Daniel Picard

Symbolic Programming

3/19/2014

Clojure Hearts

I wanted to recreate the game of hearts that can be played by friends for my project. The goal of the project was to make a game using functional concepts from Clojure while mixing in minor amount of state to create a fully functional game. This should allow for the easy creation of a game by defining the behaviors rather than focusing on the objects and state which should allow for much more rapid development. The language Clojure was chosen as it shares many similarities with Lisp, functional programming with limited object orientation, and runs on the java platform which means it is easily accessible and cross platform. Clojure was also chosen because of its ability to inherently handle concurrency which made utilizing several features of the state very easy to accomplish and reason about. This will also leave open for the possibilities of expansion into the future for the clojure code to be used in a web game interface.

The game is very easy to use and will provide on screen instructions. However the startup procedure is slightly convoluted. To create the game the user will need to enter the command (create-game). This will load the game’s resources and set the necessary atoms for the game to function. Once this is complete the game will output instructions on how to continue as shown below.

To look at your hand type (show-cards)\n  
To start the game type (start-game)\n  
To place a card type (play-card n)\n  
- n is the number from 0 of your card\n  
To see this list again type (show-options)\n  
To exit type (exit)\n\n"

The function (show-cards) will display the cards for the current player. This is accomplished by the use of an atom to dictate who the current user is which is then run against the player hands storage. It will cycle through all of the cards and display them in a nicer fashion such as “Queen of Hearts”, or “2 of Clubs”. This was capable by having a simple function that would take the value of the card and replace any value of 11 to 14 with the appropriate face card.

The function (start-game) will do three main tasks. The first is that it will grab the first player; this is done by finding the player that has the “2 of clubs” in their hand and setting the appropriate atom as the current user. It will then set the play count atom to 0 as no cards have yet to be played. The last thing it will do is run the play command which will output a message to the user.

The (exit) command will stop the game. This is accomplished by calling System.exit(); from java. Though effective as a command line tool, it will occasionally cause issues with IDE’s.

The (show-options) command will display the same option window as displayed above. This is so the user can always refer to the options again at a later point.

Lastly, the (play-card n) command will take the selected card from the deck and play it in the current trick. For this command the “n” will need to be replaced by the number of the card from the deck. If the user wanted to play the “3 of hearts” which is the 4th card in the deck they would type (play-card 3). This assumes that the user counts from 0, not from 1. The card that is played will first be verified to make sure that the card is valid for the trick. This is accomplished by ensuring that the first card played is the “2 of clubs” and that hearts are not played first. It will also verify that hearts have been broken before a heart can start a round. If any of these conditions are not met it will display a simple message informing the user to try again. If the card passes validation then it will be placed into the trick and the trick will be displayed. After all players have placed a card in the trick a winner will be displayed and the users will be asked if they would like to play the next trick.

The program has many helper functions that were not initially accessible within clojure itself. The two main functions that I created were drop-nth and index-of. The index-of function will return the index of the value within the given collection. This function was useful as it would look up the card (a map) from within the hand of the player (a vector). The function drop-nth was incredibly useful for my program. The functions within clojure are designed to remove or drop items in the collection from the front of the deck. This function will rotate the collection and remove the first element then return a map of the remaining elements.

In my program I chose to use a concept of atoms. Atoms in clojure are a way to manage independent, synchronous, shared state. I used this as a way to manage several items in my program as I could run a transaction and “reset” the atom at a later time. Each one of these transactions would occur in order they are receive by the atom so even if the program was executed concurrently they would all be managed correctly with very little setup. These atoms also allow the ability to find values within them through the use of keywords. This made their use very easy and effective for such a sort program and would be a great addition to any language I use in the future.

This project has taught me a great deal about a completely different style of programming: both functional and symbolic programming. These different styles have enabled me to create a functional program in a much shorter time than expected that is very easy to reason about its features. These features are something that I have come to love as I can write a function and know that its results will always be the same given the same input. Some functions that are written in my program do not follow this paradigm, as state was needed. Instead I incorporated the state into the functions so that I will always be given the correct value for the current state. The state transitions were very easy to use and made designing the program much simpler than expected.

A few things that would give me pause were the extensive use of lists in clojure and how to effectively utilize them for my program. Though clojure has many different list types (vectors, maps, sequences, ect..) they don’t all use the same function names for the same process. The process to discover how Clojure defines each and how each can be used and modified took a great deal of my programming time.

In this entire project was a great learning experience for both functional and symbolic programming. It required me to put aside my concepts of object oriented code and combine them with a more functional style that I feel will make me a much better programmer overall. Though I could still use many more projects to firm up the ideas I feel that this was a fantastic starting point. The project also forced me to learn another programming language which I feel will be very useful in the future that has many unique features that I feel my regular programming languages are missing.