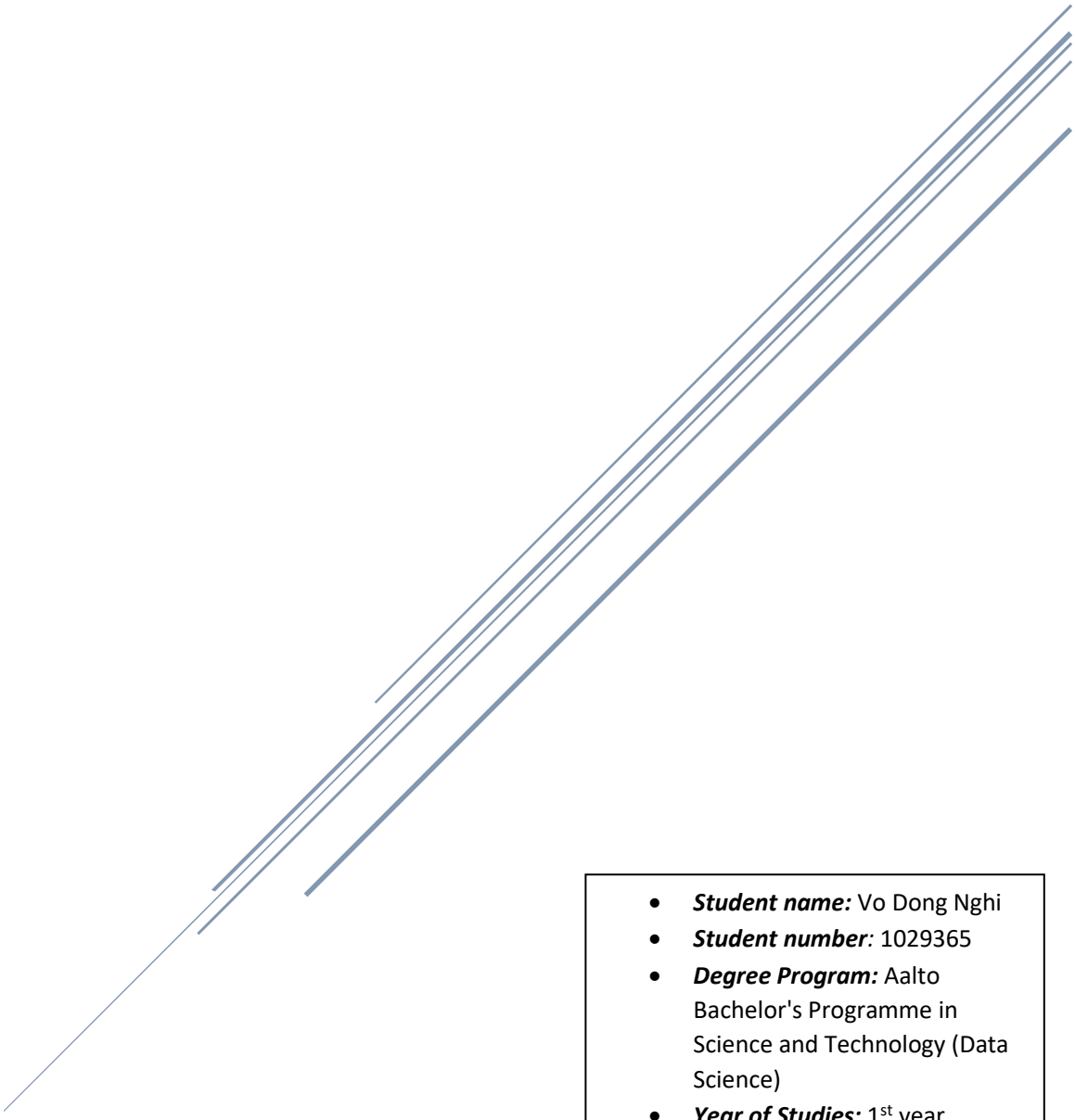
PROJECT DOCUMENT

Project Title: Casino



- ***Student name:*** Vo Dong Nghi
- ***Student number****:* 1029365
- ***Degree Program:*** Aalto Bachelor's Programme in Science and Technology (Data Science)
- ***Year of Studies:*** 1st year
- ***Date:*** 26-04-2022

Contents

1 General description 3
1.1 Game aspects 3
1.2 Application aspects 4
2 User interface 5
3 Program structure 9
3.1 Exceptions 9
3.2 GameLogic 10
3.2.1 Card Package 11
3.2.2 Move package 12
3.2.3 Player package 14
3.2.4 Round, Game, and FileManager 17
3.3 GUI Package 18
3.4 Conclusions 20
4 Algorithms 21
4.1 Verifying moves 21
4.2 Finding moves 23
4.2.1 Finding component captures 23
4.2.2 Stacking component captures 24
4.3 Making moves for computer players 26
4.3.1 Functionality 26
4.3.2 Extensibility 27
4.3.3 Efficiency 28
4.3.4 Some strategies 29
5 Data Structure 30
5.1 Vector 30
5.2 Map 31
5.3 Set 31
5.4 Tuple and other Data representation 32
6 Files 32
7 Testing 33
7.1 Unit Tests 34
7.2 Manual Tests 34

8 Known bugs and Missing Features 35
9 Best sides and Weaknesses 35
9.1 Best sides 35
9.1.1 Independency 35
9.1.2 Extensibility 36
9.1.3 Attention to details 36
9.2 Weaknesses 37
9.2.1 Time cost 37
9.2.2 Redundancy 37
10 Deviations from plan, realized process and schedule 37
11 Final Evaluation 39
12 References 39

1 General description

The aim of this project is to create a Casino game, which leads to two essential goals: first, this project should be able to recreate a real-life Casino game, with all its dealing, playing rules and scoring; and second, this project should take the application aspects into account, such as graphical interface, computer-simulated players, saving data of previous game state, …

1.1 Game aspects

As for the first goal, this project would comply with the rules of Finish / Nordic Cassino as described in the Cassino project page of A plus and Pagat.com [1]. Specifically, the gameplay could be described in parts, including:

1. **Deal:** The dealer is picked at the start of the game, which can be one of the players or a computer-controlled player. The cards are then dealt in pairs, one pair to each player, one on the table (face-up) and one to the dealer, then repeat. It is important that the dealing of the cards in the game is the same as that in real-life. Then for each round, the player to the left (clockwise) becomes the dealer. The game starts with the player to the left of the dealer making the first move, then play turn passes clockwise

2. **Play:** After the dealing, each player has 4 cards (not visible to other players) and 4 cards on the table (visible to all players). Each turn, the player always plays one card to the table. If the player captures a card or cards, the player put the card played and the card (s) captured in a separate pile. If no card is captured, the player plays his or her card on the table face up to be captured later. A played card can capture:

    - A card on the table of the same capture value
    - A set of cards on the table whose capture values add up to the capture value of the played card
    - Several cards or sets of cards that satisfy conditions 1 and/or 2 above

    If some player gets all the cards from the table at the same time, he or she gets a so called sweep which is written down. There are a couple of cards that are more valuable in the hand than in the table

    - Aces: 14 in hand, 1 on table
    - Diamonds-10: 16 in hand, 10 on table
    - Spades-2: 15 in hand, 2 on table

    The player must also take a card from the deck so that he or she always has 4 cards.

3. **End**: When every player runs out of cards, the last player to take cards from the table gets the rest of the cards from the table. After this the points are calculated and added to the existing scores. The following things grant points:

- Every sweep grants 1 point.
- Every Ace grants 1 point.
- The player with most cards gets 1 point.
- The player with most spades gets 2 points.
- The player with Diamonds-10 gets 2 points.
- The player with Spades-2 gets 1 point

One must collect points which are calculated after every round. The game continues until someone reaches 16 points.

1.2 Application aspects

As for the second goal, this project aims to create a application to satisfy the demanding level of difficulty. In particular, the Casino application should:

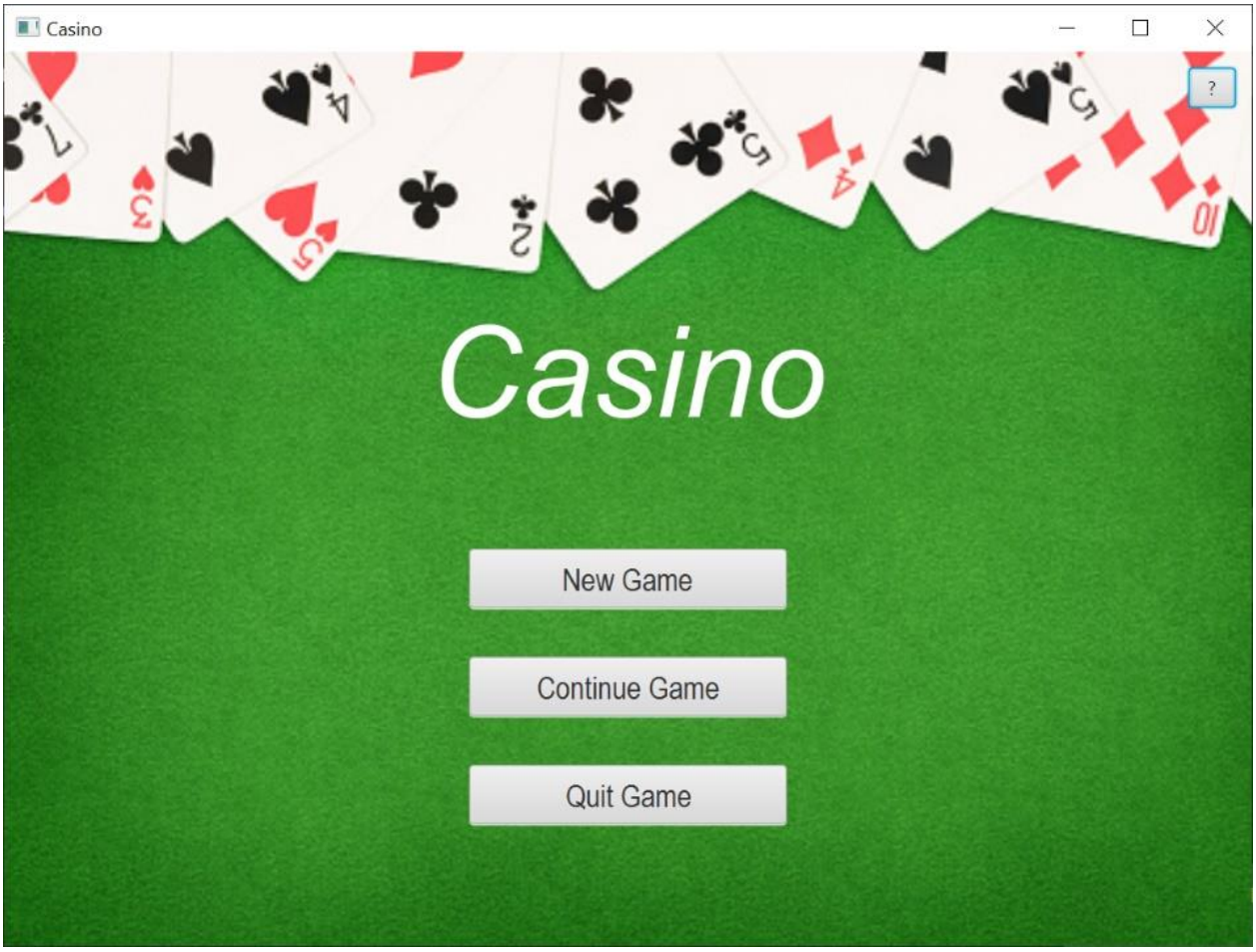
- **Have graphical user interface**

- **Allow the user to play against human and computer – controlled opponents:** The current implementation allows the user to start a game with 2 – N players, including at least 1 human player and computer – controlled opponents (0 – N). The computer – controlled opponents have access to different strategies, such as prioritizing special cards (e.g. Aces, Spade-2, Diamond-10) or sweeps, avoiding easy sweeps, building with play moves, prioritizing more captured cards and captured Spades, … The computer – controlled opponents share the same strategies, but can behave differently based on their types and the preferences of each type. For example, a "careful" computer – controlled opponent "favors" the strategy to avoid easy sweeps, while an "immediate" computer – controlled opponent "favors" strategies that help it gain the most point during the current turn, such as prioritizing special cards, more captured cards, … The type of computer – controlled opponent is chosen randomly when one is added. However, the user can choose the "mode" of each computer player, which is among 4 modes: Random (default mode), Easy, Intermediate, and Hard.

- **Save the current game and load the previous game:** The current implementation allows the user to save and load the previous game from 2 files, 1 main file and 1 back-up file. If the user quits the program appropriately (using quit buttons), the user will be prompted to choose to save or quit without saving the current game, and the saved game (if chosen to be saved) will be in both files. If the user quits inappropriately, the main file will be marked as "interrupted" and the back-up file will store the current game (if any). When loading the file, if the main file is found to contains error or being interrupted, then the user will be prompted to try loading from the back-up file or start a new game.

2 User interface

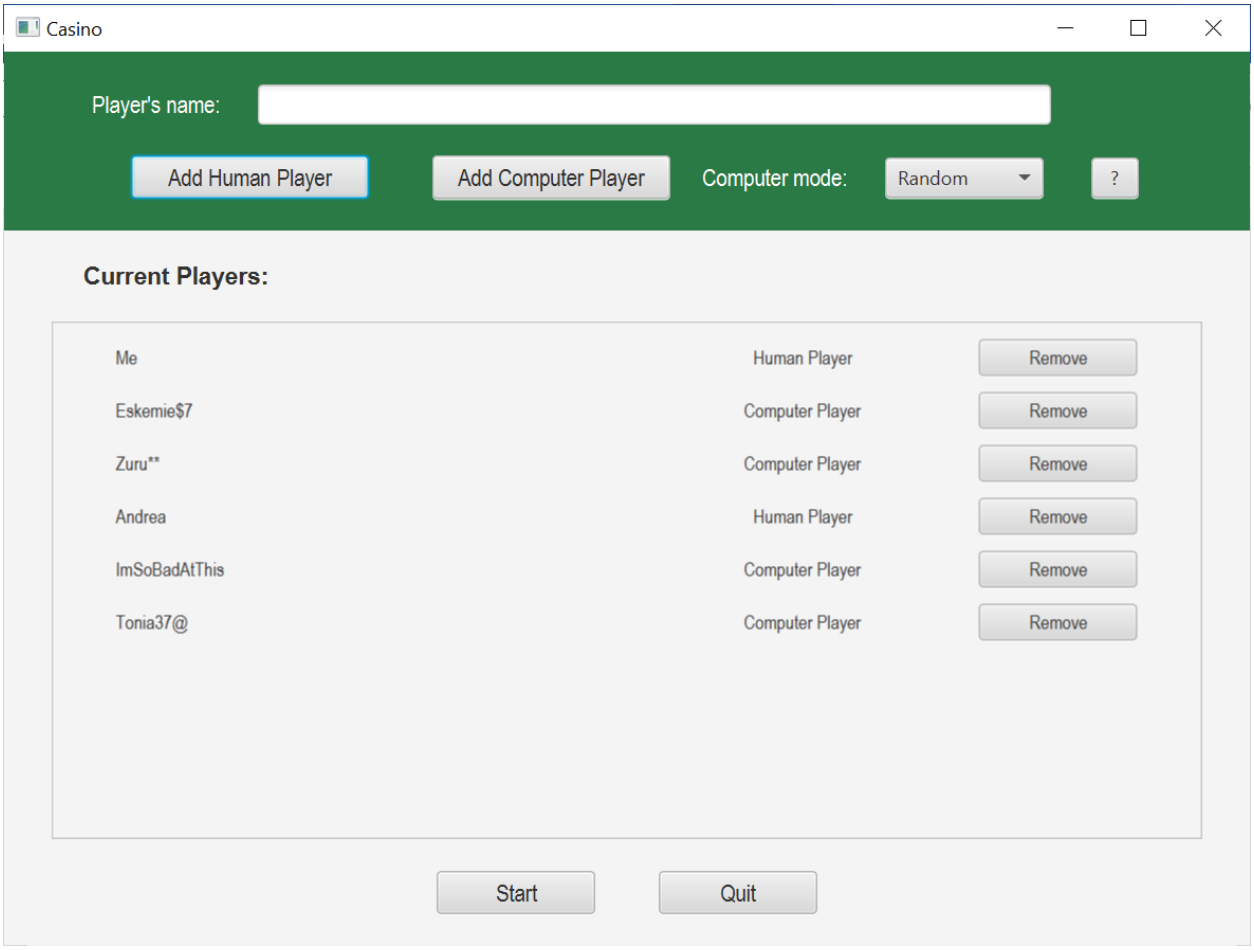
The user interface implemented is a graphical user interface using the library scalafx. The user interface has 4 main scenes, with some alerts to interact with the user. The 4 main scenes are as followed: the Menu scene, the Add Player scene, the Player View scene, and the Score scene.



Menu Scene

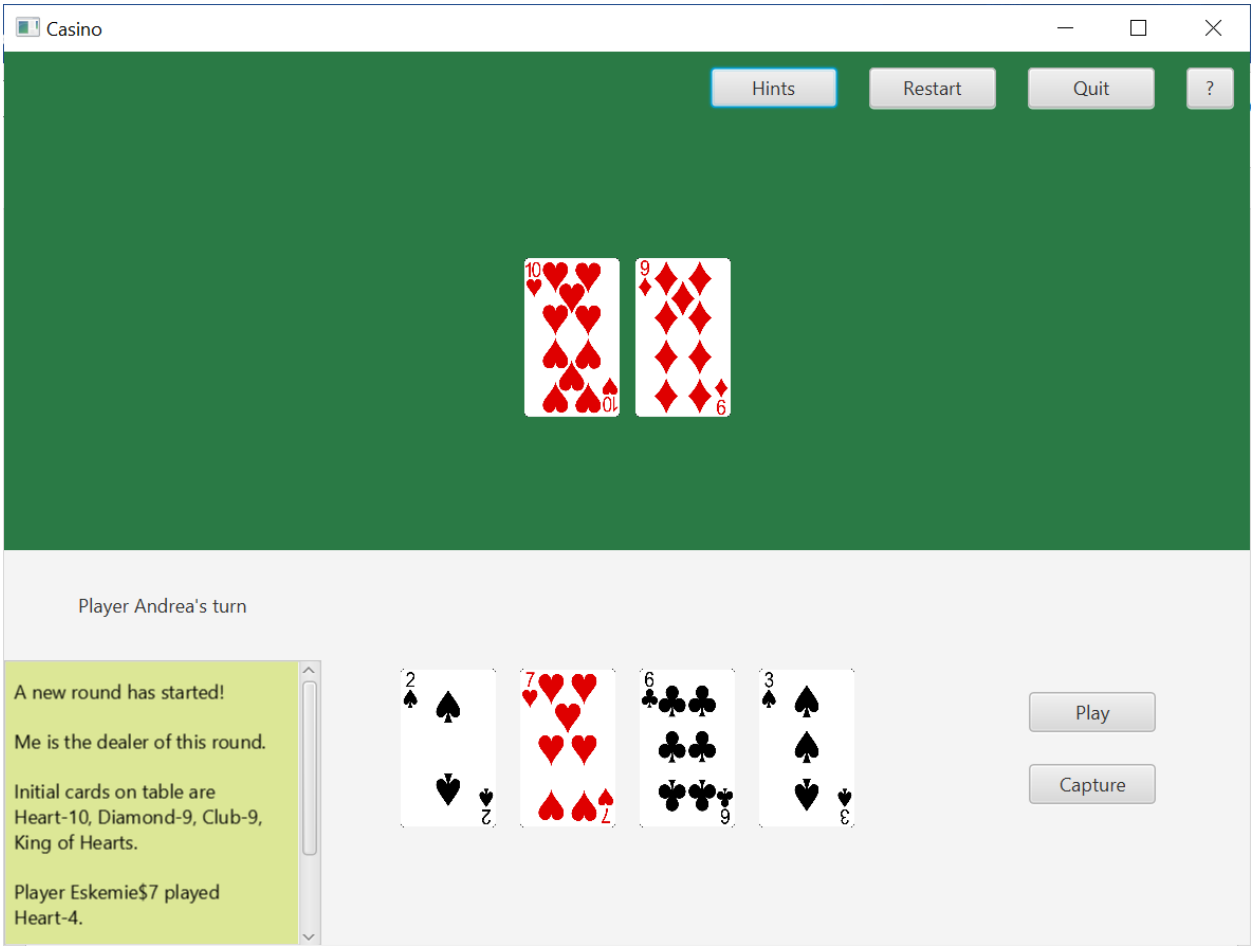
At the start of the program, the user interacts with the Menu scene, which has 2 buttons and thus 2 main functions. The user can either choose to start a new game, or continue the previous one. In the case the user chooses to start a new game, the scene is switched to the Add Player scene, otherwise it is switched to the Player View scene. However, if there are errors in file-reading, or no previous game has been saved, the user may interact with dialogues to retry loading from back-up file, or to start a new game.

If the user chooses to start a new game, the scene is switched to the Add Player scene. With this scene, the user can add human or computer players, as well as removing the added players. There is also a condition that all players should have a name, with the computer player having some default names if no name has been provided. The user can then choose to start the game when they have finished adding players. The conditions for the number of players is that at least 2 players must be added with at least 1 human player, total number of players should not exceed 12, as there would be not enough cards to start the game otherwise. If the number of players added is valid, then the game will start with the first player being the dealer and the scene will be switched to Player View scene. In the case there are any invalid user inputs (no name given for human player, or starting the game with no appropriate number of players), there will be dialogues informing the user of such mistakes.



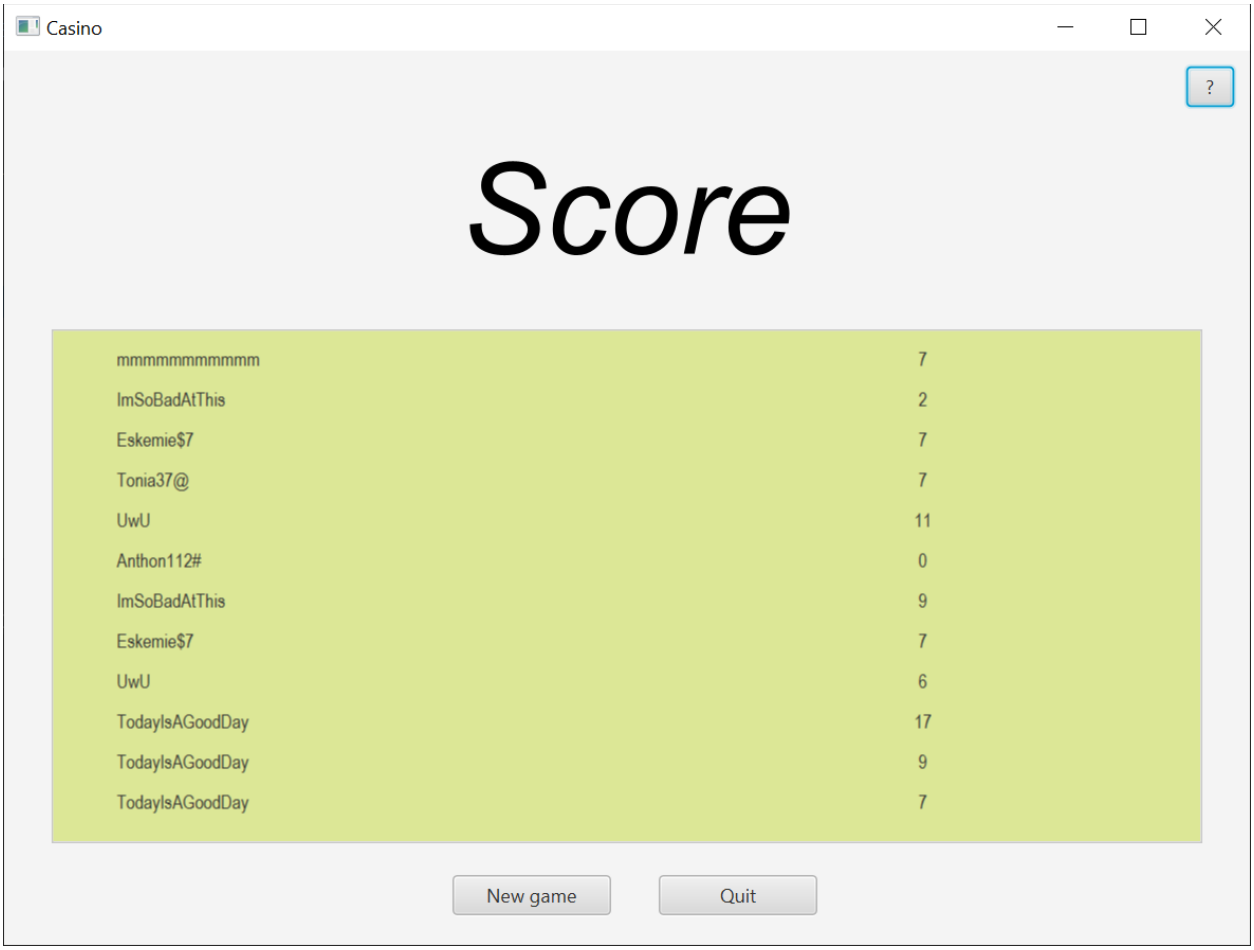
Add Player scene

If the user then chooses to start the game (and appropriate number of players has been added), the scene is switched to the Player View scene, which is the view of a single player who is having their turn (human player). This view includes the hand of that player, the current table, a game log which summarizes the events in the current round, and several buttons. The player can choose a card by clicking on the image of that card, and click again to undo the choice. When the player has chosen the cards, they can make a move using the Play or the Move button. If the move is valid, the hand and the table are updated, and an alert shows up to notify if there is a next player who has the view. Afterwards, the stage closes and a new alert shows up to notify the next player and the Player View scene is switched back. In case the move is invalid, there will be warning alerts to notify the player of their invalid choices. There are some other buttons that support the game play, such as Restart button and Hint Button. When a round or a game ends, the scene is automatically switched to the Score scene.



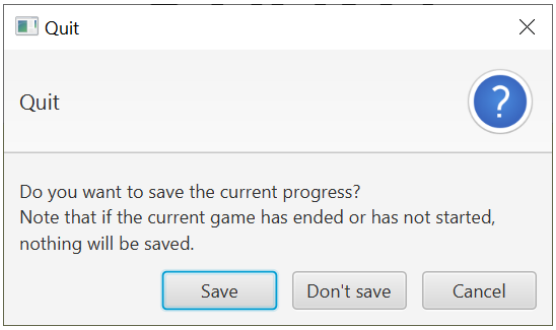
Player View scene

The Score scene shows the scores of every player after the round / game has finished. The player could use buttons to start a new round / game. There will be a notification to announce the winners (if any).

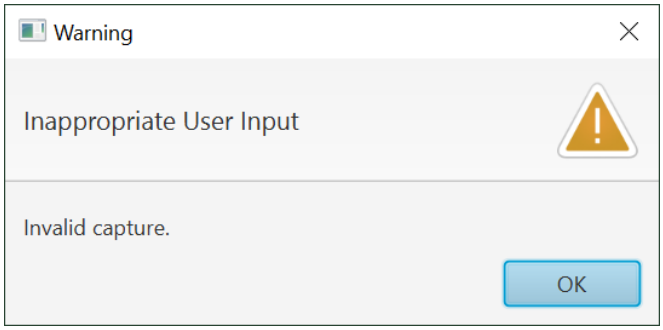


Score scene

In addition to these main scenes, the user can interact with several dialogs and notifications, such as these:

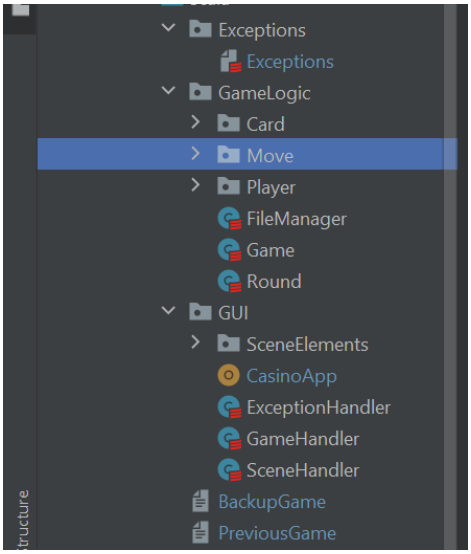


Quit game dialog



Warning alerts

3 Program structure



The general program structure can be perceived by the packaging in the source code, with 3 main packages dividing the program into 3 relatively independent parts: Exceptions, GameLogic and GUI (graphical user interface). In the Exceptions package there is a single file which stores all the used exceptions. In the GameLogic package, there are also 3 smaller packages for Card, Move, and Player, with 3 main files standing for 3 administrating classes of which functionalities are built upon "smaller" classes. In the GUI package there is the CasinoApp object which runs the program and its 3 significant helpers the ExceptionHandler, GameHandler and SceneHandler. There is also a small package SceneElements which stores the small classes used for nodes in the scene. Additionally, there are 2 files used for saving the game outside of these 3 main packages.

3.1 Exceptions

The idea for using exceptions initially stemmed from the need for a uniformed method of communication between the graphic user interface and the game logic without increasing their interference with each other. As the project progresses, exceptions become powerful identities which serves to increase the effectiveness of communication, enhance the functionalities of the program, and increase the independence of each class:

- **Communication:** There are various forms of communication between classes. One of the simplest methods to transfer the necessary information is adding more parameters to the functions. In fact, there are some functionalities that use this method within the classes of the GUI package. However, with a multi-layered program structure in which some classes being administered by another, this method of communication will require a chain of information transferring. For example, the algorithm to verify the moves is currently in the Move object, which can be accessed by Player class when creating Move instances. If the move the user made is invalid, then the result of that invalid move must be transferred through the Move object to the Player class, then to Game class which finally delivers the information to the GUI. As such, this method will become increasingly difficult to manage as more and more of these conditions, verifications or checkpoints are implemented in the program. Similarly, the method of communication with a class calling a specific function of another class based on the result of its function is discarded for the same reason. As such, exceptions are used to communicate as they can be easily delivered across the system until the function that handles them. In this case, such functions are implemented in the CasinoApp which catches the abnormalities that occurred in the GameLogic system and handles them with new user input.

- **Functionalities:** Unlike elementary data types such as String, Int, or Boolean, which are usually used for transferring information, exceptions can easily be grouped or classified, and thus allow the program to effectively identify and react to different types of exceptions. For example, the program

currently has 2 main exceptions: exceptions due to user input and due to file reading. Using inheritance in the structure of exceptions in this program, the program can easily process and react accordingly, while also reacting to unknown exceptions in a relatively meaningful way. Additionally, although the program currently does not have specific reaction to the specific exceptions under these main categories, this method leaves room for future development.

- **Independence:** As previously explained, the use of exceptions allows communication without having chains of information transferring. Therefore, the functions in one class can be implemented without relying on the results from the functions from other classes, or giving results for the following functions. Furthermore, this use also makes developing and separating several verification algorithms for each class much more convenient. For example, when making a move, the program will need to verify if the move is valid. If the usual method of communication is used, the verification method will have to be implemented fundamentally in the Player class, which then commands the Round to react based on the results. However, this would make the code clustered and much less manageable. Using exceptions, this verification method can be implemented in another class Move and when creating a move, if that move is invalid, the creation will fail as the exception halts its execution, and that exception will be automatically thrown to the function that handles it. Thus, the communication goal is achieved while the Round, Player and the new Move class are still relatively independent.

To use exceptions, a new class CasinoAppException which is a direct subclass of Exception is created. The direct subclasses of this CasinoAppException are the IllegalFile and InvalidUserInput classes, which represents the 2 main exceptions mentioned above. The subclasses of these classes are case classes standing for specific errors. For the superclass IllegalFile, the subclasses are UnknownCard, MissingCard, MissingMetaData, MissingPlayerInfo, IllegalPlayerInfo, MissingCardDeckInfo, MissingTableInfo, IllegalExpression, InterruptedFile, NoPreviousGame. For the superclass InvalidUserInput, the subclasses are NoNameChosen, NoHumanPlayer, InvalidNumPlayers, NotEnoughCardsInDeck, WrongMoveButton, NoCardInHandChosen, and InvalidCapture. Each of these classes has text assigned to it when thrown, and the getMessage function of Exception is used to retrieve the description of error for the user. All of these specific case classes will be mentioned in the functions later explained in GameLogic package. Additionally, the exceptions IOException and FileNotFoundException from java.io. are also handled.

3.2 GameLogic

The GameLogic package contains the most important operations of the game. This package contains 3 smaller packages for easy management and identifying the more closely associated classes and objects. These packages are the Card package, Move package, and Player package. It is also convenient to investigate these packages in this order as the functionalities of the classes in the latter package use those in the previous one. Outside of these packages are the classes which have more administrative roles: FileManager, Round, and Game class.

3.2.1 Card Package

The Card package revolves around the central class: Card class. This class is a case class and its role is to act as a data representation and allow easy pattern-matching for later algorithms. Card class has 4 values: suit: Suit, nameVal : String (name value, such as King, Ace, 3), numVal : Int (numerical value on the table), and valueInHand : Int (numerical value on the hand). As a case class, this class only has 3 effect-free function: giveName : String (the name used to describe the card in game log), giveImgPath : String (the corresponding image path), and giveSaveDetail : String (the format in which a card should be saved in). The cards are unchanging identities throughout the game, with each is a combination of one suit and one name value. Therefore, having method to store this combination of information is easily transfer and extract them is beneficial for the program functionalities and style, and the case class structure with immutable values and pattern matching can achieve this goal efficiently.

However, this class also requires some logic, including finding its numerical value and value in hand. Originally, I planned to implement a simple function to these values, but as the project progresses, the later algorithms increasingly need to use these values, especially the value in hand. Additionally, to implement more logic in this class is quite inconsistent with the purpose of this class as data representation. Therefore, I changed these values into the innate values of the class and utilized the Factory Pattern with a companion object Card. This implementation allows simple, intuitive method to create Card instances using a string of length 2, with the first letter being the initial of the Suit name and the second represents its name value: h2 is Heart-2, da is Ace of Diamonds, and s0 is Spade-10. Furthermore, verification method can be implemented in this object and initiated upon card creation, which detects the UnknownCard exception when loading the file. Hence, as a whole, this method increases the Card class independency and future modifications can be easily implemented. For instance, some variations of the game can assign different values for the cards, and this system can easily adjust to those values by modifying the Card class and object. This object can also be used for storing some special collections of cards that will be used for later algorithms to avoid recomputing these collections. To sum up, the Card object is added with a goal to create and verify the Card upon creation and store useful collections of cards for future use.

As such, the Card object has 2 private val allSuits: Map [String, Suit], and allValues: Map[String, Int] which are used for creation algorithm, 2 public val allCards : Vector[Card] (52 cards) and standardCards : Vector[Card] (cards with all different values in hand) which are special immutable collections of cards, 1 public function apply (cardInfo: String) : Card, and 2 private functions generateAllCards : Vector[Card] and findValueInHand (suit: Suit, numVal: Int) : Int as helper functions.

As perceived from the functions, Suit is a class. This is a simple class with 1 public val suitName: String and only 4 instances which are case objects: Heart, Diamond, Spade, Club. This structure is implemented because it is more intuitive than just using string values for suits and more convenient when using pattern-matching.

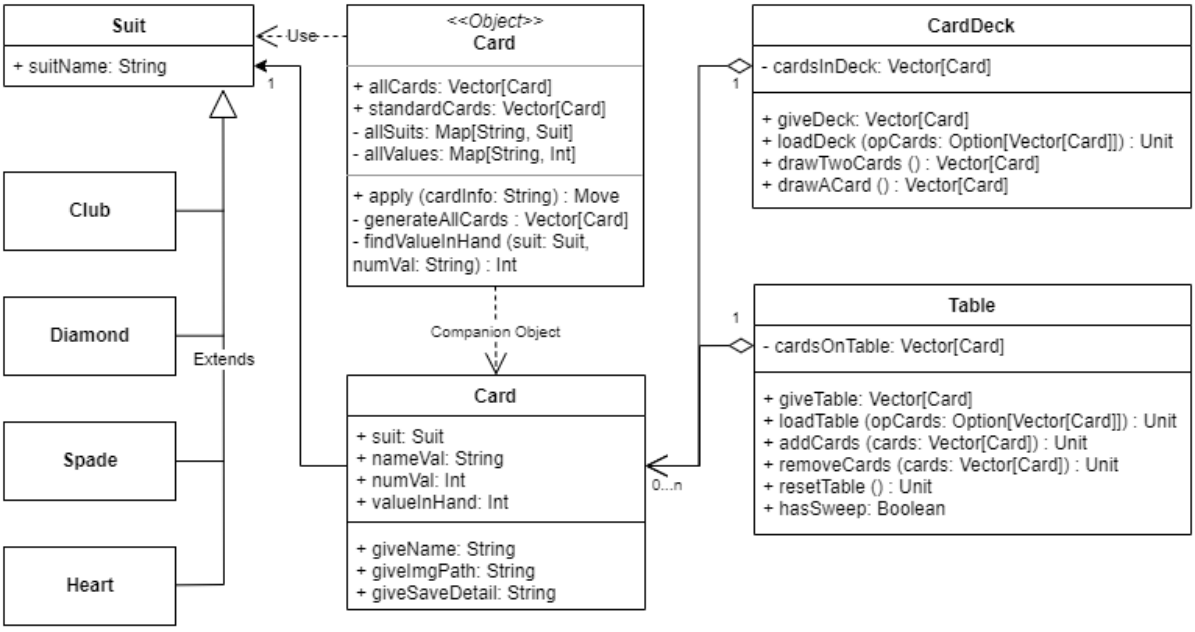
Similarly, the classes Table and CardDeck are added because it is more intuitive and manageable. The functionalities these classes provide can be replaced using more val or var in Round or Game class, but it may become much more confusing and the code in Round or Game class will be clustered. The operations in these classes are very simple and revolves around managing its core internal collection of cards. These operations will also somewhat represent their real-life entities, for instance, CardDeck class with drawACard function.

For the Table class, there is private var cardsOnTable : Vector[Card] which stores the current table and a public function giveTable : Vector[Card] to allow access to the current table. There are public functions

loadTable (opCards: Option[Vector[Card]]) : Unit which loads the table from a file, addCards (cards: Vector[Card]) : Unit, removeCards (cards: Vector[Card]) : Unit and resetTable () : Unit to change the cards on table, and hasSweep : Boolean to check for the sweep conditions.

For the CardDeck class, there is also a private var cardsInDeck : Vector[Card] and the public function giveDeck : Vector[Card] to grant access to it. There are public functions newDeck () : Unit which load the collection allCards from the Card object and shuffle it, loadDeck (opCards: Option[Vector[Card]]) : Unit to load from file, drawTwoCards () : Vector[Card] (for dealing) and drawACard () : Vector[Card] (for each turn) used to draw cards from the deck and return the drawn cards.

The UML diagram for Card package:



3.2.2 Move package

The Move package contains one major algorithm of the game: finding and verifying moves made by user. This package is consisted of 4 files, Move, Finder, Grader and MoveType, with the functionalities of the former 3 classes and objects making use of Card instances.

Similar to the Card class and object, the Move class is a case class with its companion Move object serving as a Factory to create Move instances and verify them upon creation. This is because once a "move" is created, its properties do not change and therefore, will better serve as a data representation, packaging its type, card played and cards capture for other classes to easily access and use. And as the verification algorithm is implemented in its own companion object, the Move class and object are independent which leaves room for modifications or improvements.

Hence, the Move class has 3 values: moveType: MoveType, cardPlayed : Card, cardsCaptured : Vector[Card]. Its companion object has 2 public functions apply (moveType: MoveType, cardPlayed: Card,

cardsCaptured: Vector[Card]) : Move which allows a simple, straightforward mean to create a Move (used for testing and finding moves), and apply (moveType: MoveType, cardChosen: Option[Card], cardsChosen: Vector[Card]) : Move which creates and verifies the moves based on user input. To verify the moves, the Move object also has 3 private helper functions verifyPlay (cardsChosen: Vector[Card]) : Unit, verifyCapture (cardPlayed : Card, cardsChosen : Vector[Card]) : Unit and strongVerification (cardPlayed: Card, cardsChosen: Vector[Card]) : Unit (this function is used in verifying capture moves, explained in more detail in the Algorithms Section). During the verification process, the exceptions NoCardInHandChosen, WrongMoveButton, and InvalidCapture may be thrown, and the execution halts when no Move instance has been created should these exceptions are detected.

Like Suit class, the class MoveType and its case objects are added for convenient access and pattern matching. This class has 2 case objects, which are Play and Capture.

The role of the Finder class is finding all possible moves from a "table" (a set of given cards for capture moves) and a "hand" (a set of given cards to play). While this single role is straightforward, the algorithms used to achieve this are quite complicated. Initially, I only implemented these algorithms for the computer player. However, as the project progresses, the need for these algorithms arises in other classes and even eventually the Move object itself (more details in Algorithm Section). Therefore, it seems unreasonable that these algorithms are implemented in the Computer Player class. I also consider implementing them in the Computer Player object, but as the "table" and "hand" are different for every time these algorithms are needed, then having a single object responsible for handling these changing situations is not intuitive and difficult to manage. Implementing the algorithm in the Move object is also not beneficial, as these complicated algorithm would not be easily managed and interfere with the core Factory design pattern of the Move object. Hence, I chose to create another class for this purpose which has the algorithms to find all possible moves. This choice of implementation allows reusing these complicated algorithms, as well as breaking down the algorithms into smaller tasks and therefore, simplifies the process of comparing the algorithms or modifying and upgrading the algorithm.

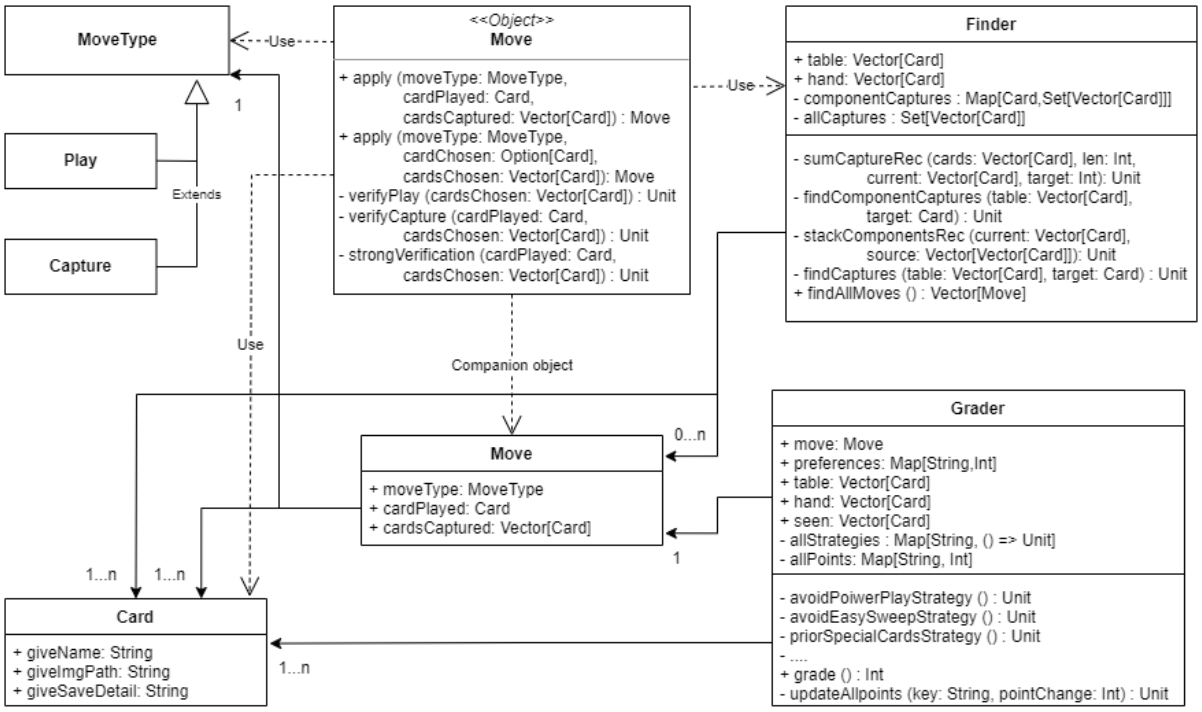
As such, the Finder class needs to be passed on the table : Vector[Card] and hand : Vector[Card]. It has 2 private val componentCaptures : Set[Vector[Card]] and allCaptures : Set[Vector[Card]]. It has 1 public function findAllMoves () : Vector[Move] and several private functions. Among these private functions, the core algorithms are in 2 recursive functions sumCaptureRec and stackComponentRec. This class and its functionalities will be explained in more details in the Algorithms Section.

Similar to Finder class, the Grader class was originally planned to be implemented in the ComputerPlayer class or object. The need for its usage does not increase, but as more and more "strategies" being added, the code in the Computer Player object becomes very clustered and difficult to manage or modify. This method also limits the extensibility of the Computer Player system itself. But with this new class for grading the moves, the process of altering or adding the grading strategies become much more simple. Additionally, this structure allows an independence class to manage the grading system which is quite crucial for the algorithm. For example, with a different class to "grade" the moves, new types of computer players can now be easily created by changing the "preferences", which are then easily passed on and processed by the Grader class. This structure is superior to the usual method of creating different subclasses of the Computer Player class in terms of extensibility. Furthermore, having a class to grade the moves also makes managing the strategies much more convenient as now they can be implemented as separate functions which share a common interface (() => Unit) and the different parameters required for specific strategies are easily passed

in the functions as the constructor of the class. This implementation will be discussed in more details in the Algorithms Section.

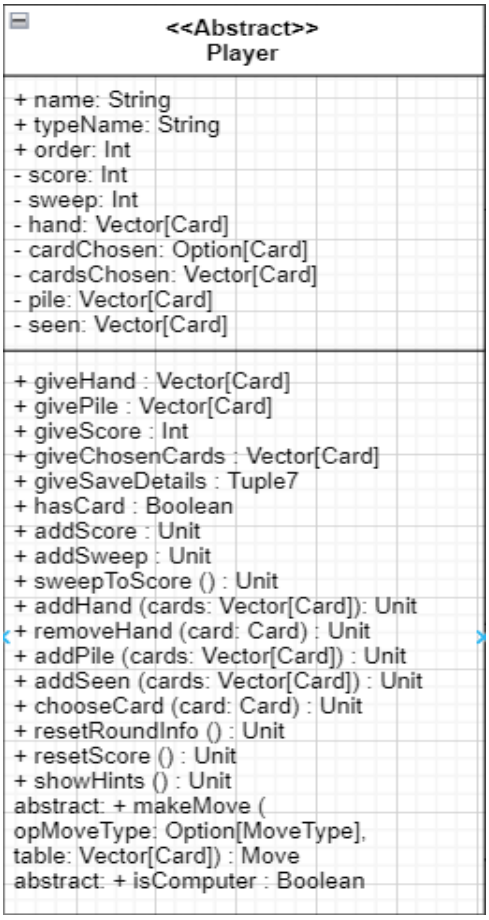
The class will be created with the parameters needed for its strategies, including move : Move, preferences : Map[String,Int], table : Vector[Card], hand : Vector[Card], seen : Vector[Card]. It has 2 private val allStrategies : Map[String, () => Unit] and allPoints : Map[String, Int]. It has 1 public method grade() : Int which returns the final score for the move and several private functions for the strategies (whose name is stored in the allStrategies) and their helper functions.

The UML Diagram for Move package and its specific relationships with Card:



3.2.3 Player package

The Player package includes the files of the Player class and object and 2 subclasses of Player class: HumanPlayer and ComputerPlayer classes with a companion object ComputerPlayer. The Player class is an abstract class with 2 abstract functions makeMove () : Move and isComputer : Boolean which will be defined in the subclasses.



The Player class is implemented to represent a player. Hence, it has all the variables representing the basic properties of a player in the Casino Game, including the name, score, number of sweeps, hand, and pile. In addition to these basic properties, in order to perform the necessary functionalities, some other val and var have been added, including: typeName, which represents the "type" of player such as "human" and different types of "computer"; order, which represents the initial order with which the player is added, from the game perspective it also represents the order of "sitting" used to determine the next player to make a move or the next dealer; cardChosen, which is the card in hand that the player is currently choosing and cardsChosen cards chosen on the table; seen, which is the cards the player have already seen to that point of the round, and its related seenValues (explained in details in Algorithms Section).

As such, all functions in Player class revolve around extracting these values and variables (the "give" function), interpreting and/or packaging them (hasCard, makeMove and giveSaveDetails), and changing them in well-defined manners (for instance, score can only be changed using addScore or resetScore; pile can only be changed using addPile or resetRoundInfo; cardChosen and cardsChosen can be changed using chooseCard or showHints, which automatically chooses cards from all moves found).

This implementation complies with the basic principles of object-oriented programming, encapsulation and abstraction, in that it manages the attributes of the Player instances so that even the variables (often changed) are protected by only predetermined ways of change could be made, and that the logic behind the functions are hidden (resetRoundInfo, makeMove).

The Player class is an abstract class, with 2 abstract functions isComputer and makeMove, It has 2 subclasses ComputerPlayer and HumanPlayer which define these functions. The isComputer function is always true or false for ComputerPlayer or HumanPlayer respectively. The second function makeMove has 2 parameters, with the opMoveType being necessary for HumanPlayer and the table necessary for ComputerPlayer. This function returns a Move.

Apart from the 2 abstract functions, no new functions are implemented in the subclasses. However, the ComputerPlayer class will have one addition attributes preferences : Map[String, Int] which is used for its makeMove function (explain in details in Algorithms Section).

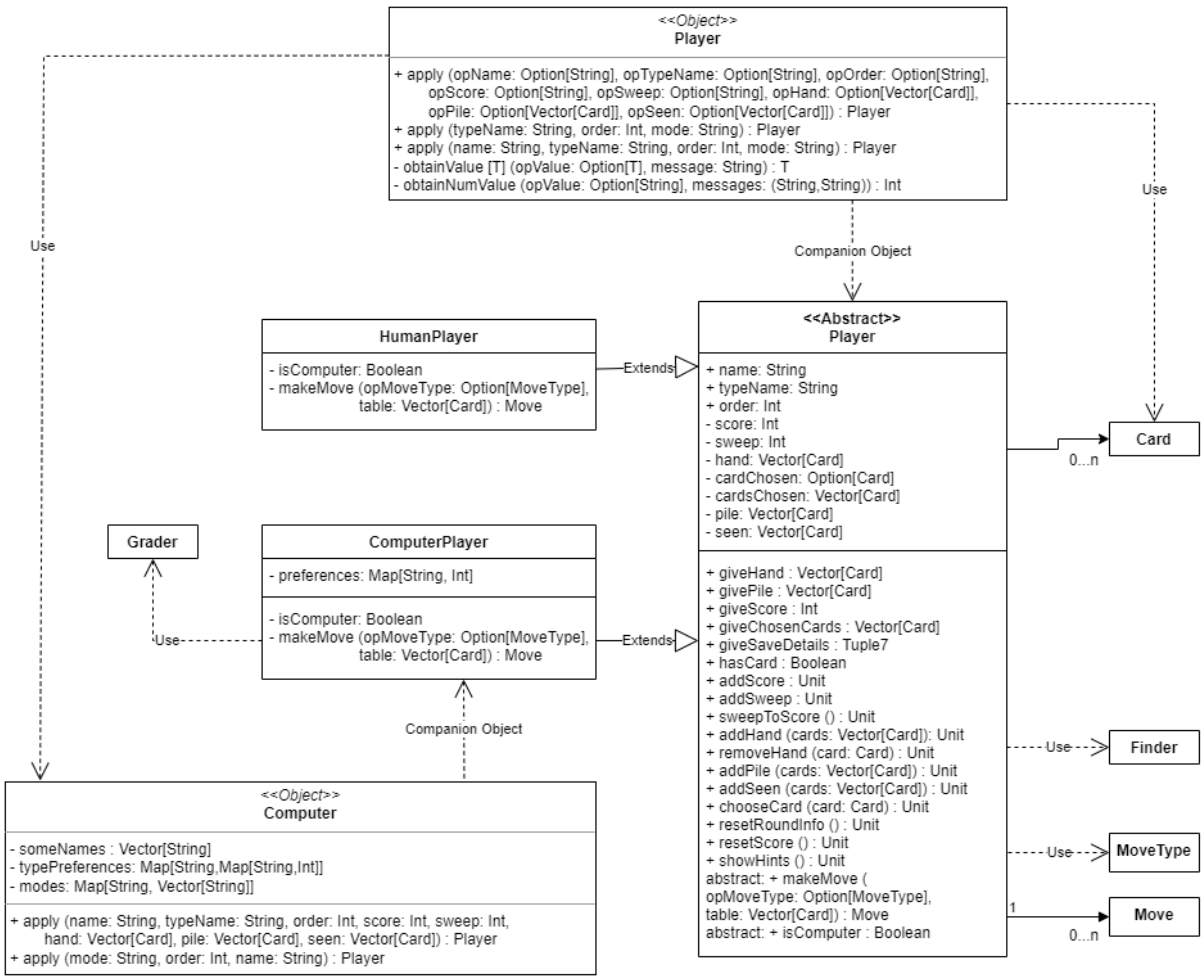
Furthermore, the Player class has a companion object Player, and the ComputerPlayer class also has a companion object ComputerPlayer. These companion objects are also used according to the Factory design pattern, but for slightly different purposes. The Player object is used to create and determine the validity of the information, or parameters for the creation of a Player instance, much like Card object. This process can throw the exceptions NoNameChosen (no name chosen for human player), IllegalPlayerInfo, and MissingPlayerInfo. The ComputerPlayer object is used to create and verify specifically the type of computer

players, used when reading the file, or create a ComputerPlayer instance with a type and preferences stored in the object, based on the mode of computer (e.g. easy, intermediate, hard). The process can throw IllegalPlayerInfo if the type is invalid. Furthermore, the Player object also uses the ComputerPlayer object when creating ComputerPlayer instances.

Similar to the above usage, this method is used to increase the independence of the classes by having their own verification method, and for the ComputerPlayer specifically, it is used to create computer players without specifying their types, which is quite similar to the common usage of Factory Design Pattern.

Hence, the Player object has 3 apply functions, 1 for creation of Player instance from reading file, 2 for creation from new user input (with name and without name). For the ComputerPlayer object, there are 3 private vals someNames : Vector[String] used to generate random names for computer players, typePreferences: Map[String,Map[String,Int]] which maps the types of computer players to their preferences (explained in details in Algorithms Section), and modes: Map[String, Vector[String]] which maps the modes to the types belonging to that mode. The ComputerPlayer object has 2 apply functions, 1 for file reading, and 1 for new user input.

The UML Diagram for Player package and its specific relationships with existing packages:



3.2.4 Round, Game, and FileManager

As the name suggests, the Round class manages the interactions that happen during one round. Therefore, it needs to "know" its table : Table, its cardDeck : CardDeck and players : Vector[Player]. Because all of these aspects are set once the round is started, they are all vals which will be passed on by the Game class to initiate a round. The Round class also keeps track of its private variables, which includes currentTurn: Int, currentView : Int, currentLastCapture : Option[Int] and currentComments : Vector[String]. Current turn, current view and current last capture store the indices of the corresponding players in the Vector of players that are initially passed on by the Game class, as these Int (or Option[Int]) type variables are simple to update and manage.

The functions in Round class revolve around the following tasks:

1. Distributing the cards and updating the variables using defined methods in cardDeck, table, and players;
2. Updating the current variables; and
3. Automatically running the computer players (if any).

Thus, its core, effectful functions are the public functions startRound () : Unit, responseMove (move : Move) : Vector[Card] (this function returns the updated hand of the player who has just finished the move for updating on GUI), endRound () : Unit. It also has 1 private function runComPlayer () : Unit which is accessed within the other functions to automatically run the computer players, and some effect-free functions to access specific values: giveCurrentTurn : Player, giveCurrentView : Player, giveCurrentComments : Vector[String], giveCurrent : Tuple3, and isComplete : Boolean (which checks if the current round is complete).

The Game class has 2 main roles:

1. Managing what happens outside of one round, such as adding player, finding the winner, …
2. Packaging data of its elements (round, player, …) according to the file format.

Therefore, the Game class "knows" its table: Table, cardDeck: CardDeck and manages players: Vector[Player] (private variable players), all of which then will be passed on to Round class. It also keeps track of its currentRound: Option[Round] (the type is Option as there are cases where no round on-going, e,g. score scene / menu scene and add Player scene), and its fileManager: FileManager. The Game also has the current variables in Round class which are used to store loaded data from file and set the variables to the Round class, along with its own current variable currentDealer which are used to determine the vector of players that will be passed on to Round class, with the first player in the vector being the dealer. There are also 2 more private variables, winners : Vector[Player] which stores the players who have won (if any), and started : Boolean (started is true if any round has been started), used to determine whether the game should be saved.

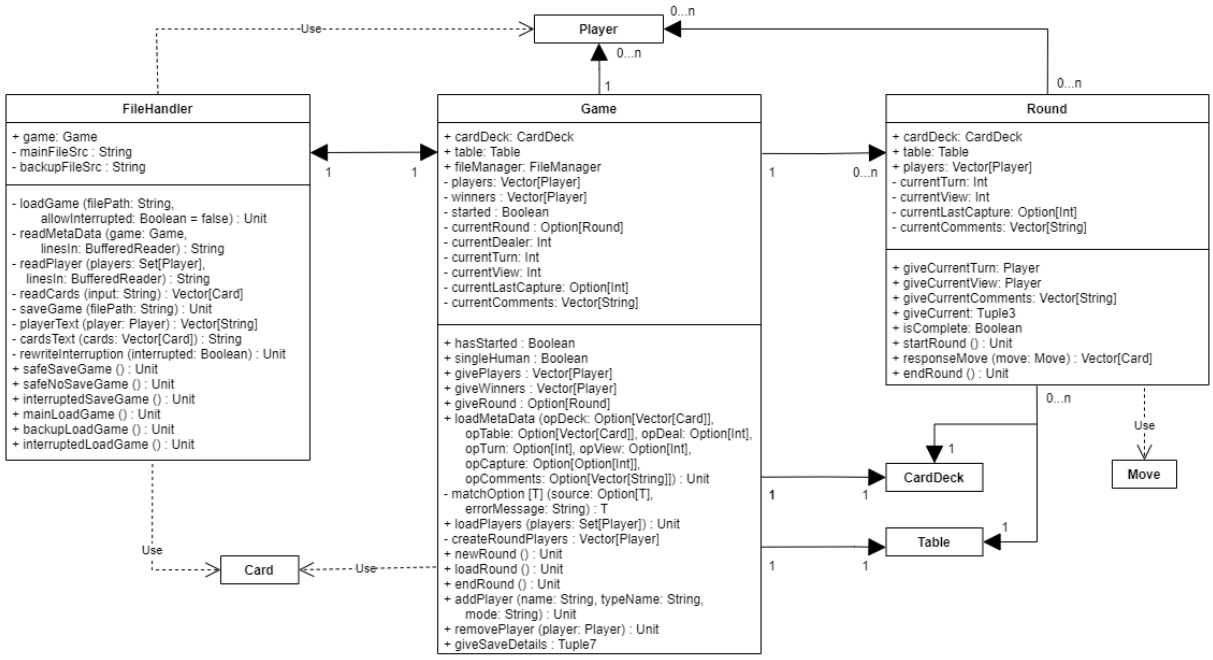
The functions of the Game also revolve around managing the interactions between Round, Player, and FileManager. The core functions of this class are loadMetaData, loadPlayers, newRound, loadRound, endRound, addPlayer and removePlayer. It also grants access to some of its attributes through effect-free "give" functions or their interpretation (singleHuman : Boolean which returns true if there is only one human player). The Game class also has some private helper functions such as createRoundPlayer or matchOption.

The FileManager class is "bounded" 1-1 with the Game class, so that every fileManager is only responsible for 1 game, and the Game instance is the constructor for FileManager class. It is not necessary for the

FileManager class to be bounded to this single Game instance, but as the FileManager class requires information from the Game class, the FileManager would need to be passed a Game instance as parameter for its function if they were not bounded together. Hence, bounding them together would be much more intuitive than passing Game instance as parameter, as the Game class will also manages its fileManger.

As mentioned in the General Description Section, the previous game can be saved and read via various methods (from main file / backup file, interrupted save / load …). These methods will be represented with different public functions of FileManager class, which include: safeSaveGame (if the App was quitted appropriately and the game is chosen to be saved), safeNoSaveGame, interruptedSaveGame (if the App was quitted inappropriately, then the current game, if any, is saved to backup file), mainLoadGame, backupLoadGame, interruptedLoadGame (load from main file allowing interrupted file). The FileManager class also has various private functions to break down its tasks.

The UML Diagram of Round, Game, FileManager class and their relationships with previous classes:



3.3 GUI Package

The GUI Package consists of the CasinoApp Object and its 3 handler classes, GameHandler, SceneHandler, and ExceptionHandler, and one additional package for SceneElements which contains classes that extend Node in scalafx.

The system for this is inspired by the Mediator Design Pattern, in which the CasinoApp is the "mediator" that encompasses the interactions between the handlers, and the 3 handler classes are the "colleagues".

Among the handlers, the GameHandler class is the class that directly interacts with the elements of package GameLogic, passing them user input and extract necessary information. Naturally, the operations of this class can be implemented within the CasinoApp. However, such implementation not only makes the code clustered, but also makes management of those interactions extremely difficult due to the multi-layered structure of the GameLogic Package. For example, if the hand of the current player who has the view needs to be displayed on the screen, the GUI will need to access to that hand by game.giveRound.get.giveCurrentView.giveHand. Creating variables for storing the current elements in CasinoApp is also not desirable as it will increase the interferences between the 2 packages, as well as undermining the core role of the CasinoApp as the mediator of tasks. Therefore, the GameHandler class is implemented with the purpose of interacting with specific elements in the GameLogic Package via the Game instance.

Hence, the GameHandler class has 1 core private variable game : Game. Its functions are either effectful which return Unit type or effect-free which are used for extracting information. It has 1 additional variable hasPrevious : Boolean which is used to determine the notification modes (e.g. if hasPrevious = false, there would be no notification to announce the user who has finished their turn).

On the other hand, the SceneHandler is the class that "stores" the templates (designs) for the scene and sends user input back to the CasinoApp. While the functionalities of this class is very simple, the templates / designs have a considerable amount of detail so "storing" them in a class is much more manageable.

The SceneHandler class has some private variables that are elements (nodes) that needs to be updated regularly such as tablePane, playerHandPane, … and a map cardLabels : Map[Card,CardLabel] that maps a card to its corresponding card labels on the screen, which is used for updating the chosen cards (making them highlighted). These private variables may be updated using the "update" functions of the SceneHandler class. This class has 4 core functions that store templates to the 4 main scenes introduced in the User Interface Section, which are menuScene, addPlayerScene, playerViewScene, and scoreScene. Buttons in the scenes are connected to the corresponding functions in SceneHandler class, which then would send user input back to the CasinoApp.

The SceneHandler also uses the classes in SceneElements Package. These classes in the package are special nodes that are either:

1. Updated regularly (e,g, tablePane, playerHandPane, …), or
2. Tied to a specific GameLogic element (e.g. cardLabel, playerPane, …)

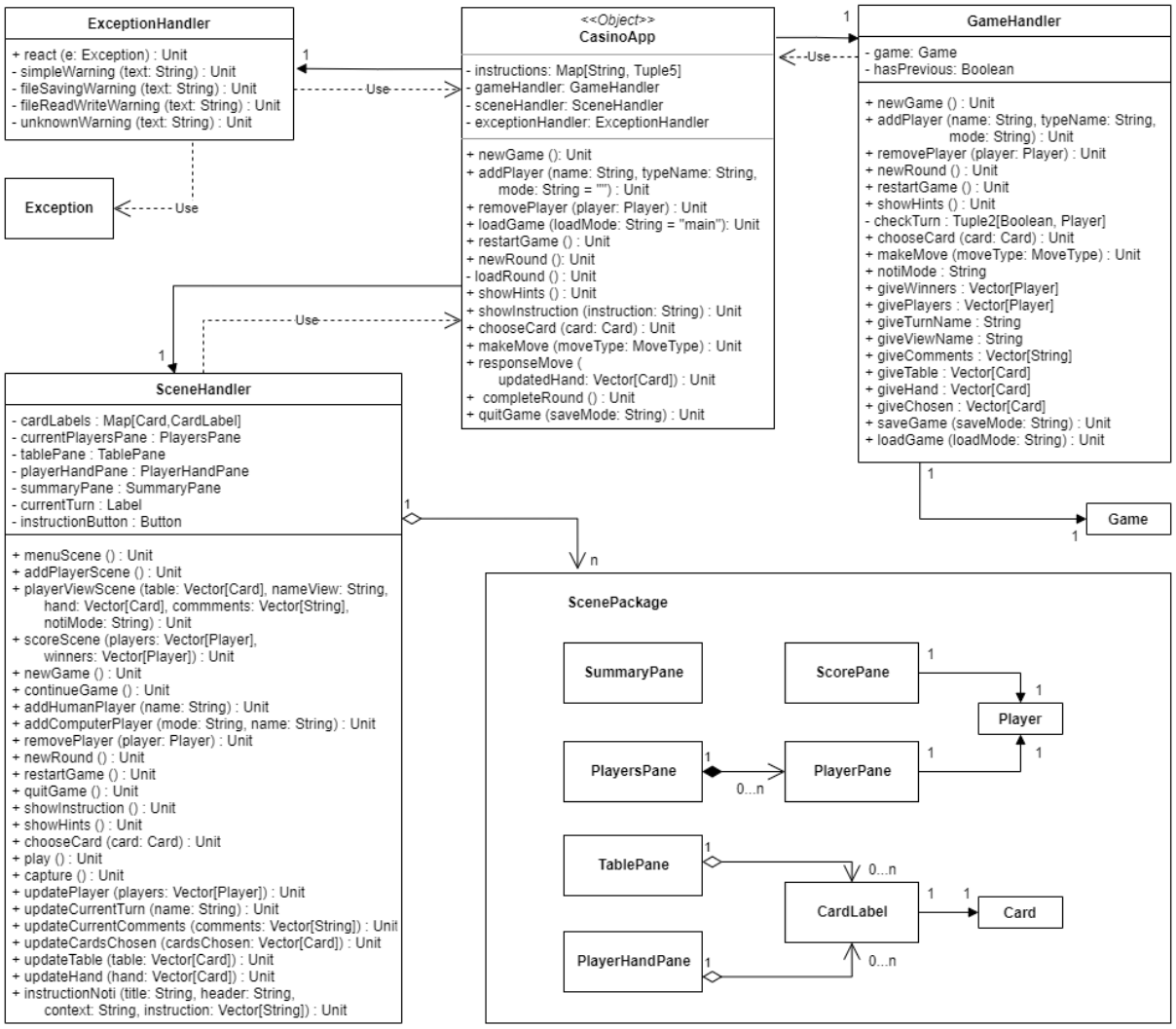
For the first case, these classes are created and separated from SceneHandler templates to prepare for future implementation, such as animation or thread. For the second case, these classes are created to display some attributes of the element (playerScorePane) or send back to the SceneHandler class the specific element that the user chose, which the SceneHandler will eventually send back to CasinoApp. While I try to avoid unnecessary interference between GUI and GameLogic, such implementation is necessary as otherwise, some functionalities would not work appropriately, e.g. choosing a card / removing a player.

Lastly, the ExceptionHandler is the class that handles the exceptions that occur at runtime. It has only 1 public function which requires an exception as a parameter and 4 private functions to react to different types of exceptions. When an exception occurred and passed on to the ExceptionHandler, it creates dialogs to inform and if necessary asks for user inputs to react to the current situation (e.g. unexpected error / file

reading error). This class is created to outsource the exception handling from the CasinoApp for easy management and further highlights its role as the mediator between handlers.

The CasinoApp in this system acts as the mediator between the handlers, with 3 private val for 3 handlers. Its functions correspond with actions from user input (e.g. start a game, start a round, make a move, …). These functions involves commanding the GameHandler to interacts with GameLogic elements with user input, updating the SceneHandler, and sending any exceptions to the ExceptionHandler. Additionally, there is a private val instructions which is a simple storage of the instructions to use the CasinoApp.

The UML Diagram of GUI Package and its relationships with other packages:



3.4 Conclusions

Hence, this program structure consists of classes and objects with clear, specific goal. The program has a "hierarchical" multi-layered structure (especially in the GameLogic Package) in which the "smaller" classes are managed by the "bigger" classes in the hierarchy, and these "smaller" classes are relatively independent

with one another and independent of the "bigger" classes, which is demonstrated by the packaging and ordered introduction of these packages. This structure is supported by the use of customized exceptions in the program and design patterns Factory Design Pattern and Mediator Pattern.

The program structure does have some deviations from the planned structured, namely the addition of Grader and Finder class in Move Package and the SceneElements in GUI Package. Such additions are implemented to further improve the independency of the classes and accommodate new algorithms. The case classes as subclasses of Card are also simplified.

4 Algorithms

The most complicated algorithms in the program involves the Move Package and ComputerPlayer class. Specifically, those algorithms are:

- Algorithm to verify the moves, implemented in Move object (1)
- Algorithm to find all the moves, implemented in Finder class (2)
- Algorithm to make move for computer players, implemented in ComputerPlayer class and Grader class. (3)

However, as the choices of algorithms are vast, I will have to consider many algorithms before deciding the most suitable algorithms based on predetermined criteria. In particular,

For (1), I would prioritize:

1. **Functionality:** which meant the algorithm must be able to perfectly determine valid and invalid moves,
2. **Efficiency:** which meant the algorithm must be able to perform with reasonable time cost.

For (2), the criteria would be the same as (1), with an addition of **manageability** when implementing, which meant the implementation should break down the algorithm into smaller parts for easy management.

For (3),

1. **Functionality** would again be prioritized, which meant the computer player should make reasonable moves
2. **Extensibility**: which meant the system should be versatile and the process of developing new types of computer players and modifying of existing ones should be simplified.
3. **Efficiency**: which meant the algorithm must be able to perform with reasonable time cost.

With these predetermined criteria, I would choose the most suitable algorithms for these tasks.

4.1 Verifying moves

There are many errors that could be detected when verifying a move, such as NoCardInHandChosen or WrongMoveButton. But the most complicated algorithm would be to verify a valid capture. There are two approaches to implement this verification method. The first one is to check if the chosen cards satisfy some specific conditions, and if all such conditions are satisfied, the move is valid. The second one is to find all

possible moves using the chosen cards, then check if there is a move that can capture all the chosen cards on table.

As a matter of fact, the first approach is much more efficient than the second approach. However, as I tried various conditions with this approach, I have yet to find the conditions to create a perfect functioning algorithm for this purpose.

The algorithm for this approach works as following:

1. To initiate the verification, first filter out the cards whose values are equal to the value in hand of the card played.
2. If there are no remaining cards, then the move is valid.
3. If there are remaining cards, then those cards must be composed of the non-overlapping subsets whose sums are exactly the value in hand of the card played.

I tried using conditions to check this, but all of these conditions are not without fault:

Originally I planned to check if the sum of all the remaining cards is divisible by the value in hand. Additionally, this simple condition can be combined with the condition that the maximum value of the cards captured must not exceed the value in hand of card played. While this can rule out the obvious invalid moves, these conditions will fail to detect the invalid move as it fails to verify if the subsets of that move sum up to exactly the value in hand of the card played. For example, these conditions will fail to detect invalid moves such as card Diamond-9 played to capture cards Diamond-8, Heart-2 and Spade-8.

Therefore, I needed a more powerful condition to verify the move. Using the above conditions as an initial filtering, this condition will verify the remaining moves that passed this initial checkpoint. The condition I tried was based on the reasoning that if the cards captured are composed of non-overlapping subsets whose sum are exactly the value in hand of card played, then the sum of the values of cards captured divided by the value in hand of card played must be the number of those subsets. Then, I used the algorithm that counts the number of non-overlapping subarrays from GeeksForGeeks to check this condition. This condition works perfectly for detecting invalid captures.

However, the critical downside of this condition is that it will also mislabel some valid moves as invalid. Because it uses the algorithm to find number of non-overlapping subarrays, then if the cards captured are not chosen in the exact order that will form those subarrays, then the count would not meet the expected count and thus the move is marked as invalid. For instance, if cards captured are chosen in the order card Diamond-7, Spade-2, Heart-8, Ace of Club with the card played Heart-9, then the move is valid via this condition, but if the cards captured are chosen in the order Diamond-7, Heart-8, Ace of Club, Spade-2, then the move is invalid via this condition.

While this downside may not pose a significant to the game progression (no invalid move is made, and player can choose the cards again until they are in an appropriate order), per the predetermined criteria, this algorithm must prioritize the functionality over efficiency and therefore, this approach is not chosen.

Nevertheless, the simple conditions (the first 2 above mentioned conditions) and the initiation that filters out same value cards are still kept as a means to increase efficiency. If these conditions are passed, the move is checked by finding all possible moves using the card in hand and the cards on table chosen, then check if there exists a move that can capture all the cards on table chosen. Thus, the final algorithm will use the finding all moves algorithm.

4.2 Finding moves

Initially, the finding moves algorithm was only intended to be used for the computer players and also has 2 approaches. One of them is similar to the conditions approach above, meaning that I would try to find collections of cards that are composed of non-overlapping subsets whose sums are exactly the value in hand of the card played. However, as I could not find the conditions effective enough to validate the moves, this approach is not chosen.

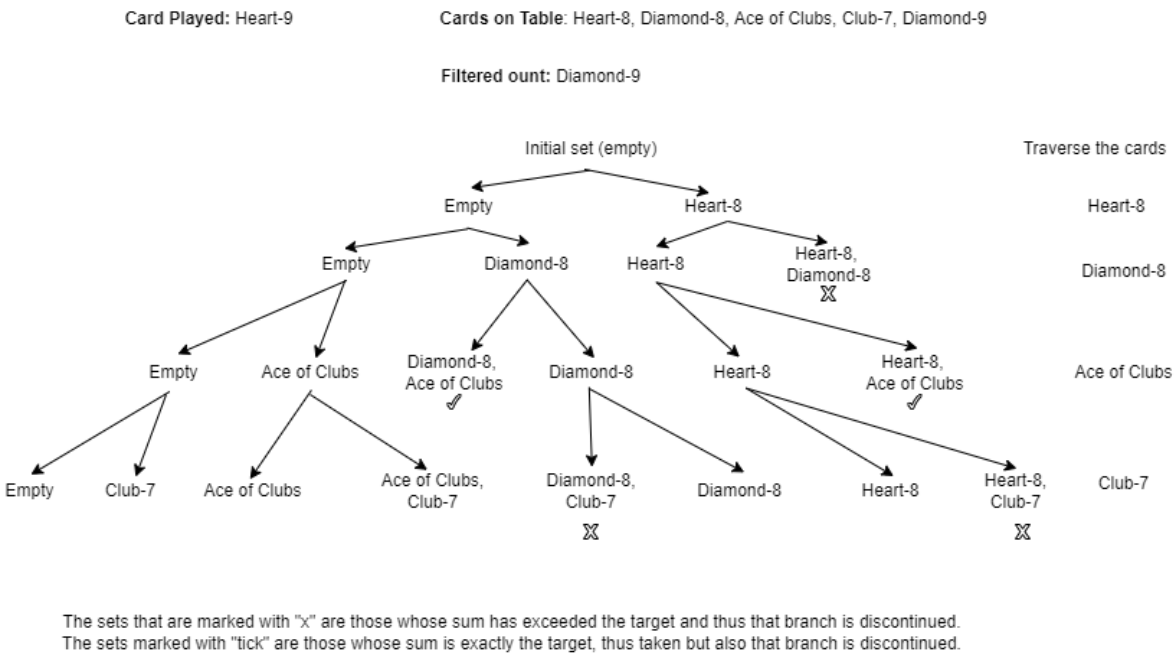
The second approach is more in a bottom up manner, that is, rather than finding possible combinations of cards that is composed of such subsets, the algorithm will find the suitable subsets and then stack these subsets together. Thus, this is the core principle of the used algorithm. To summarize:

- This algorithm will be passed in a collection of cards to be captured and a collection of cards to play.
- This algorithm is divided into 2 main tasks: finding all the subsets from the collection of cards to be captured whose sum is the exact value in hand of a specific card from the collection of cards to play, called **component captures**, and then stacking them together.
- Only **non-overlapping** components would be stacked together.
- The algorithm does not find the longest stacked collection of cards, but **multi-layered stacks**, e.g. from 1 component to n components stacked together.
- The algorithm will use the found stacked collections of captured cards for each specific card to play to create moves, then combine them with play moves, and finally return a collection of all moves from the passed-in collections.

4.2.1 Finding component captures

A **component capture** (used in program code) is a subset of cards on the table whose sum is exactly the value in hand of card played. The current used algorithm is inspired from the algorithm to print all subsets with a give sum in GeeksforGeeks. This algorithm is relatively efficient for a table of under 25 cards, and as such number is very difficult to reach, this algorithm is currently efficient enough for the game. The algorithm is also relatively simple to implement.

This algorithm is based on the concept of subsets, or that the subset of a set A can be represented by a sequence of 0s and 1s, with 1 at ith position meaning that the element at the position ith in the set A is in the subset. As such, this function will find all subsets by traversing the set and for each element, considering 2 paths: if the element is included and if not. If the element is included, then the target sum is subtracted by the value of that card element, and that element is added in the current subset. The execution halts if the target sum is 0, meaning that the sum of the current subset is exactly the value in hand of the card played, or if the final element is reached, or if the target sum is smaller than 0, meaning that the sum of the current subset has exceeded the target value. A simple diagram to visualize this algorithm would be:



However, while this algorithm is chosen as it is relatively simple but effective, it is not the most efficient and thus I may consider more complex algorithms in the future.

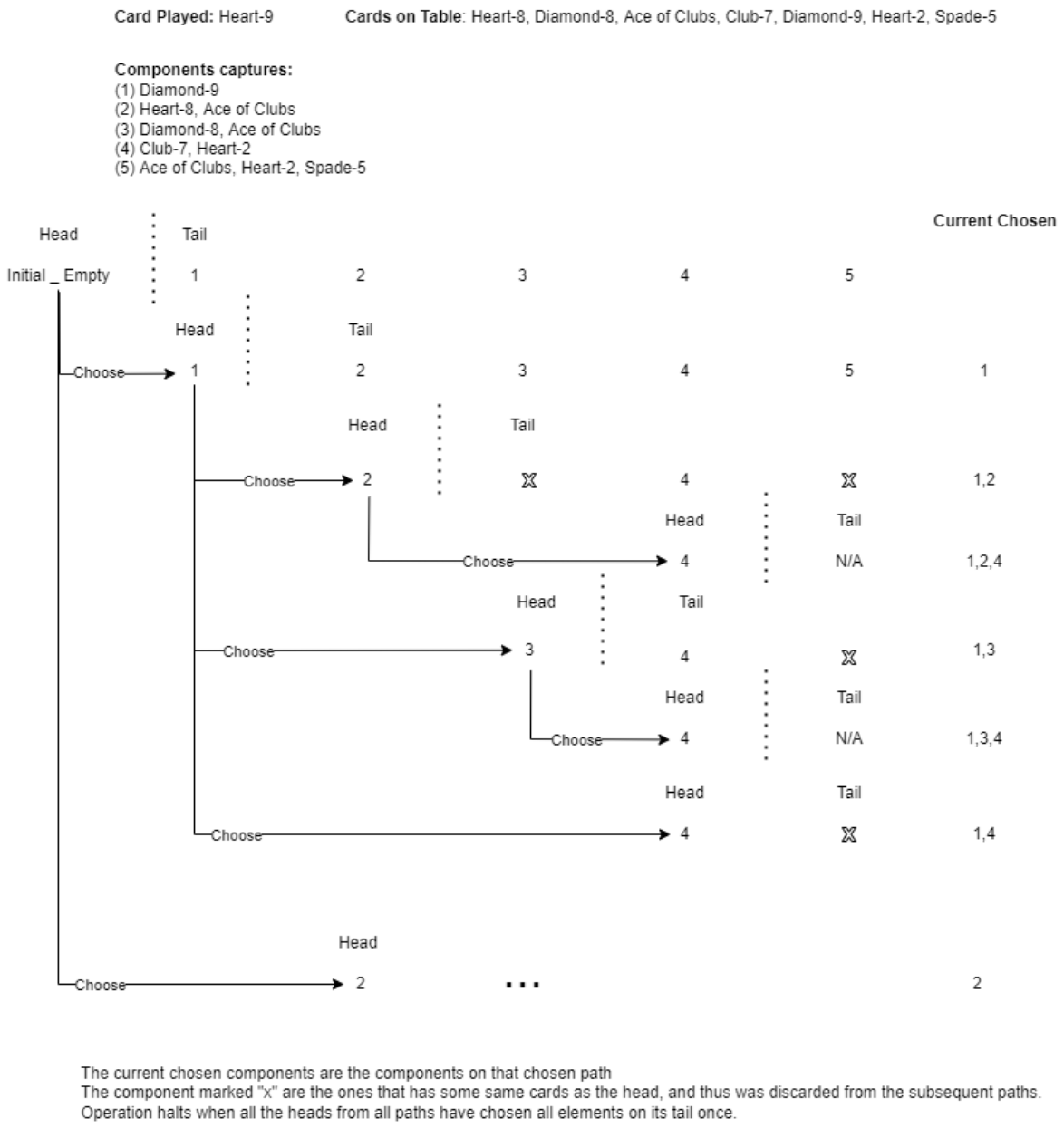
4.2.2 Stacking component captures

Again, the concept of subsets could be utilized here. However, I find that this approach would not work efficiently for stacking components. This is due to the fact that the base condition, target sum == 0 of the previous algorithm is very simple and can be passed on easily for subsequent recursive calls. This is not the case for this algorithm, as the condition to update the newly stacked subsets is that they must not be overlapping, which is known via checking the number of distinct elements. This condition in itself is rather costly, especially with higher number of cards.

Therefore, the algorithm I chose to implement is more like a problem of drawing edges of a complete graph. In a complete graph, there is an edge between any two nodes, hence, if I order the nodes in a sequence, I can find a path from the first node to the final node by recursively choosing an edge between the “head” node and one of the “tail” nodes. Hence, let the component captures be nodes. As we traverse the component captures, recursively choosing each of them as the “head” node and the remaining (not the “head” node and yet to be traversed) as the “tail”, we can obtain all the possible stacked collections of captured cards.

This algorithm performs better as it can actually filter the “tail” nodes before continue to recursively traverse them. The “tail” nodes are checked if they overlap with the current “head” node and thus the overlapping component would be filtered out, and all subsequent calls in this chain will use this filtered “tail”. As the component captures are relatively small in size, this can be checked with rather low time cost.

A simple diagram to visualize this algorithm would be:



I already made an efficiency test for the Finder class which checks different implementations for this class and compares the time complexity of those implementation. I tested the performance for both of these algorithms and chosen the second one which is based on the "head" and "tail" nodes. While for smaller number of cards, (from 0 to 9), the first algorithm would have lower time cost, for higher number of cards (12-18) the second algorithm performs significantly better than the first and for the number of cards >= 19, the first algorithm almost comes to a standstill (In one test which searches for moves 10 times with 19 cards on table and 4 cards in hand, the first algorithm ran for 33 minutes, while the second algorithm run for only 20 seconds). Admittedly, most of the game would involve dealing with small number of cards, reaching those higher number of cards (12-25) is still possible, especially with strategies such as build play.

Additionally, for the smaller number of cards, the difference in their performances in quite insignificant (only around an average of 4 milliseconds).

4.3 Making moves for computer players

As a matter of fact, to decide suitable moves, the computer player should have some "strategies", for example, avoiding easy sweeps, building, prioritizing special cards, … All of the reasoning in this Section would be made based on this predicate.

As the finding all moves algorithm is already implemented, the computer player would only need to decide which move should be chosen. Per the criteria, the algorithm used for this should:

1. Be able to systematically choose the best move.
2. Allow easy implementation of new types of computer players (different behaviors) and modification of existing ones.
3. Perform with reasonable cost.

The current implementation is based on the idea of a grading system. The algorithm would now choose the best move by first finding all possible moves, then grading them and finally the move with the most points is chosen. This points could be positive or negative (e.g. all moves can lead to easy sweep, but the algorithm would choose the one that is least likely to).

The grading system functions as following: All the strategies used in this system would be stored in the `Grader` class, which would need a `Move` instance, preferences, table, hand, seen, … to be passed in as its constructors. For every `Move`, a `Grader` instance is created for that move and passes the `Move` through all necessary strategies while adding up its points. The total points of a `Move` would be returned at the end of the system.

This implementation is chosen mainly based on the first 2 criteria, then, it is improved to satisfy the third.

4.3.1 Functionality

The grading system provides an intuitive, systematic and strategic method to decide the best move. In particular, it allows computer player to:

- **Measure how "fitting" one move is to a strategy:** Compared to other methods such as filtering out the unsuitable moves or choosing arbitrary moves from a collections of move, the grading system does not classify the move as fit or unfit, but measure how "fitting" it is to a strategy. For example, if the strategy in question is prioritizing special cards (Aces, Spade-2, Diamond-10), then obviously the move that captures 2 Aces or Diamond-10 will be more valuable and more "fitting" to the strategy than only capturing 1 Ace or Spade-2. A classifier approach cannot make the difference between these moves, but the grading system can and will award points correspondingly

- **Combining different strategies systematically:** Let us compare with a different approach: the computer player decides the moves solely based on some strategies in its function, or perhaps the `ComputerPlayer` class has many subclasses which have different `makeMove` function. While this is feasible, this approach would result in 2 scenarios: either the function itself is based on 1 strategy (which is obviously undesirable and in some cases is not even sensible, e.g. build play strategy), or

that repetitions occur across the functions of these subclasses. Meanwhile, the grading system could easily solve this combination challenge. As all the computer players share a common grading system, each move can be passed through all or some of the strategies and the results from these strategies are combined together at the final stage, creating a final point for the move. This implementation will allow simple combination of strategies as well as avoiding repetition in the program code.

- **Strategic combinations:** Let us combine the previously mentioned other approaches. If a specific type of computer player has 2 or more strategies, and if a move that performs exceptionally well with one strategy and badly with another, then this approach could not make the most suitable decision. For example, if capture move can successfully capture special cards, but will leave easy sweep options on the table, naturally this move is still more beneficial as it immediately grant the player score(s), while the next player may not even have the card(s) used to make a sweep. However, if the previously described approach is taken, then the move would be classified as unsuitable via one strategy and suitable via another, hence, the program would need to consider each and every combination from these results to make the final decision (e.g., if had special cards, choose the move, else if did not leave easy sweeps, …). Meanwhile, the grading system will measure how "fitting" one move is to a strategy and combine different results by simply adding them together. With the previous move, for example, the grading system will +300 points for each special card and -50 points for every cards that can capture all the remaining cards on table and make a sweep (for instance, 4 such cards can leave a -200 points). That may leaves the move with a total of around +100 points, which is desirable and still more than e.g. the priorMoreCardsstrategy which will award +10 points for every card captured.

Thus, the grading system is chosen as for the algorithm to make moves for computer players as it provides such an intuitive, systematic and strategic method to measure how well a move will perform based on its strategies.

4.3.2 Extensibility

As described in the previous part, the grading system allows combinations of results from different strategies by adding these results together, creating a final point. As such, it can viewed as a linear system, in which the total point is calculated by:

$$y = \sum_{i=1}^{n} \beta_i x_i$$

With y being the final point, n being the total number of strategies, x_i being the resulting measurement of how "fitting" the move is with the i^{th} strategy, and β_i is the "weight" of that strategy. For instance, with the basic grading system, the measurements are added together creating a final point, which means that the "weight" of all strategies are equal to 1 (results added once).

As such, changing these "weights" will result in different final points for the same move, and thus, generating different "behaviors". Therefore, it can be claimed that this system can create infinitely many behaviors just by changing its "weights", which would also mean that using this system, infinitely many

types of computer players can be created just by assigning each type to a different set of "weights", or, as in the implementation, **preferences** of a type.

Therefore, this system is far more superior to the usual use of subclasses in terms of extensibility. While sharing a common grading system, the computer players can have different behaviors just by changing their preferences towards the strategy. Using this system, the program currently has up to 11 different types of computer players, each representing some different ways to change the "weights" and the number only stops there as I could not test out all the behaviors of these types. Out of these 11 types, the "special computer" and the "spade computer" are the most competent players, these computer players "know" all the strategies and give heavy weights to the strategies prioritizing special cards or spade cards. Some other types such as "fair computer" (equal weights), "lookahead computer" (prefer build play) or "more computer" (prefer more cards captured strategy) have less noteworthy but relatively stable performances. Some types have very special behaviors and effects on the game, such as the "careful computer" and the "bold computer". The "careful computer" gives heavy weight for strategies such as avoiding power play or avoiding easy sweeps, and thus makes gaining points very difficult for other players, especially its next-in-turn player. On the contrary, the "bold computer" gives less weights for those strategies and thus its performances fluctuate significantly based on the current situation. Other types of computer players including "dumb computer" or "random computer" are also added to make the game easier.

One additional advantage of this system is that as the types of computer players are not subclasses but rather a set of weights, these sets can be stored and assigned a type name, which can be easily accessed and grouped if necessary. This allows me to conveniently develop the "mode" feature with which the user can choose the modes of the computer players. The default mode is Random which chooses a type name from all types, then in the Easy mode there are "dumb computer" and "random computer", … For each mode, the program will randomly choose a type name from those classified as within that mode, and then using the type name, the program would find the set assigned with that name. This creation algorithm is in the ComputerPlayer object, along with the available types and preferences.

4.3.3 Efficiency

However, this system will admittedly function with higher cost than the other approaches mentioned above. Nevertheless, per the predetermined criteria and the various benefits that this system provides, I would choose this system and attempt to optimize it instead.

Per the previous description, one major cause of significant time cost of this system is that it has to compute all the measurements for each strategy, while in some cases this might not be necessary and the computation of these unnecessary strategies may be quite costly. Furthermore, while in principle the preferences should specify all weights for different strategies, this could be quite inconvenient with the growing number of strategies. Therefore, some adjustments are made to improve the implementation of this algorithm.

To effectively access the strategies functions and assign them to different weights, the most intuitive method is to use maps and keywords. Thus, each strategy is assigned a keyword, and the computer player will also map these keywords to the strategy weights when passing the preferences to the Grader class. To achieve both of the aforementioned goals, I introduced a new, different keyword "all", and also create "strategy points" using the keywords for each strategy, rather than updating the computed measurement and weight directly to the total points variable.

The keyword "all" stands for using all strategies, and it also has weight associated with it as a means to regulate the importance of a preferred strategy. In the case this keyword is included, all the strategies would

be computed once. Then, for specific strategies (if any) that has additional weights, the resulting measurement for these strategies would be extracted from the "strategy points", multiplied with the additional weights and added back. The total points would be the sum of these "strategy points".

In the case the keyword "all" is not included in the preferences, only the strategies that is included in the preferences would be computed. As such, if the strategy is not in the preferences, the process of computing that unnecessary strategy, and then multiplying it with 0 would be emitted. Hence, in this case the time cost of the system would be closer to the time cost of the previously mentioned approaches which also only compute the strategies stored in the internal functions of subclasses.

4.3.4 Some strategies

The Grader class currently has up to 10 different strategies which diverse levels of complexity. Some are extraordinarily simple, such as priorRandomStrategy (which gives random points to the moves), or priorCaptureStrategy (which gives 1 additional point for capture moves). Some requires more logic such as priorMoreSpadesStrategy (which counts the number of captured Spade cards and gives point accordingly), or priorSpecialCardsStrategy (which traverse the collection of captured cards combined with card played and use pattern-matching to find the special cards). However, the most complicated strategies of these are the priorBuildPlayStrategy, which plays a card to build possible capture moves for some next turns, and the avoidEasySweepStrategy, which gives penalty if the move can lead to easy sweep options.

In real-life situations, a build play move is to play 1 or more cards in order to create a capture move using them and the existing cards on the table. Therefore, the core principle of the priorBuildPlayStrategy is to hypothetically play a card, then consider the possible moves with that played card on the table and the remaining cards on hand. These moves, naturally will also include the next play move and its "build potential", thus creating a recursive chain until no more move is found (all cards have been hypothetically played). The current grading system allows quite simple implementation of this strategy. For every play move, a hypothetical table and hand are created which represent the table and hand after the move, then the algorithm would use the Finder class to find all possible moves, and for each of these moves, create a Grader for them. This Grader, however, would only consider some specific strategies (with a function called halfGrade() : Int). This is because some strategies would be not reasonable to use in this hypothetical situation such as the priorSweepStrategy, or the avoidSweepStrategy, … The Grader gives the points to these hypothetical moves and the move with maximum point is taken into account. The points gained from this strategy will be the maximum point after being applied a fixed penalty (as it is only hypothetical).

For the strategy to avoid easy sweeps, however, the principle is more complicated, as it is quite difficult to determine what combinations of cards on the table would lead to easy sweep options. Naturally, these conclusions can be drawn from some assumptions, for example, if there are many low value cards on the table then it is more possible to have a sweep. However, while these assumptions maybe somewhat sensible, their accuracy fluctuate significantly in different scenarios. Therefore, the algorithm I chose would give penalties to moves with easy sweep options according to the relative possibility of those easy sweeps using the concept of "**seen values**". In particular, the algorithm functions based on the following:

- A **seen card** is a card that in a "human" sense, a player has seen. Those are the cards that has been on the player's hand, or the table, or the cards that other players have played. A **seen value** is the value in hand of a seen card.
- A **seen card** cannot be used by other players to capture the cards on the table.

- A card can capture other cards based on its value in hand. Hence, if a card can capture one collection of card, then a card with similar value in hand should also be able to capture that collection.

Based on these identities and arguments, it is safe to assume that the more cards of a value in hand are seen, the less likely another card with that value in hand can be used to capture in the next turn and make a sweep. With this, the challenge can be solved by first finding the possible values in hand of the cards that could make a sweep with the remaining (hypothetical) table, then giving penalties based on how likely such values in hand can be used (based on how many cards of that value is left unseen). The first step can easily be completed by using the `standardCards` collection from the `Card` object which contains all cards with all values in hand (from 2 to 16), and try to find a move from each of them that would capture all the remaining cards on the table (quite similar to the strong verification in the `Move` object). If such move exists, the values in hand of such cards would be added to a collection and form the "sweep values". Then, for each of the sweep values, check how possible that value could be used by checking how many cards left with that value which remain unseen and give penalties accordingly. This is how the current implementation solve the problem of avoiding easy sweeps.

5 Data Structure

The current program makes use of 3 types of data structures: `Vector`, `Map` and `Set`. `Vector` is used predominantly across the program, while `Map` and `Set` are used situationally.

5.1 Vector

The data structure `Vector` is used predominantly throughout the program because of its immutability and its nature as indexed sequence.

- **Immutability**: As previously described in the program structure, the program is consisted of various classes and objects which must function based on several frequently updated variables, yet must also be able to communicate and allow access to those variables (e.g. players, cards on hand, cards on table, …). Therefore, an immutable data structure is desirable for the goal of transferring such data across the program. With an immutable structure as `Vector`, if such a collection is transferred, we can ensure that the original collection would not be changed by external classes. This choice of data structure protects the independency of each class by limiting their interference with internal states of others, and thus makes the program much more maintainable and comply with the basic principle abstraction of object-oriented programming.

- **Sequence:** The nature of a sequence of the data structure `Vector` means that whenever the variable that stores a `Vector` is changed, the change would be predictable in terms of the position of the changed element(s). For example, when adding a card to the hand, the variable hand of the player is changed to a `Vector` that is composed of the previous `Vector` and the newly added card, with this card added at the end of the sequence. Or for the `CardDeck` class, the internal cards in deck variable would only be shuffled once and then when the `drawACard` function is called, the "head" of this collection is returned when variable will store the "tail" of the collection. Such predictability is quite convenient for management or testing, while also provides some additional benefits for the

gameplay. For instance, the variable currentComments of the round is a collection of separate lines of comments that should be added in a manner such that the new comments should always be at the end of the sequence; or when adding cards to the table or hand, intuitively, the newly added card (s) should be at the end of the collection for display on the screen.

- **Indexed elements:** Indexed elements are crucial for some functionalities. For instance, the variables currentDealer, currentTurn and currentView function based on the indexed elements of Vector. These variables do not store specific players, but their indices in the player Vector. This allows easy updating of these values and extracting the corresponding players. Furthermore, the indexed sequence nature also proves very beneficial when traversing the elements and splitting collection into "head" and "tail" for the stacking system in the finding all moves algorithm.

Moreover, the data structure Vector is chosen rather than List which also has these qualities because its performance takes effectively constant time for all the necessary operations based on the Scala documentation about performance characteristics of collections.

5.2 Map

Both mutable and immutable Maps are used in the program. Map is used because it provides an intuitive method for binding values to simple keywords, which would then simplify the process of communication between classes (rather than specific values, only keywords would be transferred) while also keeping the level of abstraction (the class that commands a certain class with keywords does not "know" or has access to the specific values). Additionally, values from Map can be looked up and updated with reasonable time.

One use case is the Map used for storing the preferences of a computer player which maps the keyword for that strategy to a weight, and for storing the keyword and the name of the strategy function in the Grader class. When a Grader class is passed on the preferences, it would combine the strategies and the weight using these keywords.

Another use case is the Map for storing modes of computer players and the type name of the specific types of computers. The mode is obtained from the user input, and then the ComputerPlayer object (factory) will decide randomly from the pool of types bound with this mode.

5.3 Set

Mutable Set is used in this program. As a Set is mutable, it is not a safe method for transferring collections across the classes, but its mutability is vital for some algorithms or functions.

For example, the Finder class whose single goal is to find and return all possible moves would benefit considerably from using a mutable collection set throughout its recursive smaller algorithms. The approach using immutable data structures would be rather inefficient in this case, as that current results of these recursive functions would need to be passed on throughout the recursive calls, extracted, stored and combined together. Therefore, Set is used to store component captures and all captures for the Finder class. On additional benefit of this usage is that the Set data structure also ensures that there are no duplicate elements, which is important for this algorithm.

Another use case for Set is for the reading file function of the FileManage. A Set is used to store the read players before loading all of them into the Game instance. This "pasting" manner of loading ensure that the initial loading would not affect the subsequent loadings should the need arise (e.g. bug would occur if players are added to the Game 1 by 1, and if an error is detected after that, subsequent calls for loading would have abnormal number of players). However, the function to read information about a single player should be separate from the main function to break down the task, and a global variable for storing this collection in the FileManager seems quite unreasonable. Therefore, a mutable Set which is passed on the reading player information function would effectively solve this challenge.

5.4 Tuple and other Data representation

Apart from the aforementioned data structures, the program also makes use of tuples as they are very convenient for packaging different types of data together to transfer these necessary data for a function of another class. One notable usage of tuples is for packaging the information that should be saved for file saving function.

Additionally, there are some different data representations used in the program: Card class and Move class. These data representations package the data in a well-defined manner and to be transferred across the program.

6 Files

The used files in this program are text files with format inspired from the exercise HumanWritableIO in the course. Thus, the format is relatively easy to understand and read. In particular, a sample file would be:

```
CASINO Save file

#game metadata
CardsInDeck: s7hjd2s8c8s6d8c9hqcjdqcqhkdac3c7d6s9d5djsah9d3h0h5had4c0s5c6sjskd9s0
CardsOnTable: h6h2ca
CurrentDealer: 0
CurrentTurn: 0
CurrentView: 0
CurrentLastCapture: 2

#comments
A new round has started!
me is the dealer of this round.
Initial cards on table are Heart-6, Spade-4, Heart-2, Spade-3.
Player Zuru** played Ace of Clubs.
Player Tom45* played Diamond-7 and captured Spade-3, Spade-4.

#player
Name: me
TypeName: human
Order: 0
Score: 0
Sweep: 0
Hand: c4c2s2h7
Pile:
Seen: c4c2s2h7h6s4h2s3cad7

#player
Name: Zuru**
TypeName: lookahead computer
```

```
Order: 1
Score: 0
Sweep: 0
Hand: d0dksqh4
Pile:
Seen: d0dksqcah6s4h2s3h4d7

#player
Name: Tom45*
TypeName: random computer
Order: 2
Score: 0
Sweep: 0
Hand: c5h8h3ck
Pile: d7s3s4
Seen: c5h8d7h3h6s4h2s3cack
```

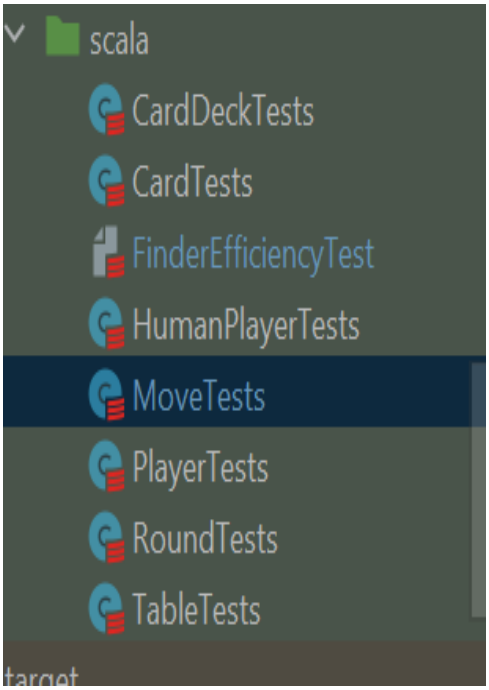
Some notes about the file reading and saving are:

- The accepted file format would always start with “CASINO Save file”
- The second line could be an empty line or “Interrupted” in the main file if the previous game was quitted inappropriately.
- The file is read in “chunks” of data, with these chunks being either the game metadata chunk, a comment chunk or a player chunk.
- The current dealer, current turn and current view always have some Int values, while the current last capture could be None, denoted in the file as “currentLastCapture: ”
- The current comments are composed of separate lines of comments.
- Various conditions would be checked when reading a file, such as missing chunks, missing meta data, abnormal number of players, invalid player types, missing cards, …
- Changes from the planned file format are the addition of “Order” attribute for players and a separate comment block.

7 Testing

The testing for this program does not deviate much from the project plan. The fundamental smaller classes (Card, Move, CardDeck, Round, …) are tested using unit tests, while Game, FileManager, ComputerPlayer and GUI are tested using manual tests.

7.1 Unit Tests



Several unit tests have been written to test almost all the functionalities of the fundamental classes and objects. These tests include:

- CardTests which tested the creation of an instance of Card;
- CardDeckTests and TableTests to test all the functionalities of these 2 classes;
- MoveTests to test the creation and verification of a Move instance;
- PlayerTests to test the creation of a player instance, extracting information from this class and changing its variables using defined functions;
- HumanPlayerTests to test the functionality to make a Move using chosen cards;
- FinderEfficiencyTest to test the efficiency test of several implementation for Finder class, as well as cross checking their results.

I frequently revisited these unit tests with every new implementation. This is the core logic of the game, involving exchanging of cards, points adding, move verification and thus it is of utmost importance that the testing of these classes must be conducted clearly and systematically.

One aspect I did not expect, however, is how difficult it is to test the Move class. As previously described in the Algorithms Section, many choices of implementation failed for some very specific choice of cards chosen and some minor details (order of selection). Some such details were overlooked when I first implemented the unit tests for the Move class and fortunately I was able to detect the faults as I progressed with manual tests later on.

Apart from this issue, all the tests worked quite effectively and were very useful in detecting faults. Currently, the program can pass all of the written unit tests.

7.2 Manual Tests

All the other classes (Game, FileManager, ComputerPlayer, Grader and Finder) are tested manually. This is because it is simpler to check the FileManager functionalities as the file format is comprehensible to human, and the set-up required for testing these classes are quite complicated. For instance, the Grader now has up to 10 strategies, along with some randomness in points (priorRandomStrategy), or the ComputerPlayer is created with randomly chosen type. As such, these classes are tested via manual tests, specifically:

- File loading and saving functionalities were tested and improved via game test-runs using user interface. All file loading and saving modes were tested, different methods and quitting time are used and resulting files are checked. Error detection is also tested by deliberately changing the file.
- Algorithms in Finder and Grader classes are tested using one by one using Scala Reply.
- ComputerPlayer class was tested via game test-runs using user interface.
- Naturally, the GUI Package is tested with manual tests via user interface. The manual tests also helped to configure some design choices while implementing the GUI.

8 Known bugs and Missing Features

There are currently no known bugs in the game logic itself, but there is a small bug in the graphical user interface. Whenever I used setScene, the size of the scene would not be configured according to the size of the stage. As such, I had to close the stage and show it again after switching scenes, even when it is not necessary.

There are no missing features per the project requirements. But I would like to develop more features should I continue working on the project. These features include simple animation (screen update after each separate computer player's moves), perfecting the "hints" system, simple fixing error file, and limited time for each move.

For the first feature, I have already implemented the GUI Package with separate classes for scene elements which need frequent updates and thus would be useful for this purpose, but I would need to further familiarize myself with scalafx and Thread. For the second feature, I would have to add another variable "hint count" and manage this variable, but it would not be too difficult. The third feature would require more logic and algorithm, but with the current implementation which separate different types of exceptions when reading a file, I believe this feature is quite possible. The final feature would be more challenging though, as it would require me to implement a "ticker", and a method to make automated move for human player, called when this "ticker" reaches the time limit.

9 Best sides and Weaknesses

9.1 Best sides

9.1.1 Independency

As previously described, the program structure is consisted of classes and objects with specific, clear roles. As explained, although some classes such as Table, CardDeck, Grader, … could be implemented in some other classes, these are made separate classes in order to allow all classes in the program to focus solely on their intended goals. And as the tasks are properly distributed, the level of independency between classes also increases, as interference between classes is now properly restricted through the means of well-defined functions. Additionally, private var and private functions are used extensively in the program to further restrict unnecessary interference and comply with the principle of abstraction.

Moreover, the program makes use of the design patterns which increase the independency of the classes. For example, the Factory Design Pattern allows the verification logic to be stored within the class itself, as well as the choice of creation (Human vs computer players, different types of computer players) being kept hidden. Apart from the Factory Design Pattern, the GUI Package also uses the Mediator Pattern with the

CasinoApp being the mediator for 3 handlers, which thus encourages each handler to focus on its core role. Hence, both of these design patterns serve to further increase the independency of each class and the level of abstraction in the program.

Furthermore, the program utilizes the exceptions to transfer the results of verification methods across the program. This implementation reduces the need for cumbersome chain of information transferring and thus allows the multi-layered hierarchy structure in which "bigger" classes manage "smaller" classes without having those "smaller" classes calling functions from "bigger" classes when the need arises. Naturally, this choice of implementation decreases the dependency of these "smaller" classes on "bigger" classes and other classes. This is also extremely useful when developing the program, as the "smaller" classes can have their functionalities implemented and tested in full before moving up the "hierarchy".

Independency is also important in the class itself to break down tasks using private functions. These functions also form independent parts to bigger algorithms. This implementation is very beneficial for the classes Finder and FileManager whose main functionalities involve solving the problem with various steps and smaller algorithms.

9.1.2 Extensibility

The program is implemented with high regards for its extensibility. The high level of independency within the program also serves this purpose. In fact, many new features were added in the last weeks, such as giving hints, giving instructions, restarting game and round, saving and loading modes, computer modes, ordering of players and verification of such orders, different saving comments method to accommodate special characters in comments … And the implementation process was relatively simple.

Additionally, some choices of implementation are yet to be utilized in the program, but have great potential to facilitate many new features, such as the current system of specific exceptions in the game, or the various scene elements which can be updated separately.

And most importantly, I believe the system of computer players is the most noteworthy demonstration of how extensibility is one of the core values of this program. Instead of choosing the usual methods of using subclasses to represent different types of computer players, I attempted and succeeded to find a system that can create infinitely many types of computer players using very simple adjustment of weights. Additionally, the separate implementation of the Grader class also makes room for easy adding new and modifications of existing strategies.

9.1.3 Attention to details

The current program desires to accommodate the user to the best of its capabilities. As such, there is a great attention to details to deliver good user experience. This is demonstrated by the external features of showing instructions, able to quit game at any point of the game (midround, after a round, start a game, …) without causing problem in file saving/reading, specific explanations of invalid user inputs, … to the implementations of the game logic. For instance, some implementations are made more complicated to accommodate rare but possible situations (many players with same number of cards in pile (spades in pile), tied winners, …) or improve user experience (notifications to change to the next player in view, game log/comments, showing the previous player in view their updated hand and the table, chosen card in hand could only be 1 at a time, …). Another example is how the FileManager separates the comments block out of the game metadata to accommodate special characters, or how it reads the name to allow special

characters (":") in the name. One notable demonstration of the level of attention to details is the mode of notifications to announce the next player in view. If the game has only 1 human player, there would be no notifications at all. If the game has more than 1 human players, there would be only 1 notification to announce the player in view at the start of the round, but for the following players their would also be an additional notification to announce them to change to next player at the end of their turn.

9.2 Weaknesses

9.2.1 Time cost

For the time being, the current implementation serves its purposes well with reasonable time cost. I have tried testing the game with the maximum number of computer players (11) and it works with hardly any delay (less than 1 second). However, when I tried testing the system with a table of 25 cards and 4 cards on hand, it took up to 4 seconds to find and grade all moves. While this number is quite difficult to reach, as the grading system expands with more and more strategies added, the need for lowering the time cost would inevitably arise.

I believe that one major cause of this issue is the algorithm used for finding all component captures based on the concept of subsets, because when I tested this algorithm in Scala Reply with increasing number of cards, I found that for around 25 to 30, the algorithm becomes very slow. As such, I would like to change this algorithm for some other algorithms using dynamic programming should I continue with the project. With the current implementation of the Finder class in which the main algorithm has been divided into smaller algorithms, this modification would be quite convenient.

Furthermore, I think the current method for efficiency test could be improved. Currently, I only tested the time cost by measuring the time difference between the operations. I would like to investigate further methods for measuring time cost and other aspects regarding efficiency.

9.2.2 Redundancy

While admittedly some structures or implementations in the program are deliberately designed to leave room for future development, there are some implementations that is currently not used and rather redundant in the program, for example, the wide range of specific exceptions, or the updating functions of scene elements, or the `NotEnoughCardsInDeck` used for dealing in `CardDeck`, or the use of "current" variables which are not frequently updated in the `Game` class, …

This problem would be fixed by either implementing these new features, or simplifying the program. However, it would need some time to determine which implementations are unnecessary while not diminishing the extensibility and independency of the program, and what consequences (bugs) may occur if those implementations are discarded.

10 Deviations from plan, realized process and schedule

While the estimated time for implementation of the classes does not drastically deviated from the plan. After my demo session, I decided to first implement the features to satisfy the moderate requirements first, then implement the computer player system last. This change of plan actually better supported my project as I could plan ahead and spare lot of time to develop my computer player system.

In particular, the realized process is:

Week	Main Classes / Objects	Goals	Tasks description
1	Suit, Card, CardDeck, Table, Move	Implement the simple classes and "smaller" classes.	• Implement the Suit class and its case objects (no significant methods) • Implement the Card class • Implement methods for extracting data from Card class (get suit / value / numerical value / imgPath) • Implement the Card object (factory) and verification algorithm • Implement the CardDeck and Table (mainly acts as collection of cards) • Create suitable exceptions involving these classes (wrong notation for cards) • Implement the Move class with case objects of MoveType and corresponding case classes • Implement the method for verification within the Move class • Implement the Move object that creates the Move instance and verifies it upon creation • Test the logic (unit tests)
2	Player, HumanPlayer, Round, Game, FileManager	Implement the more complex classes in GameLogic.	• Implement the abstract Player class • Implement the Player object (factory) with different methods to create and verify the creation of a Player instance • Implement the HumanPlayer class (with necessary methods to make a move) • Implement the Round class • Implement the Game class (incomplete) • Implement FileManager class (incomplete) • Create suitable exceptions • Test the logic (unit tests)
3-4	Game, Round, FileManager CasinoApp, SceneHandler, GameHandler, ExceptionHandler	Complete the GameLogic Package. Implement the App and GUI elements.	• Complete the GameLogic Package • Implement the CasinoApp • Implement the GameHandler class • Implement the SceneHandler class • Implement the ExceptionHandler class • Create suitable exceptions • Test the app / system testing (manual tests)
5-6	ComputerPlayer, Finder, Grader	Implement the computer player system	• Implement the Finder class • Implement the Grader class • Implement ComputerPlayer class (make a move automatically) and ComputerPlayer object • Test the operations using mostly manual tests in Scala Reply

7	All	Improving program code	• Debugging • Efficiency tests • Implement some new features: e.g. different save/load modes, computer modes, hint, …
8	All	Finishing project	• Continue debugging • Commenting and style check • Prepare project document

11 Final Evaluation

Overall, I am quite satisfied with the quality of the program. The structure is well-designed with independent classes, reasonable use of design patterns, and well-defined means of interactions between classes. The program is currently very manageable and leaves room for future developments with its core values of independency and extensibility. The algorithms used are carefully considered and implemented. I also experimented with more unusual choices of implementations (Factory Design Pattern for verification, customized exceptions to communicate abnormalities from different layers of program structure, design of computer players based on linear system, …), and the outcomes are satisfactory.

While there are no major shortcomings, I would like to further improve the program. Should I continue working on the project, I would implement the missing features and minimize the presented weaknesses (changing the current variables in Game class, testing dynamic programming to reduce time cost, …). Additionally, I would like to test the performance of more data structures, as the current data structures are mainly chosen because of their functionalities, and perhaps more effective data structures could be used. As such, the current program structure would support me in adding these new features and modifications as it is as previously mentioned, implemented with independency and extensibility held in high regards.

I am quite content with the current project. I was given the chance to apply and experiment with the introduced concepts and ideas in the course content, as well as interesting algorithms and systems. While I could not implemented all the features I originally planned to, the development of the core system actually far exceeded my expectations.

12 References

There are 2 algorithms that I used were inspired by some similar problem in GeeksforGeeks. The first one (not used) is the counting of subarrays with given sum: https://www.geeksforgeeks.org/find-subarray-with-given-sum/; while the second one is finding subsets with given sum: https://www.geeksforgeeks.org/recursive-program-to-print-all-subsets-with-given-sum/?ref=lbp

The performances of data structure are based on the Scala Documentation: https://docs.scala-lang.org/overviews/collections/performance-characteristics.html

For the design patterns, I mainly take the inspiration and learn the structure via the course material (both O1 and studio A), and from the two websites Source making (https://sourcemaking.com/design_patterns) and GeeksforGeeks (https://www.geeksforgeeks.org/software-design-patterns/).

I am still new to graphics and ScalaFX, so I find the Youtube channel of Mark Lewis very helpful. This is where he posts various videos about swing Scala and ScalaFX, (https://www.youtube.com/channel/UCEvjiWkK2BoIH819T-buioQ).