Assignment 2

Team number: 6
Team members:

Name	Student Nr.	Email
Sifra ter Wee	2626840	s.v.ter.wee@student.vu.nl
Gal Cohen	2658531	g.cohen@student.vu.nl
Shahrin Rahman	2660701	s.rahman@student.vu.nl
Saman Shahbazi	2018969	s.shahbazi@student.acta.nl

Implemented feature

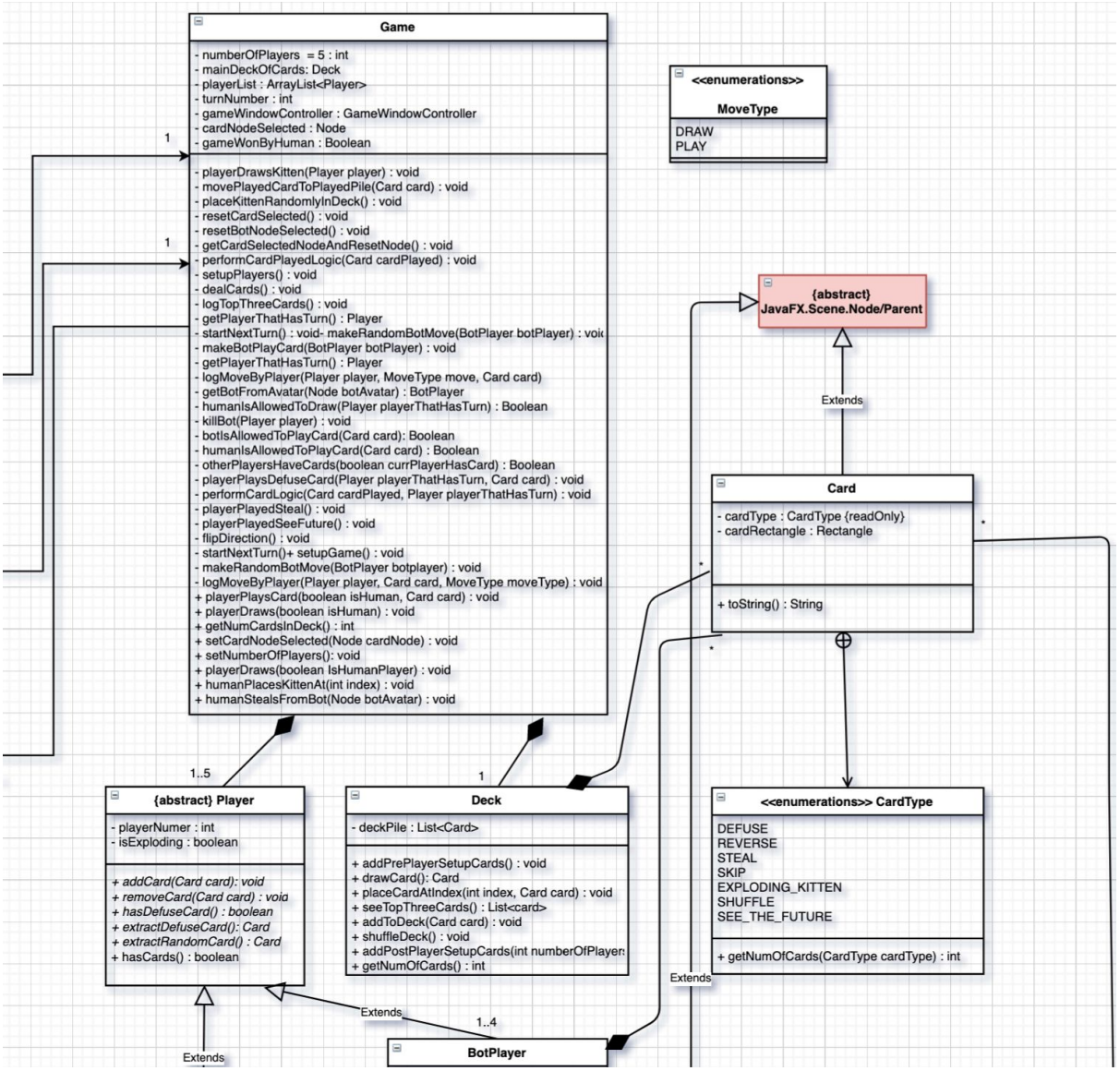
ID	Short name	Description
F1	Start screen	The player is greeted with a start screen where they can select game options [select number of players] or start the game.
F2	Set up	The starting cards are dealt according to the rules (n-1 Exploding Kittens… etc). The deck is shuffled and the players get 8 starting cards in total.
F3	Game semantic	A player can make an action using the GUI. The available actions are the following: draw, play. The user can play a: 1. Defuse card 2. Skip card 3. Reverse 4. See the future 5. Steal Moreover, the user can place the exploding kitten card at a location in the deck.
F4	Winning/losing	A player loses when they don't have a defuse card and they draw an exploding kitten card. A player wins when they are the last one standing.

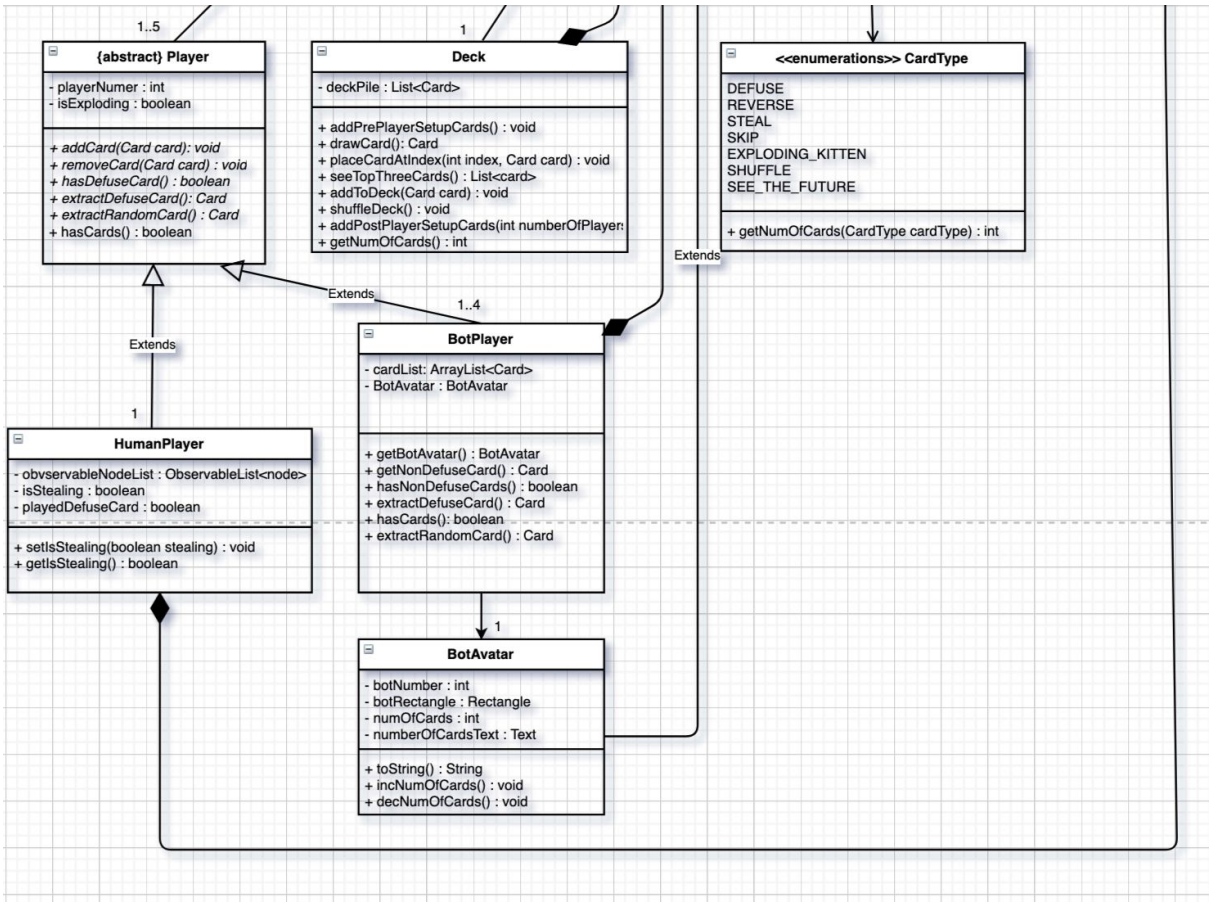
Used modeling tool:

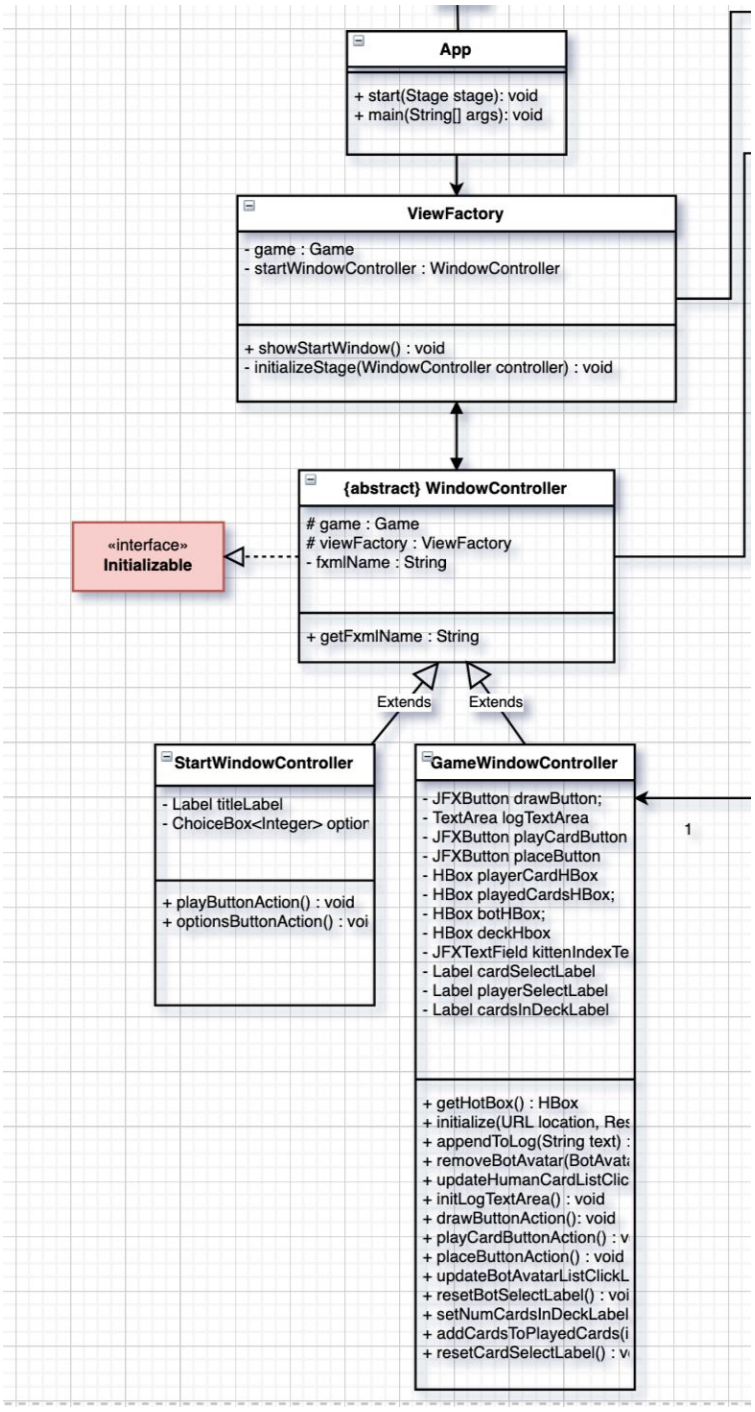
- Draw.io (app.diagram.net)

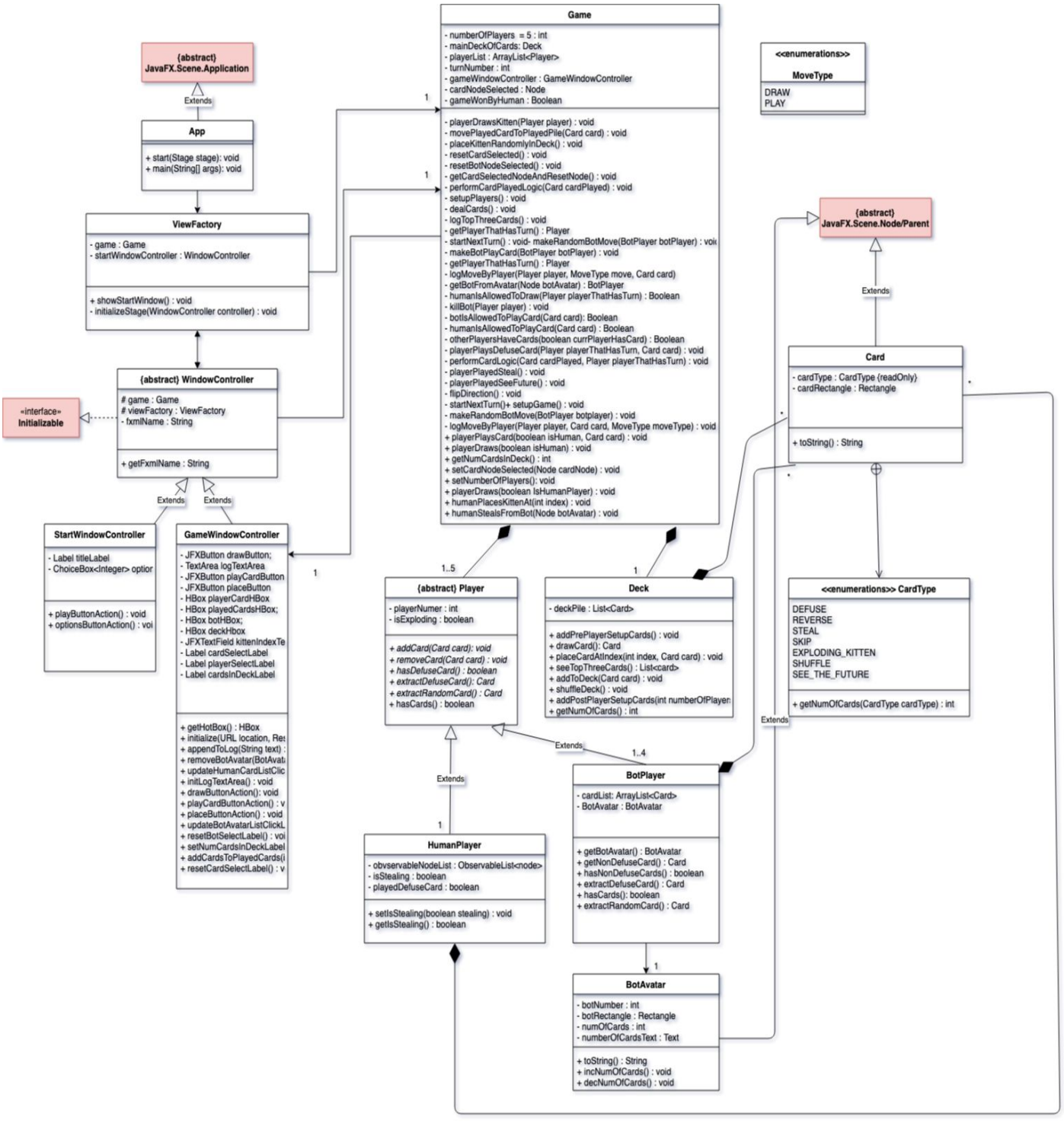
Class diagram

Authors: All


Game
- numberOfPlayers = 5 : int
- mainDeckOfCards: Deck
- playerList : ArrayList<Player>
- turnNumber : int
- gameWindowController : GameWindowController
- cardNodeSelected : Node
- gameWonByHuman : Boolean
- playerDrawsKitten(Player player) : void
- movePlayedCardToPlayedPile(Card card) : void
- placeKittenRandomlyInDeck() : void
- resetCardSelected() : void
- resetBotNodeSelected() : void
- getCardSelectedNodeAndResetNode() : void
- performCardPlayedLogic(Card cardPlayed) : void
- setupPlayers() : void
- dealCards() : void
- logTopThreeCards() : void
- getPlayerThatHasTurn() : Player
- startNextTurn() : void- makeRandomBotMove(BotPlayer botPlayer) : void
- makeBotPlayCard(BotPlayer botPlayer) : void
- getPlayerThatHasTurn() : Player
- logMoveByPlayer(Player player, MoveType move, Card card)
- getBotFromAvatar(Node botAvatar) : BotPlayer
- humanIsAllowedToDraw(Player playerThatHasTurn) : Boolean
- killBot(Player player) : void
- botIsAllowedToPlayCard(Card card): Boolean
- humanIsAllowedToPlayCard(Card card) : Boolean
- otherPlayersHaveCards(boolean currPlayerHasCard) : Boolean
- playerPlaysDefuseCard(Player playerThatHasTurn, Card card) : void
- performCardLogic(Card cardPlayed, Player playerThatHasTurn) : void
- playerPlayedSteal() : void
- playerPlayedSeeFuture() : void
- flipDirection() : void
- startNextTurn()+ setupGame() : void
- makeRandomBotMove(BotPlayer botplayer) : void
- logMoveByPlayer(Player player, Card card, MoveType moveType) : void
+ playerPlaysCard(boolean isHuman, Card card) : void
+ playerDraws(boolean isHuman) : void
+ getNumCardsInDeck() : int
+ setCardNodeSelected(Node cardNode) : void
+ setNumberOfPlayers(): void
+ playerDraws(boolean IsHumanPlayer) : void
+ humanPlacesKittenAt(int index) : void
+ humanStealsFromBot(Node botAvatar) : void
1
1
<<enumerations>>
MoveType
DRAW
PLAY
{abstract}
JavaFX.Scene.Node/Parent
Extends
Card
- cardType : CardType {readOnly}
- cardRectangle : Rectangle
+ toString() : String
*
*
*
1..5
1
{abstract} Player
- playerNumer : int
- isExploding : boolean
+ addCard(Card card): void
+ removeCard(Card card) : void
+ hasDefuseCard() : boolean
+ extractDefuseCard(): Card
+ extractRandomCard() : Card
+ hasCards() : boolean
Deck
- deckPile : List<Card>
+ addPrePlayerSetupCards() : void
+ drawCard(): Card
+ placeCardAtIndex(int index, Card card) : void
+ seeTopThreeCards() : List<card>
+ addToDeck(Card card) : void
+ shuffleDeck() : void
+ addPostPlayerSetupCards(int numberOfPlayers
+ getNumOfCards() : int
<<enumerations>> CardType
DEFUSE
REVERSE
STEAL
SKIP
EXPLODING_KITTEN
SHUFFLE
SEE_THE_FUTURE
+ getNumOfCards(CardType cardType) : int
Extends
Extends
1..4
Extends
BotPlayer


1..5
1
{abstract} Player
- playerNumer : int
- isExploding : boolean
+ addCard(Card card): void
+ removeCard(Card card) : void
+ hasDefuseCard() : boolean
+ extractDefuseCard(): Card
+ extractRandomCard() : Card
+ hasCards() : boolean
Deck
- deckPile : List<Card>
+ addPrePlayerSetupCards() : void
+ drawCard(): Card
+ placeCardAtIndex(int index, Card card) : void
+ seeTopThreeCards() : List<card>
+ addToDeck(Card card) : void
+ shuffleDeck() : void
+ addPostPlayerSetupCards(int numberOfPlayers
+ getNumOfCards() : int
<<enumerations>> CardType
DEFUSE
REVERSE
STEAL
SKIP
EXPLODING_KITTEN
SHUFFLE
SEE_THE_FUTURE
+ getNumOfCards(CardType cardType) : int
Extends
Extends
1..4
Extends
1
BotPlayer
- cardList: ArrayList<Card>
- BotAvatar : BotAvatar
+ getBotAvatar() : BotAvatar
+ getNonDefuseCard() : Card
+ hasNonDefuseCards() : boolean
+ extractDefuseCard() : Card
+ hasCards(): boolean
+ extractRandomCard() : Card
HumanPlayer
- obvservableNodeList : ObservableList<node>
- isStealing : boolean
- playedDefuseCard : boolean
+ setIsStealing(boolean stealing) : void
+ getIsStealing() : boolean
1
BotAvatar
- botNumber : int
- botRectangle : Rectangle
- numOfCards : int
- numberOfCardsText : Text
+ toString() : String
+ incNumOfCards() : void
+ decNumOfCards() : void


App
+ start(Stage stage): void
+ main(String[] args): void
ViewFactory
- game : Game
- startWindowController : WindowController
+ showStartWindow() : void
- initializeStage(WindowController controller) : void
{abstract} WindowController
game : Game
viewFactory : ViewFactory
- fxmlName : String
+ getFxmlName : String
«interface»
Initializable
Extends
Extends
StartWindowController
- Label titleLabel
- ChoiceBox<Integer> optior
+ playButtonAction() : void
+ optionsButtonAction() : voi
GameWindowController
- JFXButton drawButton;
- TextArea logTextArea
- JFXButton playCardButton
- JFXButton placeButton
- HBox playerCardHBox
- HBox playedCardsHBox;
- HBox botHBox;
- HBox deckHbox
- JFXTextField kittenIndexTe
- Label cardSelectLabel
- Label playerSelectLabel
- Label cardsInDeckLabel
+ getHotBox() : HBox
+ initialize(URL location, Res
+ appendToLog(String text) :
+ removeBotAvatar(BotAvata
+ updateHumanCardListClic
+ initLogTextArea() : void
+ drawButtonAction(): void
+ playCardButtonAction() : v
+ placeButtonAction() : void
+ updateBotAvatarListClickL
+ resetBotSelectLabel() : voi
+ setNumCardsInDeckLabel
+ addCardsToPlayedCards(i
+ resetCardSelectLabel() : v
1



Brief description of the class diagram

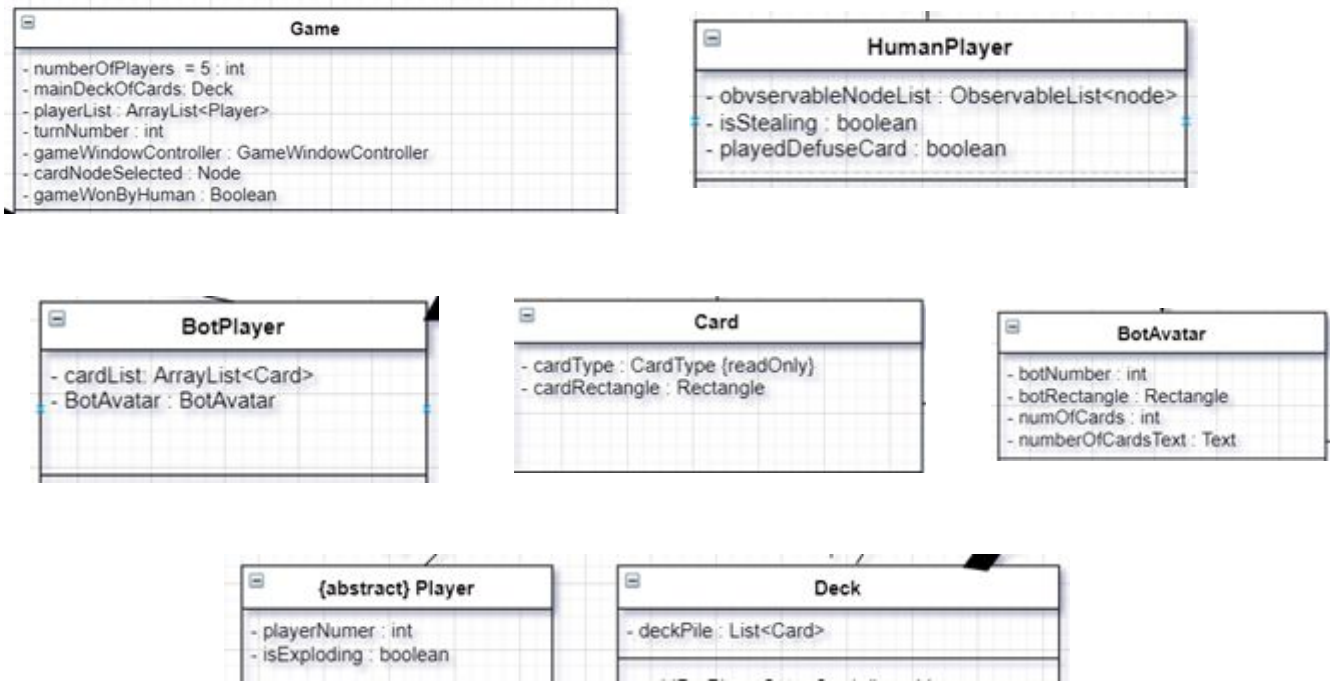
As we can see from the given UML class diagram, our game 'Exploding Kittens' is mainly based on about 13 classes – Game, App, ViewFactory, WindowController,StartWindowController, GameWindowController, Player, Deck,

HumanPlayer, BotPlayer, BotAvatar, Card, and CardType etc. The mentioned classes are interrelated to each other in specific ways by encapsulating all the relevant data of particulars objects and operations through a systematic manner. The given diagram also comprises interface and abstract classes as well as the implementation of inheritance. Each of the classes is responsible for a specific task, and it contains a number of attributes and methods to serve this purpose. For example, the Game class is responsible for storing significant information of our whole game where it deals with the number of players, maintaining the deck and its cards, keeping record of turn numbers and many more. It also defines a number of methods that are responsible for retrieving related data in response to requests from both inside and outside the class. The similar scenarios also apply to other classes to maintain the systematic process of the game.

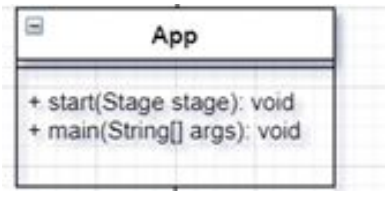
Brief description of most relevant attributes

From the attached class diagram it can be seen that there are multiple public, private, and protected attributes that we have used to design our game.

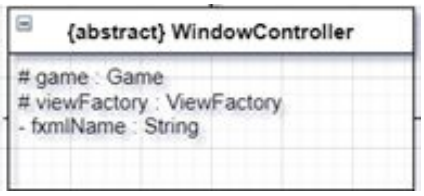
Most relevant private attributes are as follows:



Most relevant public attributes are as follows:



Most relevant public attributes are as follows:



Thus, our class diagram suggests a number of attributes that reflect the semantics of our game. For example, in the Game class, apart from defining the private attribute numberOfPlayers with type integer, we also initialise this attribute with value 5. Similarly, the attributes mainDeckOfCards with type Deck responsible for setting up cards in the deck pile, playerList with type ArrayList responsible for creating an array of list of the total players, turnNumber with type integer responsible for keeping track of the played turns of each player and so on are defined in the same way. On the other hand, the HumanPlayer class has the attributes of obersableNodeList with type ObservableList that takes care of creating the list for the human players in the game, isStealing and playedDefuseCard with boolean type that work like checkers do equally similar jobs in our game. Furthermore, the Player class has the attribute isExploding of boolean type that checks for each player's status whether they are exploding or not. This is how every attribute in our class diagram is responsible for each specific roles in order to have a complete implementation of the game.

Brief description of most relevant operations

The following operations seem to us to carry the most significance within the game:

- setupGame()
- performCardLogic()
- playerPlaysCard()
- playerDraws()

These functions are all contained within the Game class and they allow the game to move forward and develop as it is played. Here we will briefly describe the functionality of these operations.

The first function sets up the game by instantiating the deck and the players. It also fills the deck with cards in order to provide the players with their first five starting cards. Additionally, it interacts with the GUI so the correct amount of players are displayed on the screen.

The second function - performCardLogic() - takes care of the logic behind each type of card. There are seven different types of cards, and so seven different actions have to be performed. The function also moves a played card to the played pile by removing it from the player's hand. Thereby maintaining the integrity of the game.

The third function determines if and when a player can play a card. A player is not allowed to play any card at any given moment, so this needs to be checked. Furthermore, a player is not allowed to play a card when it is not their turn. If this is the case, the playerPlaysCard function rejects this attempt and sends a message to the GUI, which will then reset the card to its original location. The same thing occurs when a bot player tries to play out of turn. If a player is allowed to play the selected card and it is in fact their turn, then this function calls the performCardLogic function described above.

Lastly, playerDraws() allows new cards to enter the game from the deck. This is essential to the game, as without this, only the setup of the game would be possible. Additionally, this function determines whether the card that was drawn is an Exploding Kitten card. If this is the case, it will call the necessary functions that will respond according to the rules of the game.

Most important associations

In this diagram, multiple associations are depicted. The most important association is the composition of Game, Deck and Player. As is shown in the diagram the multiplicity of Game and Deck is one: there can only be one game, and in one game there can only be one deck. Without Game, Deck and Player cannot exist. So this is a one-to-one relationship. The relationship between Game and Player is slightly different than the relationship between Game and Deck, in the sense that there can be multiple players in a single game. So the multiplicity between Player and Game is not one, but one to five. Which constitutes a one-to-many aggregation (one game has many players). However, in this implementation there can only ever be a single human player, and up to four bot players. Therefore the multiplicity of BotPlayer is between one and four, and the multiplicity of HumanPlayer is one.

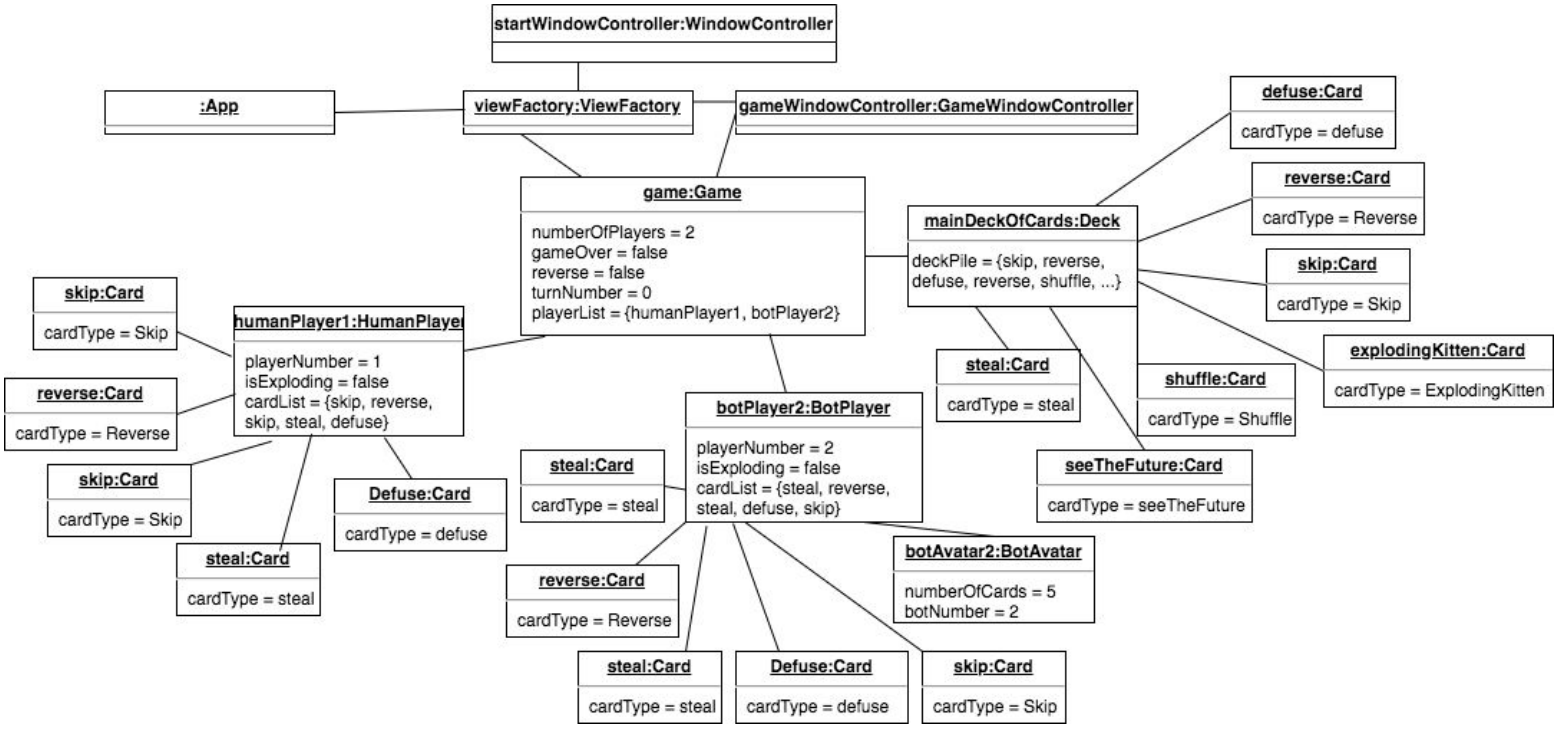
These two types of players inherit methods and properties from their parent abstract class Player. In our implementation only bot players and human players can be instantiated, therefore the Player class is indeed abstract. Another example of inheritance in the class diagram is that of the Window Controller and its subclasses StartWindowController and GameWindowController.

The relationship between Card and Deck is also a composition and a one-to-many aggregation. An instance of a Card cannot exist after the deletion of the Deck. A deck can theoretically contain all cards, which is indicated by the asterisk. The same goes for the relationship between Card and HumanPlayer or BotPlayer. If a player is deleted, then so are their related instances of Card, so this relationship is also a one-to-many aggregation (one player has many cards). A HumanPlayer or BotPlayer could theoretically possess all instances of Card, which is also indicated by an asterisk.

Lastly, the relationship between Card and CardType is a nested relationship, because the enumeration CardType resides within the Card class.

Object diagram

Authors: Sam and Gal

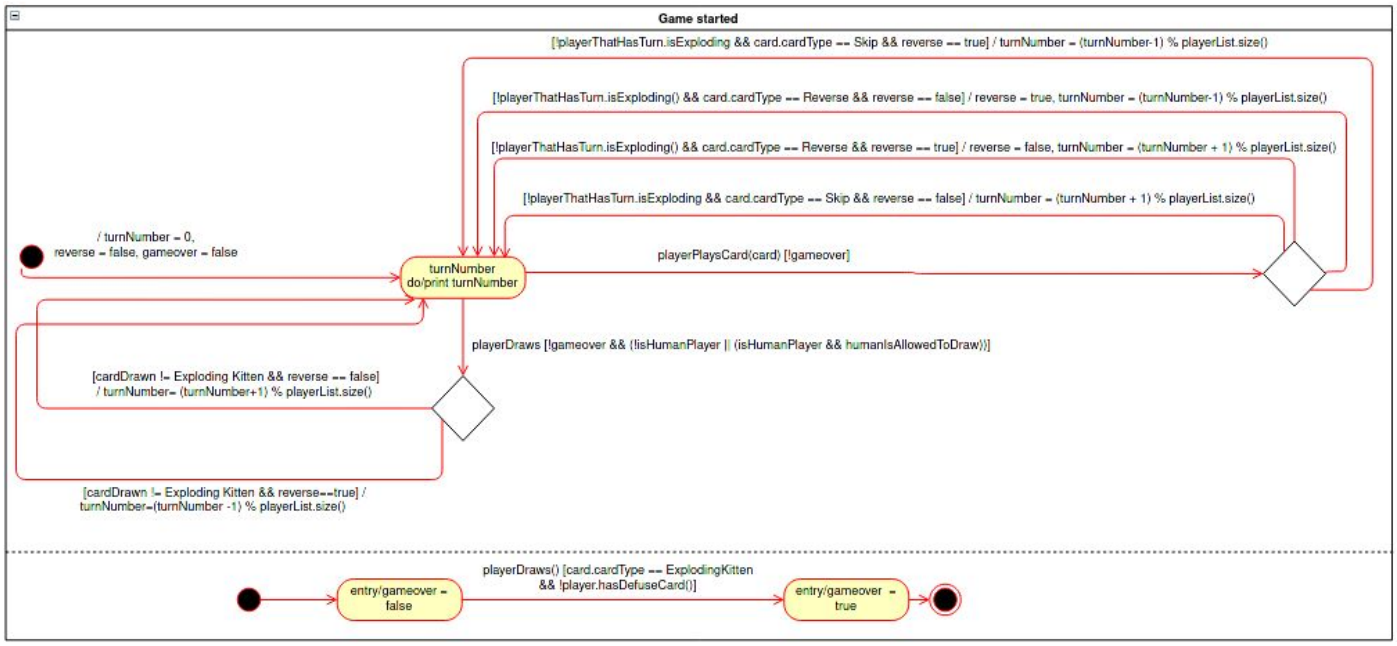


The presented Object diagram depicts a snapshot of the Exploding Kitten game in action. Creating the Object diagram helped us reason about the underlying mechanisms of our implementation of the game, which attributes each class should have and how they interact with one another. By observing the diagram, we can clearly see a strongly connected main component which is the game:Game Object. The game:Game object determines the tone for several other classes associated with it. This key element holds several attributes. One of which is numberOfPlayers. The numberOfPlayers determines how many players would play the game. The first player is the HumanPlayer and the rest are BotPlayers. In this snapshot, we chose to model a situation where only 2 players are playing the game and therefore there are one human player and one bot player in the diagram. Each player is modeled as a separate object and associated with other objects. In this snapshot, each of the players is associated with several types of cards as the attribute cardList holds the different cards. Similarly, the object mainDeckOrCards:Deck has an association with each card type as well. That is because the Deck object holds the deckPile attribute which contains instances of the cards.

State machine diagrams

Authors: Sam, Shahrin

Game: game.*turnNumber* | game.*gameover* states



In the figure above, the states *turnNumber* and *gameover* in the *Game* class have been visualized in a composite orthogonal state. For this figure, a general overview of the operations that affect the turn-number and gameover states has been modeled. The events that trigger the transitions are:

1. *playerDraws()*
2. *playerPlays()*

The model attempts to depict the state changes that occur after the player chooses one of the aforementioned events.

<u>Guards:</u>
Initially, the *turnNumber* is set to zero and *gameover* is set to *false*. At any point in the game, the player can only play or draw if the game is not over. Therefore all the guards that affect the turn-number check whether the game is over. Moreover, if a player has drawn an exploding kitten (player.*isExploding)* their *PlayCard* action is only limited to playing a defuse card, therefore all the events associated with the turn-number have a condition to check whether the player is allowed to play a card (*humanIsAllowedToDraw* checks for this in the *draw* action). Finally, the reverse condition dictates whether the turn-number is incremented or decremented.

The guards for the *playerDraws* events only allow a change in the turn-number as long as the drawn card is not an exploding kitten, in which case the turn-number is not immediately affected.

The gameover state is only changed when the event is of *playerDraws* type with the guards: the drawn card is an exploding kitten *and* the player that draws the card has no defuse cards.

PlayerPlays(card) events and activities:

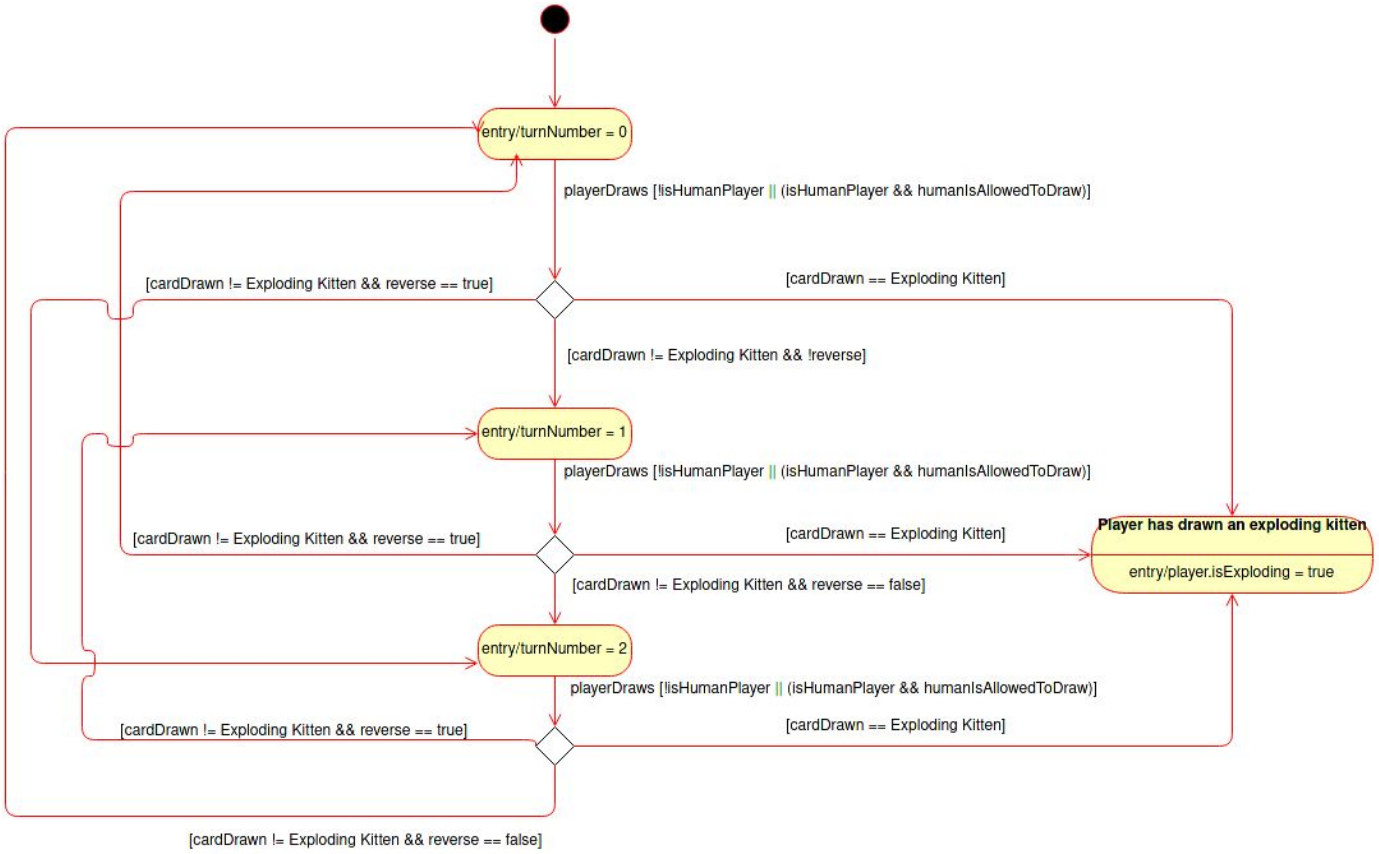
The turn number is increased or decreased by incrementing or decrementing the turn number and performing a modulo on it with the *playerList.size*.

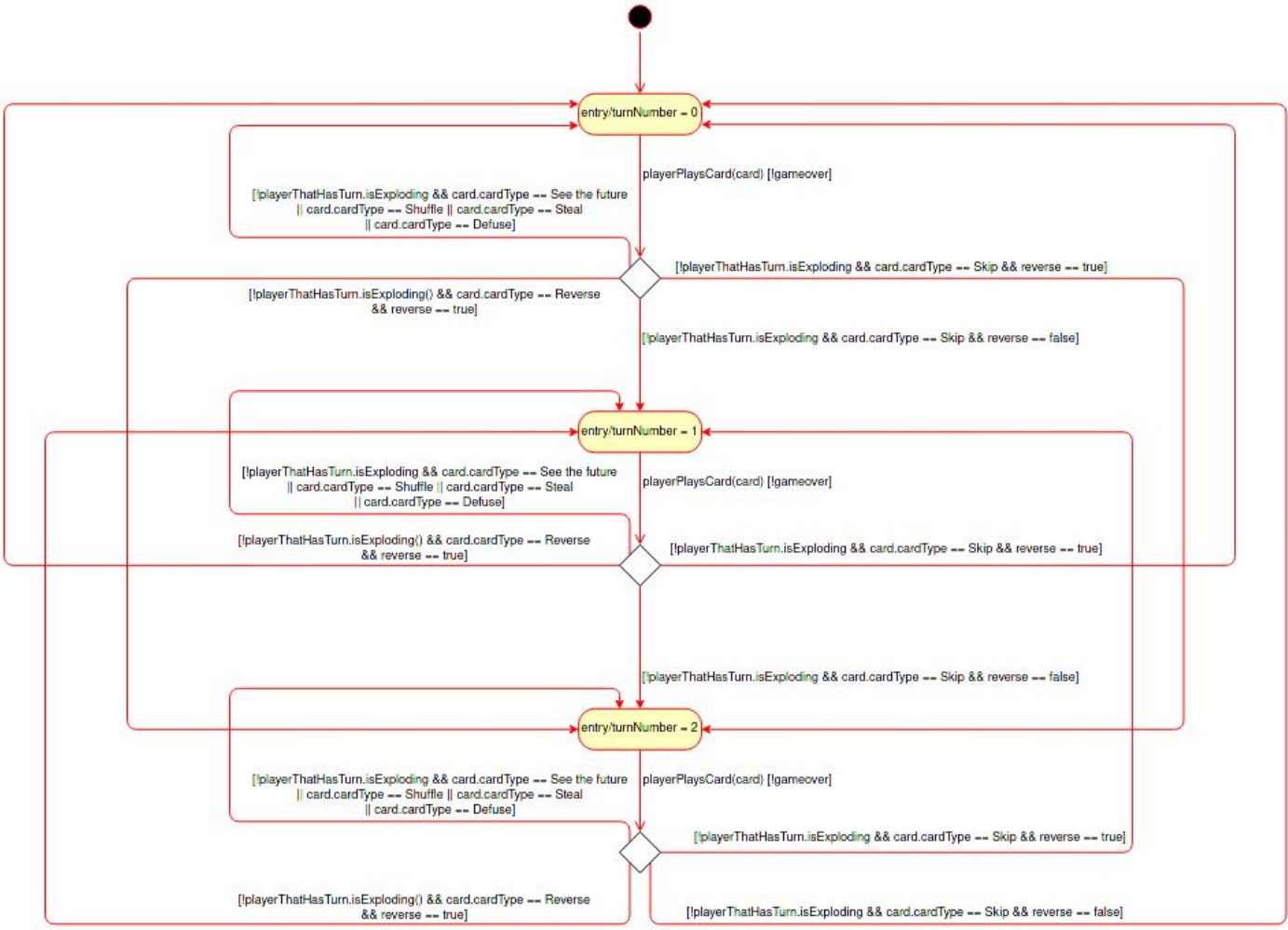
Not all played cards affect the turn-number state. The cards that do affect the turn-number are the skip and the reverse card. Turn-numbers are first incremented if reverse is disabled (*false*) and first decremented if reverse is enabled. When a skip card has been played, the turn-number changes according to the reverse state. The same holds true for the reverse card however, with the reverse card flips the reverse state before changing the turn number.

PlayerDraws(card) events and activities:

The turn-number is incremented and decremented in the same manner as the *playerPlays* events.

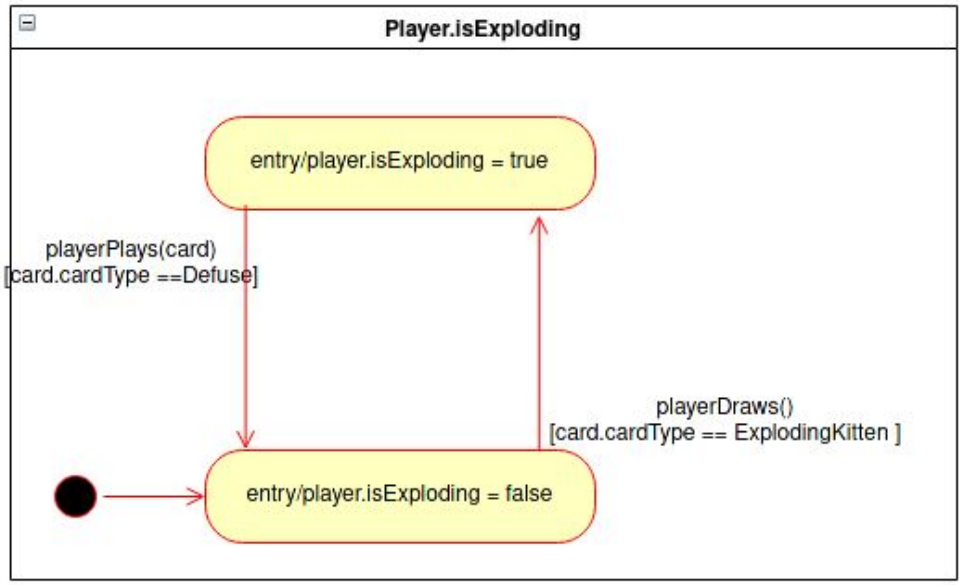
Game: game.turnNumber up (with specific states)





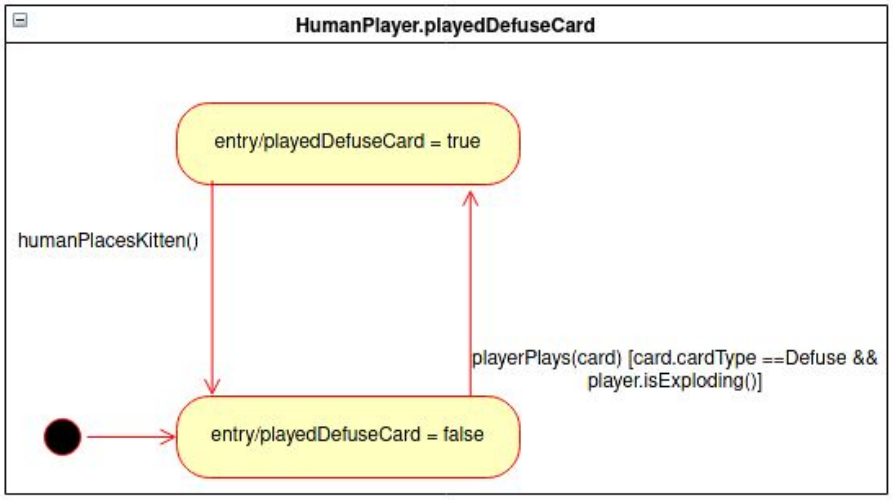
The above figures represent the same state as previously described (turn-number) - however with this model we try to explicitly state the different turn numbers with this model. The models are for the events *playerDraws* and *playerPlays* respectively.

Player: *isExploding* state



The *isExploding* state changes from true to false or vice versa depending on whether the player plays a defuse card and the drawn card is an exploding kitten card or not. As it can be seen from the given figure, if an exploding kitten card is drawn by the player, then player.isExploding is set to true. On the other hand, if a defuse card is played by the player, then plater.isExploding is set to false. In short, the figure shows the state transition of a player in the game of the particular moment where the player is either exploding or not exploding.

HumanPlayer: *playedDefuseCard* state

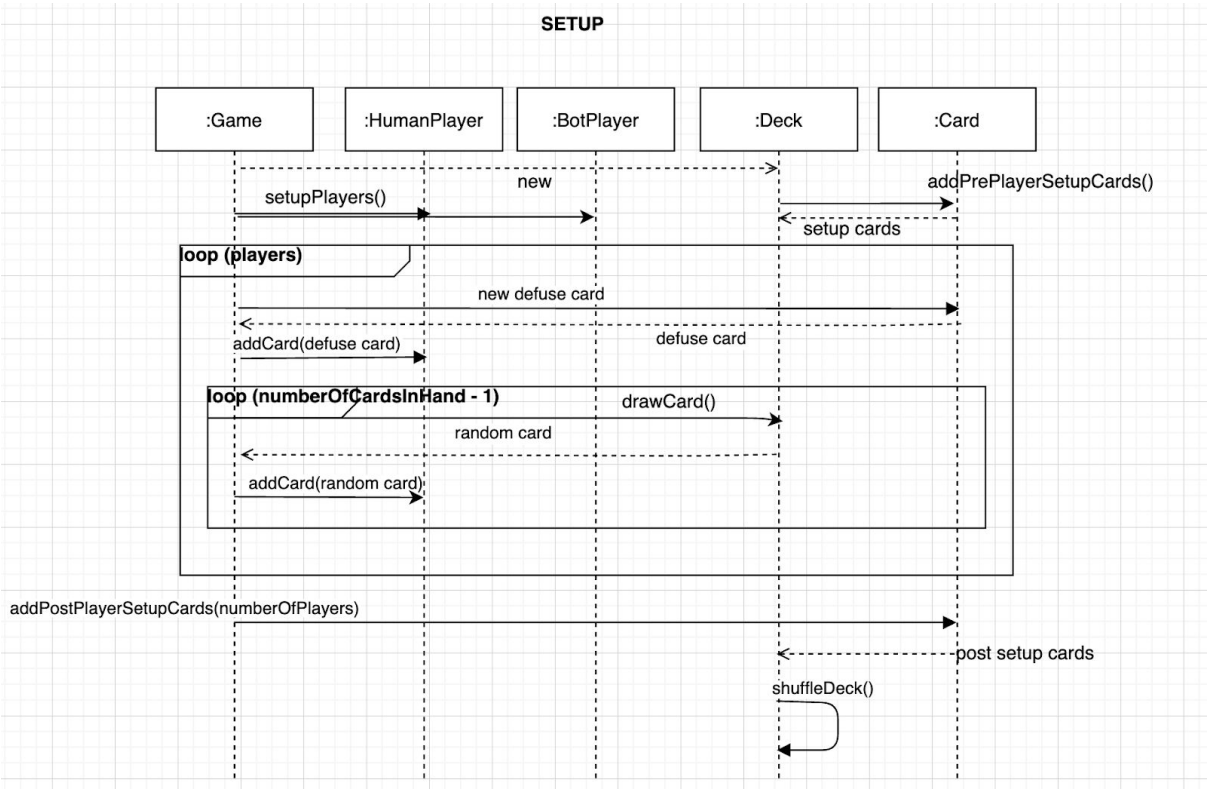


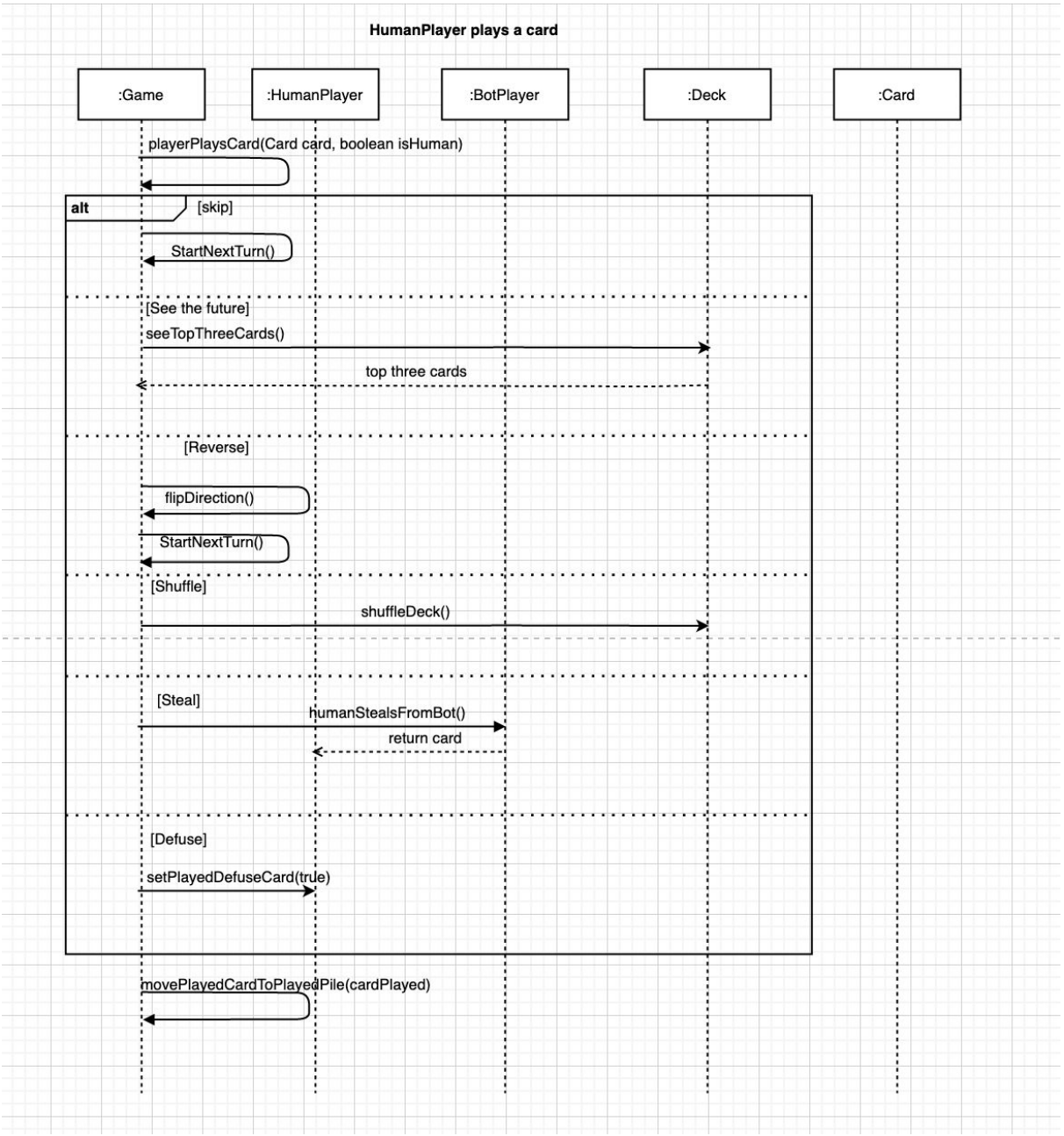
The *playedDefuseCard* state changes from true to false or vice versa depending on some certain scenarios. In the scenario of having player.isExploding() set to true, if the player

plays the defuse card, then the playedDefuseCard is also set to true. On the other hand, in the scenario of having humanPlacesKitten() set to true, if the player places the exploding kitten card back in the deck after playing the defuse card, then playedDefuseCard is again set back to false and new turn starts for a different player. In short, the figure shows the state transitions of a player in the game of the particular moment where it checks for the status of whether a defuse card has been played by the player or not.

Sequence diagrams

Gal, Sifra





Human Player Plays A Card

The HumanPlayer Plays A Card sequence diagram is an interaction overview diagram that describes the process of the user (a human player) playing a card. The diagram is composed of five object lifelines that interact with one another through messages. The five classes are Game, HumanPlayer, BotPlayer, Deck, and Card.

When a user chooses to play a card, they select a card from the GUI interface. The GUI interface interacts directly from the Game class and therefore isn't mentioned in the diagram as a class on its own. Our diagram begins with the Game class calling the playerPlaysCard function. This function checks which player plays the current turn, if this player is in fact a human, then the function continues by executing their selected card. There are seven different types of cards in the deck, however, as the exploding kitten card cannot be held in a

player's card pile, a player can only play six different types of cards. We model the implementation's switch statement using the alternative interaction fragment.

In the case of a Skip card, the Game class executes the StartNextTurn() function which skips the player's turn. When See The Future Card is played, the Game class interacts with the Deck class through the function playerPlayedSeeFuture(), which displays the top three cards of the deck back to Game. The Reverse card calls the flipDirection() function that changes the direction of the game. When the Shuffle card is played, the Game class interacts with the Deck class using the function shuffleDeck(), this function simply shuffles the deck's cards in a random manner. The Steal card executes the function HumanStealsFromBot(), which sends a message directly from the Game to the BotPlayer class, this function takes one card from a bot player and adds it to the human cards pile. The last card in the alt fragment is the Defuse card. When the Defuse card is played, the Game class interacts with the HumanPlayer class through the function playerPlaysDefuseCard(Card card), the function lets the human player defuse an exploding kitten card, then prompts the user from an index to reinsert the exploding kitten back into the deck.

After a card has been chosen and played the function movePlayedCardToPlayedPile(cardPlayed) is called. which moves the played card to the played cards pile. At this point, the human player has the option to repeat the process by selecting another card or continuing their turn by drawing a card when applicable.

Setup

The first sequence diagram depicts the setup of the game. Firstly, the deck is instantiated by the Game class. During instantiation, the function addPrePlayerSetupCards() is called, which fills the deck with cards except for cards with the type Exploding Kitten and Defuse. Then the Game class interacts with the HumanPlayer and the BotPlayer through the function setupPlayers(). This function creates an instance of the HumanPlayer and as many instances of the BotPlayer class as the user opts to play against. After this has been done, the cards are dealt through two loops. Each player takes 5 cards in total. In the outer loop the game class calls the function addCard(), which in turn creates a new defuse card. Each player then adds one defuse card to their hand. In the inner loop the game class calls drawCard(). This function causes the deck to return a random card from the series of cards that were generated by addPrePlayerSetupCards(). Each player adds 4 cards to their hand. Then after the loop, the game class calls the addPostPlayerSetupCards(numberOfPlayers) function. This loads the Exploding Kittens and Defuse cards into the deck. The number of Defuse cards and Exploding Kittens cards is dependent upon the number of players in the game, which is why they were not added in the addPrePlayerSetupCards() function. After having put the kittens and defuse cards in the deck, the deck is shuffled and the game can start.

Implementation

Strategy: UML → Java

The given steps in the modeling process (class, object, state machine and sequence diagrams) guided us with an intuitive approach to designing the game.

By starting off with the Class Diagram, we were able to think about the necessary classes that were needed to play a simple game of Exploding Kittens. Classes such as *Player, Deck, Card* & *Game* were the among the first classes to be modeled in the class diagram. Several methods such as adding a card to a player, removing cards from the deck and having methods to set up the game were quickly added to our class diagram. This provided us with a good starting point for the implementation. On a high level, a class diagram translates naturally to an object oriented programming language such as Java and implementing the respective classes in the code proved to be a trivial task.

However, classes with just function declarations and uninitialized values are far from a working implementation. As we started working on the implementation we frequently had to go back to the models and adjust our design. Some attributes and methods proved to be redundant and others were missing altogether from the models. Moreover, some parts of the model were only added *after* we had learned how to make a specific library work. In our case, this was especially true with the JavaFX library, as we now had to incorporate Controller classes for the GUI and have it interact with the game.

After having the class diagram in a relatively final and stable state, we were able to pick up the pace with the other models, however the back and forth between modeling and implementing the code (and vice versa) continued all throughout the development process.

By modeling the state diagram we were able to reduce redundancy in our implementation and concretely define the conditions needed for state changes to occur. By using the sequence diagram model it was easier to reduce unnecessary interaction between our objects and therefore also reduce redundancy in the implementation.

Key obstacles and their solutions

Initially, several problems required solutions. These were as follows:

1. How does the game decide whose turn it is to play?

Solution:
By implementing a turn number in the *Game* class and incrementing/decrementing the number and performing a modulo operation on it with the size of the *playerList*, the game is able to tell which player is currently allowed to make a move. Since the size of the *playerList* is dynamic (increasing when a player is added and decreasing when a player has been eliminated), this simple solution is scalable for any number of players. The turn number behavior is vital to the (turn based) game as it enforces fair play and normal game behavior.

2. How does a player choose which card to play and how do we implement this? How can a player choose which player to steal from?

Solution:
The GUI has been implemented with event-listeners so that the player can click on the card they want to play (with a label showing their selection) and by clicking on the *Play* button the user can finalize their action. Similarly, when a player plays a *Steal* card they can click on the Bot avatar they'd like to steal from.

3. How does a player keep track of the game (whose turn it is, what the other players played?)

Solution:
By keeping a text area which logs the actions played by the other players, as well as logging whose turn it is, we hoped to achieve smooth communication between the game and the player.

4. How does a player place the exploding kitten back in the deck?

Solution:
Through the use of a text input area, the user is able to choose an index for where in the deck they want to place the kitten.

Project settings

- Modular JavaFX project using Maven
- Project SDK/language level: 11
- (Fat) jar built using the Maven Shade plugin (multiplatform)

Main Java class

Location: $softwaredesign.Launcher$

Jar File

Location: $target/Software-design-vu-2020-1.0-SNAPSHOT.jar$

Demo footage

https://youtu.be/582FucPh5EU

Time logs

Member	Activity	Week number	Hours
All of us	Class diagram	3, 4 and 5	6
All of us	Class diagram description	3, 4 and 5	2

Gal, Sifra	Sequence diagram description	3, 4 and 5	3
Gal, Sifra	Sequence diagram	3, 4 and 5	5
Shahrin, Sam	State machine diagram	3, 4 and 5	5
Shahrin, Sam	State machine diagram description	3, 4 and 5	2
Sam, Gal	Object diagram	3, 4 and 5	2
Sam, Gal	Object diagram description	3, 4 and 5	2
Sam	Implementation	3, 4 and 5	18
Sam	Implementation description	3, 4 and 5	1
Gal	Implementation	3, 4 and 5	4.66
Sifra	Implementation	3, 4 and 5	6
Shahrin	Implementation	3, 4 and 5	6
All of us	Mentor meeting	3, 4 and 5	0.5
		TOTAL	63.16