Nim Game Application

Jordan Barron

University of West Georgia

July 22, 2020

Nim Game Application

This project is a computer application that when run, serves as a simulation of the game Nim. As an overview, Nim is a strategy game believed to have originated in China, albeit this is inconclusive of evidence so technically the origin is unknown. The game has been referenced as far back as the 16th century, around the same time as the popular game "Rock, Paper, Scissors", so it is possible that Nim was the first game ever recorded in history.

In terms of play for this application, one human player is required. The human player will play against the computer player by taking turns. Items on the field of play (board) only include a pile of sticks. On each turn, the player must remove (nim) between 1 and 3 sticks from the pile. Whichever players is left with one stick to remove from the pile is declared the loser. In another sense, the goal is to not be the player left with the last stick. This is known as Misère play. Please note there are many alternate ways to play, but Misère is the one designed for this application.

In addition, there are several options included to enhance the human player's experience with the game and increase replayability, the first of which is the strategy settings. This feature allows the user to adjust the strategy the computer player takes to try and win the game. Because Nim is a strategy game, it has a mathematical element, so playing against different strategies adjusts the way the user must play to win. Next is the flexible pile option, which allows the user to adjust the initial size of the pile before playing. This helps for a shorter or longer game. Following is the coin flip option, which randomly selects which player goes first. Going first is an important consideration and can drastically alter the winner. Finally, the last options are the automated computer player and quick think timer. The automated computer is used so that the human player does not have to manually make the computer take its turn. The quick think timer is the time limit the human player is given to nim sticks from the pile. This is used to increase the

pace of place and allow the human player to make quick decisions and not have an unlimited time to figure out or come up with a strategy.

In the future, there may be more options included to the application. Standard play is an option that can be include that will make the player with the last stick in the pile the winner. Another is multi-pile where the field of play include multiple piles from which to choose from. All these features can provide a more immersive experience as development continues.

In terms of the actual programming and development of the application, current programming methodologies and practices are used to create sustainable and maintainable code. The first concept to highlight is the Design by Contract methodology created by Bertrand Meyer. This concept was important to implement because every supplier method should be responsible for checking its own preconditions. This ensures that any client methods that call the supplier methods are free from that obligation and guaranteed a correct result. Also, any errors that occur can be pinpointed to the exact method.

One good example of use of Design by Contract is in the construction of the Pile class. The Pile class constructor takes in a parameter that will set the number of sticks in the pile for the game. The Pile class also includes a method to remove sticks and takes in a parameter of how many sticks to take from the pile. Given that the pile cannot be negative, the Pile class constructor was set with the precondition that the parameter passed must be an integer greater than 0 and the remove sticks method was set with the precondition that the parameter passed must be an integer greater than 0 and less that the number of sticks in the pile . In the event the precondition was not met an error would be thrown.

By setting up the supplier methods this way, it provides the benefit of freeing the methods from having to check for any integers outside of the precondition as well as fulfilling

it's obligation to any client methods that call it. For example, in the Game class there is a set pile client method that makes a call to the Pile class constructor. The client method would be guaranteed a pile greater than 0 because it is bound to the output of the supplier method. If an error were to occur during this process in some way, it would be from the supplier method. This is the importance of Design by Contract. The supplier method has an obligation to the client method that any preconditions in the supplier method work correctly. Without this set up, if there were an error it could be from the supplier or the client.

The next concept to highlight is the use of the Model-View-Controller architectural design. The code of the application was separated into different folders called model, view, and controller respectively. This design pattern has become widely used on development teams because it provides rapid development. It was important to implement this in the application to ensure that the internal information and logic is separate from the information that the user can see and interact with. It also allows scalability because if this application ever needs more functionality or data then developers can be assigned clearly defined tasks. For example, one developer can oversee the model while another works on the view.

Within the application, a model folder was created to control all logic and data for the game. This folder is used to store the strategies of the game the computer player will use as well as how many sticks each one removes from the pile. It also stores the behaviors that define a player such as taking their turn and taking a certain number of sticks from the pile. Next is the pile which stores the number of sticks the pile is made of as well as the behavior of losing sticks. Following is the countdown which defines the data of the timer and the actual counting of the timer. Finally, is the game model that represents the start of a new game and controls the flow of the interaction between the players and the pile.

The view folder was created to store the presentation information that the user can physically see and interact with. Using JavaFX, the user interface is split into five parts. First is the menu which stores the code to display and interact with the menu. The user can use the menu to exit the application, change the computer strategy, get information about the game, etc. Second is the status section which stores the code to display and adjust the pile size. The user can interact with the pile before a game starts and adjust it based on the data provided from the model. During actual play of the game the user can see the pile size change. Third is the computer player and human player section. Both sides contain their name, number of sticks taken on their turn, a countdown before their turn is over, and listeners that when activated disable the user from interacting. The only difference is on the human player side there is a drop down that allows the user to select the sticks to take as well as the countdown is visible. Lastly is the Nim section. This section is a culmination of all the other section into one with the addition of a section that allows the user to select who should go first. This section is the one that provides that actual display that the user sees.

The controller folder was created to store the information that executes the application itself. This folder contains a main file that initializes the game from the model as well as the nim pane from the view. All these folders work in tandem and provide clear objectives individually to create the whole of the application.

Now, there is an additional folder in the application outside of the model, view, and controller. That folder is named test and leads into the next important concept called Unit Testing. It was important to implement unit testing because manual testing (Putting System.out.println() throughout certain parts of the application to make sure it is working) has it’s limitations, such as unnecessary code and inability to truly know if values/functions work

correctly. Unit testing on the other hand involves testing the fine grain logic of the smallest components/functions which significantly reduces the overall bugs, especially when dealing with very complex applications.

As far as the application, there are thirty test cases with an average of three test functions per case for a total of 90 +/- 5 tests. Each test case is specific to only one function of a class and there are only test cases for classes in the model's folder. For example, the countdown class contains a function that checks when the time is up (int <= 0). The countdown initially starts at 15 and each time a function is called (countTimeDown) the countdown variable decreases by one. Using manual testing, there would be no way to create a representation of an actual countdown unless a method was constructed and called the function 15 times. This would work but it causes unnecessary clutter in the model folder because the function will never be used in the application.

This is where the unit tests come in. Using a unit test, we can essentially set up a separate spot where simulations can be run. An additional benefit of unit testing, and in this case Junit testing, is the use of test case specific functions. The Junit library has a function called assertEquals that takes in two parameters. The actual output and the expected output. By using this, inferences can be made as to why the actual output is different that the expectation. It provides an ease of access that makes the process of testing more efficient.

Strategy Design Pattern is another important concept that was used in the build of the application. This concept is good to use because it allows for the use of different concrete implementations of an interface at runtime. For the application, this was used to create the selection and implementation of the computer players strategy of play during the game. This was important because with the strategies being implemented in that way it would not be possible for

the computer player to change state (strategies) during a game and would lead to a loss of dynamisms.

The strategy folder inside the model folder is where the Strategy Design Pattern takes place. Inside, there are four files, titled CautiousStrategy, GreedyStrategy, RandomStrategy, and NumberOfSticksStrategy. NumberOfSticksStrategy serve as the abstract interface that when called must implement the howManySticks function. The other strategies are concrete classes that each implement the howManySticks function differently. CautiousStrategy takes 1 from the pile, GreedyStrategy take the maximum number allowed on a turn from the pile (3 for this application), and RandomStrategy takes a random number between 1 and the maximum number allowed on a turn from the pile.

Afterwards, a variable of type NumberOfSticksStrategy is created inside the Computer Player class. Here shows the importance of Strategy Design. Without Strategy Design, multiple Computer Player classes would have to be created just to use a different strategy (class ComputerPlayerGreedyStrategy, class ComputerPlayerRandomStrategy). This is not inherently bad, but it is ineffective. In the future, if a new implementation involved adding power ups, more classes would have to be created. Because in theory the Computer Player should be a single object, this would not scale well. By using Strategy Design, it makes the interface a variable, allowing the implementation of methods inside of the target class that changes the interface between its different concrete class implementations, all while keeping the same single object.

While there are many other concepts used in the application, the final one to highlight is Behavior Driven Development (BDD). This one is more of refined approach to unit testing but is equally important for several reasons. The first is that this is a methodology that is part of the agile software development process, which is becoming widely more popular on business

development teams, and because this application uses the current business practices it's important to include it. Also, there can never be enough testing on an application, and different ways of testing are an additional benefit to finding bugs. In large businesses there are entire Quality Assurance teams dedicated solely to using different tests to find and exploit bugs.

In the application there is one JUnit test case that was developed using BDD, named ExpertStrategyGetHowManySticks. This test case was used to test the function of a new strategy; one more complex than all the other strategies. Because of the complexity, it would be ideal to use the BDD principles, which involves defining the unit test first, writing the test to fail, and iteratively implementing the function and rerunning the unit test so that it passes. The ExpertStrategy is designed to be the most difficult setting, and thus needs to consider many more mathematical outcomes during the game.

Using BDD, the test initially failed, the ExpertStrategy function was implemented with simple logic, the test was run again and passed. On the next iteration, additional logic was added to the function and tested so that the additional and previous logic passed. This process continued until there function logic was complex, but all tests passed. This is the important of BDD. Testing iteratively allows implementation to started small and built into something complex, while at the same time reducing bugs because testing was done and successful at every step. With regular unit testing, it can be difficult to find the exact line where the logic of a function goes wrong, especially for ones that are substantially more complex.