PROJECT TECHNICAL PLAN

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

CLASS STRUCTURE

1. Identifying Classes and Objects

At the beginning of the plan, the key classes and some additional classes are identified. While the key classes will not have major changes, other classes may be adjusted, added or discarded as some functionalities are considered.

From the description of the Casino game, the essential components of a functional game are cards and the players, hence the most fundamental classes are Game, Round, Player and Card. From the perspective of an application, the program also needs to load and save the game, as well as create a functional user interface and initiate the application. Therefore, the CasinoApp object and FileManager class are also necessary. These are the key classes and objects of the program.

Delving into the Player class, there are two types of players: human and computer-controlled players. The two types differ in the way they make their moves. Human players wait for the user's input, while the computer-controlled player decides by themselves how they should make their move. However, the program needs the results of their moves to proceed, regardless of player types, so Player should be an abstract class, with HumanPlayer and ComputerPlayer as subclasses with different behaviors (methods) for the same call (makeMove). Additionally, a Move class should be added with PlayMove and CaptureMove as subclasses (case class) to easily classify, verify (avoid inappropriate captures for instance) and extract information about the player's move.

As for the Card class, subclasses including Two, Three, Four, … Queen, King, Ace will be implemented as case classes to utilize pattern matching (useful for extracting card values, scoring, …). Also, a Suit class is added with 4 case objects, including Heart, Spade, Diamond, Club for simple management and some more utilities (explained later on). Furthermore, it would be more convenient to implement CardDeck and Table class to handle some functionalities regarding the Card class. CardDeck class represents the cards in the initial pile, starting with 52 cards at the beginning and 0 at the end where all the cards have been drawn. This class has functionalities such as creating 52 cards, shuffling, … Meanwhile, the Table class acts as a collection for the cards currently on the table, with functionalities such as adding and removing cards, checking for sweeps, … The functionalities of these classes can still be implemented in the Game class instead, but using other classes is more manageable and intuitive.

2. Key functionalities

Before the game starts, the CasinoApp will create an instance of the Game class. The CasinoApp should have a newGame and continueGame methods which prompts different reactions in the user interface (form to add new players for example), and affect the Game instance. Therefore, the Game class should have methods including loadGame to continue the previous game, newGame, addPlayer, startRound to start a new game / round (the initiation of a new game are divided into different parts to synchronize with the user interface). To complete these methods and carry out other functionalities, the Game class must store information about its file manager (FileManager class), its players (Player class), the card deck (CardDeck class) and cards on table (Table class); and the current round (Round class).

The FileManager class is rather simple. This class needs 2 main methods to fulfill its role: loadGame (game: Game) and saveGame (game: Game) with the former to modify the game according to an existing file and the latter to save the current game into the file.

Instances of Player class must be created at the start of the game, whether if it is a new game or previous. However, the creation of these instances is complicated by the two subclasses, with the HumanPlayer class requiring a ‘name’ argument. Hence, the factory method with a companion object Player would be very efficient to create and verify the instance upon creation. If there are missing or inappropriate arguments (when reading from file or adding players manually), such creation would throw exceptions.

Throughout the program structure, the factory method as well as verification upon creation using exceptions would be utilized. This is because the game and application require different subclasses, such as HumanPlayer and ComputerPlayer, the subclasses of Card so the factory method would be convenient for the creation of instances from those classes. Moreover, this method allows the algorithms for verification to be implemented within the class and initiated upon creation. This makes such algorithms accessible to all situations that require the verification. For example, if the algorithm for verifying appropriate Player instance is solely implemented in the code for file reading, then another algorithm or the same algorithm must be copied for checking the user’s mistakes in creating players (missing names for human players, for example). This advantage will be demonstrated clearly with the move verification.

To easily identify and deal with different problems during the program such as players creation, the abstract class CasinoAppException, which extends to class Exception, is added along with several case classes as subclass to represent possible exceptions of the program. In the creation of Player class instances, the exceptions MissingPlayerInfo and UnknownPlayerType can be thrown. After the creation of Player class instances manually (user input), the exception NotEnoughPlayers can be thrown.

Delving deeper into the Player class, this class should know its name, type (Human/Computer), scores, number of sweeps, cards on Hand, cards in Pile and cards seen (currently on the table). While some of this information can be stored elsewhere (Game/Round class), this information will be used for different classes in several functionalities (score calculation, user interface) and hence, storing them in Player class would provide more versatility. Additionally, to make the code more manageable, the class PlayerType with HumanType and ComputerType case object are added. To update or show the variables to other classes, the corresponding methods needed are updateCardSeen(Vector[Card]), addScore (score: Int), addSweep (), addHand (card: Card), getPile: Vector[Card]). Also, while the name, type and scores are preserved, other variables are information limited within the round, so another method resetRoundInfo () is implemented to reset these variables.

The fundamental method for Player class will be an abstract method makeMove () which should return an appropriate move (Move class), accompanied by the method possibleMove () to identify possible moves based on cards seen and cards on hand. While the computer-controlled players make their moves automatically, human players require user’s input to make moves, so it is reasonable to add chosenCard and chosenCards as variables and corresponding methods to handle them. Furthermore, the computer-controlled players will need different strategies (explained further in Algorithms) so coding them in a companion object will be helpful.

The CardDeck and Table classes are less sophisticated. The main functionalities of the CardDeck class is to create a new deck with 52 cards (method newDeck()), load the deck of the previous game (method loadDeck (Vector[Card])), shuffle the deck (method shuffle ()) and allow players to draw cards (method drawCards (numCard: Int): Vector[Card]). Meanwhile, the Table classes acts mainly as a collection for the cards on the table, with method addCards (cards: Vector[Card]) to add cards, removeCards (cards: Vector[Card]) to remove cards when there is a capture move, and checkSweep : Boolean when a sweep happens. These classes and the player class should also be able to check if there are no cards left, with a method hasCard: Boolean.

The Round class acts as a mediator between the Player, the CardDeck and the Table classes and manages all activities during a round. A Round instance is created by the Game instance, with its CardDeck and Table instances directly passed on. The Round instance also knows the players from the Game, but in a particular order so that the first player in the collection is the one to make the first move (the person next to the dealer in a new round or the current player in turn from the previous game). The Round class should be able to respond to a move made by players, hence the method respondMove (move: Move), which will remove or add card (s) to the table, remove the played card from player's hand, draw a new card from deck and add to player's hand, and add the cards captured to player's pile, as well as check if the round has ended. The class will also need a nextMove() method to start the next player's turn. Implementing this method separately allows more room for user interface responses, such as a pop-up message to announce the next player's turn or covering the previous player's card. Considering the move, it is noteworthy that the ComputerPlayer makes their moves without waiting for the user's input, so another method runComputerPlayer () should be implemented so that the program runs past the computer players' moves automatically until the next human player.

At the end of the round, the last player who captured a card takes all remaining cards on the table, and the scores are calculated. This requires a method endRound () of the Round class to perform these actions, as well as resetting some variables of players (cards) and show the results of the round. To announce the other classes of its completion, a variable complete is added and a corresponding method isComplete is implemented. A corresponding endRound () method should also be implemented in the Game class to determine if one of players won, and prepare to start the next round. If the winner is found or the user quits the program mid-game, an endGame () method should be implemented, a variable complete should be added to the Game class along with the corresponding isComplete method.

Apart from these classes, there are classes that act mainly as special storages of information, including Card class and Move class. The Card class is divided into case subclasses according to their values (2, 3, 4, …, Jack, Queen, King, Ace) and a Suit class with 4 suits as case object is implemented to fully utilize the match case structure. As a result, the methods of Card class are quite basic, including getSuit: Suit, getNumVal: Int (numerical value of a class, explained in Algorithms), getName: String (get the name of the card, 'Jack of Hearts' or 'Diamond-10' for example).

The Move class is implemented in a similar structure to Player class, with a verification algorithm implemented to initiate upon creation with factory method and a companion object. However, as the verification involves more complicated issues than missing information for instantiation, the method verify () is implemented in the Move class which the Move object call to throw exceptions if an error occurs. Exceptions for the instantiation of this class are: NoCardInHandChosen, WrongMoveButton, InvalidPlay and InvalidCapture. This class also has case subclasses PlayMove and CaptureMove to utilize match case structure, along with trait MoveType with PlayMoveType and CaptureMoveType as case objects. The Move should also know its player to easily transfer the data.

3. User Interface Elements

The CasinoApp is the most important element of the user interface, acting as a between all other user interface elements and the game logic. The task will be divided between the interface elements, game elements and the CasinoApp will act as a mediator, using the mediator design pattern. This makes the program easier to maintain and test. This type of structures also allow easy addition or adjustments of functionalities.

To effectively communicate with the existing game logic classes, the CasinoApp must know its current game, current round and current player and current player in view (the computer-controlled players do not have screen time during their moves). This information can be easily stored and extracted using corresponding variables and methods from the Game class and Round class (extracted from Game class). However, implementing these variables and related logic in the CasinoApp would make the code rather cluttered and these classes cannot effectively call the methods of CasinoApp without allowing the user interface to interfere too much with the game logic. Therefore, a new GameHandler class will be created for handling the internal game logic with all the former variables. An instance of this class will be stored in the CasinoApp as the first "colleague" of the mediator CasinoApp.

The next important member of this structure is the SceneHandler class. While the main stage is fixed, the scene of it can be changed according to different stages of the application, such as showing a starting menu, adding the players. Therefore, a SceneHandler to manage such scenes would be more convenient and effective. Another essential part are the exceptions, so the class ExceptionHandler should also be added to detect the exceptions that occur and notify the user, as well as wait for user's input on the matter via customized alerts.

The SceneHandler and ExceptionHandler classes can also act quite similarly to a factory that store the templates or special customization that can be utilized to create certain items when the method is called. For the SceneHandler class, such methods are createMenuScene (), createAddPlayerScene (), createPlayerViewScene (hand: Vector[Card], table: Vector[Card], turnName: String, summary: String); for the ExceptionHandler class, there are faultFileReadingAlert (), faultPlayerInputAlert() for specific corresponding exceptions, and a factory method to create an alert according to exception input createExceptionAlert (e: Exception). Because the creation of dialogues in scalaFx requires an owner, the stage: JFXApp.PrimaryStage should also be stored as a variable of the ExceptionHandler class.

However, these handler classes should not only create the suitable products, but also manage them. Therefore, additional classes are added to help these handler classes delegate the code for formatting elsewhere, especially with the SceneHandler class. The formatting for one scene is rather complex with different design details and therefore, to effectively manage these scenes, it is reasonable to store them in the StartMenuScene, AddPlayerScene, PlayerViewScene classes, which are subclasses of Scene. These classes should store the designs as well as the controllers of the scenes such as label, button, … However, it should be stressed that these classes only store the 'format', and the behaviors / methods with these controllers will lead to a call for suitable methods in the handler, which in turn, invoke methods of the CasinoApp.

Most of the communication between user interface elements are between the CasinoApp, SceneHandler class and GamHandler class. Therefore, most of their methods will correspond to each other.

For the SceneHandler class, the methods to response to user input include newGame () to prompt the CasinoApp to generate a new game, continueGame () to load the previous and startRound () to start the game; addHumanPlayer (playerName: String) and addComputerPlayer () to add players; chooseCard (card: Card) to choose a card; playMove (), captureMove () to make a move. All the methods in this class will be invoked by clicks on labels / buttons and are solely used for transferring information back to CasinoApp. Additionally, the SceneHandler class should also have a getScene (): Scene to allow the CasinoApp to extract its scene. This class also have a chosenDisplay (cards: Card) method that makes the chosen cards glow, and a createNextTurnNoti () to pop up a message about the next player's turn (human players)

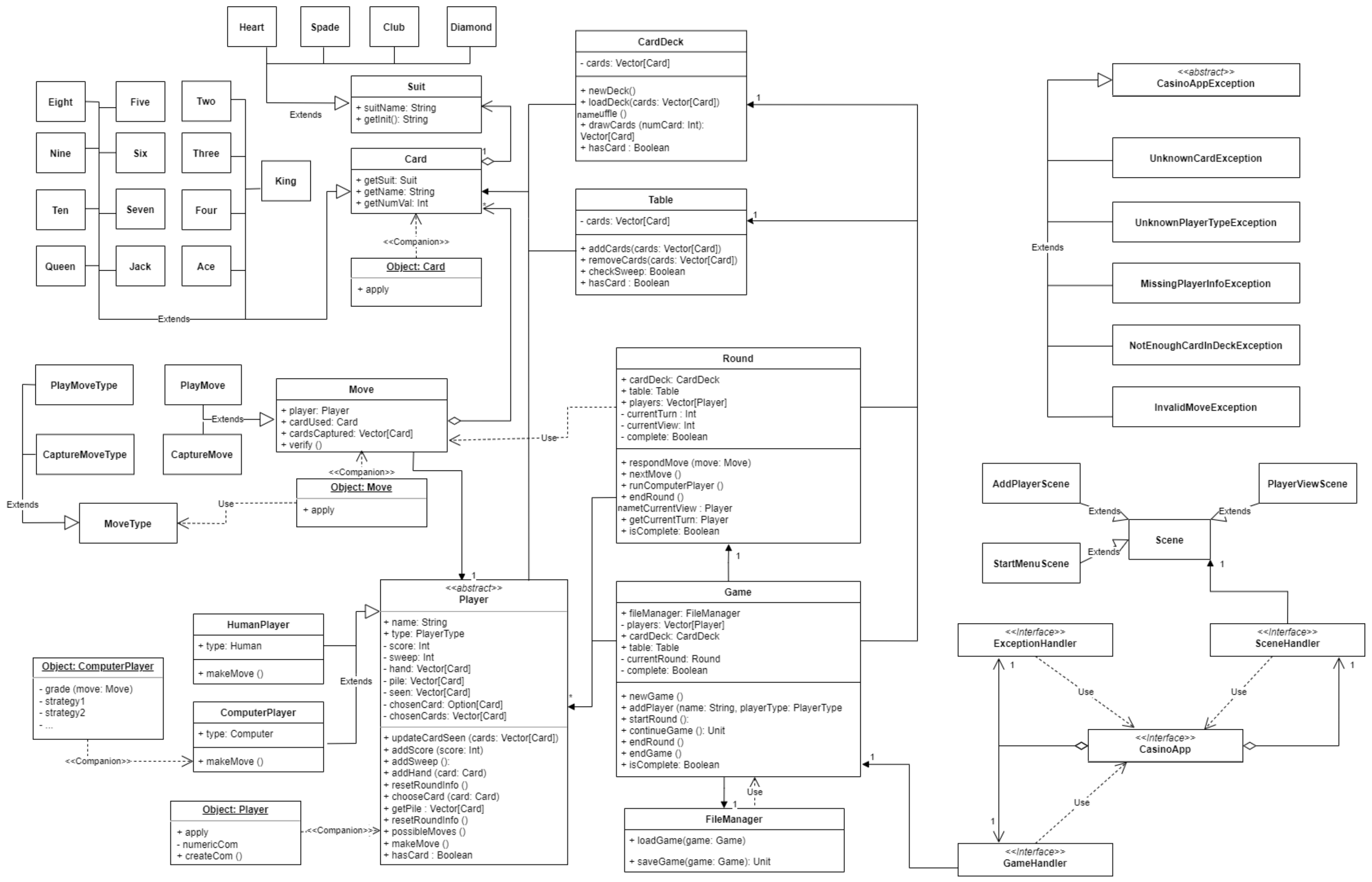
The CasinoApp will have the same first 8 methods as class SceneHandler, transferring the information and prompting actions between the GameHandler class as well as prompting the next action from SceneHandler. It will also have a run () method to run through the application (from the start menu). Additionally, the nextMove () and runComputerPlayer () methods will be implemented to handle the computer-controlled players. The startRound(), endRound () and endGame () methods will also be implemented.

For the GameHandler class, there will be methods will synchronize well with the logic of the Game, with the same methods as the CasinoApp, except for addPlayer (playerType: PlayerType, name: String) and addPlayer (PlayerType); and a method makeMove () to prompt the player card to make a move. It also does not have a run method.

For the ExceptionHandler class, such methods include newGame () if an error occurs in file reading and the user chooses to play a new game, and quit () if the user chooses to quit; confirmFault () to confirm the mistakes of the user. The try – catch structure that involves this class will be implemented in necessary methods of the CasinoApp.

4. UML Diagram

The UML diagram is quite oversized to fit in the document, so an URL is added for better view: https://viewer.diagrams.net/?tags=%7B%7D&highlight=0000ff&edit=_blank&layers=1&nav=1&title=Casino#Uhttps%3A%2F%2Fdrive.google.com%2Fuc%3Fid%3D1GSzW4iwJv49zCktFVtzKRQorPWb0Wh6Q%26export%3Ddownload


Heart
Spade
Club
Diamond
Suit
+ suitName: String
+ getInit(): String
Extends
Eight
Five
Two
Nine
Six
Three
King
Ten
Seven
Four
Queen
Jack
Ace
Extends
Card
+ getSuit: Suit
+ getName: String
+ getNumVal: Int
1
<<Companion>>
Object: Card
+ apply
CardDeck
- cards: Vector[Card]
+ newDeck()
+ loadDeck(cards: Vector[Card])
nameuffle ()
+ drawCards (numCard: Int): Vector[Card]
+ hasCard : Boolean
1
Table
- cards: Vector[Card]
+ addCards(cards: Vector[Card])
+ removeCards(cards: Vector[Card])
+ checkSweep: Boolean
+ hasCard : Boolean
1
<<abstract>>
CasinoAppException
UnknownCardException
UnknownPlayerTypeException
MissingPlayerInfoException
NotEnoughCardInDeckException
InvalidMoveException
Extends
PlayMoveType
CaptureMoveType
Extends
MoveType
PlayMove
CaptureMove
Extends
Move
+ player: Player
+ cardUsed: Card
+ cardsCaptured: Vector[Card]
+ verify ()
<<Companion>>
Object: Move
+ apply
Use
Use
Round
+ cardDeck: CardDeck
+ table: Table
+ players: Vector[Player]
- currentTurn : Int
- currentView: Int
- complete: Boolean
+ respondMove (move: Move)
+ nextMove ()
+ runComputerPlayer ()
+ endRound ()
nametCurrentView : Player
+ getCurrentTurn: Player
+ isComplete: Boolean
AddPlayerScene
PlayerViewScene
Extends
Extends
Scene
StartMenuScene
Extends
1
1
<<abstract>>
Player
+ name: String
+ type: PlayerType
- score: Int
- sweep: Int
- hand: Vector[Card]
- pile: Vector[Card]
- seen: Vector[Card]
- chosenCard: Option[Card]
- chosenCards: Vector[Card]
+ updateCardSeen (cards: Vector[Card])
+ addScore (score: Int)
+ addSweep ():
+ addHand (card: Card)
+ resetRoundInfo ()
+ chooseCard (card: Card)
+ getPile : Vector[Card]
+ resetRoundInfo ()
+ possibleMoves ()
+ makeMove ()
+ hasCard : Boolean
*
HumanPlayer
+ type: Human
+ makeMove ()
Extends
ComputerPlayer
+ type: Computer
+ makeMove ()
Object: ComputerPlayer
- grade (move: Move)
- strategy1
- strategy2
- ...
<<Companion>>
Object: Player
+ apply
- numericCom
+ createCom ()
<<Companion>>
1
Game
+ fileManager: FileManager
- players: Vector[Player]
+ cardDeck: CardDeck
+ table: Table
- currentRound: Round
- complete: Boolean
+ newGame ()
+ addPlayer (name: String, playerType: PlayerType
+ startRound ():
+ continueGame (): Unit
+ endRound ()
+ endGame ()
+ isComplete: Boolean
1
Use
FileManager
+ loadGame(game: Game)
+ saveGame(game: Game): Unit
<<Interface>>
ExceptionHandler
<<Interface>>
SceneHandler
1
Use
Use
1
<<Interface>>
CasinoApp
1
Use
1
1
<<Interface>>
GameHandler

USE CASE DESCRIPTION

When the application runs, a menu will appear to allow the user to choose to start a new game or continue the previous one. If the user chooses to start a new game, he or she can add players (human or computer) at will. The updated players will appear on screen and when all the players are added, the user can choose to start the game. Additionally, the user will be notified if there are not enough players, or missing names or other faults made when adding players, as well as faults in reading the file for the previous game which will prompt the user whether to start a new game.

At this phase, the operation mainly involves the CasinoApp and the handler classes. The CasinoApp will 'run' to start the program and prompt the SceneHandler to create a start menu. If the user chooses to continue the game, the SceneHandler will prompt the CasinoApp to continue the previous game and thus, the CasinoApp will prompt the GameHandler to call the loadGame () method of its Game instance, ask for the necessary information for the GameHandler to prompt the SceneHandler to create the scene from the player's viewpoint. On the other hand, if the user chooses to start a new game, the newGame () method is called instead and the CasinoApp will request a scene to ask for players from SceneHandler. With each player added, the CasinoApp will retrieve information from the SceneHandler and prompt the GameHandler to make suitable additions to the Game instance. In all the duration of the process, any faults (by the user or the file) will be detected by the CasinoApp and handed over to the ExceptionHandler to notify and wait for the user's reaction.

While that is how the structure delivers input and output, the system that makes real adjustments and verify input revolves around classes related to game logic. In the case the user chooses to continue the previous game, the method loadGame () of the Game instance is called, and the FileManager class will load the game from the file, while in the 'new game' scenario, the newGame () method reset players list, round, the CardDeck and the Table (more useful later on) and the user will add players manually. In both cases, the creation and verification of some classes is carried out, including the Card class and the Player class, which will verify itself upon creation and notify the system if some errors (exceptions) occur. These instances will be then added to CardDeck, Table and players list.

Then the first round starts, the user (s) will complete their moves during their respective turns. During each turn, the player (user) will have their cards on hand, cards on the table displayed on the screen. The player chooses the cards with mouse clicks and makes the move with buttons (play and capture). After their moves, a new card is drawn and added to their hand. The screen changes to the next human player (user) at their turn after notifying the user of the next human player's turn.

Again, the input and output are transmitted within the system with the handlers and the CasinoApp with the following direction: SceneHandler -> CasinoApp -> GameHandler -> access inner game logic -> CasinoApp -> SceneHandler, with errors detected by CasinoApp -> ExceptionHandler to notify the user. Inside the game logic, the Player class creates and delivers the Move instance (verified upon creation) to the Round class. The Round class then distributes the cards between Player's hand, Player's pile and Table according to the Move instance and draws a card from CardDeck to the Player's hand (if possible). After each move, the completion of the Round is checked and if the Round is yet to be complete, the next move is prompted. In case of computer-controlled players, the CasinoApp will prompt itself to run through computer-controlled players, prompt the GameHandler to use the inner logic to make the moves and deliver the results back to CasinoApp for output on screen via SceneHandler.

When the round ends, the user will be notified of the result of the round. If the winning condition is met, the game is over, the result of the game is announced, and the user is asked whether to start a new game. Else, the user is prompted to start the next round or save and quit. More about the quitting function, if the user quit the game mid-round, the game will still be saved automatically.

The ending of the round is checked with its inner logic after every move. If the round has ended, the Round class will calculate scores of players and add the new scores to the player's existing scores. The ending of the Round is checked by the GameHandler method which will then prompt and give the information to the CasinoApp to announce the user on screen with SceneHandler and ask for a new Round (if the game has not ended). Then, CasinoApp also prompts the GameHandler method to activate the endRound () method of the Game and if the winning condition is met, the GameHandler prompts the CasinoApp to again, display the result of the game and ask for a new game. The user input is retrieved with the SceneHandler and prompt reactions from CasinoApp. If a new round or new game is requested, the similar procedures are followed. If the player quits and the game has not ended, the CasinoApp will prompt the GameHandler to activate the endGame () method of game, which will (if not ended) save the game to a file using the FileManager.

ALGORITHMS

1. Initiating the Game

At the start of the program, whether it is loading a game from a file or starting a new game, this phase involves instantiating objects from various classes. The most common faults that occur during this phase are faults in the file itself (wrong notation for the cards, missing player information) and the faults when the user adds players manually.

The approach for the instantiation and verification of classes is to let the classes self-verify upon creation with the factory method and companion objects. This approach allows reusing such verification in different scenarios, as well as allowing various methods to create the same object (reading from file, adding manually, adding with program logic). This will also allow easy creation of case objects useful for pattern matching (used with cards for scoring and other logic).

The verification with these classes mostly involves detecting missing or unrecognized parameters, which is simply missing names or wrong notation of suits and can be easily detected. Also, the game must also check if there are missing players (whole player's block) by storing the number of players separately and compare that to the final number of players or missing cards (adding up the size of cards created).

2. Validating moves

One of the most fundamental tasks of this program is to allow and ensure the player (user) to make appropriate moves. There are two types of move that could be made during a turn: to play a card without capturing any cards and to capture card (s) on the table. The former move can be performed by the program simply by removing the chosen card from the player's hand and adding the card to the table. The latter move is more sophisticated to handle. This task starts with extracting the numerical value for the cards. This phase is complicated by the changing numerical values of particular cards:

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

Two among the methods to solve the problem are to assign a 'state' variable to the card or to implement two different functions for extracting numerical value, of which the latter is preferred because this method would simplify the class Card and reduce its interference with the logic of the game. This way, the Card class and its instances can be more easily managed with 'case class' and provide utilities for other functions (points calculation, for instance). Therefore, the default numerical value, or the numerical value stored in the Card itself, will be the numerical value of the card on the table. A different function for extracting numerical values from the player's hand can be conveniently implemented using the 'match-case' structure.

After extracting the numerical value of the class, the program needs to verify the choice (s) of card (s) of the player. Specifically, per the game's rule, the card (s) that can be captured is:

- A card of the same value (Ace, King, 8, 4, …) as the played card
- Card (s) that sum up to the value as the played card
- Both cases in a single turn

One approach is to check the move using the list of cards chosen, while the other is to find and store all possible combinations of card used and cards captured in another object which is instantiated when the program start. The list of cards used and cards captured is then checked using the available combinations.

However, there is a way to combine these approaches to obtain optimized performance, while also allow reusing the method for computer-controlled players: that is to check and create capture moves with the cards of the same value, then check if the list of the numerical values of the remaining cards contains a certain slice of numerical values that sum up to the value of the card used, created and stored beforehand. If some combinations are found, the program finds the index and extracts the corresponding combinations of cards. If more than one combination is found for the card, the algorithm will use the recursive method and loop through all combinations. For example, the first combination is removed from the collection and the cards are removed from cards seen, then recheck if the second combination is still contained in the remaining cards seen, if the second combination is still possible, that combination is merged into the first and the cycle continue until the largest combination (with first combination) is found. The operation then continues with the second combination and the remaining combinations to find the all (multileveled) possible combinations and make moves out of them. The result of this sum capture move is then merged with the same value capture move (if possible). While this method may seem cumbersome and time costly at first, there are many ways to improve it such as narrowing down the numerical list based on real-life situations, or setting up more conditions in the algorithm, such as only try merging the combinations of which size will not sum up to a number larger than the cards seen size.

This method shall allow the computers to detect various faults when the human user make a move, including using wrong buttons, invalid captures, not capturing enough cards (two cards of same value on the table but only one card is captured), and no card in hand chosen. This algorithm will be implemented in the Player class, and the faults are then detected and reported by the Move object when creating instances of Move class. This approach is very convenient because it allows the algorithm to be used in different scenarios (checking the player's move, finding possible moves, the "hint giving" algorithm). However, as the project progresses, if this approach is tested to be requiring significant time lost, it can still be replaced by other simple checking conditions (free play, if capture then remove same value cards and adds up the remaining cards, condition passed if sum divide by card chosen value have 0 remainder). But with this the finding possible method must still be added to Computer-controlled player, and the "hint" function cannot be implemented.

The algorithm necessary to create and store all possible collections that sums up to a targeted number is inspired by an answer on StackOverflow by @Richard Fearn (linked in preferences). This algorithm uses the recursive methods that call itself until the partial sum reaches or becomes higher than the target. Because the largest value of a card used is 16 (Diamonds-10), this method will not require paramount time cost. The method will be implemented in the Player companion object for easy management.

After the appropriate move has been made, the program will have to check and record "sweeps" for later scoring at the end of each round. This condition is checked easily by checking the collection of cards on the table.

3. Scoring

Per the game's rule, scores are calculated as following:

- 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

Hence, all of these conditions can be checked via going through the collection of cards in the pile of each player with the exception of the "sweep" condition. Therefore, the number of sweeps will be recorded by the players every turn. After a round, except for points from "sweeps" the score calculation is carried out in the Round class because there are some score-gaining conditions that need information from other players. Because of the case class structure, the conditions are checked easily by going through each player's pile.

4. Computer-controlled opponents

The decision - making of computer-controlled opponents will be divided into two parts: finding all possible moves and finding the best move. The former task was already discussed in the previous part and the algorithm is implemented in a shared method in Player class.

For the next task, after finding all possible moves, the computer-controlled player needs to determine the best move for this turn. One of the methods for this is to implement a "grading system". This system consists of different criteria, each of which will give a point correspondingly as each move goes through this system. For example, a move that allows the player to get an Ace class or special cards (spades) will get a lot of points from the corresponding criteria, the number of cards in each move will gain some points in a different criterion, the moves that leaves easier sweeps will get minus points from another criterion, a move that uses a powerful cards (cards with large values for instance) will get minus points if it is still early in the game, … After going through the "grading system", the move with the highest point is chosen.

This approach allows easy addition of strategies used (implemented just by adding another method in the grading system), as well as some interesting features such as different "preferences" of computer-player, which multiplies the scores gained from the method preferred. This system will likely be implemented in a companion object for easy access and management.

DATA STRUCTURES

For the data structures readily available in Scala, most methods will use Vector (immutable) and some will use Map (immutable) and Option. This is because Vector has great performance when accessing with indexes (useful for going through the player's turn), and quite stable performances in various situations (head, tail, apply, append, going through all elements …) compared to other types of collection. Also, its immutable nature is a simple yet powerful idea to improve the style and readability of the program. This allows a stark distinction between the fixed and changing variables which makes the code more manageable and easier to track when and where changes to collections are made.

However, in some situations, the Map (immutable) will be more convenient. For example, it is reasonable to use a map to store the numerical values and the collection of values that sum up to them (used in finding possible moves), or to return the data about the moves and their respective scores after going through the grading system, or to describe the collection of card labels in the SceneHandler to make the label glow when the corresponding card is chose. Furthermore, the Option data structure will also be useful when dealing with problems (missing information when reading file), or simply creating necessary variables which may not always be in existence (a round of the game, chosen cards on hand) and indicating that existence.

Apart from these readily available structures, the program also makes use of case classes and objects as units to store and transfer data. For the case objects, the program uses Heart, Diamond, Club, Spade to indicate suits, PlayMoveType and CaptureMoveType to describe the move types, Human and Computer to describes player types. For the case classes, there are classes to describe the cards with different values (Two, Three, … King, Queen, Ace) and PlayMove and CaptureMove. All these case classes and objects are added to utilize pattern matching and ease in transferring and extracting packages of data.

SCHEDULE

The estimated time for the project would be approximately two months, so it will be reasonable to divide the project schedule into 8 weeks.

Week	Main Classes / Objects	Goals	Tasks description

1	Suit, Card, CardDeck, Table	Implement the simple classes and objects (no complex methods / algorithms)	• Implement the Suit class and its case objects (no significant methods) • Implement the Card class and its case classes • Implement methods for extracting data from Card class (get suit / value / numerical value / imgPath) • Implement the method compare () to easily sort the cards • 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) • Test the logic (mostly automated tests)
2 - 3	Move, Player, HumanPlayer, ComputerPlayer	Implement the classes that requires complex algorithms	• 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 • Implement the abstract Player class with PlayerType case objects • 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 ComputerPlayer class (make a move automatically) and ComputerPlayer object to store the grading system • Create suitable exceptions • Test the logic (mostly automated tests)
4	Game, Round, FileManager	Implement the classes that dictates the other classes	• Implement the Round class • Implement the Game class • Implement FileManager class • Create suitable exceptions • Test the logic (automated tests / manual tests)
5-7	CasinoApp, SceneHandler, GameHandler, ExceptionHandler	Implement the App and GUI elements. System testing	Note: For this phase, the classes are implemented simultaneously • Implement the CasinoApp • Implement the GameHandler class • Implement the SceneHandler class • Implement the ExceptionHandler class • Test the app / system testing (automated / manual tests)
8	Undetermined	Complete the project	• Time to spare if unexpected situations happen • Prepare / complete the documentation / demo / project-related elements

TESTING PLAN

1. System testing

The system testing will be carried out via manual tests using the user interface.

The first test case will test the starting new game function, with the action of choosing to start a new game in the start menu, adding some players, deliberately making mistakes (missing player's name when adding human players) then start the round and quit, checking the saved file. During this round of testing, the following outcomes must be met:

- There must be automatic transition to add player scene from start menu when choosing to start the game
- The mistakes made must be detected and the program should notify the user via user interface
- The file must be saved with the players added
- The cards must add up to 52 cards, cards must be shuffled
- Each player must have 4 cards on hand, 0 cards in pile
- On the table are four cards

The next test case will test file reading using the previous file. The tester repeatedly chooses to continue the game, with the original file and some alterations (deliberately makes mistakes). During this round of testing, the following outcomes must be met:

- The program successfully continues the game with the unaltered file
- The program detects the mistakes, reports them and prompt the user to start a new game

The next test case will test the make moves function. A file game will be created so that the mistakes can be tested (suitable cards on the hand/table), sweeps can be made and the moves of computer-controlled player can be predicted. The user chooses to load the file, the tester deliberately makes mistakes and the planned moves, then quits and checks the file. During this round of testing, the following outcomes must be met:

- The program allows appropriate moves
- The program detects invalid moves and notify the user
- The computer-controlled players make the predicted move
- The file is updated according to the move

The last test case will test the end game and end round functions. A file game will be created that saves a game at the very last stage. The user chooses to load the file, the tester makes the planned moves. During this round of testing, the following outcomes must be met:

- The round ends after some moves
- The round score is added to each player's scores
- The player's scores are calculated and the winner is found

- The game ends and the results are displayed
- The program prompts the player to a new game, which will leads to the add player scene again

2. Unit testing

During the first stages (from week 1 to 3), most of the unit tests can be done with relatively simple setups. For example, during week 1, unit tests revolve around instantiating a card from the `Card` class and object, which should take inputs such as suit name (Heart, Spade, …), card value (2, Jack, Ace, …) and return a suitable card. The test can also results in the `UnknownCardException` should the inputs are incorrect. This would be useful when reading from file later on.

During week 2 – 3, while the complexity of the algorithms is increased, the setup for unit test is still simple. For example if the `makeMove` method (one of the most important method in the game) is tested, the inputs would be the cards chosen (added to the player instance using its method `chooseCard (card: Card)`), the output would be the move or some exceptions being thrown. For the `makeMove` method of the computer-controlled player, the inputs are cards (added to hand and cards seen) and the output should be the expected move.

From week 4 onwards, the tests will require more complicated setups as more classes are involved. For week 4, however, the unit test can still be used in some method, such as `responseMove (move: Move)` method of `Round` class which should take the Move as an input and makes appropriate adjustments to the players, `CardDeck` and `Table`.

REFERENCES AND LINKS

First and foremost, the specifically mentioned algorithm to find the combinations that sum up to a certain number is posted by @Richard Fearn on StackOverflow (https://stackoverflow.com/questions/4632322/finding-all-possible-combinations-of-numbers-to-reach-a-given-sum), where he also added more users of the website and a preference to a lecture of Stanford on Programming. I usually use this website to find algorithms, how to use a method, …

For the design patterns mentioned in the plan, 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 also frequently search for methods or codes in GeeksforGeeks. For the choice of data structure, I use the measurement from Scala Documentation, Performance Characteristics (https://docs.scala-lang.org/overviews/collections/performance-characteristics.html).

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, or Scala in general (https://www.youtube.com/channel/UCEvjiWkK2BoIH819T-buioQ).