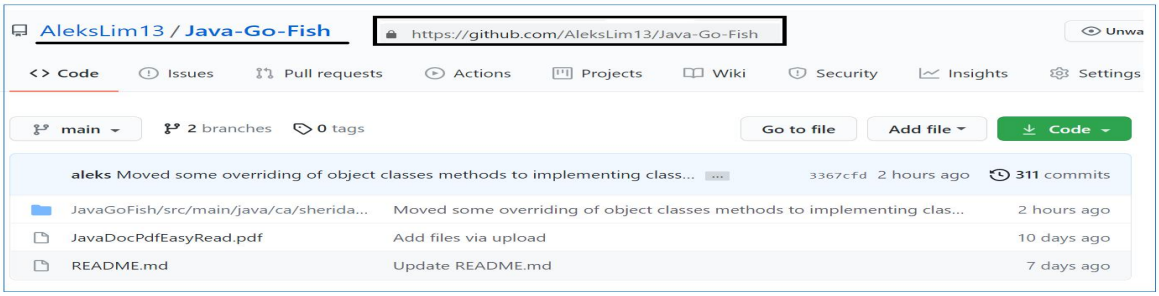
Student: Aleks Lim
College: Sheridan College
Date: April 12, 2021
Semester: Winter 2021
Repository: https://github.com/AleksLim13/Java-Go-Fish.git

Github Repository:



Problem Domain:

The present system addresses users desire for computer based games. The game is called Go Fish and is a specific example of a card game. Black Jack and Poker are other examples of card games. People want to play these game for leisure as a way to take their minds to a different place. Sometimes people want to play games for downtime where they need a less demanding tasks to bring their minds to a resting state. Games are more recently being used to improve mental faculties like logical thinking or memory recall in those who want to see cognitive improvements. The computer gaming industry is a billion dollar industry. I enjoyed developing this game for the duration of my Winter semester.

Programming Language:

We can write the card game in Java or Python programming languages. Python is noted to be used in games where interaction needs to be modeled digitally. There's nine different frameworks for developing games in Python that I can think of offhand. Not many computer games are written in Java however, there's lots of jobs out there for Java developers needed to build enterprise applications for business operations or services. Both are object oriented that make them suitable for developing larger applications. I chose java because that's what I was trained to use the most. I personally want to be a Java developer because I like code quite a bit and those developers are in high demand.

Project Structure:

The application is organized into five packages. The packages are labeled Cards, Players, Start, Turns, and Utility. There's eight abstract classes that form the template of the games object oriented design. Each of the abstract classes form functions of the game that any card game will require. There are ten concrete classes that extend their relevant abstract classes. There is also two classes for routine input and output that occurs for providing necessary feedback. I developed the application in Netbeans and it was worked on for four months before writing this final deliverable. This is a console application as there is not a GUI like XML or HTML.

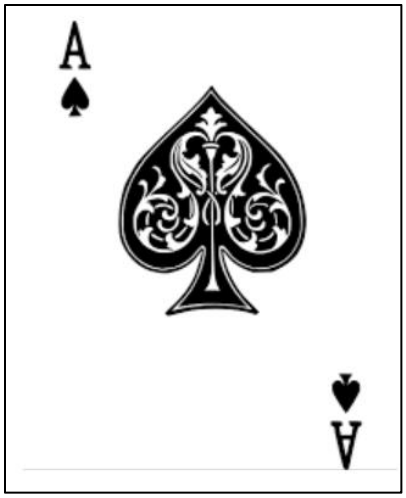
Software Requirements:

Building a Card game as software calls for ways to indicate different players and store information for them related to a running game. Several changing variables need to be tracked throughout the game. There needs to be a mechanism for modeling a deck of Cards and building each players hand. When playing a card game not every player has their turn at the same time so turns need to be managed individually. A dealer needs to be modeled so tasks that can influence a persons advantage that need to happen routinely to advance the game falls to a non biased individual. A lot of the time card games are played for money and card games as software are no exception to this rule. During a game, each player has to analyze their cards and determine how their going to group cards and what cards they want

versus don't want. So, this thinking needs to be modeled digitally. This project calls more for programming in algorithms and calculations.

ACard Class:

This class resides in the Cards package. The ACard class is an abstract class because we can use it in other games that use cards. We modeled the suits and values as String arrays with fixed values. There's two fields in this class that each concrete implementing class will have. There's a String variable for suit and one for value. Each instance of ACard will need to either be constructed with a suit and a value or have it set after instantiation. There's a constructor for just the value because during a players turn they only ask for a cards value and not the combination of suit and value. The values for suits and card values don't change in Go Fish so they're modeled as constants. Go Fish uses a standard deck with thirteen values that each have four suits. The suits array has four constant values as an uppercase String values. The values array has thirteen values as uppercase constant String values. Strings are easy to work with and the Collections framework has much prebuilt functionality we can use. We used regular arrays for this because there easy to perform calculations directly. We also need to know the range for suits and values for other tasks the game needs to perform like initializing the deck with fifty two cards. We chose to make both suits and values array a single centralized copy that each concrete card subclass will use jointly with other sibling card classes. There's a abstract equals method in ACard to enable quick card comparisons. We customized a equals method in the implementing card class so we can use the equals method for the state of our cards suit and value that we defined for Go Fish cards. We made the to string method abstract so concrete Card classes can provide the implementation because card suits and values can vary per card game.

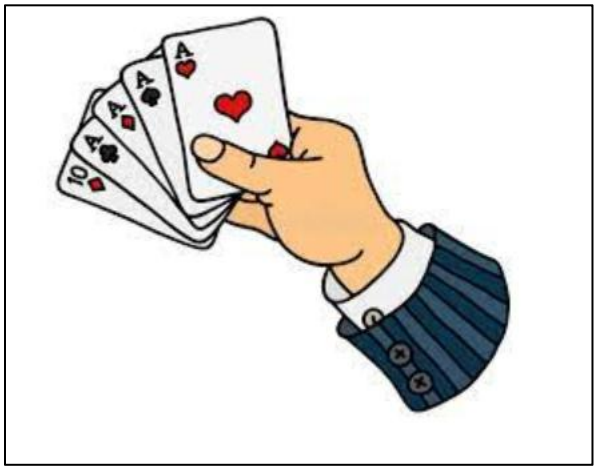


ADeck Class:

This class is abstract because other card games also need a class to model the deck used in their game example. The methods of this class are what all games that use a deck will need to do as well. This class has one global field to store the data that a deck has. It's declared at the top of the class right under the class declaration. The field is named deck and is a List construct of our ACard user defined type. The deck field is declared as final because the game needs one central copy that changes state based on players decisions throughout the game. There is one constructor for creating a deck instance of a concrete implementation. The client needs to pass in a concrete List subclass to initialize the deck field. Child classes of this superclass will have a initialize deck method to create the deck by filling up a variable of List ACard objects with thirteen values by four suits cards. It's declared abstract in this class so concrete implementations of this class will define it there. Instances of this class will have a shuffle deck method because the initialize deck method creates sets of four of each value from smallest to largest value. This means that when the initialize deck method is finished it's body of statements it the deck field of will have stacks of four of a kinds already grouped together. The deck needs to be mixed so there's a element of chance governing the game.

AHand Class:

This class is abstract because most card games require a player to get a hand or grouping of cards allocated by chance. The hand is needed to play the game where the hands contents will change dynamically based on decisions made by the owner of the hand and opponent players. This class has a global field for a deck and a scoreboard reference. There both declared as final because only one copy is needed of each for the game so they each only need to be created once. This class aggregates ADeck and AScoreBoard subclasses. There included as references instead of hand extending deck or scoreboard because a hand is not a deck and a hand is not a scoreboard. However, a hand needs a deck to create the hand and a hand needs a scoreboard to calculate duplicates. There is only one constructor for creating children of AHand class. The concrete class that extends this class will need to pass in two fresh subclasses of ADeck and AScoreBoard so the fields are initialized right away before a client tries to use them. The deck setup method of this class uses the deck reference to initialize and shuffle the deck field variable in the ADeck class. The AHand class has a method called create hand to be used after the deck has been created. The create hand method is told how many cards to create and then instantiates them and adds them to a List of ACard type. This method is only used once for each player. The start deal method is called by create hand because in addition to adding cards to a players hand, the same card needs to be removed from the deck and we chose to define that in a separate method to keep things clear and simple through separation. There are methods for finding positions of cards in a hand and then removing or adding them to or from target hands. This class also comes with a method for drawing a card from the top of the deck when a player is told to ”Go Fish.” There’s even a sort method in this class so players can see groupings of their cards for clarity.

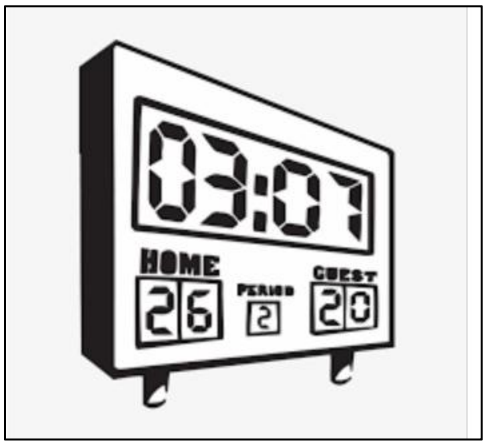


APlayer Class:

This class is abstract because all card games need to model players that have unique data for the people playing a Go Fish round. The six fields of this class reflect what a data a player needs. We distinguish players by name and id. Each player has three lists for tracking their hand, amount of four of a kinds, and duplicate cards in their hand. There is only once constructor to create subclasses of this parent class. Pass in two lists of ACard and a String so the fields are initialized before used. The print statistics method returns a String representation of all a players data. This class defines getters and setters for all fields that subclasses can use directly.

AScoreBoard Class:

This class is abstract because it could be argued that each type of card game will need to perform the operations that are defined in this class. This class has no references to other classes because it performs mostly general calculations on the a players hand. This class has four methods in total. This class has a method called get duplicates. Each player has a reference to a List variable called desirable list. When this method is called, the target players hand is searched and the duplicates are added to desirable list in addition to being in the players hand. This class has a method called find first index. This method is to remove duplicates from a hand one card at a time. This class has a method called calculate books. When a player gets four of a kind this method will remove all four cards from the players hand and place them into a different list to keep track. A player doesn't need their hand cluttered up with cards they no longer need to ask for. This class has a method called determine winner. This method will take both players of the game after the central deck runs out of cards and see which players book list variable has more four of a kinds in it.



ATurnManager Class:

This class is abstract because all card games need to conduct the operations that determine a full players turn. This class has six global field variables. The human and computer reference of type APlayer are declared as final because once their determined who they are they must stay assigned to those memory locations. The two reference fields in play and not in play are not declared final because the human and computer players each become designated as in play or not in play depending on who's turn it is. The AHand and AScoreboard reference are declared final like human and computer because we need a single centralized copy of each. There is only one constructor for creating subclasses of ATurnManager. The arguments require fresh concrete copies of the fields so their not used for initialized. There are five methods in total belonging to this class. This class has a method called human asking for a card. This method asks the human player through console input and output what card they want and expects a string input of a card value. The method will keep prompting the human for a card if they don't provide a value of one of the thirteen card values in a standard deck. When they do provide a valid value the

methods returns the card. This class has a method called computer asking for a card. This method uses the computer desirable list and hand to determine which card to ask the human player for. If the desirable list isn't empty then the computer will ask for the first duplicate card in their list. If that is empty, because their hand doesn't have any duplicates the ask for the first card the have in their hand at index 0. This class has a method called go fish. This method is passed a card in and then checks if the card is in the players hand who's being asked. It returns a yes if it's in there or a no if it is not. This class has a method called should keep going. A while loop locks in circuit of repeating behavior. First, it checks if the player who's in play is a human or a computer player. Based on the result of that check the appropriate ask method is called and the appropriate card is returned. The card from that process is passed into go fish method now and the result is stored in a boolean variable. Based on the result of that check, there are two paths available where only one can be taken. If the card is in the not currently asking player the cycle will repeat as many times as a asking players ask is successful. Hands of each player is updated accordingly dependent on which path is taken depending on the outcome of the go fish method. When the card isn't in the not asking players hand turn switcher is called to switch who's turn it is. Turn switcher belongs to this class and switches players who's designated in play with the help of a temporary APlayer variable.

AGame Class:

This class is abstract because if we make other card games we need a objective way to distinguish them. This class has a reference to ATurnManager (TM). Through the TM the game subclass can manage what needs to happen during each players turn. It's final because we only need once centralized copy and it's state doesn't need to change after it's creation. This class has only one constructor to create unique game examples. Pass in a subclass of ATurnManager to initialize the field so it's not used before being initialized.

AStart Class:

This class is abstract because each card game needs to define how it will start and play the game. There is a start method that subclasses of this class will need to define. That's where the main steps of the game will be defined. All use cases are defined in the play method. This class has a reference to AGame object.

CRoundOne Class:

This is a concrete class that extends AStart class. It inherits a reference to AGame object from AStart parent class. This is the final top layer of the game and I believe all the layers are justified up to and including this ending point. The constructor for CRoundOne calls for a concrete implementation of AGame that will be CGoFish game at run time. In the play method this class overrides has the main use cases of the application. The single copy of CGoFish has a single reference to a TM where all the use cases are triggered. First, we use the game reference to access TM. Through TM we access the reference variable hand. We then call the create hand method on the hand reference. For this method we pass in the human player and the number seven to indicate we want a hand with seven cards. The deck setup method is called automatically in the constructor we defined for hand. The human players hand field gets filled with random cards from the deck the hand class can access through it's ADeck class reference. We go through this same process for the computer player. The create hand methods is called only twice for a full game of Go Fish. That completes the first use case of our game. Next, we use the game reference to access our TM. Through our TM we access our AScoreBoard concrete implementation we passed into the TM constructor. We call the get duplicates method belonging to the AScoreBoard reference and pass in the TM's copy of the human player. The human players desirable list field gets filled with it's duplicate cards. We repeat this same process for the computer player. This completes our second use case for our game. Next, we use our AGame reference to access our TM. Through the TM we use the setter to set the TM's in play variable. We pass in our TM's copy of the human player into the TM's in play setter method. We repeat this same process but use TM's not in play setter method and pass in TM's copy of the computer player. This completes our third use case for our game. Now, a while loop locks in a circuit of behavior for the game for as long as the deck still has cards. The while loop accesses the deck through AGame -> TM -> AHand -> ADeck -> ADeck get deck to then calling the List method is empty. Then, we use the AGame reference to access our TM. Through our TM we call it's method should keep going. Flow of control then steps inside the should keep going method where a while loop locks in a sub circuit of repeating behavior. This loops will continue to ask the no in play player for desired cards until their ask isn't successful. This sub circuit will update both players hands accordingly. Then the turn switches and keeps repeating until the deck is empty. This completes the fourth use case of our game. After the turn switches, the calculate books methods is called on the previous in play player. We use the AGame reference to access our TM. Through the TM we access our AScoreBoard reference and calls it's

methods calculate books. We pass in the previous in play player to this methods that we get through our TM. This completes the fifth use case of our game. When the deck is empty the outer while loop in CRoundOne ends. Then, we use our AGame reference to access our TM. Through the TM we access our AScoreBoard reference and call it's determine winner method. We use our Tm through our AGame reference to pass in the TM's copy of human and computer player. This methods checks the size of each players book list variable to determine who's is bigger and thus made the most progress throughout the length of the game. Cest fini.