Kathryn Hodge

Professor McCarthy

CMPU 203: Software Development

October 15, 2014

Spike: Don't Pop the Bubbles!

To implement a game (and create a testing suite to ensure the game is implemented correctly) , a programmer first must look at the Game Manual to see the specifications that the code must meet. The **Spike I Manual** specifies certain conditions for the Field Of Play, Controls, Scoring System, and How To Win/Goal of The Game. If the coded implementation meets all of the specifications of the **Spike I Manual**, then the program is an accurate representation of the game. By testing the properties of a game, one before and one after user input (represented as two SpikeGame's), it can be proven that the program is implemented correctly.

For the purpose of simplifying this paper, note the following:

- All tests involving actual gameplay are in the *TestException.java* file
- All tests involving starting and restarting the game are in the S*pikeGameTest .java* file
- For all of these tests, if a condition corresponding to a specification in the Game Manual is not true, then an Exception is thrown with an appropriate input statement and the game crashes. However, if it is true, then the game continues as usual
- In previous versions, this game used to incorporate balloons, but in this latest version, the game uses bubbles instead of balloons, so some of the classes and functions are named after Balloons.
- Details about the individual spike are in the *Spike.java* file; details about individual bubbles are in the *Balloon.java* file; details about loosing lives are in the *LivesLabel.java* file; details about scores are in the *ScoreLabel.java* file
- The logic of the game is in the *SpikeGame.java* file

- If the programmer wants to run tests, the *SpikeGameTest.java* file should be run
- If the programmer wants to play the game, the *SpikeGameRun.java* file should be run

In Field Of Play, the Game Manual describes the graphics on-screen and the initial starting conditions of the game. The graphics cannot be tested because the only way the programmer can open a window is by physically playing the game. However, it is okay that this is true because the graphics only handle how the game looks - not how the game is played. To test the initial starting conditions of the game, the **testConstructor()** [13] function in *TestException.java* creates a new SpikeGame [14] and checks a bunch of conditions on it. For example, it checks if the spike starts in the top middle of the screen [16], if the balloonDataStruct is empty [19], if the game starts with two lives [22], and if the score starts at zero [25]. These are all of the starting conditions of the game, so if these conditions are true, then the game's constructor works according to the specifications of the Game Manual.

In Controls, **Spike I Manual** describes how the player controls the spike and when the bubbles move upward. According to the Manual, the player controls the spike with the left and right arrow keys that move the spike in the corresponding direction. The function *testReactTickSpikeBalloon* [125] takes in two SpikeGames, an oldSpikeGame and a NewSpikeGame, and a CharKey rnb, which keeps track of the button pressed. The function compares the differences in width for the spikes and heights for the bubbles and makes sure that these differences correspond to the specifications in the Game Manual. First, the method sets a local variable (dw) that represents the delta width of the spike to zero [126]. Then, the function asks whether rnb is the left or right arrow and changes dw to -1 or 1 respectively to shift the spike one space in the corresponding direction [126-131]. However, if the spike is at the very left or very right of the screen, then the spike should not move, even if the respective left or right button is pressed, because the spike should never move off screen [134-136]. If the spike didn't move after pressing the right or left arrow, then the delta width is set back to zero because the spike isn't suppose to move due to the fact it is at the edge of the screen

[138-142]. Lastly, the game checks if the newSpike moved to a width corresponding to the oldSpike's width + the delta width.

The second part of *testReactTickSpikeBalloon()* makes sure that the bubbles only move when the spike moves. If the delta width isn't zero, then the spike moved and all the balloons should move up the screen [144]. Each bubble is given an identity so it can be tracked through the ArrayList that stores all of the balloons shown on screen [148]. If two bubbles have the same identity, then they represent the same bubble on screen, but in different states (one before the input (right/left/other button) was pressed and the other after). The method goes through all the balloons in the old balloonDataStruct [145] and the new balloonDataStruct [147] via for-in loops and checks that all of the bubbles in the newDataStruct moved up the screen by one [150]. However, after a certain number of turns, a bubble will reach the top of the screen and be removed from the data struct. The found boolean [146, 154, 160, 163] keeps track of this removed balloon. In the two ifs at the end, the (b.height - 1)'s keep track of the balloons just entering the screen and just leaving the screen and makes sure they match up with the found boolean [139-144]. Meaning, if the removed bubble is not found, then the old bubble's height should be 1 (because it's about to be removed), and if the removed bubble is found, then the old bubble's height shouldn't be 1 because it doesn't get removed. If the delta width is zero, then the spike didn't move and so the balloons shouldn't move [167]. The function goes through all of the old bubbles [168] and all of the new bubbles [169] and using the bubbles' identity, checks that the heights of the new and old bubble are the same. If not, as with all the other tests, it throws an exception and crashes the game. For a more efficient program, after the old bubble finds its corresponding match in the new bubble database, the inner for loop breaks because there is nothing else to check for that bubble — it has already found it's singular match [153-155 && 174-176]!

The Controls section also states that a new bubble should be added to the newSpikeGame. balloonDataStruct when the spike moves and that a new bubble should not be added to the newSpikeGame.balloonDataStruct when the spike does not move. In *testAdded*, the program tests whether a new bubble was added at the appropriate time [95]. The boolean newBalloon is false when a newBalloon isn't found and true when a newBalloon is found. A new bubble is found only when it does not have a

matching identity with a balloon in b [102]. If the bubble in the newBalloonDataStruct has a match with a bubble in the oldBalloonData, then it isn't the new balloon that was added, so the program changes the oldBalloon to true because an oldBalloon was found. If a bubble isn't an oldBalloon, then it's a newBalloon [108]! A new balloon should only be created if the spike moved, so two IF statements check that if a new balloon is found, then the old spike should not be in the same place as the new spike (i.e., the spike has moved and a balloon is added) [113-120].

For the Scoring System, every time a bubble passes the top of the screen without hitting the spike, the player's score should increase by 5 points and the player's lives should stay the same. In the opposite case, every time the player hits a bubble with his/her spike, the score should stay the same and the player should lose a life. However, in all cases, the score should be nonnegative. For testing the non-negativity of the score, the function *checkIfPositiveScore* [32] takes in an old SpikeGame and a new SpikeGame and checks if each respective score is nonnegative [33]. If either score is negative, then the game crashes [34].

For every turn, one of three things happens: the spike hits a bubble, the spike avoids a bubble, or the spike neither hits or avoids the bubble because the bubbles aren't high enough on the screen yet. In the program, the function *testCollisionLivesScore* [61] makes sure that at least one of these cases is true. First, if the spike doesn't move, then the balloons don't move and the lives and score should stay the same [63-66]. If a collision is about to occur, then we use a boolean collision to know that the score or the lives should change [62, 73]. In a collision, a spike has the option of hitting a bubble or avoiding a bubble. For the first case, when the spike hits the bubble, then the game should have one less life and it's score should remain the same [74-76]. For the second case, when the spike misses the bubble, the game should have the original score + 5 and its lives should remain the same [76-77]. However, if a collision is not about to occur because all the bubbles are too far away to hit or miss the spike, then the boolean collision would be false and lives and score should stay the same [84-88]. If they are not, then throw an exception.

In How To Win, the player cannot win the game, but simply tries to get the highest score he or she can. The player loses the game when he or she pops exactly two

bubbles, not necessarily in consecutive order. In *checkGameOverLives*[39], the programs checks that when the game is over, the player should have zero lives [40-43]. If the old game is not over, then the lives of the old game should not be zero[45-48].If the new game is over, then the old game should've had one life and the new game should have zero lives [49-52]. Otherwise, if the new game is not over, then neither game should have zero lives [53-55]. If any of these conditions are not true, then an exception is thrown and the game crashes.

In addition to have tests within the game play of the game, I also have tests for how a user should start and restart the game in *SpikeGameTest.java*. To start the game, the user must press the Up button. The program checks this by calling *shouldStart,* which is a function within the logic of the game in *SpikeGame.java*, to see if the game should start and asking if the Up arrow was pressed [26-33]. The program checks that they should both be true or both be false whenever the user is at the start screen. Similar logic is used for restarting the game. To restart the game, the user must press the Down button. The program checks this by calling *shouldRestart,* which is a function within the logic of the game in *SpikeGame.java*, to see if the game should restart and asking if the Down arrow was pressed [53-60]. The program checks that they should both be true or both be false whenever the user is at the restart screen.

Every test in this testing suite uses randomly generated key presses from a function called *randomButton* [281] in *SpikeGame.java*. In the method, a created random number generator [282] picks a random number, makes it positive, and then modulos 5 [283]. Then, depending on the number, returns the Up, Down, Right, Left, or another random key. The reason Up, Down, Right, Left get more preference over the other keys is due to the fact that the game relies on the input from those buttons and not from the other keys.

In addition, for each test in *TestException.java*, I created a variable [6-11] that counts up how many times the test is run. If the test runs without an exception, then the test is successful; otherwise, the game would crash. In *SpikeGameTest.java*, I also created variables [5-6] for the Start and Restart tests.

By using properties from the Game Manual for testing, the program can be proven to be a correct representation of the game. These tests can only be run because

the implementation of the program does not mutate any objects, allowing every new object to be compared to its previous one. As shown in the console, these tests, which meet the specifications of the Game Manual, run and succeed tens of thousands of times. Thus, because the program's implementation of Spike meets the specified conditions in **Spike I Manual**, the program is an accurate representation of the game.