Disc Golf 2D Game

Jon Griesen

Introduction

The following writeup and associated deliverables are intended to satisfy the half-semester project requirements for both CIS 612 (Requirements Specification) and CIS 613 (Software Testing). In particular, the application of requirements specification and testing to the ongoing development process serves as the basis for this project.

Project Description

The core premise of this project is the ongoing development of a top-down 2D game based on the sport of disc golf. Gameplay on a given hole will consist of the player attempting to hit the basket with the disc in the fewest possible number of throws (measured as strokes) while avoiding various obstacles. Upon completion of a hole, the game will load the subsequent hole. This process will continue until the player has completed all holes on their course scorecard.

Ultimately, my hope with this project is to leverage the concepts of requirements specification and software testing to arrive at a superior final product. I would also like to point out that completion of this project will not coincide with completion of the game at large, as this will likely serve as an ongoing personal project. The extent to which I apply refinements will largely depend on how much promise is shown by the initial minimum viable product achieved throughout the course of this project.

Project Tools

- Java (Programming Language)
- LibGDX (Development Framework)
- Tiled (Level Design)
- Paint.net (Graphics)
- PreSonus StudioOne (Game Audio)

Requirements Specification (CIS 612)

Process

My initial goal with regards to requirements specification was to essentially “lock in” the scope of the game I intend to develop. I had a laundry list of ideas for game mechanics sprawled out in a text document as part of an introductory brainstorming session, and I felt that these initial ideas would translate well into a set of natural language requirements. When evaluating the existing notes, I decided to focus on functional requirements that would describe the game in a way that is less open to interpretation, at least when compared to non-functional requirements.

The requirements engineering process for this project was of a continuous nature (and likely is the closest thing I have experienced to any form of Agile development), as I needed to get an idea of how difficult it would be to implement certain mechanics within my game from a programming standpoint. I scrapped several initial ideas such as unique disc selection and curved flight paths due to difficulty of implementation within the constraints of the LibGDX framework. Additionally, I found that it was necessary to rewrite several of the surviving requirements as I better understood what the specific implementation looked like within my code.

Once I had a working set of natural language requirements, I performed a sort of "formalization" to arrive at two decision tables describing key game behavior. The specific behaviors in question included the disc within the game itself, as well as the game scoring mechanics (namely the interaction between the player scorecard for the current round and the all-time game leaderboard). The use of decision tables in both cases seemed appropriate due to the Boolean nature of the conditions to be evaluated. I felt that mapping out all possible combinations and distilling them down to a pair of reduced decision tables would help give me a better understanding of the underlying mechanics and hopefully point out any flaws in the game logic.

Benefits

The development of natural language requirements was an incredibly valuable exercise in terms of determining the ultimate direction I wanted this game to take. It was quickly apparent which ideas melded well together and which ones were not compatible, as well as where my own skills (or lack thereof) would limit my ability to implement certain functionality. In many ways I was humbled by "scope paring" rather than scope creep, and I think this will ultimately result in a more polished game. I would rather have a smaller number of mechanics that work well together than have a mess of features that do not contribute to an enjoyable gameplay experience.

Additionally, the use of the decision tables provided a convenient means of lessening the ambiguity inherent to the natural language requirements. A good example of this is present in the disc behavior decision table, as I feel the decision table provides a relatively "stateful" look at how the game is played. For example, rule 4 describes the expected behavior of the disc if it is thrown and lands in a pond. In the case of the Scorecard decision table, the rules are more straightforward, but I still believe the details of scoring behavior are more concisely conveyed via the decision table format when compared to the natural language requirements. It is also noteworthy that the reduced decision tables can serve as a basis for tests.

Drawbacks

The drawbacks associated with natural language requirements were discussed extensively in class, and I feel that a fair number of them are present in my requirements despite my best efforts. The primary factor here is the inherent ambiguity of natural language requirements, and although this ambiguity can be reduced by the formalization process necessary to convert to a decision table, a high-level overview of the requirements does not fully describe all attributes of the game. These requirements are also subject to considerable revision over the course of development, as there is no guarantee that all features that will ultimately be present in my game are captured by the current set of requirements.

With the decision tables there is the consideration of rule explosion resultant of an excessive number of conditionals. I made the decision to try and focus on relatively compartmentalized game logic (rather than the game at large) to prevent exponential bloat from taking hold. This decision was particularly important within the context of the disc behavior decision table, as taking conditions such as collision detection into account would have likely inflated the table extensively. Even in the case of my limited conditional tables, there is still a considerable amount of time and effort necessary to model out and reduce the rules within the tables (Disc: 5x conditions → 2^5 = 32 rules **|** Scorecard: 4x conditions → 2^4 = 16 rules). If I had attempted to model out all game behavior (collisions, velocity of the disc, mid-flight adjustments, etc.) any value provided by the table would have become almost impossible to extract due to its complexity. I felt a more straightforward implementation of the decision tables would be more effective for this reason.

Deliverables

Natural Language Requirements (Functional)

Menus

- The main menu shall initialize a new game when the “New Game” icon is clicked.
- The main menu shall terminate the application when the “Quit Game” icon is clicked.
- The main menu shall display the top 5 scorecards when the “Leaderboard” icon is clicked.
- The main menu shall provide the player with the ability to enter their name when starting a new game.

Scoring

- The scorecard shall be created when a new game is initialized.
- The scorecard shall track the player’s score on all holes when the game is in progress.
- The scorecard shall increment the current hole stroke value when the disc comes to a complete stop.
- The scorecard shall display on the screen when the current hole is completed.
- The leaderboard shall be read into memory from CSV when a new game is initialized.
- The leaderboard shall consist of the top five scorecards from all previous course rounds.
- The leaderboard shall write to CSV when total score is sufficient for a top 5 leaderboard placement.

Gameplay

- The aim guide shall extend from the disc to the cursor when the disc is at a complete stop.
- The aim guide shall specify the heading and velocity of the disc when the player clicks.
- The disc shall be thrown when the aim guide is active and the player clicks within the game window.
- The disc shall reset to the spot of the previous throw when it comes to a complete stop out of bounds.
- The disc shall reset to the spot of the previous throw when it comes to a complete stop within a pond.
- The disc shall come to a complete stop when contact with the basket is detected.
- The disc’s flight shall be adjusted when an “Arrow Key” is pressed and minimum velocity is achieved.
- The in-flight adjustment count shall reset to a value of 10 when the disc comes to a complete stop.
- The disc shall have a maximum of 10 in-flight adjustments per throw.
- The current hole shall be completed when contact with the basket is detected.
- The subsequent hole shall be initialized after the player scorecard has been displayed.
- The game shall end when the 9^{th} hole has been completed.

Rendering

- The game shall display sprites that are 32x32 pixels in size when the game is running.
- The game shall initialize all sprites specified with a course Tiled Map when a new hole is loaded.
- The game shall display all graphics at a consistent resolution of 1024 x 768 pixels.
- The game shall redraw all graphics a rate of 60 frames per second.

Course

- The game shall initialize game bodies based on their Tiled Map specifications when a new hole is loaded.
- The tree, rock, boundary, and basket bodies shall be susceptible to collision with the disc body.
- Optional course bodies shall include the tree, rock, pond, and boundary.
- Mandatory course bodies shall include the disc and basket.

Decision Table #1 – Disc Behavior

Step 1 – Eliminate Impossible Scenarios

Disc Behavior																															
Conditions	**R1**	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**	**R9**	**R10**	**R11**	**R12**	**R13**	**R14**	**R15**	**R16**	**R17**	**R18**	**R19**	**R20**	**R21**	**R22**	**R23**	**R24**	**R25**	**R26**	**R27**	**R28**	**R29**	**R30**	**R31**	**R32**
c1. discThrown	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
c2. discStopped	T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F	T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F
c3. discOverPond	T	T	T	T	F	F	F	F	T	T	T	T	F	F	F	F	T	T	T	T	F	F	F	F	T	T	T	T	F	F	F	F
c4. discOutOfBounds	T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F
c5. discInBasket	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F
a1. Reset Disc Position				X		X																										
a2. Display Aim Guide								X																X								
a3. Increment Score				X		X	X	X																								
a4. Apply Penalty				X		X																										
a5. End Hole							X																									

Step 2 – Condense Rules

Disc Behavior								
Conditions	**R4**	**R6**	**R7**	**R8**	**R12**	**R14**	**R16**	**R24**
c1. discThrown	T	T	T	T	T	T	T	F
c2. discStopped	T	T	T	T	F	F	F	T
c3. discOverPond	T	F	F	F	T	F	F	F
c4. discOutOfBounds	F	T	F	F	F	T	F	F
c5. discInBasket	F	F	T	F	F	F	F	F
a1. Reset Disc Position	X	X						
a2. Display Aim Guide				X				X
a3. Increment Score	X	X	X	X				
a4. Apply Penalty	X	X						
a5. End Hole			X					

Step 3 – Condense Actions

Disc Behavior						
Conditions	**R4**	**R6**	**R7**	**R8**	**R12**	**R24**
c1. discThrown	T	T	-	T	T	F
c2. discStopped	T	T	-	T	F	T
c3. discOverPond	T	-	-	F	-	-
c4. discOutOfBounds	-	T	-	F	-	-
c5. discInBasket	-	-	T	F	-	-
a1. Reset Disc Position	X	X				
a2. Display Aim Guide				X		X
a3. Increment Score	X	X	X	X		
a4. Apply Penalty	X	X				
a5. End Hole			X			

Step 4 – Final Reduced Table

Disc Behavior						
Conditions	**R4**	**R6**	**R7**	**R8**	**R12**	**R24**
c1. discThrown	T	T	-	T	T	F
c2. discStopped	T	T	-	T	F	T
c3. discOverPond	T	-	-	F	-	-
c4. discOutOfBounds	-	T	-	F	-	-
c5. discInBasket	-	-	T	F	-	-
a1. Reset Disc Position & Apply Penalty	X	X				
a2. Display Aim Guide				X		X
a3. Increment Score	X	X	X	X		
a5. End Hole			X			

System Assumptions

1. In game conditions are evaluated every single frame upon runtime (60 times per second).
2. Actions a1, a2, and a5 are performed every frame in which the conditions are satisfied.
3. Action a3 is performed a single time per rule satisfaction (rules R4, R6, R7, and R8).
4. The disc is immediately stopped upon being reset to its prior position.

Decision Table #2 – Scorecard Behavior

Step 1 – Eliminate Impossible Scenarios

Scorecard Behavior																
Conditions	**R1**	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**	**R9**	**R10**	**R11**	**R12**	**R13**	**R14**	**R15**	**R16**
c1. playerScore > Par	T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F
c2. playerScore < Par	T	T	T	T	F	F	F	F	T	T	T	T	F	F	F	F
c3. playerScore > fifthPlaceScore	T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F
c4. playerScore < fifthPlaceScore	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F
a1. Apply Scratch Certification										X	X	X		X	X	X
a2. Apply Top Five Certification							X				X				X	
a3. Write Player Score to CSV							X				X				X	

Step 2 – Condense Rules

Scorecard Behavior									
Conditions	**R6**	**R7**	**R8**	**R10**	**R11**	**R12**	**R14**	**R15**	**R16**
c1. playerScore > Par	T	T	T	F	F	F	F	F	F
c2. playerScore < Par	F	F	F	T	T	T	F	F	F
c3. playerScore > fifthPlaceScore	T	F	F	T	F	F	T	F	F
c4. playerScore < fifthPlaceScore	F	T	F	F	T	F	F	T	F
a1. Apply Scratch Certification				X	X	X	X	X	X
a2. Apply Top Five Certification		X			X			X	
a3. Write Player Score to CSV		X			X			X	

Step 3 – Condense Actions

Scorecard Behavior				
Conditions	**R6**	**R7**	**R10**	**R11**
c1. playerScore > Par	T	T	F	F
c2. playerScore < Par	-	-	-	-
c3. playerScore > fifthPlaceScore	-	-	-	-
c4. playerScore < fifthPlaceScore	F	T	F	T
a1. Apply Scratch Certification			X	X
a2. Apply Top Five Certification		X		X
a3. Write Player Score to CSV		X		X

Step 4 – Final Reduced Table

Scorecard Behavior				
Conditions	**R6**	**R7**	**R10**	**R11**
c1. playerScore > Par	T	T	F	F
c3. playerScore < fifthPlaceScore	F	T	F	T
a1. Apply Scratch Certification			X	X
a2. Apply Top Five Certification		X		X
a3. Write Player Score to CSV		X		X

System Assumptions

1. Player score is determined upon completion of a course (9 holes).
2. Par is evaluated on a course level (sum of par values for all holes on a given course).
3. Top five certification is awarded based on comparison against other scorecards for the given course.

Conclusion

After consideration of the benefits and drawbacks associated with the requirements specification methods chosen for this project, I believe that their inclusion was a net positive to the development of my game. Natural language requirements and decision tables are both appropriate specification methods to use early in the software development lifecycle, and this made sense in the context of my game development process. Although it is likely that I could have progressed further along in terms of actual game development had I neglected requirements, I sincerely believe that the current state of the product would have been sloppier and less cohesive. The natural language requirements helped limit the degree of scope creep that could have overtaken my project otherwise, and the decision tables helped ensure that the overall game logic is easily understood and representative of my expectations. The disc behavior decision table was an incredible resource for the demonstration of completeness of possible disc states, and while there are other methods that are more "stateful" at a surface level (e.g.: Petri Nets, FSM, Statecharts), I still felt that the decision table was the most appropriate tool to facilitate test generation down the road.

My ultimate takeaway is that I will strongly consider the use of requirements for future projects. It would be interesting to see the process at work when there are multiple parties present (rather than a single developer), as a key goal of requirements is to efficiently convey information and facilitate progress.

Software Testing (CIS 613)

Process

Devising a testing strategy that would benefit my game was admittedly much more difficult than the initial brainstorming process and gradual refinements associated with requirements specification. With that point made, I believe that a fair number of the hurdles to testing my 2D disc golf game are intrinsic to the process of video game development at large. Many of these hurdles are somewhat obvious in hindsight, but I overlooked almost all of them in the initial phases of determining my project direction. Particularly, video games are incredibly reliant on a high number of external libraries and frameworks to ensure core gameplay mechanics to operate as intended. Furthermore, gameplay very much occurs in real-time, and frequent calls to rendering methods that are built into the underlying game framework present a significant hinderance to testability. For these reasons there is a considerable amount of discourse on online forums surrounding the benefits and drawbacks of unit testing within a video game context, with a fair number of developers writing off the approach altogether. Regardless, there are still some proponents of unit testing for video games, and several of their articles helped shape the approach I ultimately decided to pursue with regards to testing for my game.

When considering starting points for applicable unit tests for my game, I looked to the reduced decision tables that were fleshed out during the requirements specification phase. While the disc behavior decision table provides a relatively clean and concise look at possible game conditions (which are essentially states in which the disc can exist), the issues mentioned in the preceding paragraph rapidly begin to rear their heads when attempting to test this behavior. The LibGDX framework does a considerable portion of the heavy lifting in terms of collision detection between bodies, and most importantly governs the physical attributes of the disc in motion (linear velocity, angular velocity, friction, angle of ricochet, etc.). All of this is to say that the *entire game* really needs to be running to perform any direct testing on the disc's behavior. Automation testing through Selenium or another resource was a possibility, but I ultimately concluded that there were too many moving parts to glean useful test details from this approach.

The testing inspiration that I drew from browsing game development forums was a focus on areas of the game that could be tested in isolation. For this reason, I chose to use the scorecard behavior decision table to write unit tests pertaining to the scoring system. This mechanic is much more compartmentalized when compared to the high-level behavior associated with the disc. Specifically, I chose to generate tests based on the decision table testing process. To further enhance the usefulness of the suite, I also wrote tests that verify the ordering system within the top five leaderboard spots. I consider these tests to be somewhat akin to the equivalence class testing approach. Finally, I included some exploratory test cases that evaluate the scorecard comparator method and whether it correctly accounts for alphabetical ordering of names in the event of duplicate scores within the leaderboard.

My ultimate hope with testing was to verify proper functioning of the cases in which the player's scorecard is written to the leaderboard, as well as the accuracy of the "certification" flags to be applied to a scorecard at the end of a round. Lastly, in the event a player scorecard is added to the leaderboard and written to the CSV file, I wanted to ensure that the player scorecard would appear in the correct order on the leaderboard relative to the existing scorecard data.

Important Considerations

Before discussing the specific testing approaches utilized for my project, it is important to note that a dedicated helper class called CheckCSV was utilized for all tests. The rationale behind this choice is that the game itself reads in the leaderboard via CSV file and converts all scores to an appropriate data structure (specifically an ArrayList referencing ScoreCard objects). When the user selects the "Leaderboard" option from the main menu, this resource will ultimately be accessed to display the information in real time. For this reason, I wanted to be sure that the CSV itself contains the correct representation of the top ScoreCard objects generated during runtime. The helper class provides a convenient means with which to evaluate the correctness of the CSV file, particularly in terms of leaderboard placement.

Another key consideration is the use of distinct leaderboard files saved within the game directory. The Leaderboard and ScoreCard classes that comprise the scoring system at large rely on the existence of a previously populated CSV file to function properly at runtime. There are references to three distinct leaderboard files within the game code, helper class, and unit tests.

Leaderboard Game File

- leaderboard.csv

Leaderboard Testing Files

- leaderboard_abovepar.csv
- leaderboard_belowpar.csv

At runtime within the actual game, only a single file is required (leaderboard.csv). This file is read in at the beginning of every round to populate the in-game leaderboard and is updated upon the completion of every round provided the player scorecard meets the top five requirements. However, when running unit tests on the scoring system the leaderboard must be reset to its original state after every unit test takes place. This is where the CheckCSV helper class comes in. At the beginning of each unit test, the resetLeaderboardCSV() method is called and the contents of the given leaderboard testing file are copied into the leaderboard.csv game file. This allows for a consistent starting point across unit tests.

It is also important to discuss the actual "inputs" for each unit test. I find that it is best to think of the inputs for these test cases as the leaderboard (based on one of the leaderboard testing files) and the scorecard objects that are generated for the player at the beginning of each unit test. The scorecard is comprised of the player's name, their score on each hole, the total score for the entire course, and the two certifications that may be attached to the scorecard (scratch and leaderboard).

Lastly, please note that the game assumes a course par value of 27 strokes for testing purposes. This figure is arrived at by multiplying total number of course holes (9x) by the par value of each hole (3 strokes). This par value is used as the basis for the creation of the two leaderboard testing files.

Benefits

The impact of unit test inclusion on the development process itself was most likely more helpful than the benefits derived from the completed test suites. When speaking to these indirect benefits of testing on the development process, I refer mostly to the fact that the need to include tests for the scoring system actively influenced my coding choices during implementation. An example of a resultant design decision is avoidance of any built-in scoring functionality offered by the LibGDX platform. I instead elected to write independent ScoreCard and Leaderboard classes that govern the logic surrounding the scoring system. This isolation ensures that the game itself does not need to be running to test the scoring system, and the result was a compartmentalized piece of game logic more suited to the testing methods discussed in class.

Upon review of the three testing methods used, I felt that they all were beneficial to a certain degree. My thoughts on each approach in the context of my game are as follows:

Decision Table-Based
- Solid basis of tests that ensure all possible combinations of table conditions and actions are covered.
- Strong overall code coverage despite a relatively small number of test cases (4x).

Weak Normal Equivalence Class
- Accounts for specific player scorecard placement on the leaderboard.
- Relatively easy to establish logical equivalence classes in the context of a top-five leaderboard.

Exploratory
- Tests all possible paths associated with the ScoreCard comparator method.
- Improves upon ScoreCard class code coverage relative to the other testing approaches.

The tests developed for this project were all based on specification-based (black box) techniques. This approach allowed for test creation to occur much earlier in the development process than other measures that could have been considered. On a more general level, running these tests has given me considerably greater faith in my implementation of the scoring system. Though I know it is impossible to test for all contingencies, I am pleased that I have been able to run actual game results through the scorecard system before my game even has a main menu, an in-game leaderboard, or a complete set of nine holes in place. If I had not considered testing early in the process and adjusted my approach to development accordingly this never could have occurred.

Drawbacks

Despite the many benefits of the three testing methods used, there are obviously downsides associated with each. My thoughts on each approach in the context of my game are as follows:

Decision Table-Based

- Considerable front-end effort is required to create the decision table itself.
- Only concerned with certifications and whether scorecard is written to CSV (no placement considered).

Weak Normal Equivalence Class

- No direct improvement in code coverage relative to decision table-based testing.
- Required comparatively more tests to achieve the same code coverage as decision table-based testing.

Exploratory

- Difficult to identify a testing direction without prior tests for reference.
- Strongly contingent on developer competence and understanding of the underlying system.

In addition to the specific drawbacks associated with each testing method, there are also general flaws or difficulties inherent to the testing process overall. Several of these difficulties have already been pointed out in the "Process" section of this writeup and are related to the real-time nature of video games. Beyond those considerations, the inclusion of a helper class is arguably another segment of code that could be prone to errors. I was also somewhat limited in what test case evaluation metrics could be used, as test inputs were comparatively complicated relative to the triangle problem. For example, I was not able to successfully obtain any score via PIT Mutation Testing as it proved unable to generate mutants for my code.

I am quite confident that I will run into certain scenarios that could break my scoring system (an example that comes to mind is the player providing a name that contains special characters). Thankfully, this gives me the incentive to implement some form of input verification to prevent problematic data from being written to the scorecard in the first place. With that said, this is only one of the more foreseeable scenarios.

Deliverables

Leaderboard Testing Files

- leaderboard_abovepar.csv

Name	Hole 1	Hole 2	Hole 3	Hole 4	Hole 5	Hole 6	Hole 7	Hole 8	Hole 9	Total	Scratch	Leaderboard
Andrew	4	3	3	3	3	3	3	3	3	28	FALSE	TRUE
Bailey	4	4	4	3	3	3	3	3	3	30	FALSE	TRUE
Cody	4	4	4	4	4	3	3	3	3	32	FALSE	TRUE
Daisy	4	4	4	4	4	4	4	3	3	34	FALSE	TRUE
Evan	4	4	4	4	4	4	4	4	4	36	FALSE	TRUE

- leaderboard_belowpar.csv

Name	Hole 1	Hole 2	Hole 3	Hole 4	Hole 5	Hole 6	Hole 7	Hole 8	Hole 9	Total	Scratch	Leaderboard
Andrew	2	2	2	2	2	2	2	2	2	18	TRUE	TRUE
Bailey	2	2	2	2	2	2	2	3	3	20	TRUE	TRUE
Cody	2	2	2	2	2	3	3	3	3	22	TRUE	TRUE
Daisy	2	2	2	3	3	3	3	3	3	24	TRUE	TRUE
Evan	2	3	3	3	3	3	3	3	3	26	TRUE	TRUE

Note: *ScoreCard coverage is artificially low due to the presence of game-specific methods addStroke() and getStrokeCount().*

Unit Tests (Decision Table-Based)

A total of four unit tests were developed based on the rules of the Scorecard Behavior decision table developed during the requirements specification portion of the project.

See [TestDecisionTable.java](#) for full test suite.

JUnit Coverage

TestDecisionTable (Apr 15, 2021 10:11:53 PM)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
DiscGolf	40.7 %	907	1,322	2,229
src	40.7 %	907	1,322	2,229
(default package)	40.7 %	907	1,322	2,229
TestDuplicateScore.java	0.0 %	0	793	793
TestUniqueScore.java	0.0 %	0	477	477
ScoreCard.java	76.4 %	120	37	157
CheckCSV.java	93.5 %	129	9	138
Leaderboard.java	98.2 %	333	6	339
TestDecisionTable.java	100.0 %	325	0	325

ScoreCard Class: 76.4%

```
81      static class ScoreCardComparator implements Comparator<ScoreCard> {
82          @Override
83          public int compare(ScoreCard o1, ScoreCard o2) {
84              if (o1.getTotalScore() == o2.getTotalScore()) {
85                  return o1.getPlayerName().compareTo(o2.getPlayerName());
86              }
87              return o1.getTotalScore() - o2.getTotalScore();
88          }
89      }
```

```
25      public void addStroke(int currentHole) {
26          playerScores.put(currentHole, playerScores.get(currentHole) + 1);
27          this.totalScore += 1;
28      }
29
30      public int getStrokeCount(int currentHole) {
31          return playerScores.get(currentHole);
32      }
```

Leaderboard Class: 98.2%

```
38          } catch (FileNotFoundException e) {
39              e.printStackTrace();
40          }
```

Unit Tests (Weak Normal Equivalence Class)

A total of six unit tests were developed that attempt to break out equivalence classes denoting the possible ordering of player leaderboard finishes (FIRST, SECOND, THIRD, FOURTH, FIFTH, NOT_TOP_FIVE).

See [TestUniqueScore.java](#) for full test suite.

TestUniqueScore (Apr 15, 2021 10:17:58 PM)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
DiscGolf	47.5 %	1,059	1,170	2,229
src	47.5 %	1,059	1,170	2,229
(default package)	47.5 %	1,059	1,170	2,229
TestDuplicateScore.java	0.0 %	0	793	793
TestDecisionTable.java	0.0 %	0	325	325
ScoreCard.java	76.4 %	120	37	157
CheckCSV.java	93.5 %	129	9	138
Leaderboard.java	98.2 %	333	6	339
TestUniqueScore.java	100.0 %	477	0	477

ScoreCard Class: 76.4%

```
81      static class ScoreCardComparator implements Comparator<ScoreCard> {
82          @Override
83          public int compare(ScoreCard o1, ScoreCard o2) {
84              if (o1.getTotalScore() == o2.getTotalScore()) {
85                  return o1.getPlayerName().compareTo(o2.getPlayerName());
86              }
87              return o1.getTotalScore() - o2.getTotalScore();
88          }
89      }
```

```
25      public void addStroke(int currentHole) {
26          playerScores.put(currentHole, playerScores.get(currentHole) + 1);
27          this.totalScore += 1;
28      }
29
30      public int getStrokeCount(int currentHole) {
31          return playerScores.get(currentHole);
32      }
```

Leaderboard Class: 98.2%

```
38          } catch (FileNotFoundException e) {
39              e.printStackTrace();
40          }
```

Unit Tests (Exploratory)

A total of ten unit tests were developed that attempt to isolate the ScoreCard comparator behavior and ensure that alphabetization is applied appropriately in the case of duplicate scores.

See [TestDuplicateScore.java](#) for full test suite.

TestDuplicateScore (Apr 15, 2021 10:36:01 PM)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
DiscGolf	62.0 %	1,381	848	2,229
src	62.0 %	1,381	848	2,229
(default package)	62.0 %	1,381	848	2,229
TestUniqueScore.java	0.0 %	0	477	477
TestDecisionTable.java	0.0 %	0	325	325
ScoreCard.java	80.3 %	126	31	157
CheckCSV.java	93.5 %	129	9	138
Leaderboard.java	98.2 %	333	6	339
TestDuplicateScore.java	100.0 %	793	0	793

ScoreCard Class: 80.3%

```
81    static class ScoreCardComparator implements Comparator<ScoreCard> {
82        @Override
83        public int compare(ScoreCard o1, ScoreCard o2) {
84            if (o1.getTotalScore() == o2.getTotalScore()) {
85                return o1.getPlayerName().compareTo(o2.getPlayerName());
86            }
87            return o1.getTotalScore() - o2.getTotalScore();
88        }
89    }
```

```
25    public void addStroke(int currentHole) {
26        playerScores.put(currentHole, playerScores.get(currentHole) + 1);
27        this.totalScore += 1;
28    }
29
30    public int getStrokeCount(int currentHole) {
31        return playerScores.get(currentHole);
32    }
```

Leaderboard Class: 98.2%

```
38        } catch (FileNotFoundException e) {
39            e.printStackTrace();
40        }
```

Conclusion

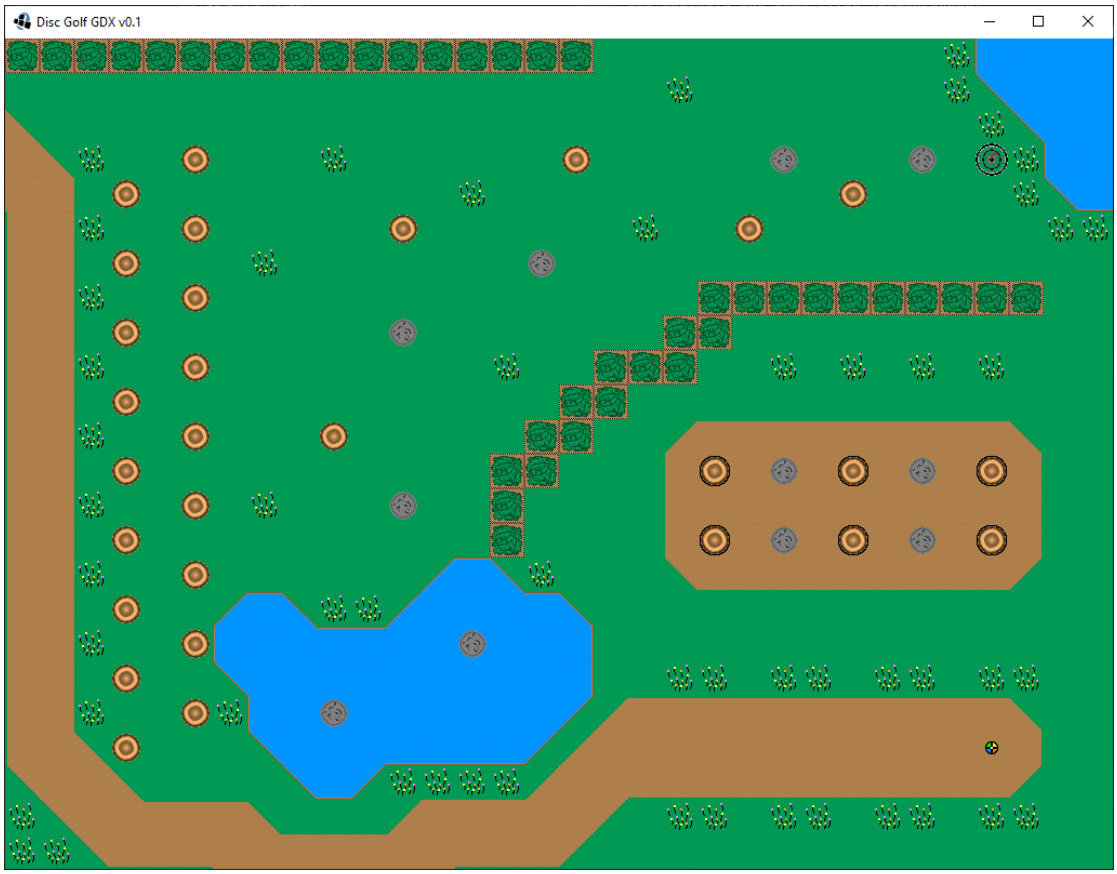
In similar fashion to requirements specification methods, I believe that the inclusion of tests was a net positive to the development of my game. While I initially intended for the decision table to serve as the basis for my entire testing approach, discovering the limitations associated with using that method exclusively helped me come up with other test cases that would better serve my testing efforts. It is clear that I was not necessarily considering the ordering of scorecards on the leaderboard when developing the decision table, and the use of other test case generation methods that would more adequately evaluate this additional behavior was important to the overall functioning of my scoring system.

My experiences with testing this game also reiterated a concept discussed extensively in class, that code coverage does not necessarily tell the entire story. At a surface level, code coverage is identical when comparing the decision table-based and equivalence class testing approaches, but the equivalence class testing approach still adds value in the form of leaderboard placement consideration. There are certainly other metrics that could evaluate the effectiveness of my test cases aside from JUnit code coverage, and they would likely reveal additional flaws in my approach. Regardless, I am still very pleased overall with the value added by my test suites. They helped give me a better feeling about the underlying functionality of the scoring system implemented within my game, and that is valuable even if the test cases are not perfect.

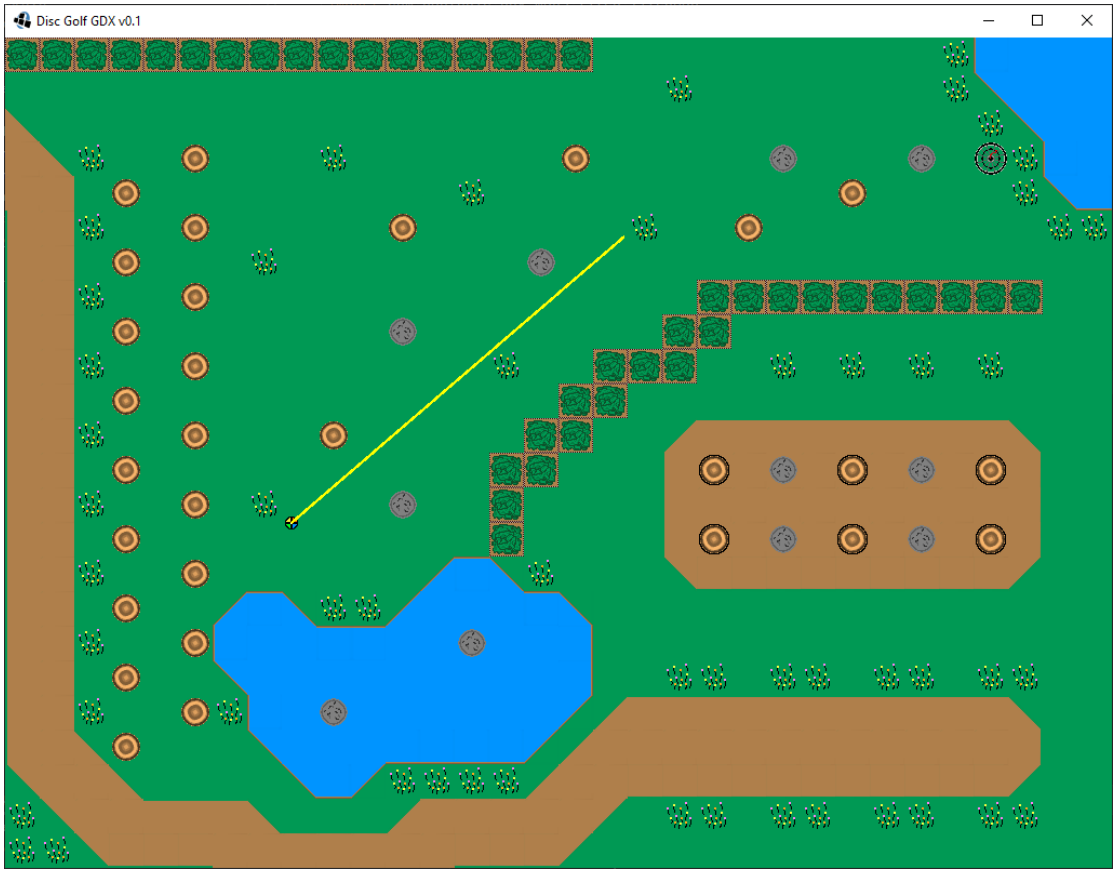
Gallery – State of Project

Gameplay Video Link: https://youtu.be/03RWa5oZcPQ

Game Window – Disc on Tee



Game Window – Aim Guide



Game Window – Disc in Basket