Test Report

1. Component Testing

We used an iterative, rather than incremental, development process during the life cycle of our system, meaning that we aimed to deliver a full system with each release and improved the functionality of each subsystem with each new release. Due to this development process model, and considering the relatively small size of the system, much of the testing that we performed falls under the category of system testing. Regardless, there were a number of components for which we performed unit tests within the context of the larger system. These component testing methods were employed for the objects from the Player, DynamicObject, and StaticObject classes in our system. These are the component tests that will be referenced in the testing process described below.

Establishing Test Objectives

- Ensure that every Player object responds to keyboard input within each of the levels via the KeyListener interface
- Verify that the randomized DynamicObject objects are continuously mobile, and that the paths appear random
- Verify that the tracker Dynamic Object objects are continuously mobile, and that the paths consistently point toward the Player object in the level
- Show that the StaticObject objects display correctly on each level
- As part of integration testing for the above components, show that Player objects may pick up StaticObject objects and be attacked by DynamicObject objects

Designing & Writing Test Cases

The key to successful testing is the design of test cases, which must be representative and thoroughly exercise all functions to demonstrate the correctness and validity of our three tested components.

With respect to the Player class, our component testing meant that we isolated the Player attributes from the system. That is, we paid no attention to the game board, the static or dynamic objects, or any other attributes of our system. We only attended to how keyboard strokes affected the movement of the Player object. The functionality of our system is sufficiently small so that testing the entire range of inputs (i.e. all keys on the keyboard) was entirely feasible. Thus, our component test for Player objects was subjecting the object to each possible keyboard selection. The test was designated as passing if the Player object responded only to the four arrow keys, moving locations in the corresponding direction on screen.

The StaticObject and DynamicObject classes each endured component testing via observational methods. Considering the randomized locations of the StaticObject and DynamicObject objects, testing over the entire coordinate range was implausible. Instead, we performed enough simulations

of the setup of the levels containing StaticObject objects (level 1) and DynamicObject objects (levels 2 and 3) in order to observe behavior of these objects within normal range and at the edge cases. The design of our test cases for these two classes was constrained by the environment in which they exist (the game board). Hence, normal range was defined as the central region of the game board and edge cases were defined to be the literal edges of the game board. For all StaticObject objects, we performed tests of level 1 setup repeatedly to observe common cases and all four edge cases. The test was marked as passing if we observed all four expected static objects in the visible game board and failing if any of the four static objects were invisible due to their coordinates. By the same token, we performed tests of level 3 setup for all DynamicObject objects to repeatedly observe the common cases and all four edge cases. The design of the tests was slightly more complicated for these objects, due to their dynamic nature. These tests were marked as passing if: (1) we observed every dynamic object to be contained within the visible game board, and (2) the dynamic objects never exited the visible game board.

We also performed integration testing between the components to satisfy our established test objectives. Specifically, we performed integration testing for the interaction of the Player and StaticObject classes and the interaction of the Player and DynamicObject classes. Note that this integration testing was framed within a continuous integration approach due to our iterative development process. The design of the integration testing between the Player object and StaticObject objects was primarily concerned with hitboxes, since a test objective was to verify that the player can pick up all static objects. We performed many boundary tests for the interaction between these two classes. The Player object (1) approached a StaticObject object from each of the four sides with the static object directly in the player path, and (2) side-swiped a StaticObject object on all four sides. For (1), the tests were marked as passing if the static object was picked up as the player moved across them. For (2), the tests were marked as passing if the static object was picked up when any portion of the visible character model contacted any portion of the visible static object model, but not picked up otherwise. For the design of the integration testing between the Player and Dynamic Object classes, we were concerned with both hitboxes and player-tracking. We performed all of the same boundary tests as mentioned above for our Player and StaticObject integration testing, but also verified player-tracking. Therefore, in addition to the aforementioned boundary testing, we also ran a test to observe if the tracker dynamic objects always moved in the direction of the Player object, regardless of their relative orientations on the game board.

Executing Tests

The component testing for our Player class was performed on an unfinished game board, since we were isolating the behavior of the Player object. We executed our test by pressing all possible keyboard entries and observing the resulting Player object movement. Our Player object only moved when any of the four arrow keys were pressed, and the movement corresponded to the direction of the arrow key. Note that this test could fall under the category of white-box testing since we are testing our code implementation more so than our system specifications by pressing every selection on the keyboard.

The component testing for our StaticObject class was repeated 20 times to achieve sufficient code execution in order to verify this component of our system. In a similar manner, the component testing for our DynamicObject class was repeated 10 times (each with a duration of at least 15 seconds) to achieve sufficient code execution for verification. The developers of the system made the determination that these simulations provided ample opportunity to test normal and boundary conditions. Note that due to the randomness employed in these classes as well as their inherent dependency on our GUI, we did not have a use for capture-and-replay or scripting tools for automating tests. Instead, we had to rely on black-box testing by merely observing that the components met system specifications.

The integration testing between the Player and StaticObject classes and the Player and DynamicObject classes was once again under our control via our ability to manipulate the Player object, so these tests fell under the category of white-box testing since we treated each direction (corresponding to unique blocks of code) during different tests. This integration testing yielded the most failed tests due to the complexity of the hitboxes.

Evaluating Test Results

The component test for the Player class passed consistently over the entire iterative development process. The component tests for the StaticObject and DynamicObject classes initially failed boundary testing at the edges of the game board due to inconsistencies between the coordinate representation of the static and dynamic objects and their visible models. After updating the code to account for that discrepancy in representation, the component tests for the StaticObject and DynamicObject classes passed and continued to pass throughout the regression testing.

The integration tests uncovered the most errors embedded in our subsystems, with nearly all of those errors associated with hitboxes. With player-object interactions at the forefront of our GUI-based system, the accuracy of the hitboxes was critical and, as such, our integration test objectives and designs were created with hitbox boundary testing in mind. These tests initially failed for the majority of our test cases because the hitbox design was universally too small. We adjusted these subsystems to interact more realistically in the GUI by continuing to tweak the hitboxes until all of the test cases for our integration testing passed, both for the Player-StaticObject integration and the Player-DynamicObject integration.

In these respects, our testing process allowed us to remove errors from our components and subsystems, thus delivering high-quality code to the system testing stage.

2. System Testing

The system testing as outlined in this section refers to the full integration testing portion of system testing, since the fourth testing category of this document expressly describes acceptance testing. In other words, in this category we focus on the verification activities of system testing rather than the validation activities.

Establishing System Testing Objectives

- Confirm flow of title screen, opening cutscene, and character selection modules
- For levels 1, 2, and 3: verify transition into level, ability to complete quest in a timely manner, and transition out of level
- Confirm flow of exiting cutscene

Designing & Writing Test Cases

Full integration testing for our system can be broken down into five major phases: the introductory sequence, each of three levels, and the exit sequence.

The introductory sequence was designed to progress through three discrete game modules: the title screen, the opening cutscene, and the character selection. The first two modules integrate the SOSGame, Board, and GameSound classes into the complete system. Our test design for these two modules followed black-box testing methods. Specifically, our test for the title screen module was to attempt every possible keyboard input, and the test passed if the enter key was the only key allowing advancement to the next module. Our test for the opening cutscene module addressed non-functional requirements. That is, our test case for the opening cutscene was a performance test of the timing variety. Timing tests are used to evaluate those requirements dealing with time to perform a function, performing the transaction and verifying that the non-functional duration requirements are met. Our test for this opening cutscene module passed if the total duration that this module was displayed was no shorter or longer than 5 seconds. Finally, the character selection module integrated the SOSGame, Board, GameSound, and Player classes into the complete system. This module accepts user input and was therefore heavily constrained within the functional requirements, yielding a larger combination of tests than for the previous two modules. Our test cases for this module dealt exclusively with keyboard input like the title screen module, but in this case our black-box testing resulting in (1) boundary testing for username length by attempting to enter 10- and 11-character usernames, (2) boundary testing for character selection by attempting to enter integers less than or equal to 1 and greater than or equal to 2, and (3) printing both character attribute selections (username and image) to stdout following completion of the character selection module. These tests passed when the username length was found to be constrained to 10 characters, character image selection was only modified with integers 1 and 2, and both character attributes in stdout exactly matched those selected within the module.

The three levels provided the most thorough opportunity for system testing, since they integrate all of the classes comprising the system: SOSGame, Board, GameSound, Player, StaticObject, and DynamicObject. Already having completed subsystem integration testing between the Player and StaticObject/DynamicObject classes (described in the Component Testing category), our primary system testing objective for each level was to verify functional entry and exit of each level and capability for completion. In other words, we needed to ensure that completion of one level resulted in our system generating the next level and that the quest assigned to the user within each level was not an impossible task. Thus, our testing method at this system level was black-box as opposed to white-box (which was implemented in some of the component testing). This black-box methodology naturally resulted from our GUI-based system that incorporates randomness. Our test suite for the transitions was observational, simply performing a number of simulations of level transitions. These

tests passed if there was nothing preventing the continuation of the game from one level to another. Our tests for ease of level completion were performance tests of the timing variety, addressing the level duration non-functional requirements described in our requirements document. The timing test passed for level 1 if average completion was under 1 minute, for level 2 if average completion was under 3 minutes, and for level 3 if average completion was under 3 minutes.

The exit sequence was designed to progress through only one discrete game module: the exiting cutscene. This module integrates the SOSGame, Board, and GameSound classes into the complete system. Similarly to the title screen module mentioned above, our test for the exiting cutscreen module was to attempt every possible keyboard input, and the test passed if the escape key was the only key permitting termination of the Java applet.

Executing Tests

The system testing for the title screen, opening cutscene, and character selection modules was straightforward to execute except for the character selection module. For this module, we had control full over the testing via user keyboard input, and therefore attempted a wide sampling of representative entries for usernames (lowercase letters, uppercase letters, numbers, special characters, backspace, combinations thereof) and verified the continuity of these usernames to stdout. These input combinations were performed in conjunction with the boundary tests that we designed and wrote for the introductory sequence.

The system testing for the three levels consisted of each of the three developers completing the game three times in a timed environment, for a total of 9 timed games. Each of the levels was individually timed, and an average was taken over the 9 trials for each level. This level average was the average used to determine whether our timing performance test on each level passed or failed. Note that for our system, three developers with three trials each is a sufficient measure of completion time. However, when system testing a larger system with a larger audience, it would be wise to use statistical methods of random sampling to find many more test subjects and only time each subject's first run-through of the game.

The system testing for the exiting cutscene immediately followed from the system testing objectives and design described above.

Evaluating Test Results

The system tests for the introductory and exit sequences passed initially and continued to pass the regression testing throughout our entire iterative development process.

We uncovered a significant error in our system while performing the function testing of each of the levels. The transition into level 2 was found to sometimes be caught in an infinite loop of restarting the level, never allowing the Player object to navigate the game board or complete the level. This error was due to random DynamicObject objects drifting to the starting position of the Player object. Since the code did not reset the dynamic object positions when the level restarted, the Player object was instantly caught upon restart of the level and the DynamicObject object was never able to leave the starting position. This error uncovered by our function tests was fixed by reassigning the

dynamic object starting locations away from the Player object starting location each time that the level restarted. After this code modification, the tests assessing transition into and out of each level passed consistently. The performance testing for level 2 described above initially failed the prescribed timing test, indicating that we had made level 2 too difficult to complete in under 3 minutes. While not an outright error in a statement of code, the logic for level 2 needed to be adjusted to lessen the difficulty of completion. Hence, we reduced the number of DynamicObject objects on the game board and ran the timing test again, following this procedure until the test as described above passed.

3. Regression Testing

Establishing Regression Testing Objectives:

- Verify that each successive demo's functionality works in accordance to the specified requirements and user stories.
- Guarantee that each successive demo's performance is at least as good as that of older demos.
- Test that new bugs are not introduced into the system for each successive and completed demo.

Designing Test Cases

When designing the test cases for our system's regression testing, we assumed that users will only use a standard QWERTY keyboard to play the game. In other words, we only need to test for user input involving keys found on this particular type of keyboard. Furthermore, we note that our system's functionality for each successive demo increased in the following manner:

- Demo 0:
  - Opening title screen is displayed until the user presses the ENTER keyboard key.
  - Opening cutscene is displayed for a set amount of time.
  - The game's background music is looped until the user quits the game.
  - Users can enter their username by using both lowercase and uppercase letters, as well as numbers.
- Demo 1:
  - User's username is displayed.
  - Level 1 is displayed.
  - The game character is displayed on the screen and users can move it in four directions by pressing the four arrow keyboard keys.
- Demo 2
  - Users can now choose between 2 game characters.
  - Static objects (i.e. level 1 only has rocks) are introduced.
  - Objective for level 1 is displayed and implemented (i.e. users must "pick up" all four rocks to display an SOS sign). We note that these four rocks are displayed randomly every time the game is reset.
  - Sounds are played when static objects are picked up.
  - Level 2 is displayed.

- Demo 3
  - Objective for level 2 is displayed and implemented (i.e. users must eat the antidote fruit after eating the poisonous fruit without touching any of the enemies or water). We note that these two types of fruits are also static objects. Furthermore, the enemies are dynamic objects that move randomly around the board.
  - When the user eats the poisonous fruit, user controls are “switched” and are set back to normal only when the game character “eats” the antidote fruit.
  - Users can quit the game by pressing the ESC keyboard key.
  - Level 3 is displayed.
- Demo 4
  - Objective for level 3 is displayed and implemented (i.e. users must pick up the flare gun without touching any of the enemies). We note that the flare gun is also a static object.
  - A second class of enemy that follows the game character is introduced.
  - The final closing cutscene emphasizing that the user completed the entire game is displayed until the user quits the game.

Writing Test Cases

It is important to stress that the test cases for our regression testing must meet the previously three outlined objectives and must support the notion of repeating tests to make sure that previously working functionality is not broken as new releases (or demos) are made. We also note that our group used an iterative development process with 2-week sprints since this was one of the required and specified time periods for us to demonstrate system progress. The following test cases for each demo provided a way for us to conduct regression testing:

- Demo 0:
  - 0TC1-When starting the game, the title screen should be displayed immediately. Press all keyboard keys (except the ESC and ENTER keyboard keys). Any keyboard key that is not ESC or ENTER should not change the state of the game at this stage. Press ESC and make sure that game is existed. Restart the game and press the ENTER key instead. Play through the rest of the game and make sure that the title screen is NEVER displayed again.
  - 0TC2-After the ENTER keyboard key is pressed during the title screen, an opening cutscene should be displayed for a total of 5 seconds. Quickly press ESC within these 5 seconds to make sure that the game exists. Restart the game but instead of pressing the ESC keyboard key, press all other keys to make sure that it does not “break” the game. Restart the game and do not press any keyboard keys; the opening cutscene should be displayed for a total of 5 seconds.
  - 0TC3-Restart the game various times to make sure that the background music “starts” playing from the beginning. Furthermore, make sure that the background music loops forever. To test this, wait a total of 10 minutes. If the background music is still playing, then the game successfully loops the soundtrack song. If, at any point, you exist the game, the background music should stop immediately.
  - 0TC4-When entering a username, press all keyboard keys in the QWERY keyboard (except ESC and ENTER). Any key that is not an uppercase/lowercase letter or a number will be ignored.

Pressing a non-alphabetic or non-numeric keyboard key will not break the game. (NOTE: Users can use the “DELETE” keyboard key to delete their previously entered letters or numbers. Furthermore, usernames must be between length 0 to 10!)

- Demo 1:
  - Regression Testing-Testing 0TC1 through 0TC4 as well as the following test cases.
  - 1TC1-After entering a username, press the ENTER keyboard key. The correct username entered by the user should be displayed. Any non-valid keys pressed should never be displayed inside the username string. Any keys pressed after a length-ten username is met should never be displayed inside the username string.
  - 1TC2-After the user presses the ENTER key to select their username, the level 1 map will be displayed. Play through the rest of the game and make sure that the level 1 map is NEVER displayed again.
  - 1TC3-During level 1 users can control the game character by pressing the four arrow keys on the keyboard. Press all other keyboard keys that are not the four arrow keys to make sure that nothing breaks. The left arrow key should move the game character left, the right arrow key should move the game character right, and so forth. In addition, the game character is not allowed to move outside the applet’s border.
- Demo 2:
  - Regression Testing-Testing 0TC1 through 0TC4, 1TC1 through 1TC3, and the following test cases.
  - 2TC1-When choosing game characters, the number 1 corresponds to the man character while the number 2 corresponds to the female character. Any other keyboard keys (with the exception of ESC and ENTER) should be ignored since they do not represent any valid character selection. Test that pressing any other keys do not break the game.
  - 2TC2-A total of four rocks should be displayed during level 1. Restart the game several times to make sure that they are displayed randomly with each gameplay.
  - 2TC3-Users are allowed to pick up these rocks and can only advance to the next level when all four rocks are picked up. Pick up the four rocks in different order in each replay. The game level should increment only when the last rock is picked up. Play through the rest of the game and make sure that the rocks are NEVER displayed again.
  - 2TC4-Every time a rock is picked up a sound should be played. That is, a sound to indicate that a static object is picked up should only be played a total of 4 times.
  - 2TC5-Play through the rest of the game and make sure that the level 2 map is NEVER displayed again.
- Demo 3:
  - Regression Testing-Testing 0TC1 through 0TC4, 1TC1 through 1TC3, 2TC1 through 2TC5, and the following test cases.
  - 3TC1-Run into every enemy and every piece of water on the screen on purpose since this should automatically “reset” level 2.
  - 3TC2-Every time the poisonous fruit is consumed the arrow keys will be “switched.” The controls are set back to normal ONLY when the antidote fruit is consumed.

- 3TC3-Pressing the ESC keyboard key should exit the game, regardless at what “level” or “place in time” the user is on.
- 3TC4-Play through the rest of the game and make sure that the level 3 map is NEVER displayed again.

- Demo 4:
  - Regression Testing-Testing 0TC1 through 0TC4, 1TC1 through 1TC3, 2TC1 through 2TC5, 3TC1 through 3TC4, and the following test cases.
  - 4TC1-Make sure that the new “class” of enemies follow the game character and that they do not move “randomly” throughout the screen.
  - 4TC2-Finishing the game will ONLY happen when the flare gun is picked up.
  - 4TC3-Run into both “classes” of enemies on purpose since this should automatically “reset” level 3.
  - 4TC5-After level 3 is completed, only the final closing cutscene should be displayed. All keyboard keys (except ESC) will be ignored and should not break the game.

Executing Tests

During each iterative 2-week sprint, we used these regression test cases to verify that each successive demo’s functionality worked in accordance to the specified requirements and user stories. Furthermore, each test case was executed to make sure that no new bugs were introduced into the system. It is important to note that each regression test case was assigned to a specific group member since this added another level of “testing” (i.e. a group member would know easier when our system passed or failed a regression test case). To execute all regression test cases, our group members had to restart the game dozens of times in order to make sure that user input did not break the game. Additionally, our group did not used any regression testing tools and focused more on system testing rather than component testing.

Evaluating Test Results

When adding new functionality within each demo, each group member gave either a passing or failing score for each of their assigned regression test case. Such score was computed by using a somewhat qualitative analysis on how the system interacted with user input. Furthermore, each group member used the black-box testing methodology since looking at each other's code was discouraged. A demo was labeled as successful if and only if both the regression and new test cases had a passing score. Because each demo built on top of the previous demos, a demo labeled as “successful” allowed us to conclude that the performance of our system was at least as good as that of older releases.

4. Acceptance Testing

Establishing Acceptance Testing Objectives

- Use system testing to validate if system features were considered accepted by the user.
- Construct the system testing (validation) using the acceptance criteria on our user stories.

Designing & Writing Test Cases

We note that the validation portion of our system testing used the acceptance tests found in our user stories page since these user stories acted as our requirement specification. In other words, our user stories defined how our proposed system (which, in this case, is a game) shall behave in terms of user input. It is important to note that our reasoning behind why we used such methodology to construct our system testing's validation portion was because we needed to replay the game many times as "customers/users" and not as "developers." That is, we needed to see our system from the perspective of customers and users (e.g. game players, etc.). Because of this assumption, we can use both our user stories and acceptance criteria to validate our system. Such system validation would inform us as developers which features are considered accepted by users. The following ten acceptance tests guided the validation portion of our system testing:

1. Playthrough of the game with timing on how long each level takes to complete just by completing all tasks with minimal sidetracking will mark the minimum completion time. Additional playthrough of the game with conditions of both completing the tasks and full exploration will mark the maximum completion time.
2. The user spends more time and executes more actions for each successive level.
3. After the opening cutscene, give the player the option to choose between the two available characters. Once chosen, the game will continue with the character that the player originally picked.
4. While playing the game, users will be able to move the game character in all four directions by using only the up, down, left, and right keyboard keys (i.e. the up key moves the character north, the down key moves the character south, and so forth). If other keys move the game character at all, the acceptance test will fail.
5. Each number corresponds to a specific game character (e.g. "1" represents choosing the male game character, "2" represents choosing the female game character, etc.). If each number maps to the correct game character that can be chosen by the user, then the acceptance test will have been passed.
6. Play the entire game, from beginning to end. The opening and closing cutscenes should display at the right times and should enhance the game's storyline consistency.
7. Restart the game multiple times and verify that the object locations and environments change (e.g. rocks that need to be collected on level 1 are displayed at random locations every time the game is restarted).
8. Replay the game several times to make sure that user input is not delayed and that the game character moves smoothly.
9. Test a variety of sounds to find the best matched one for the given scenarios.
10. Test a variety of sounds to find the best matched one for the given action. For example, if the game character eats a fruit, then a "slurping" sound effect should be played through the computer's speakers.

We note that each corresponding acceptance test directly correlates to a single and specific user story. If an acceptance test passes, then its corresponding system feature is validated and accepted by that

user story. The following ten user stories correlate directly (in that order) to the previously ten mentioned acceptance tests:

- 1.1: Game Play Time to Completion
- 1.2: Increasing Challenge as Levels Progress
- 2.1: Choosing a Character
- 3.2: Using Arrow Keys to Move the Game Character
- 3.3: Using the Number "1" and Number "2" to Choose the Game Character
- 5.1: Opening and Closing Cutscenes
- 6.1: Variability of Object Locations and Environment
- 7.1: Smooth Character Movements
- 8.1: Background Music
- 8.2: Sound Effects

(NOTE: Each description as to whether each user story is accepted as completely implemented or not will be stated on the following "Evaluating Test Results" section.)

Executing Tests

It is important to reiterate that while executing all the acceptance tests, each group member viewed the system from the perspective of customers and users in order to promote an unbiased conclusion as to whether an acceptance test was passed. Furthermore, all group members used mostly "qualitative" factors to conclude if our system successfully passed a specific acceptance test. While executing all acceptance tests, our group also made sure that each team member gave a passing "score" for each specific acceptance test. In other words, each acceptance test must have passed successfully according to each group member, otherwise its corresponding feature cannot be validated. Because every game player has different standards for what constitutes as quality gameplay, it is important to implement such methodologies in order to decrease the amount of "discrepancy" between qualitative opinions among individuals.

Evaluating Test Results

The following list describes whether each user story is accepted as completely implemented or not by using the previously mentioned acceptance tests.

- 1.1: Game Play Time to Completion
  - Directly correlates to the acceptance test 1.
  - Our group completed the game around a mean time of 14 minutes. This is a "reasonable" time for completion. Therefore, the corresponding acceptance test passes and the user story is accepted as completely implemented.
  - Our system's feature of reasonable time to completion is accepted by the user.
- 1.2: Increasing Challenge as Levels Progress
  - Directly correlates to the acceptance test 2.
  - The average time to complete each level increased as each level progressed for each individual group member. We assume that increase in time directly correlates to increase in

difficulty. Therefore, the corresponding acceptance test passes and the user story is accepted as completely implemented.
    - Our system's feature of increasing challenge as levels progress is accepted by the user.
- 2.1: Choosing a Character
    - Directly correlates to the acceptance test 3.
    - The user can choose between two characters. That chosen character is displayed throughout the entire game. Therefore, the corresponding acceptance test passes and the user story is accepted as completely implemented.
    - Our system's feature of choosing a character is accepted by the user.
- 3.2: Using Arrow Keys to Move the Game Character
    - Directly correlates to the acceptance test 4.
    - The user can move the game character in all four directions by using the four arrow keyboard keys. The up key corresponds to moving the game character up, the down arrow key corresponds to moving the game character down, and so forth. Therefore, the corresponding acceptance test passes and the user story is accepted as completely implemented.
    - Our system's feature of using arrow keys to move the game character is accepted by the user.
- 3.3: Using the Number "1" and Number "2" to Choose the Game Character
    - Directly correlates to the acceptance test 5.
    - The user can choose character 1 by pressing the "1" numeric keyboard key or character 2 by pressing the "2" numeric keyboard key. Therefore, the corresponding acceptance test passes and the user story is accepted as completely implemented.
    - Our system's feature of using numbers 1 and 2 to select the game character is accepted by the user.
- 5.1: Opening and Closing Cutscenes
    - Directly correlates to the acceptance test 6.
    - The game displays an opening cutscene when it starts and displays a closing cutscene when the user completes the last level. These cutscenes enhance the game's storyline consistency since it explains to the user what is going on. Therefore, the corresponding acceptance test passes and the user story is accepted as completely implemented.
    - Our system's feature of including opening and closing cutscenes is accepted by the user.
- 6.1: Variability of Object Locations and Environment
    - Directly correlates to the acceptance test 7.
    - The game was restarted dozens of times. For each restart, each individual rock was displayed at a random location on level 1. Therefore, the corresponding acceptance test passes and the user story is accepted as completely implemented.
    - Our system's feature of variability of object locations and environment is accepted by the user.
- 7.1: Smooth Character Movements
    - Directly correlates to the acceptance test 8.
    - Each team member agreed that the game character moved "smoothly" throughout the board for each game level. Therefore, the corresponding acceptance test passes and the user story is accepted as completely implemented.
    - Our system's feature of smooth character movements is accepted by the user.

- 8.1: Background Music
  - Directly correlates to the acceptance test 9.
  - The game plays an appropriate background music on a loop until the player exists the game. Therefore, the corresponding acceptance test passes and the user story is accepted as completely implemented.
  - Our system’s feature of background music is accepted by the user.
- 8.2: Sound Effects
  - Directly correlates to the acceptance test 10.
  - The game plays an appropriate sound when the game character picks up a rock or eats a poisonous/antidote fruit. Therefore, the corresponding acceptance test passes and the user story is accepted as completely implemented.
  - Our system’s feature of sound effects is accepted by the user.

We note that all user stories (and acceptance tests for that matter) are accepted as completely implemented. Therefore, we can conclude that all our system’s features that are captured by these acceptance tests are validated and accepted by our system’s targeted users.