Group Q: CSC495 (Programming Languages) Project Final Begun and Edited 4/4/2018 - 5/4/2018

Raymond Woods
rjwoods3@ncsu.edu

Wenting Zheng
wzheng8@ncsu.edu

Garrett Watts
gcwatts@ncsu.edu

1. INTRODUCTION

The purpose of this project was to develop and program a set of card games using Python in multiple iterations as dictated by the course requirements. Our team chose to implement Bartok (with a few added rules for added complexity) and King's Corner (also known as Kings in the Corner). For the first iteration of the project, we set out to code both games separately in a "quick and dirty" manner with the single goal of making games that run and play properly. Although we accomplished this iteration's goals satisfactorily, our implementation was incredibly naive, which allowed us to move into the next iteration.

The successive iteration involved taking our code for the two games and heavily abstracting them into "essence and details" so that a lot of the code that is similar between the two games are implemented as such. In chronological order, we first separated all the code into multiple class files that can be used by two games. Any code that would be different between the two games were implemented as child classes of the relevant classes. Next we utilized feature extraction to pull certain pieces of code out and turn them into human-readable functions, allowing for easier utilization of code and future programmers to use the code, approaching the idea of Domain Specific Language. Next, a large change to our games was to turn the game while loops (which dictated the flow on structure of a game instance) into a Finite State Machine. The Machine class implemented state maintenance and transitions in a way that each game could provide its own set of states and work the same way. Finally, we added a (rather simple and dumb) Artificial Intelligence by adding code to the Player class that chooses plays in the game on its own, allowing for someone to play alone against a computer player rather than only having the option of two to four players, which is how it was originally.

The following sections will demonstrate our findings over the course of this second iteration in terms of "essence and details." We will describe what pieces of code were the same or similar enough between the two games and could be utilized by both (the essence), as well as the differences between the two games that required extra implementation through child classes or other forms of externally supplied information (the details). Each section will pertain to a particular implementation category as was described above (class separation, feature extraction, finite state machine, and artificial intelligence), and will be done chronologically. This will lead to some relevant aspects not being relevant at a certain point or being changed in later sections; for example, during the discussion on class separation, the finite state machine had yet to be implemented, so certain aspects of the classes will be changed by that point. Naturally these changes will be discussed at the point at which they would become relevant.

2. CLASS SEPARATION

2.1 Main Classes

As mentioned before, in order to better abstract our games, we decided to refactor our project from two single files for each game into a network of class files dedicated to each object utilized by the games. This list of classes were the following:

1. Game: which would start the card game chosen by the user, either Bartok or Kings Corner
2. Board: which represents the playing field with which players would interact using cards
3. Card: which represents a single card in the game
4. Field: which represents a single "playing field" found on a board
5. Player: which represents a player involved with a game (either human or AI, though at this point we had not implemented AI)
6. Rules: which contains many "rules" related to a card game that determine how and if a game can be conducted properly

In addition, functions that were previously written to provide functionality were moved to classes where it made more sense for them to be. This mostly moved them to the new Rules class, but also included player and board.

2.2 Multiplicative Classes

From these six main files, we established each essence that would be utilized by both card games; however, we also determined that certain files would have to be different for each game in certain ways, bringing us to the difference in details. We determined that in both games and in all aspects, the Card, Field, and Player classes would have no differences. A Card will not be or act different between two games (cards can have different meanings, but these distinctions are made in Rules) and contain the same values every time, that of suit, color, rank, etc. A field will contain stacks of cards, and for our two games, the only aspects that matter are the cards on top and on bottom, so the class is very simple itself. A Player keeps track of its name, number, and hand, while also providing functionality for drawing cards and creating a new hand, all of which are relevant to both games (though after the inclusion of game AI, some subclasses will be created for Player, and this will be discussed later).

2.3 Inherited Classes

The remaining files were determined to contain significant enough differences in the details of implementation to warrant the creation of child classes. Board was abstracted into BartokBoard and KingsCornerBoard, Rules was abstracted into BartokRules and KingsCornerRules, and, although they were not implemented as literal child classes, Bartok and KingsCorner are separate

classes that are chosen and run based on the distinction made in the Game class. In these cases, functions or initializations that differ between the subclasses are implemented as abstract in the parent class (simply written with a single line of 'pass') and are deliberated in the child classes; however, any functions that would perform the same in both games are simply written in the parent class and ignored in the child class.

2.3.1 Board

The only real similarity in the game board between the two games is that they contain a deck from which the players draw. Otherwise, the game boards are structured differently and contain a significant different in the number of fields required. Bartok only needs two fields, being the deck and the field of play, while Kings Corner requires nine fields, including the deck, the four North South East and West directional fields, as well as the four corners C1, C2, C3, and C4. Because of this, the only thing we could really combine in the parent class is that we initialize the deck; therefore, the child classes KingsCornerBoard must handle the initialization of the deck and the creation of the empty eight other fields, and BartokBoard handles the deck and the field of play. There very well might be some sort of paradigm that would allow for the board creation to be the same for both games and only differ in terms of parameters passed during class creation, but an issue arises later in terms of representing the board in the form of ASCII. Somehow there would have to be determination of which game you are playing, so it made more sense for us to simply have subclasses for each game.

2.3.2 Rules

Rules is a much bigger piece of pie than Board due to the number of functions that need to be considered. The following is a list of the functions that we determined could be defined in the parent class Rules because they apply to both games:

1. isWinnerByNoHand: returns the name of the given player if they have no cards in their hand
2. checkHandSize: checks that the player isn't trying to play a card outside of the bounds of his hand
3. printPlayerHandNumbers: prints the number of cards in each players hands
4. printPlayerHand: prints a list of cards in the given player's hand
5. printBoard: prints the instance of a player's turn, including the board, player hand count, and the list of cards in the player's hand
6. shuffle: shuffles the given deck
7. reshuffle: shuffles the cards from the discard pile back into the deck
8. createDeck: creates a deck by creating all 52 cards then shuffling. Returns the deck.

However, we were unable to define some functions in the main Rules class because they work differently in each game:

1. play: plays a card from the player's hand to the board
2. tryToPlay: attempts to play a card, if possible
3. printHelp: prints the help text that explains commands
4. determineIfPlayerHasWon: decides if the player has won the specific game
5. printBoardSpace: prints the ASCII representation of the game board
6. move: moves a stack from one field on top of a stack in another field (note, this is only actually used by King's Corner)

We also eventually ended up with many smaller helper functions, but those will be discussed in the later section about feature extraction.

There are certainly similarities among the two games where we can reuse functions. Error checking, such as checking a player is playing a card they actually have in their hand, printing the game status (the player's hand and other player's card counts, but not the board space), shuffling, drawing cards, and creating the deck are all shared among the games and utilize the same code; however, there are other similarities that can not easily be generalized for both games. Both games involve playing a card to the board in a certain way, but because the conditions and semantics of each game's "play" being so different, there isn't much code we can share between the two and end up having to separate them anyway. Further, King's Corner contains an entire aspect of play that is foreign to Bartok (that of moving stacks from a field to another), so this is a lot of logic that must be implemented separately because Bartok has no use for it.

3. FEATURE EXTRACTION

Feature Extraction describes taking a large set of code and reducing it into a set of features that are descriptive and understandable by humans, intended to make it easier for use in the future. We kept this in mind when we undertook refactoring our code by creating many smaller, descriptive functions that perform the same functionality as the code before. In some cases, we found some features that could be reused multiple times, which is a usual goal behind feature extraction.

3.1 Possibility and Action

A main paradigm we thought about in feature extraction is possibility and action. Possibility describes features that determine if a certain action is possible, and action describes the actual execution of a certain action. By defining these two parts for an action, it becomes simple for future use: first use the function to determine if you can do *something*, then use the function that actually does *something*. Often, they can even have multiple levels of hierarchy in function call. Let's take King's Corner as an example, specifically attempting to move a non-king stack from one field to another. The relevant code at the top layer of the "move" function looks like this:

```
elif self.isValidCardMove(sourceField, destinationField, board):
    self.moveStack(sourceField, destinationField, board)
```

To the reader, it's easily understandable how to do this. You check if, given the two fields and the board, this is a valid card move. If this is true, you can move a stack from the source field to the destination field. The moveStack function is simple, but the isValidCardMove function became a tree of feature-extracted, human-readable conditions; it returns true if "destinationFieldIsEmpty", "cardColorsAlternate", and "cardValuesDecrementByOne." These features are more underantable than a series of code that could fill the function. By separating into three readable features, it is easily understandable

by fellow coders and future utilizers in the case of developing a domain specific language.

3.2 Consolidation of Details

The act of feature extraction supports the separation of the details from the essence by more clearly defining the implementation of details and consolidating the high level interpretations of these details. We can analyze the previously mentioned move function in King's Corner. Below is the code for the move function before feature extraction, followed by the code after feature extraction.

```
def move(self, src_field, des_field):

    #check if src_field is empty
    if self.boardDic[src_field].bot.val == 0:
        print('\nError: Source field is empty\n')
        return 0
    else:
        # move Ks to any corner(empty)
        if (des_field == 'c1' or des_field == 'c2' or des_field == 'c3' or des_field == 'c4') and self.boardDic[src_field].bot.val == 13 and self.boardDic[des_field].top.val == 0:
            self.boardDic[des_field].top = self.boardDic[src_field].top
            self.boardDic[des_field].bot = self.boardDic[src_field].bot
            self.boardDic[src_field].top = self.boardDic[src_field].bot = Card(0, 'empty', 'empty')
        elif self.boardDic[des_field].top != 0 and self.boardDic[src_field].bot.color != self.boardDic[des_field].top.color and self.boardDic[src_field].bot.val == self.boardDic[des_field].top.val
            self.boardDic[des_field].top = self.boardDic[src_field].top
            self.boardDic[src_field].top = self.boardDic[src_field].bot = Card(0, 'empty', 'empty')
        else:
            print('\nError: Specifed move is impossible, any card placed on top of another must be of opposite color and one lower in rank\n')
            return 0

    return 1
```

```
def move(self, sourceField, destinationField, board):
    # check if src_field is empty
    if self.sourceFieldIsEmpty(sourceField, board) == 0:
        return 0
    else:
        # move Ks to any corner(empty)
        if self.isValidKingMove(sourceField, destinationField, board):
            self.moveKingStack(sourceField, destinationField, board)
        elif self.isValidCardMove(sourceField, destinationField, board):
            self.moveStack(sourceField, destinationField, board)
        else:
            print(
                '\nError: Specifed move is impossible, any card placed o
            return 0
    return 1
```

Due to feature extraction, the function has become shorter, more compact, more understandable and readable by humans, and much more reproducible. In the case of a different programmer implementing a game similar to King's Corner that involves moving cards in stacks (for example, Solitaire, whose rules are incredibly similar), the feature extracted code makes it infinitely easier to recreate and alter to fit the needs of the new game. This demonstrates the beauty of feature extraction; the move function can now be considered as "the essence", and the "details" have no been allocated to specifically named functions that can be altered if the game would need to be altered in terms of rules, but not in terms of what a "move" means at the core.

4. THE FINITE STATE MACHINE

The next step we took in our approach to abstracting the essence into details was creating a finite state machine to handle the motions of the game. Rather than relying on a loop in each game that would play until the loop is broken, we created a machine that would ingest a set of states then begin and finish the game by transitioning through these states until an end state was reached. The Machine class is created using all the parameters needed for the original game (rules, players, etc.) and then all the possible states for the game are created before running the machine. The State class is located within the Machine class and contains functions for execution of the current state as well as check, which determines if the player has won, indicating it is time to be in a final state. Because the machine runs through the various states in almost the same way internally provided that the states have been properly created and passed in, there was a lot of essence that was the same across both of our games. Therefore, most of the code related to the state machine could be inside the machine class and used for both games, and the only real difference outside of the machine is that each game created its set of states differently. Because the states are related to rules and the current state of the game, Kings Corner added states of EndTurn, Move, Play, and Help, while Bartok added Draw, Play, and Help. The states that are named similarly but have different details were determined by passing a separate function that had been defined in Rules, BartokRules, or KingsCornerRules in previous iterations. Further, each machine required a different kind of start state to properly function due to the nature of the differences in draw and turn sequencing in the games. These are defined as State subclasses inside Machine, called BartokStartState and KingStartState. Like mentioned, once these states are created and passed into the machine, the machine will run using the same code until the end successfully.

This internal state determination works pretty similar to the loops in previous iterations, but is easily abstracted for both games utilizing the state implementations. For each game, the following occurs:

1. Machine starts in the given start state
2. It starts to loop until the user ends their turn
    a. In Bartok, this will be after any single valid play
    b. In Kings Corner this is when the user indicates the end of their turn
3. In any input is given from the user, it will go to that state if it is valid
4. turnend = self.currentState.execute(currentPlayer, self.board, move)
    a. This executes the rule for the current state and returns if the turn ends or not
5. winner = self.currentState.check(self.rules, currentPlayer)
    a. This determines if there is a winner and exits the game

This makes it easier to implement future games, as someone can reuse rules or states, and creating states requires only defining a rule and turning into a state. Perhaps it is still too complex for new users to attempt to use, but it is a major step towards the goal of encapsulation in the form of a Domain Specific Language.

5. THE ARTIFICIAL INTELLIGENCE

Previous iterations only supported games of two to four human players, and we wanted to add functionality for a player to play against a computer controlled Artificial Intelligence in the case of the player not having anyone else to play with. This can also aid in the programmers' goals of testing future iterations or game transcript generation, which can effectively (given specific goals) cut testing time in half. Before discussing the details of the AI, it is worth noting the restrictions of our implementation:

1. This Artificial Intelligence is not that "intelligent." The AI effectively runs through a checklist of possible moves and performs the first possible move it determines. A more intelligent AI could utilize game trees and more complex game strategizing in order to prove a more formidable opponent, but this is a rather large proposal.
2. We only have support for a single AI at once. Theoretically, allowing for play with multiple computer players is much more possible than the above point, but would require careful attention to synchronization between local and global memory pieces related to each computer player.
3. Nothing about our AI displays any form of learning or adaptation, because this course is not about AI and our programmers did not have the effort to spare on such a feat.

Because a computer controlled player would just be a more directed player, we added subclasses to Player, named KingsAIPlayer and BartokAIPlayer. The essence of player remains the same, allowing for players to interact with the game board and making moves and plays until they are the winner or loser, maintaining the structure as defined by our finite state machine. The details become increasingly complex due to the lack of human interception of rule decision. The player defined drawFromDeck and newHand remain the same for the AI players, because this literal functional interaction with the board won't change; however, the nextstep function must be redefined, as the next step is no longer determined by human input, but computer decision.

The computer-controlled decision making shares similar essence; the computer sequentially runs through a list of possible moves and plays the first possible move. Otherwise, it ends the turn. Naturally, the details differ between the two games because the set of rules and possible moves are not the same. Bartok is much more simple in determination:

1. Iterate through the player's hand
2. If any of the cards are playable on the current pile (they have a card of the same suit or rank) they play the first one that is found
3. Otherwise, the player ends their turn by drawing

This is all handled in validIndexToPlay, whose return value determines which play value is returned in nextstep.

Kings Corner is a bit more complex due to the increased number of possible playing fields, the two possible types of "plays", and the fact that the player's turn will not end until they decide it to be (though this last point is handled by our finite state machine). nextstep first attempts to play a card from the hand before trying to move a stack by calling the validPlay function:

1. Iterates through the player's hand, and if the card is a King, plays on the first open corner field
2. Otherwise, plays a card to the first empty field found (this is part of the problem with our AI not being too intelligent)
3. Otherwise, iterates through all non-empty fields and plays a card in the first valid field if it is one rank smaller and a different color than the card on top of the stack
4. Otherwise, no card can be played, so it returns to attempt to move a stack

Next, nextstep attempts to move a stack with the validMove function:

1. Iterates through all the NEWS fields and, if the bottom card is a king, iterates through the corners. If a corner is empty, that corner will receive the stack containing the king. This is really only a play possible if a king is dealt to the NEWS field at game creation
2. Otherwise, iterates every NEWS field and iterates through every single other stack and, if the NEWS stack can be moved on top of the other stack, it does so
3. Otherwise, it returns and at this point, the player would end its turn

The point being that the essence for both games is the same, but like the human-controlled players, the details are so distinct that the only abstraction that can be done here is related to reutilizing the feature extracted functions from previous iterations.

6. CONCLUSION

Over the course of this project, we have transformed a "quick and dirty" naive implementation of two card games into an abstracted, organized project. Careful attention was paid to the essence of the two games and how they are similar in function, as well as the details of each game that made them different from each other, requiring specific implementation to differentiate them. This was accomplished in several iterations, each with a separate goal of abstraction:

1. Went from two massive and gross files to a web of many files that separated and combined functionality
2. Turned a lot of difficult-to-read blocks of code into combinations of human-readable feature-extracted functions
3. Went from each game having uncomfortably similar while loops for game state to having a single Finite State Machine that can be used by multiple games by passing in different play-defined states
4. Created an Artificial Intelligence that allowed for players to play against a computer-controlled player, which makes it easier for players to enjoy and for programmers to test and generate game transcriptions

6.1 Class Separation

From our two massive game files, it was obvious we needed to separate them into multiple classes, each taking care of functionality that is related to the named object. This allowed us to reuse redundant code that was found among both games, as well as to separate functionality into places that future programmers could use to create more card games that used

similar structures, such as the Card class. This is the one aspect that I felt was incredibly obvious to us, and we wonder why we didn't bother to do this in the initial stage when designing our dirty versions, as object oriented programming was drilled into us since the beginning of our degree studies.

6.2 Feature Extraction

A key factor in creating a domain specific language (in our case, we don't quite reach that, though we made changes that would aid in the transformation to one) is feature extraction. We took large, difficult to understand blocks of code (that could have appeared multiple times with different values) and turned them into human-readable functions that can be reused for other purposes, and are understandable by other programmers just by looking at the name. Many functionalities related to "is this move possible" as well as "Great, now let's execute that move" were created in Rules (as well as its child classes KingsCornerRules and BartokRules) to facilitate ease and readability of the game's rules and functionality. This laid a great framework for further development, as it allowed for the implementation of the Finite State Machine and AI to be accelerated due to the explanative nature of the feature extracted functions. This feature extraction was mostly spearheaded by one programmer, but were easily restructured and reused by one of our other programmers who mostly lead development of the FSM and AI.

6.3 Finite State Machine

The Finite State Machine allowed us to take two almost similar game loops from each game and turn them into a single machine that keeps track of states and transitions. The games would differ by giving the machine a different set of states relevant to the game, and the machine would handle transitioning between them in the same way, using the same code; of course, there were some minor differences within the FSM that had to be taken into account in the internals. The starting states required slightly different initialization, and is reflected in the internal initialization functions. This was the largest step in our goals of abstracting our two games, as it works towards this paradigm of having a machine that performs in the same manner for multiple games and simply requires different input.

6.4 Artificial Intelligence

While the AI itself wasn't exactly conducive to the goals of abstracting the essence of these games, it did end up displaying elements of abstraction in hindsight. Creating AI required the institution of Player subclasses for each game that handled the rules for each game automatically, rather than a human-controlled player that makes choices. Originally, Player is used as is for both games because the rules of the game are handled elsewhere, but implementing it this way required splitting the AI logic into two classes. Arguably, this might have been counter-productive to abstraction, but allowed for a nice example of how the details will vary from the essence for multiple games if it was not already displayed effectively through the rest of the project.

6.5 Further Thoughts

There are many directions we could take this project if we were to continue beyond the conclusion of the assignment. The biggest ideas are as follows:

1. The biggest plan would be to develop a few more card games using the code we created for these two games. We would then see how much of our abstraction is relevant, how much code we could reuse, and how much new code we would have to develop for the institution of these new games. It is likely that development would take considerably less time than the original games due to the previous development.
2. Similar to how the Finite State Machine works almost identically for each game internally given properly constructed states, we could create an Artificial Intelligence that parallels this. Somehow we could create a machine that plays the game given a set of rules we decide upon.

There are also some smaller details that we could consider:

1. We could allow players to play against multiple AI players, or even a game of entirely AI players
2. We could create support for more "human" elements, such as risk, fear, nervousness, etc. Quantifying humanity is rather difficult, however

Ultimately, seeing our incredibly smelly code turn into a well-constructed web of essence and details has been enlightening. This project has allowed us to see into the process of the creation of Domain Specific Languages and understand the choices made when considering the structures, functions, and approaches that makes a DSL understandable, relevant, and intuitive to users.