Technical Documentation for Nim Game

CS6910

Summer 2014

Christina Bill

**Application Description**

Nim is a two person mathematical strategy game that involves each player taking turns removing objects from a pile. There are multiple versions of the game and depending on the version you are playing, either the person who gets the last object wins or the person who is forced to take the last object loses (Nim, n.d).

The game’s origins are unknown but it closely resembles the Chinese game “Tsyan-shizi” and has 16th century European references. The current name “Nim” was created by Charles L. Bouton in 1901. Charles was a Harvard Mathematician who understood and explained the theory of the game. Nim type games exist all over the world and have a lot of diversity and variations in implementation. It was never fully explained where the current name “Nim” came from, however, it has been thought to have been taken from the German word “nimm” which means “take” (Nim, n.d).

This version of Nim is played as a misère game. This means the player who is forced to pick the last object loses. As the Player plays against the Computer, the Player has different options for the game as to how they want to play the game. The Player can choose the pile size to be 9, 16 or 21. Having 21 as the default pile size, if no other choice has not been made. The Player can also choose between picking 1, 2 or 3 objects from the pile. Also, the player can choose the strategy or difficulty of the game. The strategies are: “Cautious”, “Random”, “Greedy” or “Expert”.

The “Cautious” strategy sets the Computer to always choose one object from the pile. This is the safest strategy and since you always know what the Computer will do and it is mathematically easier figure out how to beat the Computer. The “Random” strategy is more difficult as the Computer will randomly choose the number of objects between one and three to be removed. The “Greedy” strategy sets the Computer to always choose to remove three objects from the pile. This is mathematically easier to beat the Computer with this strategy rather than the “Random” strategy because you always know the Computer will choose 3. However, it’s slightly more difficult than the “CautiousStrategy” simply from a mathematical standpoint. And the “Expert” strategy uses an algorithm to best choose the number of objects between one and three for the Computer to pick that will give the Computer the best chance in winning. This is the hardest way to play the game.

As the game starts, the user is prompted with the directions on the game and if they would like to see the directions again upon game reset. At any time during the game, the player can reset the game, re-read the directions and read the “About” page that gives the author and version. The player also has the choice of which player moves first or to let the first player be chosen at random.

The actual application is broken down into four panes. The top pane shows start of game choices such as the pile size and who starts first. The left pane shows the Player, which gives the player the choice of how many objects to remove from pile (one, two or three). The right pane shows the Computer, which shows how many objects the computer took from the pile. And the bottom pane shows the game information, if it’s the start of a new game, how many sticks are left while game is in play or who won the game.

**Concepts Learned and Implementation**

**JUnit Testing**

JUnit testing is important as it tests the method or portion of a program’s behavior. By verifying the output of each JUnit test, the programmer can determine if the methods themselves work. All tests must pass, however, if the programmer finds that the tests do not pass, it is much easier to narrow down and fix the method than it is would be to run the program and try to trouble shoot the entire program for the correctness of one method or section of a program. After determining that the methods correctly, any other issues that may arise after testing, can be determined that the problem is not within the method itself.

Some basic ideas of JUnit testing that are important are that the tests should be small and simple to run, and tests should be independent of each other and never rely on other tests. A big advantage of these tests is that they are automated and the programmer does not have to manually check the output to see if the method works correctly. A disadvantage is that it does not prove that the code itself is correct on an integration level, such as a problem between two classes (Unit Testing, n.d).

Nim makes use of JUnit testing for the methods that return a value. The classes that implement JUnit tests are: “CautiousStrategy”, “GreedyStrategy”, “ExpertStrategy”, “Game”, “ComputerPlayer”, “HumanPlayer” and “Pile”.

Before the final implementation the programmer should make sure that the tests are re-run to check that nothing has inadvertently changed. There is also the possibly a portion of the code was changed intentionally, in that case the tests should reflect the new changes.

**Separation of Concerns**

Separation of Concerns is a design principle that separates a program into different sections, each section containing a separate “concern”. A “concern” is a set of ideas/information/data that in the specific code (“Separation of Concerns”, n.d). In the Nim project, “separation of concerns” is used throughout the program.

On a package level, we have different packages that hold similar classes. For example the View package holds all the classes that are related to the Gui or output the user sees on the screen. On a class level, each class is separated by the portion of the game it represents.

In the View package, each class represents its own “view”. A few examples from Nim are: “ComputerPlayerPanel”, “HumanPlayerPanel” and “NewGamePanel”. The “ComputerPlayerPlanel” represents the left panel of the Gui output that shows the information regarding how many sticks the computer has taken. The “HumanPlayerPanel” represents the right panel of the Gui output that shows the information regarding how many sticks are available to take. The “NewGamePanel” represents the top panel that shows the choices the Player can make at the start of the game.

Separation of Concerns makes development and maintenance easier by allowing individual sections to be developed or updated separately as well as being able to manage the code in more efficiently. Also from the standpoint of looking at someone else’s existing code, it makes it easier to understand if the code is broken out by separate responsibilities.

**Model-View-Controller Pattern**

The model-view-controller pattern is design pattern that divides a program into different parts (Model, View, and Controller) (“MVC”, n.d). . It is a more specific pattern for showing “Separation of Concerns”. The Controller sends commands to the Model. The Model notifies and allows the View to provide the output.

In more detail the Controller is the starting point of the application and translates the user’s interactions for the Model and View. In the Nim project, the Controller has one class “ApplicationController” that starts the program.

The Model represents the data and how it is accessed and manipulated. In the Nim project, the Model classes include: “AbstractPlayer”, “ComputerPlayer”, “HumanPlayer”, “Game”, “Pile” and “Player”. These manipulate the data of who the current player is, how many objects are taken from the pile and overall game play.

The View decides how to visually represent the model. The View listens and updates the output as the data from the Model as it changes. In the Nim project, the View classes include: “ComputerPlayerPanel”, “HumanPlayerPanel”, “GameStatusPanel”, “NewGamePanel”, “NimContentPanel”, “NimHelpDialog”, “NimMenuBar”, and “Gui. The “Gui” class pulls the other classes into one gui game representation of panels and menus.

The Model-View-Controller Pattern is vastly common and has several advantages such as it makes maintenance easier to maintain and you can have multiple views for one set of data or Model.

**Strategy Design**

Strategy Design is a design pattern that can be used with a program has multiple algorithms to a specific task and the user, in this case the Player, chooses which algorithm, or Strategy, to be used (“Strategy Design”, n.d).

In the Nim project, there are four Strategies: “Cautious”, “Random”, “Greedy” and “Expert” in the “Strategies” package. Each strategy determines how many objects the Computer will pick from the pile. They are broken down into four classes: “CautiousStrategy”, “GreedyStrategy”, “RandomStrategy”, and “ExpertStrategy”. The Player makes the choice at the beginning of the game as to which strategy they want to use. Whatever choice of strategy the Player makes, that choice gets passed to the “ComputerPlayer” class “setStrategy()” method. This method sets the strategy for the game. If no strategy has been chosen, then the “Cautious” strategy is the default.

Each of these strategies uses a different algorithm for how many objects the Computer will choose. The “Cautious” strategy sets the Computer to always choose one object from the pile. The “Greedy” strategy sets the Computer to always choose to remove three objects. The “Random” strategy is more difficult as the Computer will randomly choose the number of objects between one and three to be removed. And the “Expert” strategy uses an algorithm to best choose the number of objects between one and three for the Computer to pick that will give the Computer the best chance in winning.

The Strategy Design is useful in the fact that you can have multiple strategies that are interchangeable.

**Factory Method Pattern**

Factory Method Pattern allows you to create multiple objects (concrete classes) based on a superclass or interface. The concrete class extends a superclass and accepts one or more parameters to create the object. This pattern is beneficial because creating objects can be complicated, and recreating similar objects can lead to code duplication. By having an abstract class that initially creates the object, the programmer can define separate objects that override specifics of the abstract object to the detail of the new object. Then the new object has been created with less code. If a programmer has multiple objects of the same kind, it’s beneficial to use this approach. According to Wikipedia, “The factory method pattern relies on inheritance, as object creation is delegated to subclasses that implement the factory method to create objects” (“Factory Method Pattern”, n.d).

In the Nim project, there is the interface “Player” that defines the interfaces for the players. There is the abstract class “AbstractPlayer” that implements the “Player” class. Since the interface has empty method bodies, the abstract class fills in the method bodies to create the overall Player object. A class that implements an interface must implement all methods or an abstract class should be created.

The concrete classes that extend the abstract class are “ComputerPlayer” and “HumanPlayer”. These classes take the abstract class and create the new player objects based on the specific requirements of each player. For these two classes, the names differ and possibly the number of objects taken from the pile.

**References**

MVC. (n.d). In Wikipedia. Retrieved July 21, 2014, from *http://en.wikipedia.org/wiki/Model-View-Controller*

Separation of Concerns. (n.d). In Wikipedia. Retrieved July 21, 2014, from *http://en.wikipedia.org/wiki/Seperation\_of\_Concerns*

Factory Method Pattern. (n.d). In Wikipedia. Retrieved July 21, 2014, from *http://en.wikipedia.org/wiki/Factory\_Method\_Pattern*

Strategy Design. (n.d). In Wikipedia. Retrieved July 21, 2014, from *http://en.wikipedia.org/wiki/Strategy\_Pattern*

JUnit Testing. (n.d). In Wikipedia. Retrieved July 21, 2014, from *http://en.wikipedia.org/wiki/Unit\_Testing*

JUnit Testing. (n.d). In Wikipedia. Retrieved July 21, 2014, from *http://en.wikipedia.org/wiki/Nim*