Assignment 3

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

Summary of changes of Assignment 2

Author(s): All

Type	Issue	Solution
Implementation	1) "[SEVERE] Game class feels like a god class with almost 400 lines of code that has too much responsibility. Try to split its responsibilities so to better define its role"	1) Some of the responsibilities from the *Game* class have been abstracted to the *Card* and *Command* classes. The *Game* class has been reduced in size by ~50 lines. However, the *Game* class remains relatively large at 350 lines. After implementing the command design pattern and the factories, it was decided that splitting the Game class into multiple classes would make the design larger than desired. Therefore, a choice was made to implement the new design patterns, decrease the game

		class slightly in size but keep it as one class.
Class diagram	1) "Note that ideally in the textual description you give an rational about the design decisions and evaluate them. For example, why did you make Player an abstract class and have botPlayer and Human player inherit from it?" 2) "one-to-many composition* in the diagram :)" 3) "True, but the asterisk means infinitely many. So in theory the player could have more cards than were initially in the deck"	1) 1)The textual description has been updated for our class diagram in accordance with the given feedback where the brief description of attributes covers how abstraction has effect on the attributes 2) The same applies for the rest, such as the brief description of associations has been updated where the multiplicities of cards of human player and bot player has been fixed.
State machine diagram	1) "[SEVERE] Right now you just model how the value of an variable changes, not the object. So I dont think turnNumber is suited to be an abstract state for game. The variable decides whose turn it is, so a more suitable states would be Turn or NotTurn for a Player object (which would be in a separate diagram with different transition events)." 2) "Those states are suitable for the game object. Make sure to name the states though"	1) In our new diagrams, we made significant changes where we renamed every state. Instead of modeling the turnNumber, we choose to model the HumanPlayer object. Which has the states: hasTurn, notTurn, playerExploding and playerDead. 2) Similarly, we updated our game over diagram completely by getting rid of the following two diagrams: isExploding and playedDefuseCard where the new diagram shows the

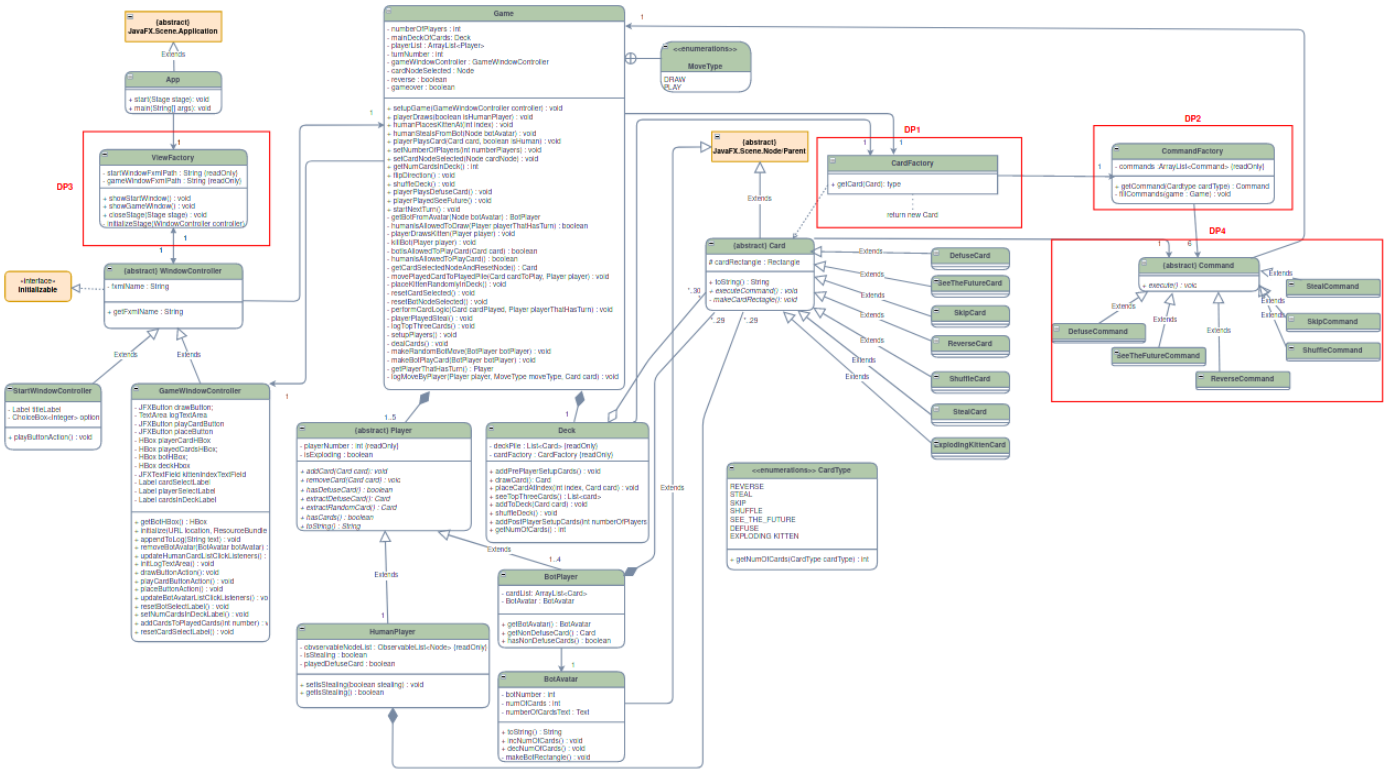
	3) "[SEVERE] You may be confusing activity diagrams with state machine diagrams since this models the flow of the execution where each node is an activity, whereas a state machine diagram models the flow from state to state" 4) "[SEVERE] Make sure to name the states (nouns) e.g. Dead and Alive"	flow between the states of the object game.
Sequence Diagram	1) "[SEVERE] Both setUpPlayers() and addPrePlayersSetup Cards() are too simplistic. Note that both are function calls within Game and Deck, respectively, which both use a loop to setUp and create the respective Player and Cards objects. This is not shown in the diagram and currently is not very informative. " 2) "Note that when a object is created with the constructor, it is shown by letting the arrow point to the box. I may have unintentionally ill advised you to show it like this in one of our meetings, which is why this wont affect your grade (I misunderstood the scenario you were giving)." 3) "[SEVERE] How can you accurately show the sequence of	1) The new diagrams are more complex and display each of the functions which are called using Setup. We have added a loop for the setUpNumber of players and for the addPrePlayerSetupCards(). 2) We fixed this by letting an arrow point to the box itself. 3) We have added the user figure and showed the interaction between the user and the GUI. 4) We got rid of the card in the second diagram as it wasn't needed. 5) Same as 3. 6) We have decided to model see the future card instead. 7) The new sequence diagrams are more detailed and we believe illustrates the ideas better.

	interactions when a HumanPlayer plays a card without a user as stick figure and GameWindowController? For example, here the user triggers the playCardButtonAction in GameWindowController that then calls the playerPlaysCard. Thereby you also show how GameWindowController comes into play in the system as a whole." 4) "You can get rid of Cards object when there is no interaction with it" 5) "[SEVERE] This is again where the user is missing. The top three cards are shown to the user, not Game" 6) "The above comments applies to this section as well. E.g. "Select a player to steal a card from!" should be printed from gameWindowController and then the user selects a player. ""Showing it would have made the sequence diagram much more accurate and clearer" "i.e. by setting reverse to its negation." 7) "Which is not shown in the sequence diagram. Generally, I think both sequence	

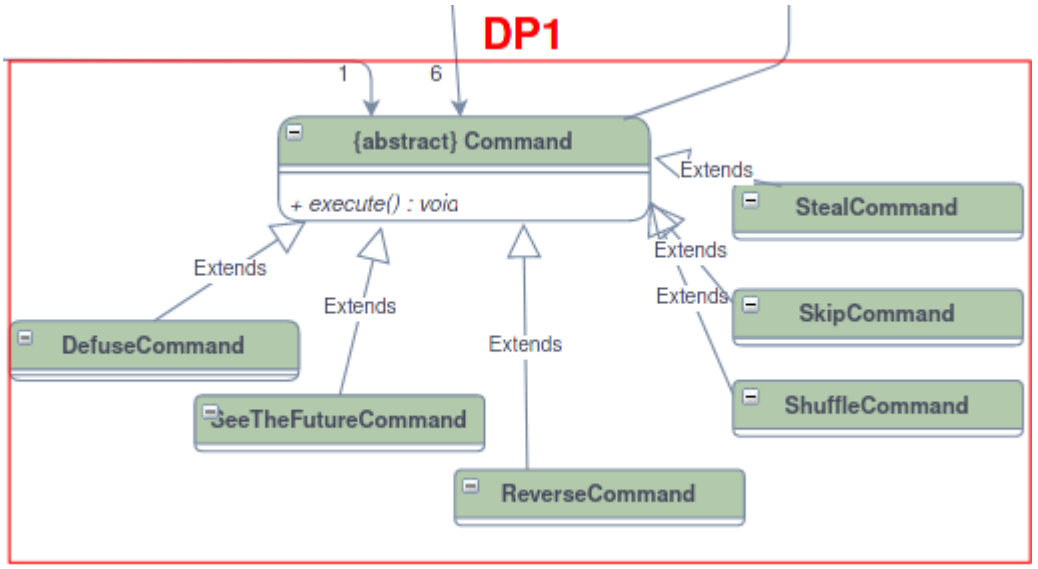
	diagrams are too simplistic" "Exactly, but I got this from the code and not from the sequence diagram"	

Application of design patterns

Author(s): Sam



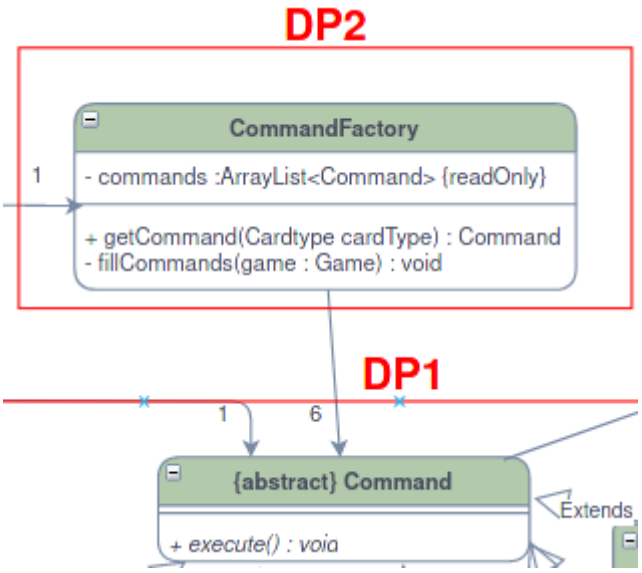
Command: a behavioral design pattern (DP1)



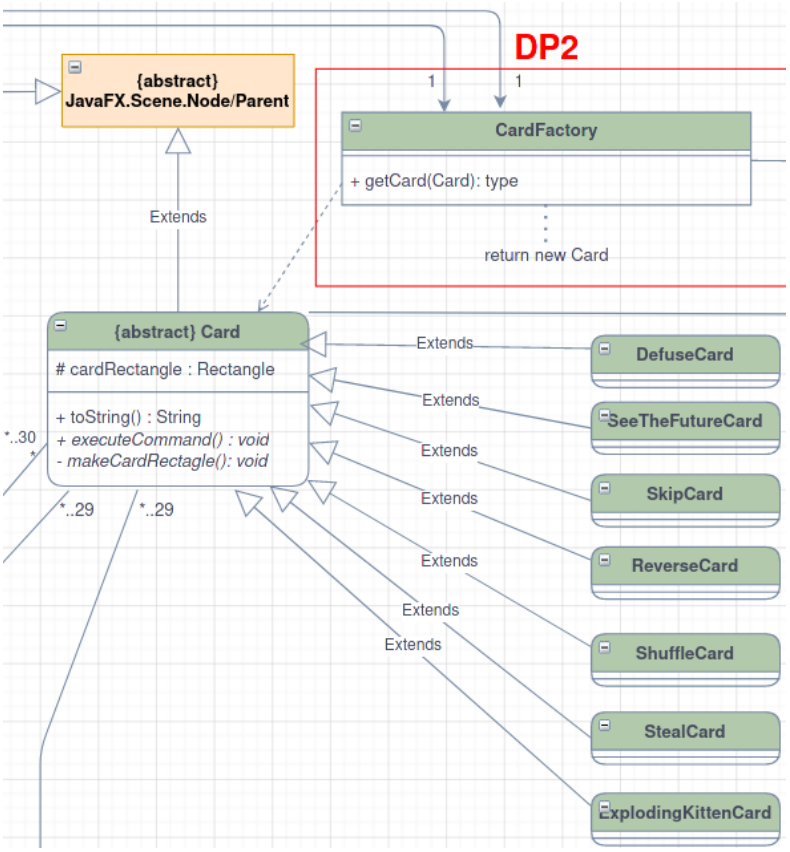
	DP1
Design pattern	Command
Problem	In the previous implementation, the actions of the cards were not encapsulated in the cards themselves. Instead, in the *game* class, a check would be done on the type of card and this would perform the required action when the card was played.
Solution	To encapsulate the actions and make them independent from the way of their execution, the new implementation uses the *command* design pattern to implement a command into each card. Each card has a *command* attribute which performs the required action of the card. Thus encapsulating the action of the card into the card itself. Furthermore, another level of abstraction is added where each card has an executable command, regardless of the card.
Intended use	At runtime, when a player plays a specific card, the card's *command* is retrieved in the game class and executed.
Constraints	Naturally, this implementation requires the *game* class to execute the command of each card, where previously the process had to be initiated inside the class itself. Therefore, the commands need to have access to the Game class. This is a design trade-off. Complexity is increased in favor of abstraction.
Additional remarks	-

Factory: a creational design pattern (DP2)

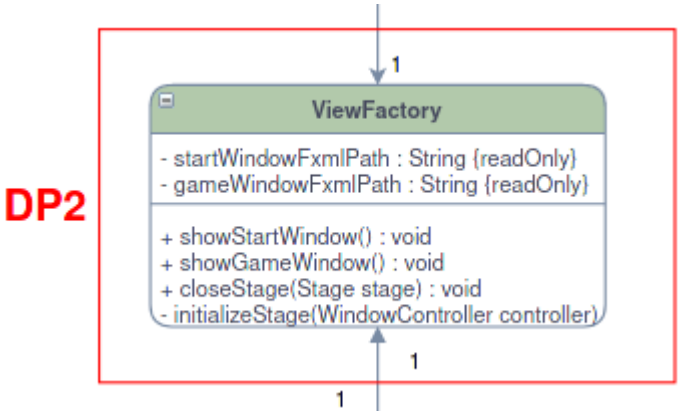
CommandFactory:



CardFactory:



ViewFactory:

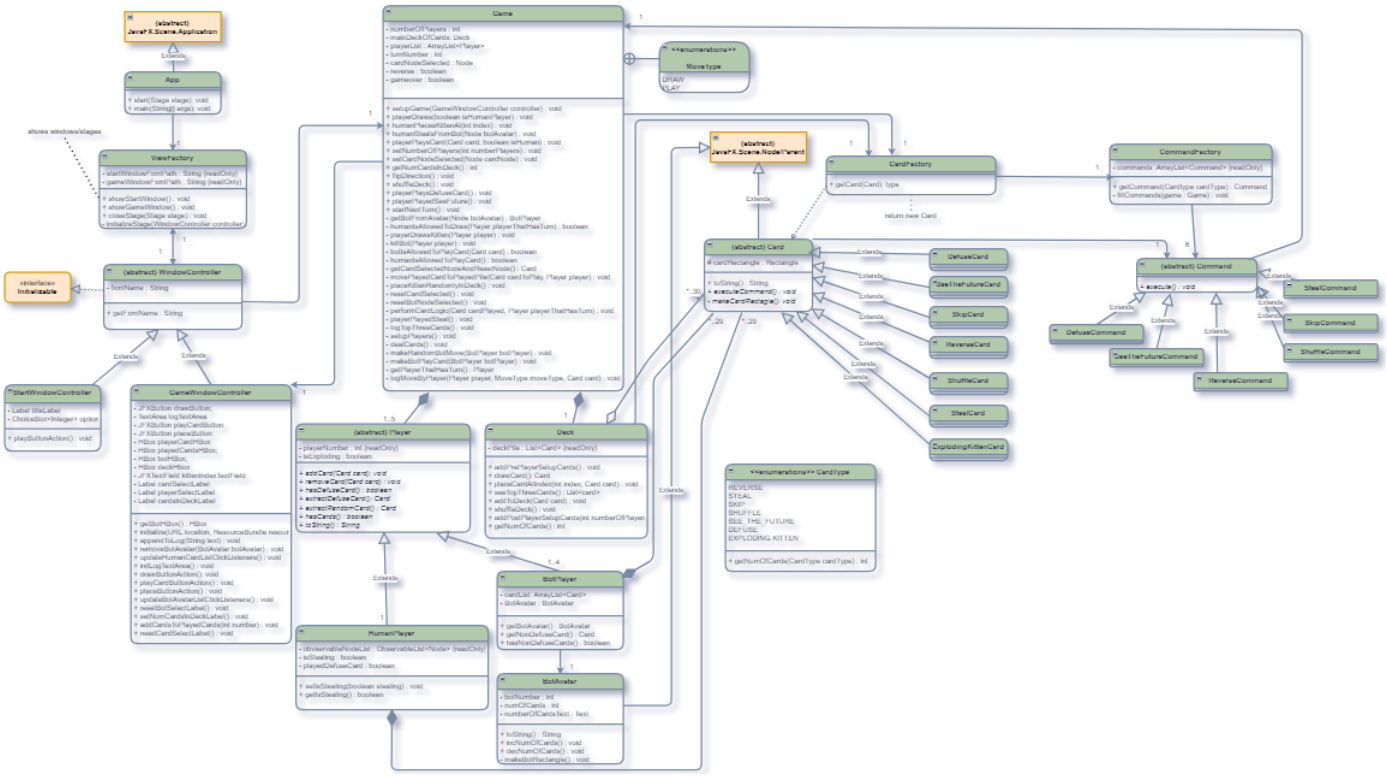


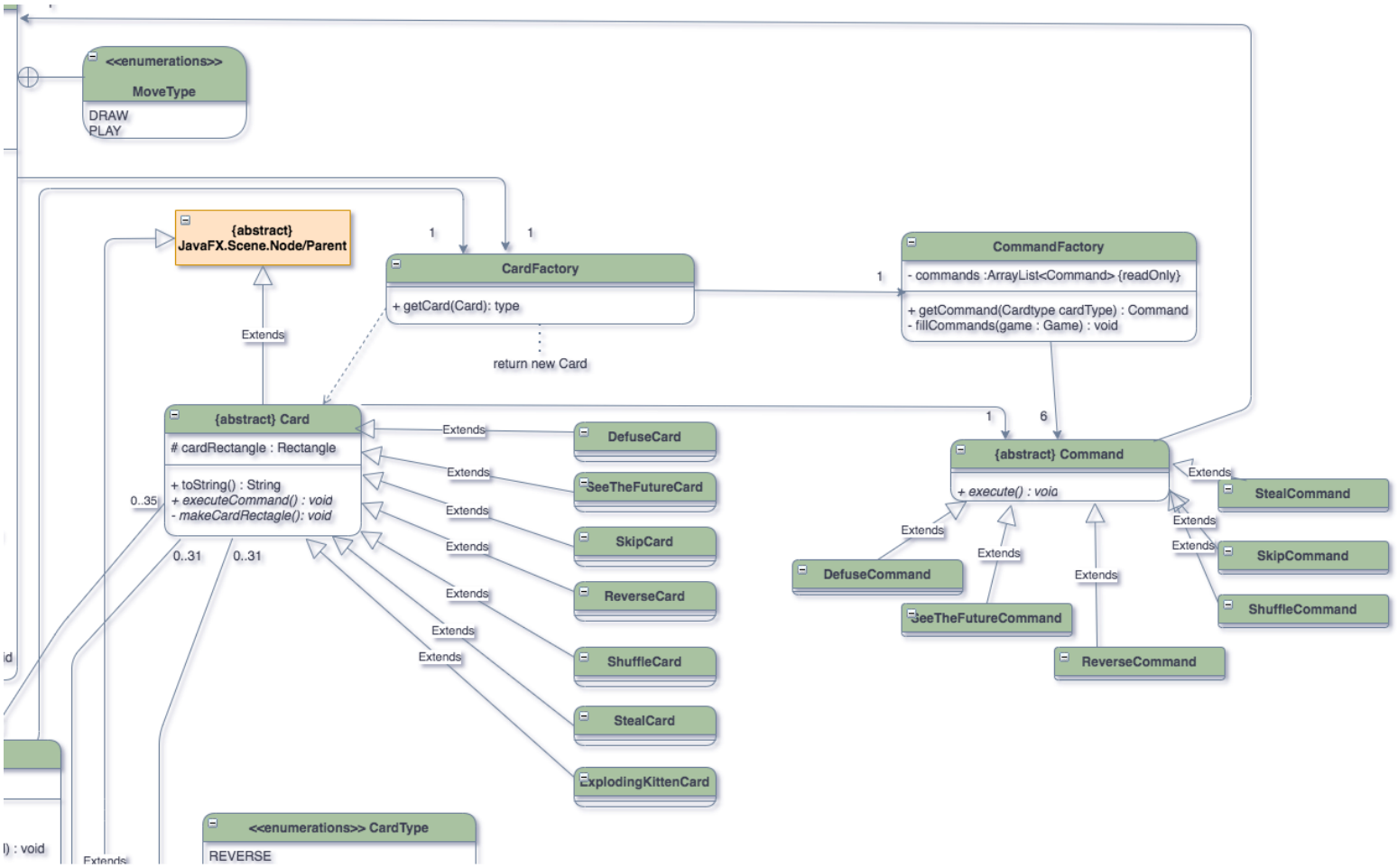
	DP2
Design pattern	Factory
Problem	The initial implementation would have been satisfactory had we decided that the game will not grow any larger than it is. However, since the game should be modular enough to be able to support the implementation of new cards/expansion packs added to the game, we had to make use of factory design patterns to allow it to do so. This would increase complexity in the short term but allow for growth in the long term. The problems which led to the creation of the factories were: - growth in the implementation would introduce redundancy in the code and design dependencies - lack of centralization *and* customization in object and window creation - hard-coded object creation Specifically, without a factory for showing the stages StartWindow and GameWindow, the stages would have to be initialized redundantly in the class that wishes to initialize them. This redundancy would grow with the number of stages that are added to the game. For card creation, the new operator was used extensively in various classes and thus hard coded everywhere. This again would introduce exceedingly more clutter in the code if new cards were to be added to the game. Moreover, since cards have *commands*, each card would have to have to have an association link to the *game* class. From a design perspective, we wanted to avoid this dependency from *Card* to *Game*.
Solution	For the stages (views), cards and commands, respectively a *ViewFactory, CardFactory and CommandFactory* were implemented. By abstracting the creation of views and creating the ViewFactory,

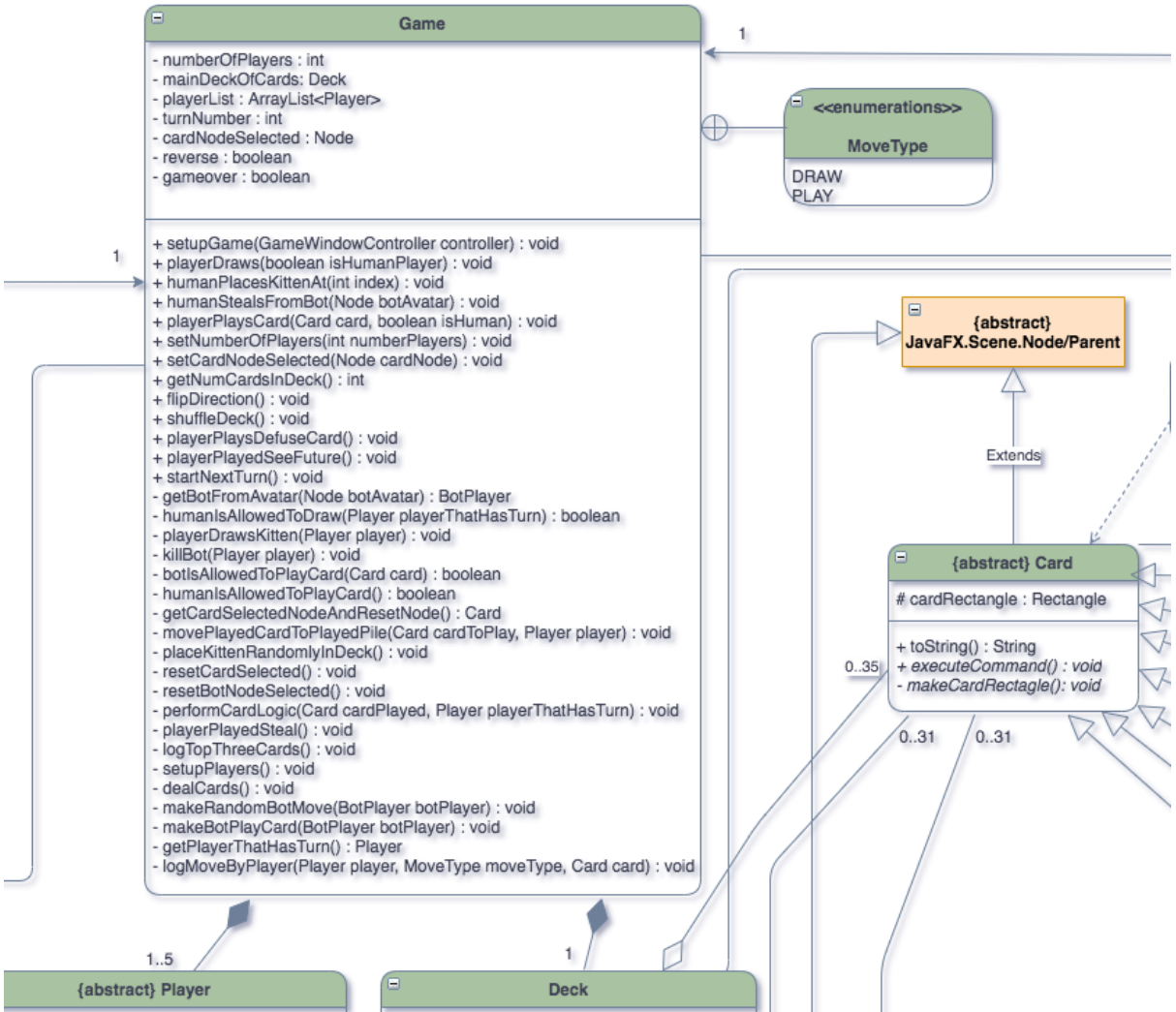
	redundancy in the code was reduced when a new window was prompted to load. By abstracting the creation of cards and commands and creating the CardFactory and CommandFactory, respectively, we limited the hardcoding of card creation, allowed for greater customizability in the future and eliminated the coupling from card to class by letting the CommandFactory take over that responsibility and initializing commands on construction with the instance of the game class.
Intended use	ViewFactory: At runtime *showStartWindow()* and *showGameWindow()* can be called to load the respective stages. These methods are intended to be called when a new window needs to be loaded. In our implementation, the methods are called initially after launching the app and after clicking the *playButton*, respectively. CardFactory: *getCard(CardType)* returns a new (subclass of the) Card object. Classes requiring (new) cards can make use of the CardFactory to do so. In our implementation, the factory method is called in the *Game* class and naturally in the *Deck* class. CommandFactory: *getCommand(CardType)* returns a new Command specific to the CardType. The only class that has access to the CommandFactory is the CardFactory. The factory method is only called when new cards are created and a command object is needed for the card.
Constraints	Currently, the *CommandFactory* is simple and depends on the CardType enumeration. It requires the ExplodingKittento be the last value in the enum, moving the ExplodingKitten to a different place in the enumeration would break the logic. In the future a more complex solution is required to solve this dependency.
Additional remarks	

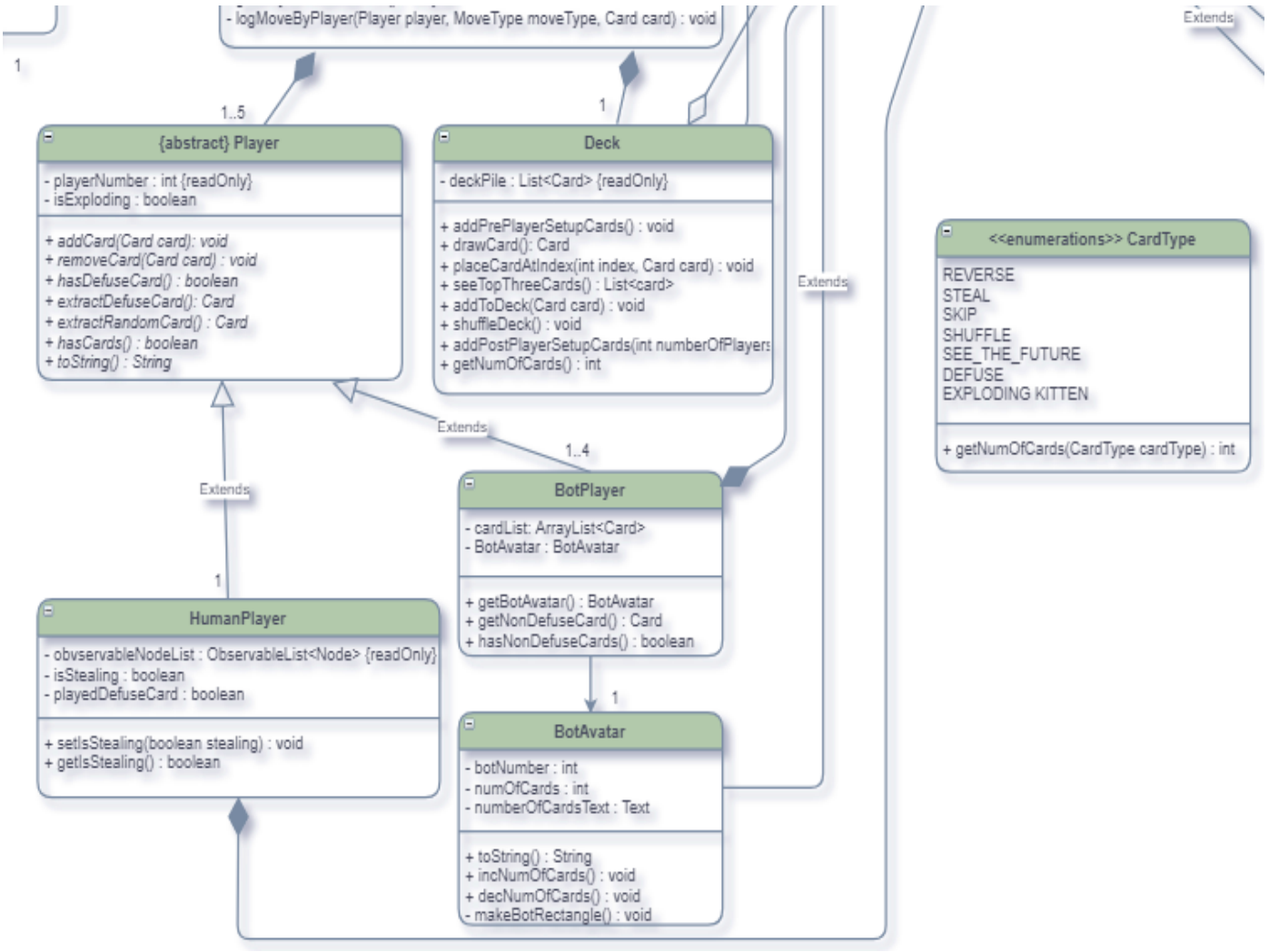
Class diagram

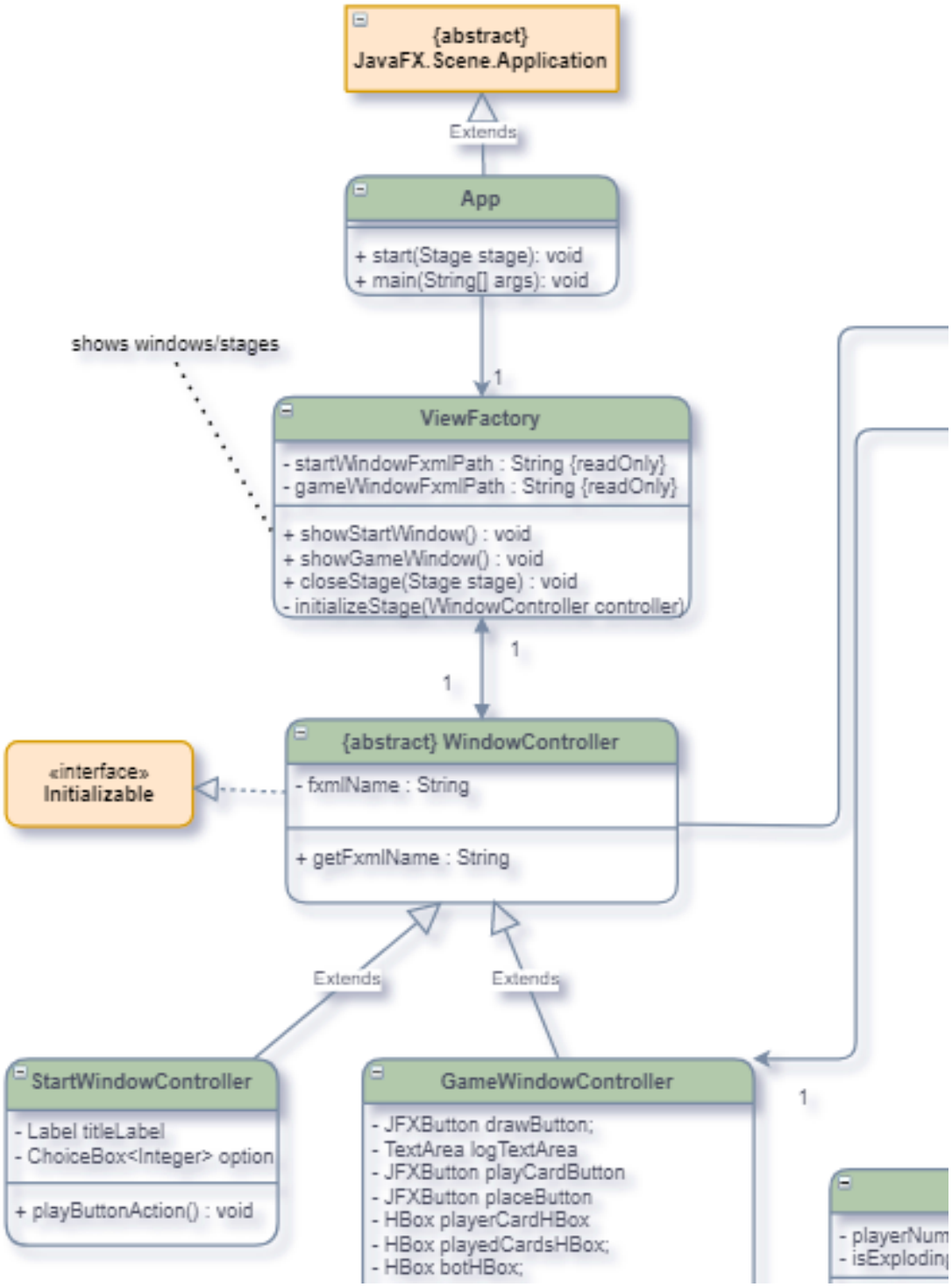
Author(s): Shahrin, Sifra, Gal

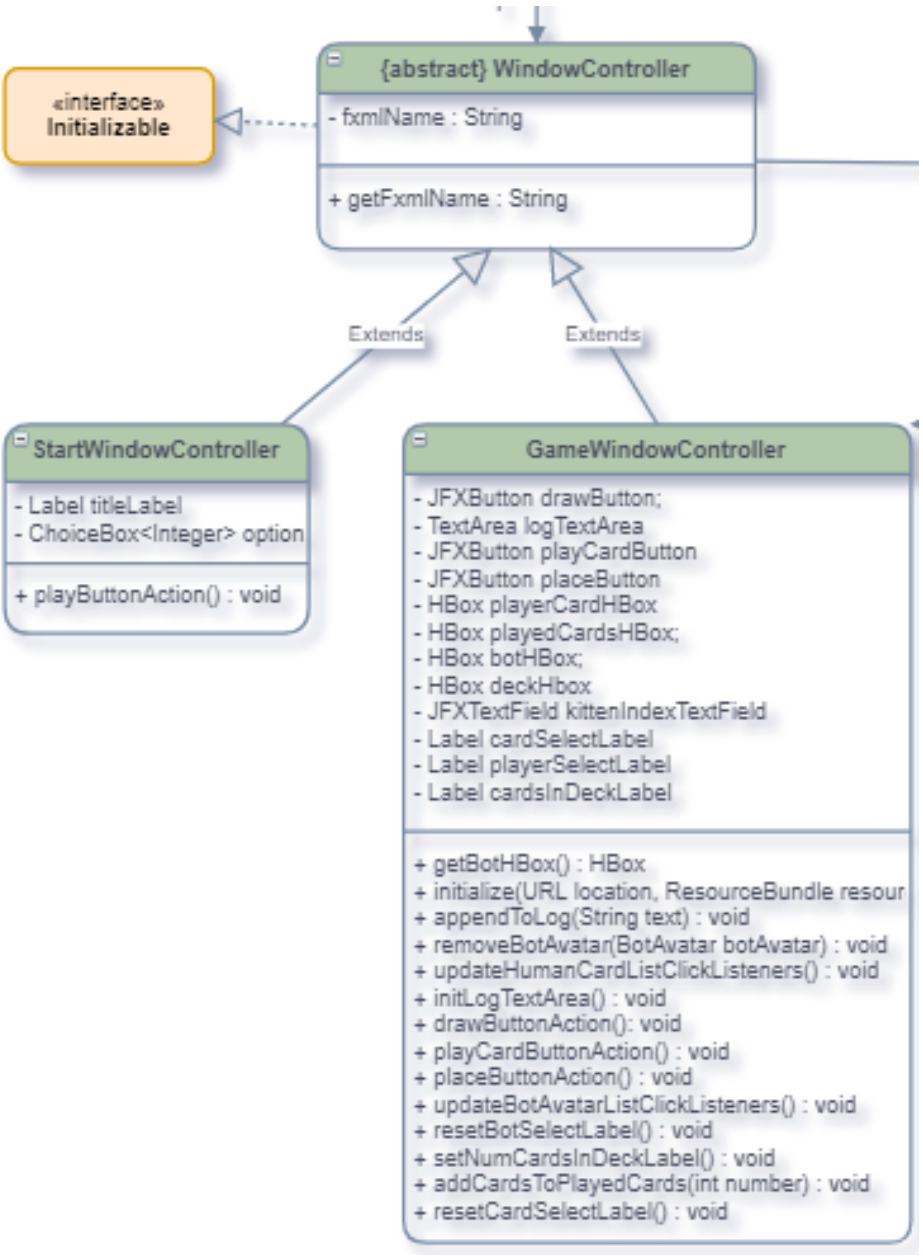

(abstract) JavaFX.Scene.Application
App
+ start(Stage stage) : void
+ main(String[] args) : void
shows windows/stages
ViewFactory
- startWindowFxmlPath : String {readOnly}
- gameWindowFxmlPath : String {readOnly}
+ showStartWindow() : void
+ showGameWindow() : void
+ closeStage(Stage stage) : void
- initializeStage(WindowController controller)
<interface> Initializable
(abstract) WindowController
- fxmlName : String
+ getFxmlName : String
StartWindowController
- Label titleLabel
- ChoiceBox<Integer> optionBox
+ playButtonAction() : void
GameWindowController
- JFXButton drawButton
- TextArea logTextArea
- JFXButton playCardButton
- JFXButton placeButton
- HBox playerCardHBox
- HBox playedCardsHBox
- HBox botHBox
- HBox deckHBox
- JFXTextField kittenIndexTextField
- Label cardSelectLabel
- Label playerSelectLabel
- Label cardsInDeckLabel
+ getBotHBox() : HBox
+ initialize(URL location, ResourceBundle resources)
+ appendToLog(String text) : void
+ removeBotAvatar(BotAvatar botAvatar) : void
+ updateHumanCardListClickListeners() : void
+ initLogTextArea() : void
+ drawButtonAction() : void
+ playCardButtonAction() : void
+ placeButtonAction() : void
+ updateBotAvatarListClickListeners() : void
+ resetBotSelectLabel() : void
+ setNumCardsInDeckLabel() : void
+ addCardsToPlayedCards(int number) : void
+ resetCardSelectLabel() : void
Game
- numberOfPlayers : Int
- mainDeckOfCards : Deck
- playerList : ArrayList<Player>
- turnNumber : Int
- cardNodeSelected : Node
- reverse : boolean
- gameover : boolean
+ setupGame(GameWindowController controller) : void
+ playerDrawsCard(boolean isHumanPlayer) : void
+ humanPlacesKittenAt(Int index) : void
+ humanStealsFromBot(Node botAvatar) : void
+ playerPlaysCard(Card card, boolean isHuman) : void
+ setNumberOfPlayers(int numberPlayers) : void
+ setCardNodeSelected(Node cardNode) : void
+ getNumCardsInDeck() : Int
+ flipDirection() : void
+ shuffleDeck() : void
+ playerPlaysDefuseCard() : void
+ playerPlayedSeeTheFuture() : void
+ startNextTurn() : void
- getBotFromAvatar(Node botAvatar) : BotPlayer
- humansAllowedToDraw(Player player ThatHasTurn) : boolean
- playerDrewaKitten(Player player) : void
- killBot(Player player) : void
- botIsAllowedToPlayCard(Card card) : boolean
- humansAllowedToPlayCard() : boolean
- getCardSelectedNodeAndResetNode() : Card
- movePlayedCardToPlayedFlexCard cardToPlay, Player player) : void
- placeKittenRandomlyInDeck() : void
- resetCardSelected() : void
- resetBotNodeSelected() : void
- performCardLogic(Card card?played, Player player ThatHasTurn) : void
- playerPlayedSteal() : void
- log topThreeCards() : void
- setup(Players) : void
- dealCards() : void
- makeRandomBotMove(BotPlayer bot?player) : void
- makeBotPlayCard(BotPlayer bot?player) : void
- getPlayerThatHasTurn() : Player
- logMoveByPlayer(Player player, MoveType moveType, Card card) : void
<<enumerations>> MoveType
DRAW
PLAY
(abstract) JavaFX.Scene.Node?parent
CardFactory
+ getCard(Card) : type
return new Card
CommandFactory
- commands :ArrayList<Command> {readOnly}
+ getCommand(CardType cardType) : Command
- fillCommands(game : Game) : void
(abstract) Card
cardRectangle : Rectangle
+ toString() : String
+ executeCommand() : void
- makeCardRectangle() :void
DefuseCard
SeeTheFutureCard
SkipCard
ReverseCard
ShuffleCard
StealCard
ExplodingKittenCard
(abstract) Command
+ execute() : void
DefuseCommand
SeeTheFutureCommand
StealCommand
SkipCommand
ShuffleCommand
ReverseCommand
(abstract) ?Player
- playerNumber : Int {readOnly}
- isExploding : boolean
+ addCard(Card card) :void
+ removeCard(Card card) : void
+ hasDefuseCard() : boolean
+ extractDefuseCard() : Card
+ extractRandomCard() : Card
+ hasCards() : boolean
+ toString() : String
Deck
- deck[Pile : List<Card> {readOnly}
+ addOne?ToPlayerSetupCards() : void
+ drawCard() : Card
+ placeCardAtIndex(int index, Card card) : void
+ seeTopThreeCards() : List<card>
+ addToDeck(Card card) : void
+ shuffleDeck() : void
+ addOne?ToPlayerSetupCards(int numberOfPlayers)
+ getNumOfCards() : Int
<<enumerations>> CardType
REVERSE
STEAL
SKIP
SHUFFLE
SEE_THE_FUTURE
DEFUSE
EXPLODING KITTEN
+ getNumOfCards(CardType cardType) : Int
HumanPlayer
- observableNodeList : ObservableList<Node> {readOnly}
- isStealing : boolean
- playedDefuseCard : boolean
+ setIsStealing(boolean stealing) : void
+ getIsStealing() : boolean
BotPlayer
- cardList: ArrayList<Card>
- botAvatar : BotAvatar
+ getBotAvatar() : BotAvatar
+ getNonDefuseCard() : Card
+ hasNonDefuseCards() : boolean
BotAvatar
- botNumber : Int
- numOfCards : Int
- numberOfCardsText : Text
+ toString() : String
+ incNumOfCards() : void
+ decNumOfCards() : void
- makeBotRectangle() : void


<<enumerations>>
MoveType
DRAW
PLAY
{abstract}
JavaFX.Scene.Node/Parent
CardFactory
+ getCard(Card): type
return new Card
CommandFactory
- commands :ArrayList<Command> {readOnly}
+ getCommand(Cardtype cardType) : Command
- fillCommands(game : Game) : void
Extends
{abstract} Card
cardRectangle : Rectangle
+ toString() : String
+ executeCommand() : void
- makeCardRectagle(): void
0..35
0..31
0..31
DefuseCard
SeeTheFutureCard
SkipCard
ReverseCard
ShuffleCard
StealCard
ExplodingKittenCard
{abstract} Command
+ execute() : voia
DefuseCommand
SeeTheFutureCommand
ReverseCommand
StealCommand
SkipCommand
ShuffleCommand
<<enumerations>> CardType
REVERSE


Game
- numberOfPlayers : int
- mainDeckOfCards: Deck
- playerList : ArrayList<Player>
- turnNumber : int
- cardNodeSelected : Node
- reverse : boolean
- gameover : boolean
+ setupGame(GameWindowController controller) : void
+ playerDraws(boolean isHumanPlayer) : void
+ humanPlacesKittenAt(int index) : void
+ humanStealsFromBot(Node botAvatar) : void
+ playerPlaysCard(Card card, boolean isHuman) : void
+ setNumberOfPlayers(int numberPlayers) : void
+ setCardNodeSelected(Node cardNode) : void
+ getNumCardsInDeck() : int
+ flipDirection() : void
+ shuffleDeck() : void
+ playerPlaysDefuseCard() : void
+ playerPlayedSeeFuture() : void
+ startNextTurn() : void
- getBotFromAvatar(Node botAvatar) : BotPlayer
- humanIsAllowedToDraw(Player playerThatHasTurn) : boolean
- playerDrawsKitten(Player player) : void
- killBot(Player player) : void
- botIsAllowedToPlayCard(Card card) : boolean
- humanIsAllowedToPlayCard() : boolean
- getCardSelectedNodeAndResetNode() : Card
- movePlayedCardToPlayedPile(Card cardToPlay, Player player) : void
- placeKittenRandomlyInDeck() : void
- resetCardSelected() : void
- resetBotNodeSelected() : void
- performCardLogic(Card cardPlayed, Player playerThatHasTurn) : void
- playerPlayedSteal() : void
- logTopThreeCards() : void
- setupPlayers() : void
- dealCards() : void
- makeRandomBotMove(BotPlayer botPlayer) : void
- makeBotPlayCard(BotPlayer botPlayer) : void
- getPlayerThatHasTurn() : Player
- logMoveByPlayer(Player player, MoveType moveType, Card card) : void
<<enumerations>>
MoveType
DRAW
PLAY
{abstract}
JavaFX.Scene.Node/Parent
Extends
{abstract} Card
cardRectangle : Rectangle
+ toString() : String
+ executeCommand() : void
- makeCardRectagle(): void
0..35
0..31
0..31
1..5
{abstract} Player
Deck


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


{abstract}
JavaFX.Scene.Application
Extends
App
+ start(Stage stage): void
+ main(String[] args): void
shows windows/stages
1
ViewFactory
- startWindowFxmlPath : String {readOnly}
- gameWindowFxmlPath : String {readOnly}
+ showStartWindow() : void
+ showGameWindow() : void
+ closeStage(Stage stage) : void
- initializeStage(WindowController controller)
1
1
«interface»
Initializable
{abstract} WindowController
- fxmlName : String
+ getFxmlName : String
Extends
Extends
StartWindowController
- Label titleLabel
- ChoiceBox<Integer> option
+ playButtonAction() : void
GameWindowController
- JFXButton drawButton;
- TextArea logTextArea
- JFXButton playCardButton
- JFXButton placeButton
- HBox playerCardHBox
- HBox playedCardsHBox;
- HBox botHBox;
1



Brief description of the class diagram

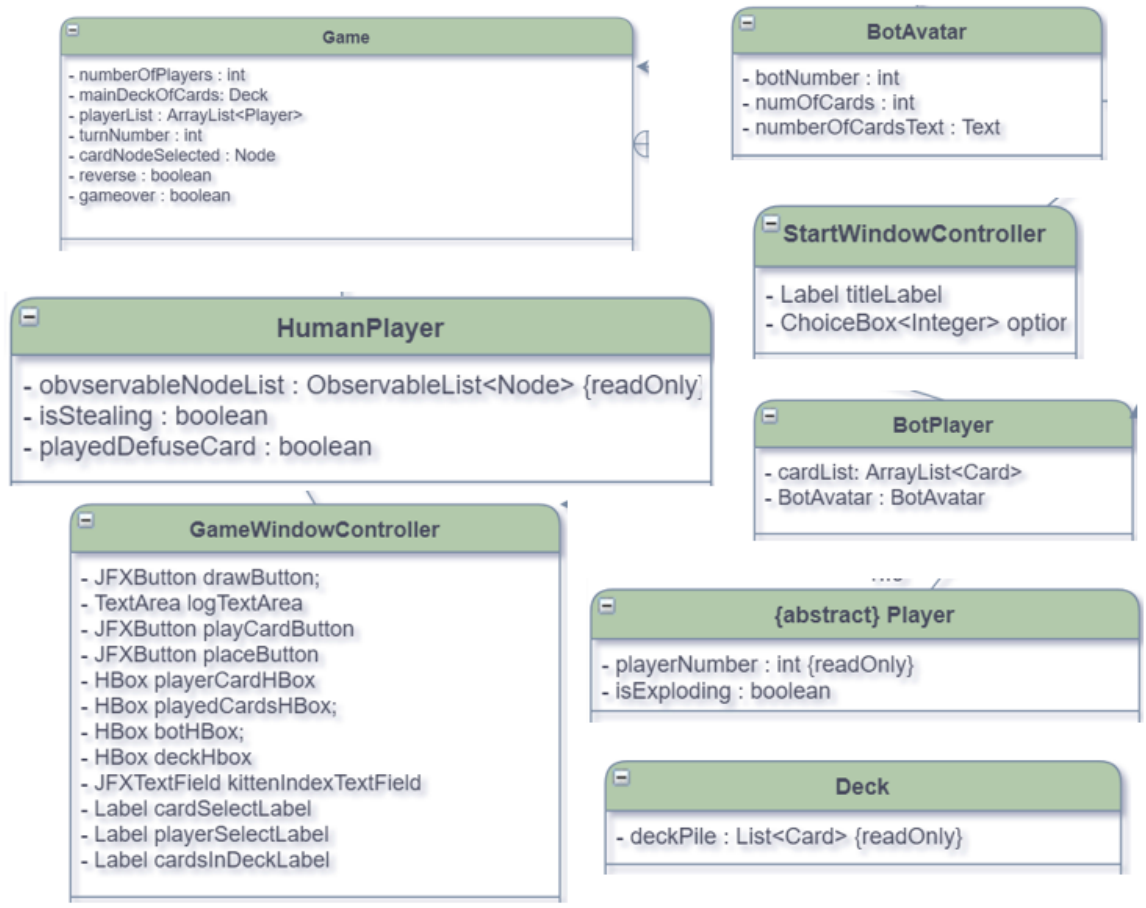
As we can see from the given UML class diagram, our game ‘Exploding Kittens’ is mainly based on about 28 classes – Game, App, ViewFactory, WindowController, StartWindowController, GameWindowController, Player, Deck, HumanPlayer, BotPlayer, BotAvatar, Card, CardFactory, Command, and CommandFactory etc. 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

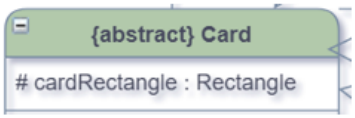
From the attached class diagram it can be seen that there are multiple public, private, and protected attributes that we have used to design our game.

Most relevant private attributes are as follows:



Similarly most relevant public attributes are as follows:

And most relevant protected attributes are as follows:



Thus, our class diagram suggests a number of attributes that reflect the semantics of our game. For example, apart from defining the private, public and protected attributes with various types, such as void, integer, String, boolean or other non-primitive types, we also define some attributes (i.e., deckPile, startWindowFxmlPath, gameWindowFxmlPath, playerNumber, commands etc) with the 'final' keyword, a non-access modifier. 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 oberservableNodeList 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. Additionally, in order to prevent an inadvertent instantiation, some classes are declared as abstract and the attributes used in such abstract classes can be accessed via creating objects of the subclasses derived by inheriting the abstract class. For example, the class Player is abstract and the classes HumanPlayer and BotPlayer are the subclasses that inherit the features of the abstract class Player and add the missing details or functionalities of the abstract class. This is how every attribute in our class diagram is responsible for each specific role 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() - executes a command. The function firstly logs the player's move, then based on the card that is selected, it executes a command. 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.

Brief description of most relevant 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. This type of relationship occurs with Command and Game as well. A command can have a single Game attribute.

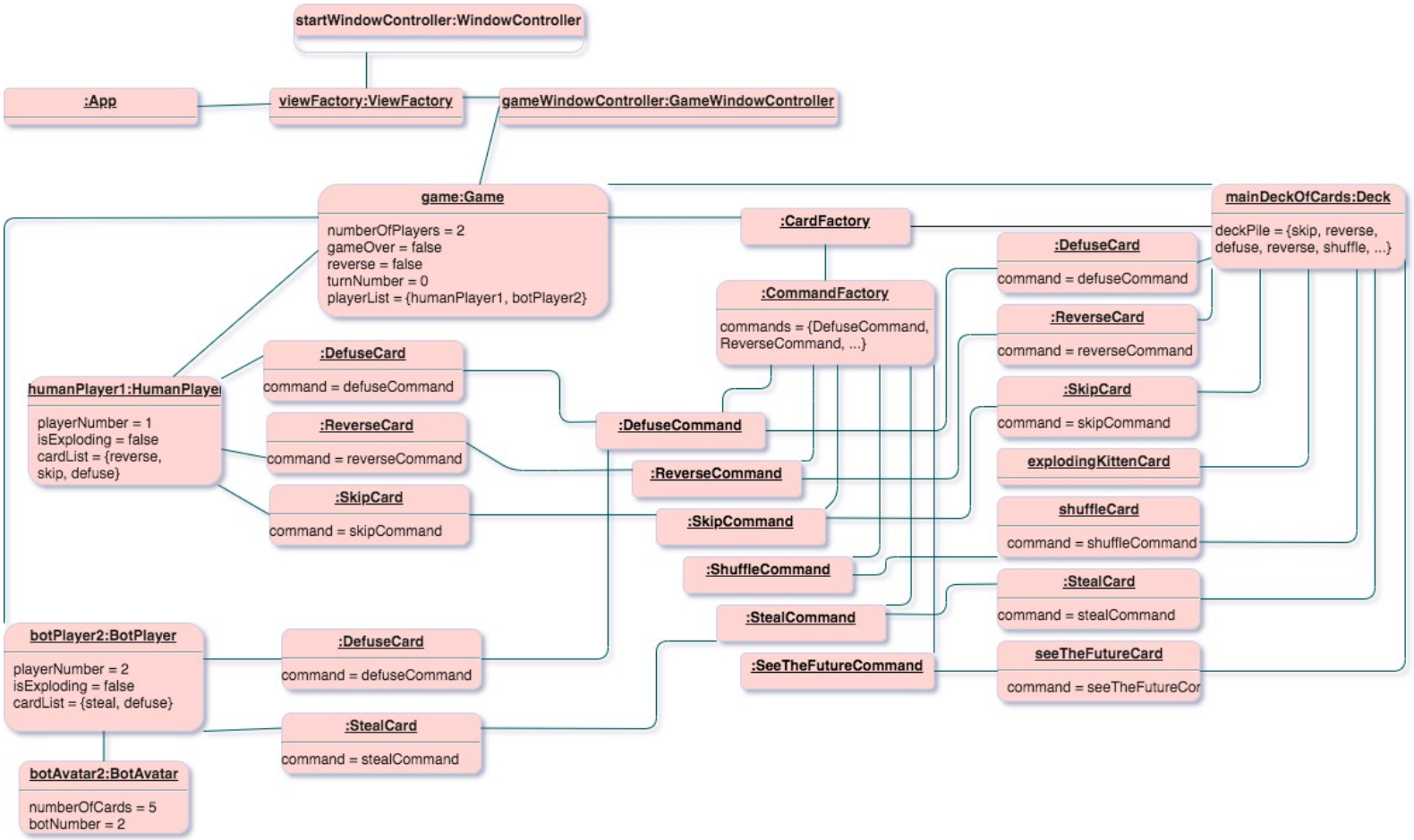
The relationship between Game and Player is slightly different than the relationship between Game and Deck, in the sense that there can be multiple players in a single game. So the multiplicity between Player and Game is not one, but one to five. Which constitutes a one-to-many composition (one game 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 and of course Command and all of its subclasses.

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 infinitely many cards, which is indicated by the asterisk, but it has to have a minimum of 30 cards. 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 infinitely many instances of Card, which is also indicated by an asterisk.

Object diagram

Author(s): Gal & Sam



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. In assignment 3 we applied the design patterns to our implementation and therefore made several changes in our Class diagram and Object diagram. These changes are depicted in the Object diagram and are explained in this document.

The game:Game object determines the tone for several other classes associated with it. This key element holds several attributes and associations with other objects. The Game object is connected to the CardFactory object which is associated with the CommandFactory object. The CommandFactory has an association with each type of card command (DefuseCommand, ShuffleCommand...etc). Additionally, the game:Game object has an association with the Deck object. The Deck object is navigable with each of the types of card objects (DefuseCard, ReverseCard ...etc). Each type of card has an association with its command. The same Commands objects are associated with the cards that each player is holding as when a player plays a card, its effect is determined and executed by the command object.

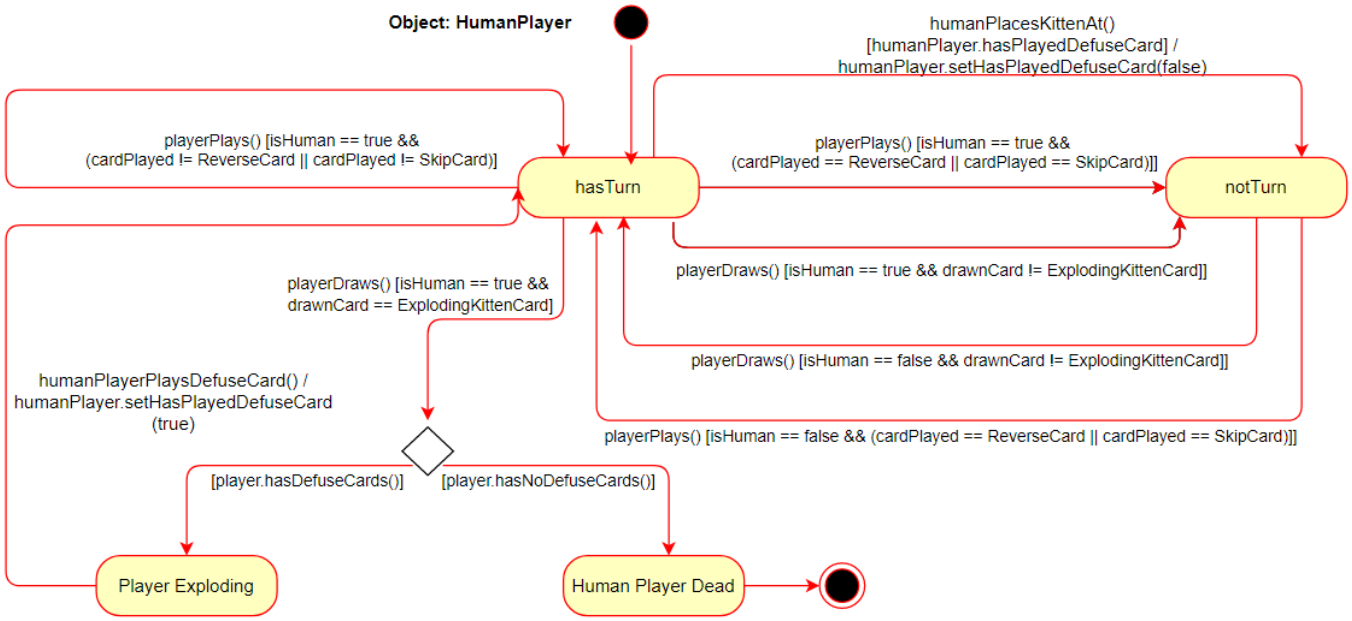
The numberOfPlayers attribute of the Game class determines how many players would play the game. The first player is the HumanPlayer and the rest are BotPlayers. In this snapshot,

we chose to model a situation where only 2 players are playing the game and therefore there is 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 card classes, each card has its own class.

State machine diagrams

Author(s): Shahrin

Class: HumanPlayer



In the figure above, the diagram shows that the HumanPlayer object under the HumanPlayer class can be in one of the following four states: hasTurn, notTurn, Player Exploding and Human Player Dead. For this figure, a general overview of the operations that affect the mentioned states has been modeled. By illustrating the relation of events, guards, and activities in states and transitions. the order of the whole system, in which the activities are executed, can be depicted.

Events/Triggers:
As it can be seen from the diagram, the events that trigger the state transitions are playerDraws(), playerPlays(), humanPlayerPlaysDefuseCard(), and humanPlacesKittenAt().

Guards:
At the beginning, the state machine is in the state hasTurn where five possible transitions can be seen to change the mentioned source state (hasTurn) to a different target state.

The turns are changed when state changes from hasTurn to notTurn because of three possible guards:

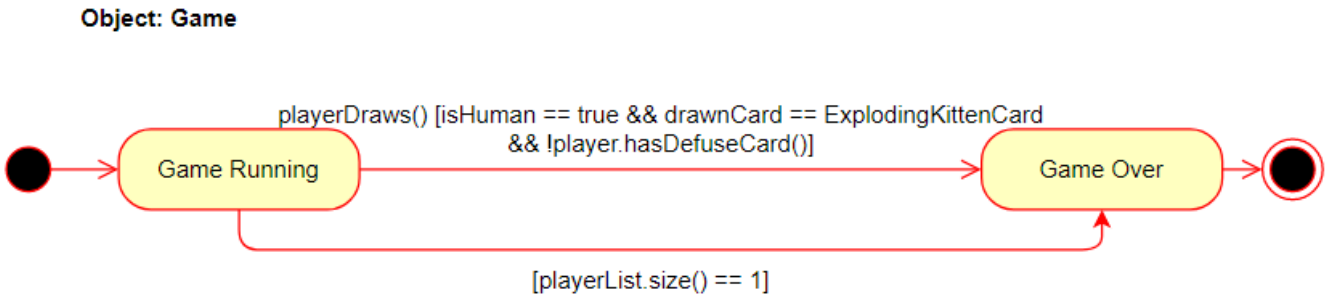
1. A reverse card or a skip card is played by the human player.
2. A defuse card is played by the human player.
3. A card has been drawn by the player that has turn and the drawn card is not exploding kitten.

Depending on whether a player has a defuse card or not, the state hasTurn is changed to either Player Exploding or Human Player Dead when the player draws an exploding kitten card. If the player has a defuse card the state hasTurn is changed to the state Player Exploding and in the opposite scenario, the state is changed to Human Player Dead, which leads to the final state of the diagram.

For the following guards a turn does not get changed:
1. A different card (i.e., Shuffle card, See the future card, Steal card etc) except for a reverse card and skip card is played by the human player.
2. Human player has a defuse card to play when the human player is exploding.

Activities/Effects:
The state hasTurn is manipulated by two kinds of effects or activities:
1. Human player's played defuse card is set to false (In this case, the state changes).
2. Human player's played defuse card is set to true (In this case, the state remains the same).

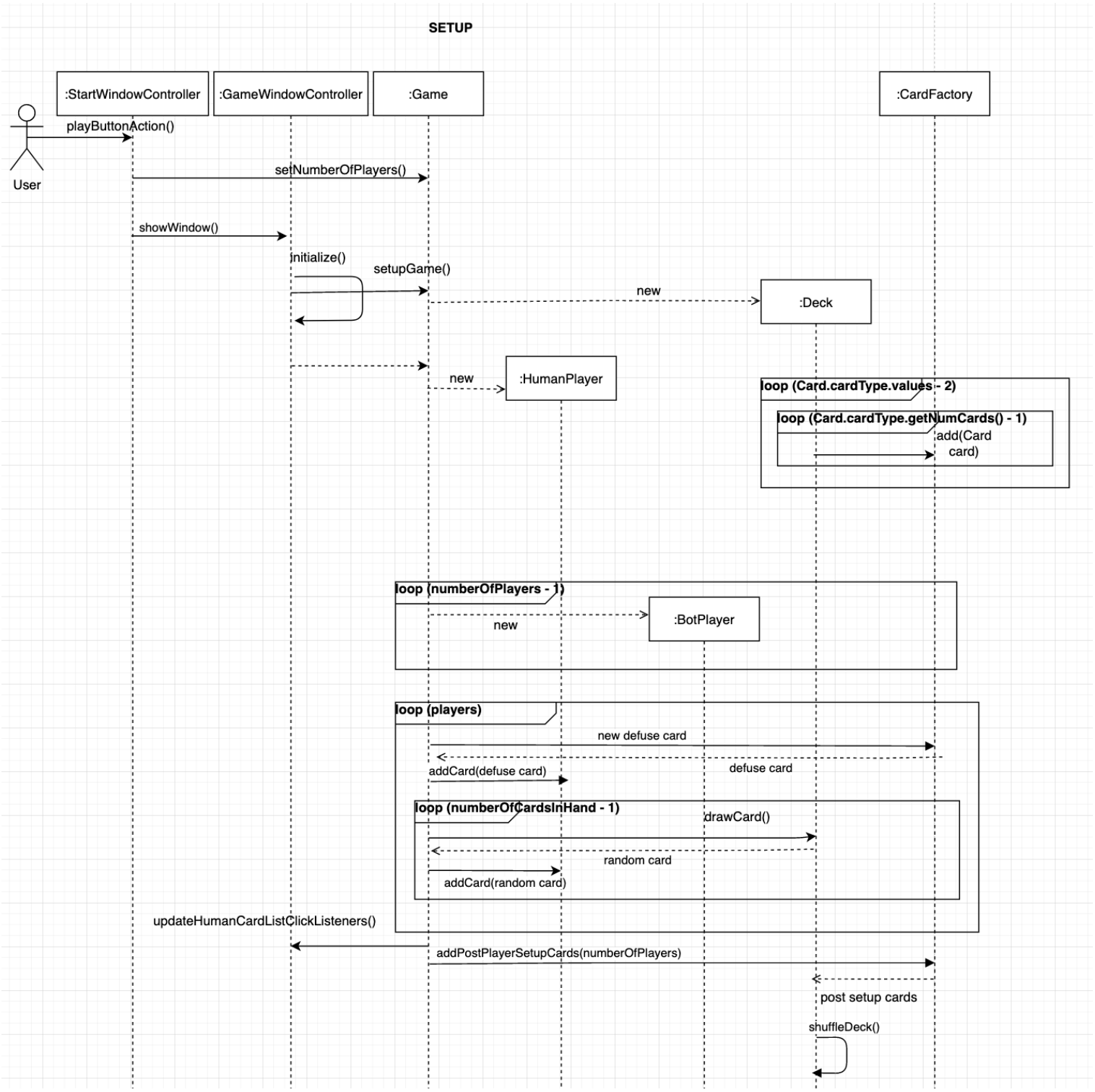
Class: Game



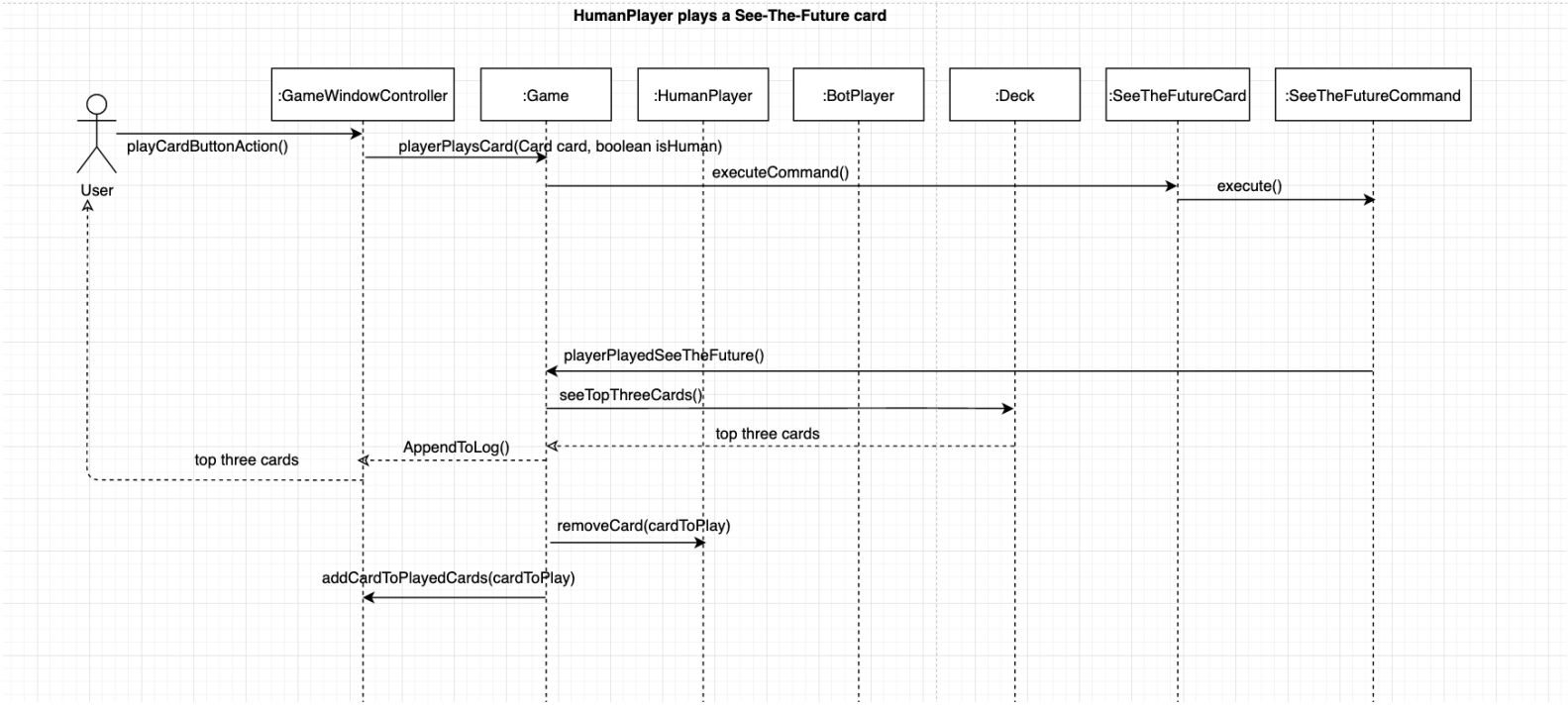
The diagram given above shows that the Game object under the Game class can be in one of the following two states: Game Running and Game Over. The state Game Running is changed to the state Game Over because of the event playerDraws() triggering the transition. In this case, if the card drawn by the human player is an exploding kitten card and the human player does not have a defuse card, then the state changes to Game Over, which leads to the final state eventually where the game is lost by the human player. On the other hand, if the number of players becomes equal to 1, then similarly the state changes from the state Game Running to the state Game over where the game is won by the human player.

Sequence diagrams

Author(s): Sifra and Gal



The first sequence diagram depicts the setup of the game. The user sets up the game by filling in the number of players they want to play against. Once the playButtonAction function is called, the number of players is set and the game window becomes visible to the user. After this the game is initialized. This means that a new instance of the HumanPlayer and the Deck is created. The deck is then filled with cards through a nested loop. This loop is contained in the function addPrePlayerSetupCards(), 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. In the diagram this is depicted with another loop. 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.



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 seven classes are GameWindowController, Game, HumanPlayer, BotPlayer, Deck, Card and Command.

When a user chooses to play a card, they select a card from the GUI interface. This is done through the playCardButtonAction function. The Game class then calls 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. The command design pattern implements this by having a command class for each type of card. Depending on the card that the player wants to play a command is executed. In the diagram we chose to model the see-the-future card. The command then calls the seeTopThreeCards function in the game class. After which the user can see the top three cards in the deck.

In the case of a Skip card, the Game class executes executeCommand(), which then calls the SkipCommand through the SkipCard class. In the SkipCommand class StartNextTurn() is called, which skips the player's turn.

A similar thing happens when the Reverse card is played. The ReverseCard executes the ReverseCommand, which calls the flipDirection() function that changes the direction of the game.

When the Shuffle card is played, the Game class interacts with the ShuffleCard class, which in turn calls the shuffle function in the ShuffleCommand class. This shuffles the array in the Deck class.

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 is the Defuse card. When the Defuse card is played, the Game class calls the executeCommand function again. The DefuseCard class will call the execute() function on the DefuseCommand class. This in turn calls the playerPlaysDefuseCard function on the Game class. This lets the human player defuse an exploding kitten card, and 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.

Implementation

Author(s): Sam

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: $target/ExplodingKitten.jar$

Demo footage

https://youtu.be/NkFf6zxIBP4

Time logs

Member	Activity	Week number	Hours
Shahrin & Sam	Class diagram	6, 7 and 8	3
Sifra, Gal, Shahrin	Class diagram description	6, 7 and 8	1
Sifra & Gal	Sequence diagram description	6, 7 and 8	2

Sifra & Gal	Sequence diagram	6, 7 and 8	3
Shahrin & Sam	State machine diagram	6, 7 and 8	3
Shahrin	State machine diagram description	6, 7 and 8	1
Gal & Sam	Object diagram	6, 7 and 8	3
Gal	Object diagram description	6, 7 and 8	2
Sam	Implementation	6, 7 and 8	6
Sam	Implementation description	6, 7 and 8	0
Sifra	Implementation	6, 7 and 8	2
Gal	Implementation	6, 7 and 8	2
Shahrin	Implementation	6, 7 and 8	2
Sam	Application of design patterns description	6, 7 and 8	2
All of us	Summary of changes of Assignment 2 description	6, 7 and 8	1
All of us	Mentor meeting	6, 7 and 8	0.5
		TOTAL	33.5