentRow to 20 so that it is at the bottom. We first updateGame and use assertEquals to check if the millisPerCylce is equal to 1 / 1.035 * 1000 to get the correct value after the gameSpeed is increased by 0.035. (**PASSED**)
- **currentRowShouldBe() -** Test to see if the currentRow gets updated after a single updateGame gets called. We test this by first resetting the game and setting the block type to I. We then update the game. We use the assertEquals method to test the currentRow of the block against 1. If the block starts at currentRow 0, then when we updateGame, the row should change to 1. (**PASSED**)

-public boolean isValidAndEmpty(TileType type, int x, int y, int rotation)

- **shouldBeValidAndEmpty() -** Testing to see if a a location is currently occupied by another piece. We do this by using addPiece, setting type to an I block, and setting its x to 1, its y to 2, and its rotation to 1. We then use the assertEquals method to check if when we add another piece, in this case tile type I at location 4,5 and rotation 1; then there should be no collision between the two pieces added because they are at different locations. We compare these locations and if it isValidAndEmpty, then it is true and the test PASSES, otherwise it fails. Meaning that the location isValidaAndEmpty. (**PASSED**)
- **shouldNotBeValidAndEmpty() -** Testing to see if two positions are colliding with each other. We do this similar to the last test, but instead we use the same values for tileType, and we use the same x,y,position (TypeI,2,3,1). In this case we check the values and if isValidAndEmpty returns FALSE, then the test PASSES, meaning that the position is already taken. (**PASSED**)

-public void addPiece(TileType type, int x, int y, int rotation)

- **shouldAddPiece() -** We are testing whether a piece should be added or not. First we rest the game, then we add a piece at position x,y,rotation (1,1,0). We then use

asserEquals to compare the TileType.values()[0] against getTime(1,2). If these are equal then that means that the piece we just added already exists and we do not need to add another piece yet. (**PASSED**)

- **pieceShouldBeOutOfBounds() -** Test to see if a piece that we add is out of bounds or within play. We do this by adding a piece at the location of x,y,position (-1,-2,0). This position is out of bounds of the game board. We then use assertEquals to compare getTime(-1,-1) with the piece we just added to see if it exists. In this case there is an exception. (**FAILED**)
    - This test does not pass. This means there there is an issue with the positioning of blocks using negative values and that the piece does not hold position at negative values.

-public void checkLines()

- l**ineShouldBeZero() -** This will test how many block lines there are after a reset and when a single block drops to the bottom. There should be no lines as we have only dropped one block. We do this with the assertEquals method and compare checkLines to zero. (**FAILED**)
    - This test does not pass. This means that there is an issue with the checkLines() function because check lines is getting a different integer than zero when a block lands at the bottom.