Kathryn Hodge

Professor McCarthy

203 Software Development

December 10, 2014

Game 2: Meteor Shower

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 **Meteor Shower I Manual** specifies certain conditions for the Field Of Play, Modes of Play, Controls, Scoring System, Persistent Player Attributes, and How To Win/Goal of The Game. If the coded implementation meets all of the specifications of the **Meteor Shower 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 World's), it can be proven that the program is implemented correctly.

====== ====== ====== ====== ====== ====== ====== ====== ======

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

- All tests are in the *TestFunctions.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 a fair amount of my tests, the functions take in an old game, a new game, and a key, and then test whether the new game did the right thing given what key was pressed and what the state of the game was before.
- The two modes of the game are separated as different class files, *MeteorShowerRM* and *MeteorShowerHM* respectively.
- Details about the individual planes are in the *PlaneRM.java* and *PlaneHM.java* files; details about the lasers, meteors, and explosions are in their respectively files as well. Details about lives are in the *Lives.java*

file, details about scores are in the *Score.java* file, and details about the *Col--lideable* and *Tickable* Interfaces are in the interface files.

- If the programmer wants to run tests, the *TestMeteorShower.java* file should be run
- If the programmer wants to play the game, the *RunMeteorShower.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, this is fine because the graphics only handle how the game looks - not the logic of the game (i.e., how the game is played). While the user plays the game, he/she actually switches between two different "Worlds" in GameWorlds. Every time the player starts the game, goes into the Bonus Round, and comes back to the Regular Mode, he/she change which "World" he/she is playing in, and the program creates a new instance of the appropriate World. This makes the constructor of each world extremely important! To test the initial starting conditions of the game, *testConstructorRM()* [396] and *test-ConstructorHM()* [443] functions in *TestFunctions.java* create a new World, each in their respective modes, and check a bunch of conditions dealing with each mode. For example, for the Regular Mode, the test checks if the game starts with three lives, [417] and in the Bonus Mode, the test checks if there are no meteors or lasers on screen when it starts [452]. If the game's constructor passes all the tests from the manual, then the game begins correctly.

====== ====== ====== ====== ====== ====== ====== ====== ======

In **Modes of Play,** the Manual describes two different modes, Regular Mode and Bonus Mode. In the Regular Mode, the player moves the plane right and left and shoots red or blue lasers. Different color meteors fly up from the bottom of the screen and the player should try to shoot red meteors with red lasers and blue meteors with blue lasers. If the user shoots 10 meteors correctly in a row, the player gets a Bonus Round Key. When initiated by the user, the player's laser and the meteors both turn orange, and the plane starts in the center of the screen. The plane now moves up and down and the me-

teors move across (left to right and right to left) the screen. The plane physically moves up and down and the changes direction (right or left) to shoot lasers at the meteors. Because everything is the same color, the player should try to shoot as many meteors as he or she can. A player exits the Bonus Mode once he/she collides his/her spaceship into 5 meteors and goes back to Regular Mode. Testing a type of mode is very broad, so instead I will test each mode in terms of its controls, scoring system, persistent player attributes, and how a player can win or lose the game.

====== ====== ====== ====== ====== ====== ====== ====== ======

In **Controls**, the Manual describes how the player controls the plane and when the lasers are shot. In Regular Mode, the player controls the plane with the left and right arrow keys that move the plane one space in the corresponding direction. If the player does not press the left or right arrow key, the plane should not move. My tester *testPlaneMoveRightAndLeftRM()* [136] takes in an old plane, new plane, and a string and checks if the plane moves accordingly. It also makes sure that the plane does not move off-screen. To shoot a laser, the player must press the 'S' key. My tester *testShootLaser()* is supposed to check if a laser is shot after the player presses the appropriate key, but because my update function is very complicated, the only way to test this property is to use the same algorithm as I used in the program, which doesn't really test anything *[*481*].* If I were to redo the program, I would break up the update() function so that the different components could be tested in smaller pieces. In addition, the player can change the color of the plane's laser by pressing 'D.' *TestSwitchLaserColorRM* checks to make sure that when the user presses the appropriate key, the color is changed and when that key is not pressed, the key stays the same [257]. If the user shoots 10 meteors correctly in a row, the player gets a Bonus Round Key, which can be triggered by pressing 0. This is checked by *testTriggerHyperSpeedMode* [599]. If all of these test succeed, then the implementation of Regular Mode's controls is correct.

In the Bonus Mode, the player can press the up and down arrows to moves the plane up and down the screen. My tester *testPlaneMoveRightAndLeftHM()* [162] takes in an old plane, new plane, and a string and checks if the plane moves accordingly. It also makes sure that the plane does not move off-screen. This test also checks that the plane's width never changes. If the player presses left and right, the plane's correspond-

ing lasers change directions. For a laser, its direction is decided on its creation based upon the plane's direction, meaning a laser can't change its direction in mid-air. In the *testPlaneLaserDirectionHM* [297] test, we create a new instance of the laser with different planes and check that the movements are correct. As with the other mode, a player shoots a laser by pressing 'S,' and *testShootLaser()* checks this in the same way the Regular Mode test did[512].

The game also has independent non-player actors that are controlled by the game. In Regular Mode, the meteors fly downward from the top of the screen automatically, and my *testMeteorMoveRM* function checks if the meteors float down one pixel every tick [210]. A meteor can be either red or blue, so my *testMeteorColorRM* also checks that this is true [189]. In addition, the meteors in this mode should not react to user input, and *reactMeteorRM* makes sure this is true[246].

In the Bonus Round, the meteors fly across the screen (left to right and right to left) automatically and my *testMeteorMoveHM* function makes sure they do this correctly [224]. However, the meteors do react to the plane's movement. If the plane moves up, the meteors flying left to right start going up and across and the meteors flying right to left start going down and across. If the plane moves down, the meteors flying left to right go down and across and the meteors flying right to left go up and across. The *testPlaneMeteorMoveHM* function tests this complicated AI element [739]. It only tests one meteor and one plane rather than sequencing through a series of meteors and planes because by testing an arbitrary meteor and plane, the test proves that any meteor or plane combination will succeed. **I**n addition, every meteor should be same color in this mode, and this is checked by *testMeteorColorHM* [197].

For both modes, my lasers automatically move diagonal, across, or up the screen and my *testLaserMoveRM* [277] and *testLaserMoveHM* [287] check that the lasers are ticking correctly.

====== ====== ====== ====== ====== ====== ====== ====== ======

In the **Scoring System**, the manual describes how a player can get points and lose lives in each mode. In Regular Mode, every time the player shoots a meteor with the same colored laser (i.e. a red meteor with a red laser or a blue meteor with a blue laser), the player gets 10 points and the player's lives stay the same. Every time the

player shoots a meteor with a different colored laser (i.e., a red meteor with a blue laser or a blue meteor with a red laser), the player loses 10 points and the player's lives stay the same. Every time a meteor passes the bottom of the screen, the player should lose a life and the score should remain the same. All of this logic is checked during collisions in *testCollisionRegularMode* [622] because the only time the score or lives will change is when there is a collision, whether it's a plane and a meteor, a meteor and a laser, etc. In essence, a collision or the lack of a collision decides the score and lives of the next tick. However, like my *testShootLaser*() test, this tester involves nearly every element of my game so the only way to test it is to do the same algorithm as I coded the original with, which doesn't really test anything at all. A test is supposed to come at the same problem from a different direction and see the program solves the problem, but if the test is the same as the program, then it doesn't really prove anything. In the future, I want to break up my programs so that they don't just rely on one big function so that I can test them more easily.

In the Bonus Round, every meteor hit gives the player 10 points. In addition, if a meteor hits the plane, the 'Regular' lives should stay the same. The player should never lose points or lives in the Hyper-speed round, but after hitting 5 meteors with its plane, the player exits Hyper-Speed Mode back into Regular Mode. Similar to collision test for regular mode, the bonus round's scoring test depends on its collisions and its score and lives are tested in *testCollisionHyperMode* [679]. This means it is hard to test because collisions involve nearly every element of my game.

====== ====== ====== ====== ====== ====== ====== ====== ======

As for the **Persistent Attributes**, the manual describes that each player has its own laser color, series of bonus round keys, lives tally, and score. Above, I have described how each of these attributes are tested within the program via the Controls and Scoring System, so to save space, I will summarize what I have said above. For the laser color, the player controls the color of their laser and those associated tests prove it works correctly. In Controls, I discussed how a player can get a bonus round key, how that key can trigger the other mode, and those respective tests. As for lives tally and score, they and their specific tests were discussed in the Scoring System section.

====== ====== ====== ====== ====== ====== ====== ====== ======

In **How To Win**, the manual discusses how the player cannot win the game, but simply tries to get the highest score he or she can. The player loses the game when (in Regular Mode) three meteors (total) pass the bottom of the screen, causing the player to lose all three lives.This invariant is checked in *testGameOverLives()* [714].

Nearly every test in this testing suite uses randomly generated key presses from a function called *randomButton* [115]. In the method, a created random number generator, and picks a random number between 0 and 5 [117]. Then, depending on the number, returns the Up, Down, Right, Left, S, D, or another random key. The reason Up, Down, Right, Left, S, and D 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. The 0 key is not included because the Bonus Round keys are tested separately from the functions that use these random buttons. For my randomized tests, I have a function called testingIndividualComponent() [311] that checks the individual components of the game with the very specific tests, and I have two functions, *verifyInvariantsRM* [768] and *verifyInvariantsHM* [781] that act as wrapper and use my individual test functions to verify all of invariants between an unreacted and unticked game and a reacted and ticked game. The *verifyInvariants* tests can be run with random data and while the game is playing, which is very helpful for identifying errors in the game because they can be seen visually with graphics.

In addition, for each test in *TestFunctions.java*, I created a variable [88] that counts up how many times the test is run. If the test runs without an exception, then the test is successful and the function adds one to the static variable associated with that test; otherwise, the game would crash. In the end, the number variable is printed out and in the console, the programmer can see how many times the test was run. If the variable was printed out, then we can assume the test was successful each time it was run or else the program would have crashed.

By using properties from the Game Manual for testing, the program proves 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 many times. Thus, because the

program's implementation of Meteor Shower meets the specified conditions of its manual, the program is an accurate representation of the game.