Assignment 2

Team number: **ExplodingKittens - Team 3**

Name	**Student Nr.**	**Email**
Muhammad Ahsan	2663138	m.ahsan@student.vu.nl
Hao Qin	2658357	h.qin@student.vu.nl
Björn Oosterwijk	2651414	b.w.oosterwijk@student.vu.nl
Ashish Upadhaya	2635493	a.upadhaya@student.vu.nl

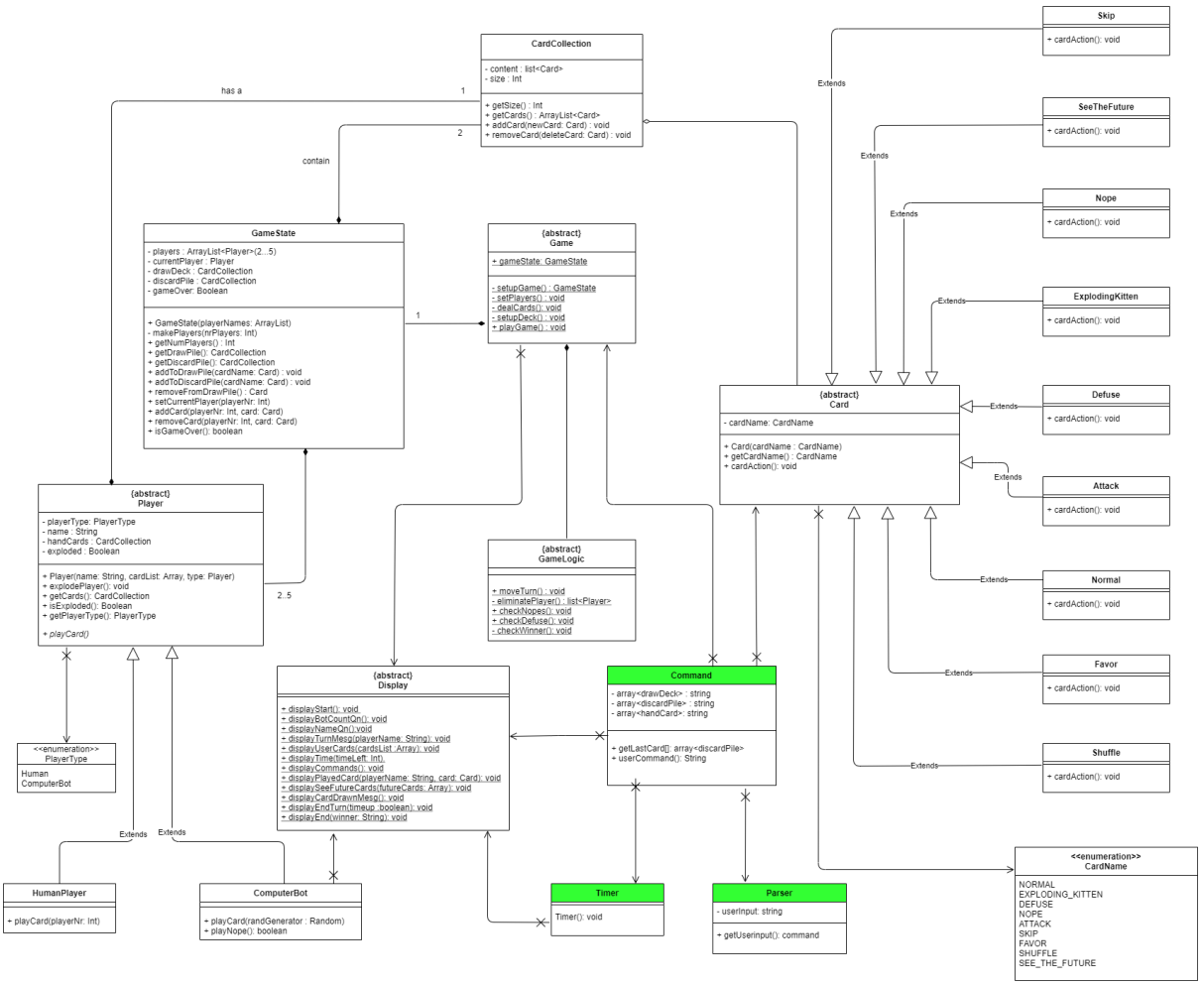
Implemented feature

ID	Short name	Description
F4	Commands	The player can play cards by issuing command-line commands following this syntax: command-name [target-cards]*. The available command-names are the following: - draw: this function draws a card for the current player - play: use the function to play a card in hand - pick: the player can pick another player to perform an action on, or one of the cards from another player - quit: player can leave in the middle of the game We have to make use of the parser and take input by the parser and check if it is a valid command and send the valid command to another function which will perform the action. For example if you want to play the skip card, then you type in “play skip”, the parser will check if it consists of one of the following actions in the array of available actions; {play [cardType], draw, pick [person], quit}. If so then it will pass the input to the action function which will perform the action. If it contains an invalid command it will simply return that “command is not valid” and the person will be deleted from the game or in some cases the game will be ended to discuss the rules. For the implementation part for this assignment we have built different features just in one command class. For example if a player gives a command, then the command will just be executed within the same class.
F5	Time limit	Each player has a time-limit for each turn, for example 1 minute. The time limit starts counting when the turn begins, and after the time-limit ends the system auto-ends the current player’s turn and makes the player draw a card. If nothing happens then the turn will be ended for that person and random command to the parser. It is thus important to be quick in this game. But for assignment 2 we were not able to completely fix this. Therefore we have decided to notify the player the duration of the game every 2 minutes.

Used modeling tool

For this assignment we are using draw.io

Class diagram *Author(s): Hao Qin, Ahsan, Björn*



The **CardName** is an enumeration, which lists out all kinds of card names in this game. When the game is set up, no strange or mistaken card names can be added to the game.

The **Card** is an abstract type class which can determine the class pattern as a superclass of all the game cards. This pattern will allow the system to be extended easily later, add new cards with specific card actions.
cardName is a protected **CardName** type that defines the card name, in the other subclasses cards, this *cardName* attribute will be defined as a private final type, which means after the Card be created in the game, its cardName shall not be changed during the game process. Each subclass of **Card** shall have a constructor with a **CardName** type field with default **CardName**, then after the specific **Card** subclass objects have been created, the **CardName** shall be set and cannot be changed during the game.
getCardName():CardName method will return the **CardName** of the card.
cardAction():void, this method will return nothing. During the current player's turn, if this player input a command to play a card, the program will check if this card could be played or not. If no other players will play a **Nope** card to stop this played card, then the *cardAction()* will be called in the game. Some cardAction need to input the currentPlayer and targetPlayer as parameters, some other cardAction do not need these parameters, the method will be written as override and overload methods in different subclasses.

This **Skip** card is a subclass inherited from the **Card** class.
cardName is a private final **CardName** type that defines the card name. After the card is created, the cardName shall not be changed anymore. All of the Card subclass will keep this pattern, due to the limit pages for the class diagram section, I shall not repeat mention this in the following subclasses.
cardAction():void will end the current user's turn without drawing a card from the card pile.

The **SeeTheFuture** card is a subclass of the **Card** class.
cardAction():void, after this method is called, the terminal will print out the cards' names of three cards from the top of the draw pile.

The **Nope** card is defined with the same card subclass pattern as a subclass of **Card**.
cardAction():void will cancel the relevant card's action.

ExplodingKitten card is defined with the same card subclass pattern as a subclass of **Card**. Beside following the same **Card** pattern, the **ExplodingKitten** card doesn't need to implement a detailed *cardAction()*. The program shall check the card which is drawn at the end part of the current player's turn. If this card is **ExplordingKitten** card, then checking if the same player hand card list contains a **Defuse** card or not. If return true, then play this **Defuse** card, if return false, the player is explored in this current game.
cardAction():void is a method without detailed implementation in this **ExplodingKitten**.

The **Defuse** card is also a subclass of **Card** class.The card will be checked when the player draws an **ExplodingKitten** card during the game. When this **Defuse** card has been played out, the current player can insert the **ExplordingKitten** card to the draw pile.
cardAction():void, in **Defuse** card, the implement for the method *cardAction()* is not necessary. The program just needs to check if the player has a Defuse card in hand when the player draws a **ExplordingKitten** card.

This **Attack** card is a subclass inherited from the **Card** class.
cardAction():void, after this method is called, the current player's turn shall be finished without drawing a new card from the card pile, and add one extra play turn for the next player in the turn order, which means the next player has to play his turn twice.

The **Normal** card is self explanatory, it is just a normal card without specific *cardAction()*. The implementation is realized following the same subclass pattern of the **Card** class.

This **Favor** card is a subclass inherited from the **Card** class.
cardAction(currentPlayer: Player, targetPlayer: Player):void, after the current player played this card, the current player can choose one card and get this card to his hand from the target player.

This **Shuffle** card is a subclass inherited from the **Card** class.
cardAction():void, this method will let the current player shuffle all the cards in the card pile.

TimeLimit
We have added this feature to make sure that the game does not take too long to play.

timer():void, for this assignment, this function will run during the game and print out a message every 2 minutes.

Command

The command class is there to make the game understandable and stable to play. It will contain an arraylist of drawpile and discard pile and an action. There will be fixed types of commands that can be inputted(checked by the parser class), thus the commands are immutable.

playerAction: string, this includes the following available actions. {draw, pick, explodeKitten, skip, see the future normal and exit}

array<drawPile>: string, this holds all of the remaining cards from the main deck in a list.

array<discardPile>: string, this holds a list of cards that have been discarded by the users.

parseAction(): string string parse the action the the action function.

getLastCard(int): string returns the last card from the discard pile.

getHandCard[]: array<handcard>, this will return the list of cards in the players' hand.

For this project parser class will check if the given command is valid and if so, it will send the action command to the action function which will take care of the execution of the function. Note the arrays belong to Assignment 2 only and serve the purpose of 'fake' setup.

Parser

The parser class will be responsible for checking and parsing the command to the action function.

availableCommand():string, this will check if the given input is valid and if so it will pass to the action function.

getCommand(): void, this will get the input from the player.

Game

The abstract **Game** class is responsible for initializing a game. During this setup, the number of bot-players will be determined and enrolled in the game, the human-player's name will be set, the deck of cards will be created and all players will get their initial hand of cards dealt. These methods instantiate a new object of class **GameState** (which will be discussed shortly) and then initiate the actual gameplay.The class is declared abstract because there is no need for declaring objects since we implement the game setup centrally. The private methods are necessary to express the division of the setup of the game (which is rather elaborate) into smaller, easily managed methods. The Game class has a one-way association with the Display class, since it needs to display certain messages to the user. The attributes and methods for this class are the following:

gameState:GameState holds the current **GameState** object.

setupGame():GameState is the method for creating and returning the initial GameState.

setPlayers():void sets the number of players and records the player's names.

dealCards():void gives (or "deals") each player the initial handCards.

setupDeck(): void sets the initial deck (with the appropriate amount of exploding kittens).

playGame(): void is the method that enables the actual playing and calls setupGame().

GameLogic

The **GameLogic** class enables the switching of turns between players. Having this class take care of controlling the flow of the game ensures that the game dynamics are arranged centrally, thus adhering to the principles of encapsulated design. In between turns, but also

within turns, it is sometimes important to make several checks, which are also done in this class. The **Gamelogic** class has a composition relationship to the **Game**, since without the (initialized) game, game dynamics cannot logically exist. The attributes/methods are:
moveTurn(): void will end the turn of a player and enable the clockwise next player.
eliminatePlayer():void will remove a player from the game when this player is eliminated
checkNopes():void will check whether other players want to respond on a play with a **Nope**.
checkDefuse():void will be responsible for checking whether a player has a **Defuse** or dies.
checkWinner():void will be checking if there is a winner after each exploding kitten.

GameState
The **GameState** class serves as a record for the state of the game. It contains the players, the deck and the discard pile and updates these every time necessary throughout the game. Grouping the relevant objects of the game together in this class is a convenient way to logically summarise the state of the game. It has a composition relationship to the **Game** class, since without the **Game** class the GameState (and it's composite-association objects) could not exist; without a game it is impossible to have a gamestate. The attributes/methods:
players: ArrayList<Player>(2...5) will store the players that are participating in an ArrayList.
currentPlayer: Player is the Player that is currently enabled to make plays.
drawDeck: CardCollection is a **CardCollection** object that contains the draw deck.
discardPile: CardCollection is a **CardCollection** object that contains the discard pile.
gameOver: Boolean will be true once there is a winner or the quit command is used.
GameState(playerNames:ArrayList) will be the constructor for the GameState class.
makePlayers(nrPlayers: Int) will instantiate the ArrayList of players.
getNumPlayers(): *Int* will return the number of players currently in the game.
getDrawDeck(): CardCollection will return a copy of the draw deck.
getDiscardPile(): CardCollection will return a copy of the discard pile.
addToDrawDeck(cardName: Card, index: Int): void will add/insert a card to the draw deck.
addToDiscardPile(cardName: Card): void will add a card to the *discardPile*
removeFromDrawDeck(): void will remove the top card from the draw deck
setCurrentPlayer(playerNr: Int): void will change the currentPlayer attribute when necessary.
addCard(playerNr: Int, card: Card) will add a card to a player's hand when called.
removeCard(playerNr: Int, card:Card): void will remove a card from a player's hand.
isGameOver(): boolean will set the gameOver attribute to indicate that the game is over.

CardCollection
The **CardCollection** class is responsible for keeping groups of cards together and for performing actions on these groups of cards.This class will be used to represent the drawing deck, the discard pile and the cards in the hands of the players. From a modelling point of view it makes sense to have a logical container for multiple cards and this also corresponds to the physical implementation of the game.
This class has a composition relationship to the **GameState** class, since every **GameState** can only exist when there are two **CardCollection** objects present: the drawDeck and the discardPile. Furthermore, there is a composition to the **Player** class, since every player can only sensibly take part in the game with a **CardColection** object containing the cards at hand. The incorporated attributes and methods are the following:
content: ArrayList<Card> will keep track of the cards in the card collection.
size: int will effectively equal the number of cards within a card collection.
getSize(): int will return the size of the specific card collection object.

getCards(): ArrayList<Card> will return a copy of the list of cards within a card collection.
addCard(): void will add a card to a card collection when needed, e.g. after a draw.
removeCard(): void will remove a specific card from a card collection, e.g. a played card.

Display
All output is done using the **Display** class. This design is used so code reuse is possible. Other classes can call the methods in the display class and do not need to separately implement printing methods. This also enables better maintainability. The methods mostly contain print statements with no return value. For example
displayPlayedCard(playerName: String, card: Card) : if the *playerName* is Peter and *card* is Skip, will print “Peter played Skip card”.
All methods are similar and do not have any special description. The **Display** class is abstract because all methods are static and instantiating is not required. Most classes and their functions which run the game use the display class. The display class does not need to be instantiated as all methods are static. Therefore connected classes have unary association that is related classes can view/access content of display class but not in reverse.

Player (HumanPlayer and ComputerBot)
The **Player** class is an abstract type which can be either a **HumanPlayer** or **ComputerBot**. It represents the players in the game. Most attributes are self explanatory. Most relevant are
cardList : this represents the cards in the hand of the player.
exploded : which represents whether the player is in the game or not. Among the methods:
Player(name: String, cardList: Array, type: Player) : This is the constructor which constructs the player.
explodePlayer(): void : This is used to set the variable exploaded to true when the player has played an exploding kitten and does not have a diffuse card.
getCards(): CardCollection : This method will return a copy of the list of cards in the player hand.
isExploded(): Boolean : This returns the value of the *exploded* attribute that is the status of the player.
playCard(): void : The method is abstract as human and players play cards differently. The HumanPlayer class method *playCard()* uses the *play()* in **Command** class as input and checking as required. The **ComputerBot** class *playCard()* implements this by randomly picking and playing cards.
playNope(): void : The **ComputerBot** class implements one extra method that is playNope(). It is used to check and randomly play the nope card when some other player does the turn. This method is called by **GameLogic** class using the object of the bot.
The relevant associations include generalization between the **Player** and **HumansPlayer** and also with the **ComputerBot** class. This is to show abstraction and allow inheritance. The **Player** class has composition relation with **GameState** class with minimum two and maximum 5 players. Each player has a list of cards belonging to the **CardCollection** class so there is association from player to **CardCollection**.

PlayerType
This is the enumeration containing the HumanPlayer and ComputerBot. This is assigned depending on the type of player.

Object diagram

Author(s): Björn

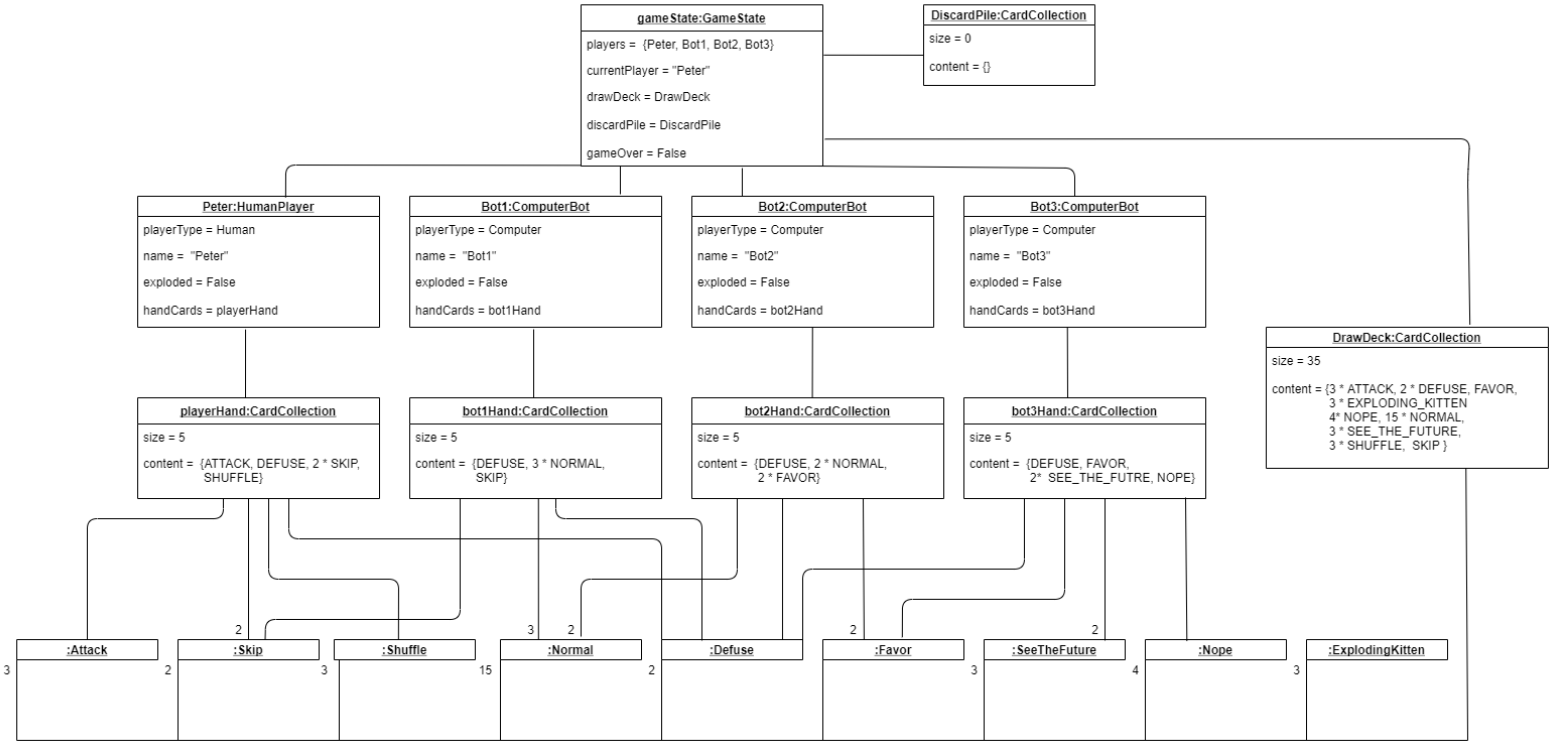


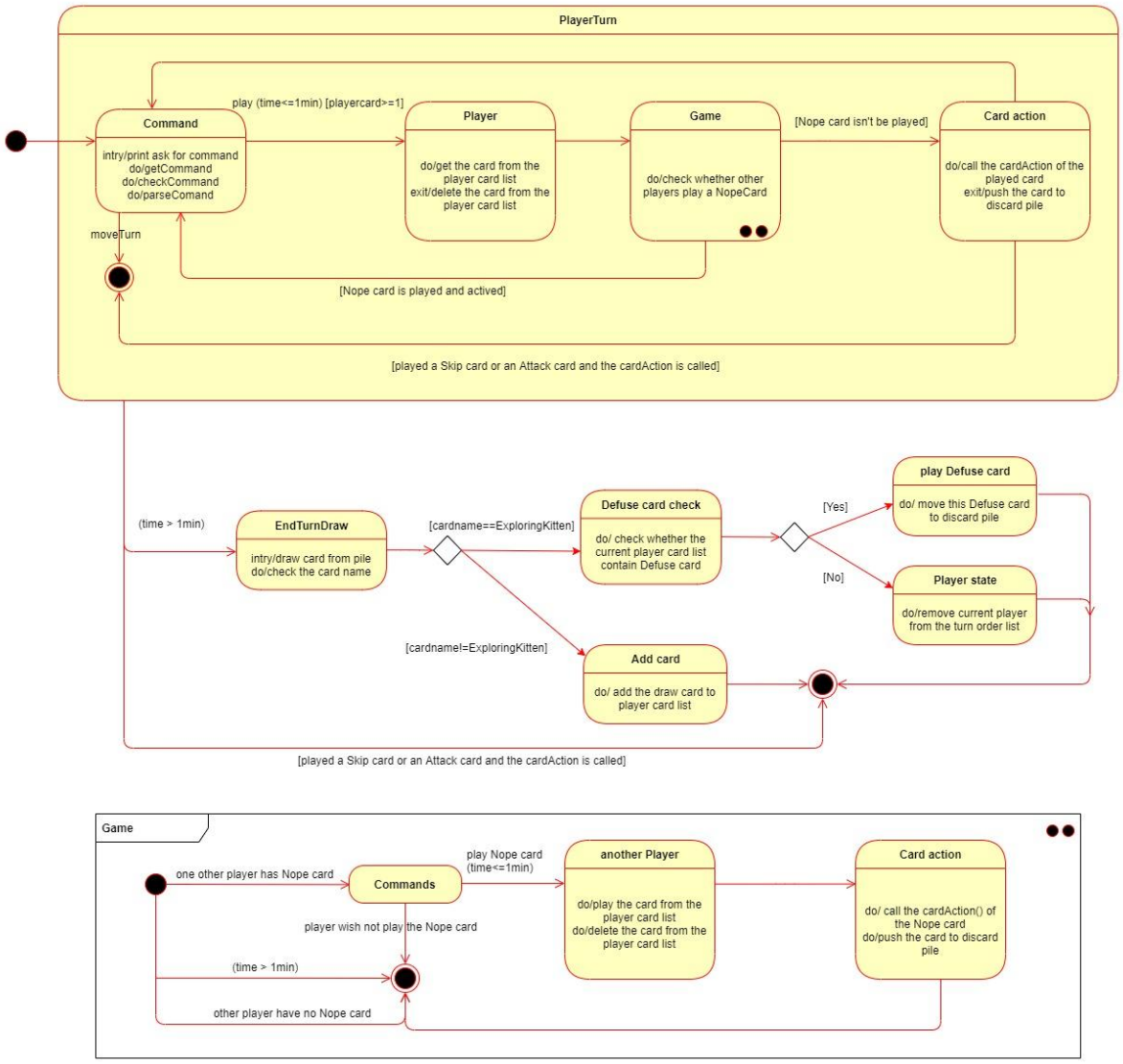
Figure representing the initial state of a game composed of 1 human and 3 bots (Diagram 2)

This chapter contains the description of a "snapshot" of the status of the system during its execution. The purpose of this object diagram is to clarify which objects are present in the system throughout a 4 player game and to identify the relationships between those objects.
The diagram represents the initial state of the game, right after the game has started. Therefore, one can see in this representation that the number of players has been decided, the players have been assigned names and have had their initial 5 hand cards dealt. In this state, the system is ready for the first player turn, which will be Peter's to play as one can see in the *gameState* object.
The *gameState* object is the central object in which the states of the other relevant objects are continuously registered and updated. One can see it contains the players and the *drawDeck* and *discardPile*. The *discardPile* is obviously still empty in this phase of the game; once cards have been played the *discardPile* content will grow. The player's hands and the *mainDeck* do "contain" cards. All players start out with 5 five cards, which are cataloged in separate **CardCollection** class objects, leaving 35 cards for the *drawDeck*. Since representing all these individual *Card* objects (55 in total) would be abundant, the diagram shows anonymous *Card* objects with the respective multiplicities indicating how many of those cards are held. With the *drawDeck*, *discardPile*, *Player objects* and player's hand cards in place, the game is ready to be played and Peter can make a first move.

State machine diagrams

Author(s): Hao Qin, Ahsan

(1) State machine diagram for a play turn of the human player



The diagram above shows the state machine diagram of the event of executing the human player turn in this card game. In the beginning of this process, the state is in the Command class, the game system waitting for the human player to input a command. After the input has been received by the Command, the method getCommand() will be executed, and then the checkCommand() will check the input command.

If the command is invalid, an error message will be printed out in the terminal, the program returns back to the Command state waitting for the command.

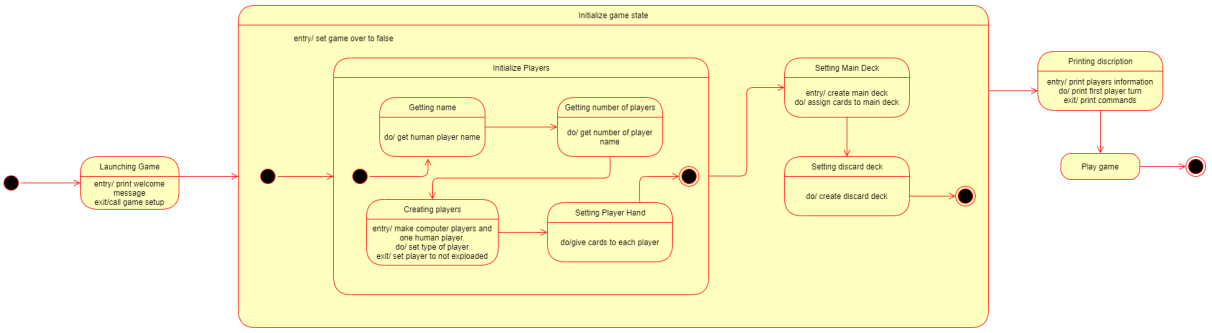
If the command is valid after the checkCommand(0 method, the parseCommand() will be called and parse the command. For example, if the player input to play a card, the state will go to the Player class to get the card from the card list, and then the Game will check with other players if they will play a Nope card, program go to the reference state Game as shown in the diagram. If no other player has a Nope card or the turn time is longer than 1 minute, the state will end and return to the player turn state. If one player has the Nope card and wishes to play it, the state will go to this player and get this Nope card. After the cardAction() of the Nope card is called, the state returns back to the player turn state.

The program now returns to the Game state. If no player plays a Nope card, then the card played by the human player will be active and call the cardAction() in the next state. If the Nope card is played in the reference Game state, then the card played by the human player will be dismissed and the state go back to the Command and wait for another command from the human player.

When this human player input the moveTurn command or the turn time is longer than 1 minute, the Player turn state will be stopped, the program will go to the EndTurnDraw state, this state will get a card from the top of the draw card pile first. After the program will check whether this card is an ExplodingKitten card or not. If this draw card is not a ExploringKitten card, the program will add this card to this current player and end the player turn. If this draw card is an ExploringKitten card, the program will check if this current player holds a Defuse card or not. If this player holds a Defuse card, the program will automatically get this Defuse card and move it to the discard pile, then let this player insert this ExploringKitten card back to the draw pile. If this current player has no Defuse card, that means the player is explored, and the program will remove this player from the turn order list, then the player can not play a turn any more.

When the current player plays a Skip card or an Attack card, and the cardAction is called, the program will skip the EndTurnDraw, and end the current turn immediately.

(2) State diagram for game setup:



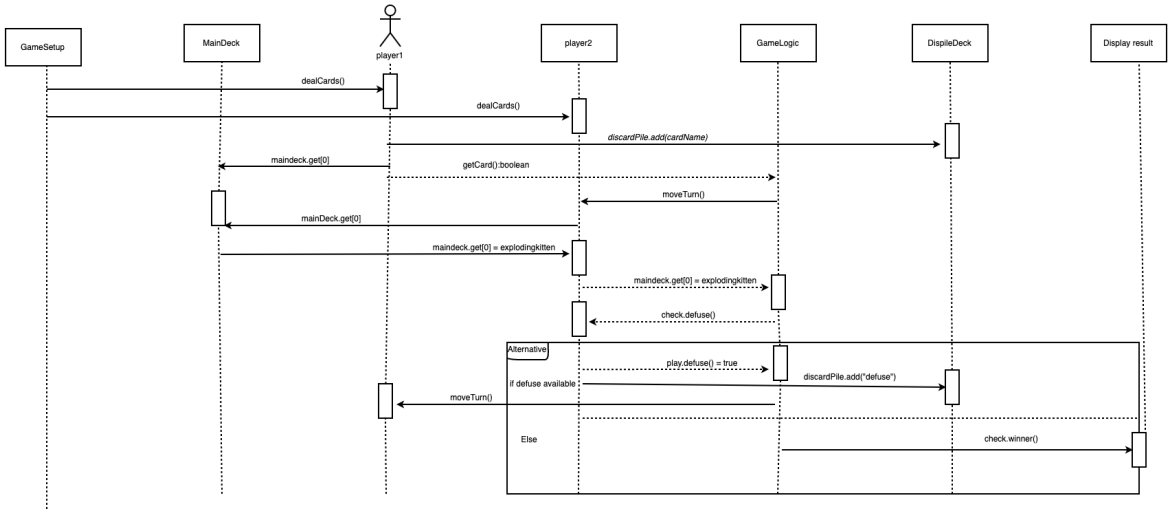
The state machine belongs to *Game* class and starts at the *setupGame* method. The game state is initialized. This happens in a specific order and different states are involved that are highlighted in the state machine. Throughout the state machine no guards are used because they are not needed. Next event only happens when the previous event is completed.

At first the game launches and prints the welcome message. Then the game set up is called. To initialise the game state object, the number of players and their names are required. Input for the name of the human player is firstly taken and then the input for the number of total players is taken. Now that the names and number of players are known the player objects can be created according to the numbers and initialised with their names, types and unexploded status. Using the player objects the cards are given to each player. After Initialising the players the main deck is set up along with the discard deck. The decks cannot be initialised before the players because the main deck is set up according to the number of players in the game. The discard deck is initialized empty. Before exiting the 'initialise game state' composite state the game over variable is set to false. Afterwards the information is printed which includes the number of players and the message for the human player to play the first turn along with the available commands. Now the game can be played (shown in state machine 1) and this completes the state machine for the game setup.

Sequence diagrams *Author(s): Ashish*

(1) Sequence diagram of playing exploding kitten

Descriptive

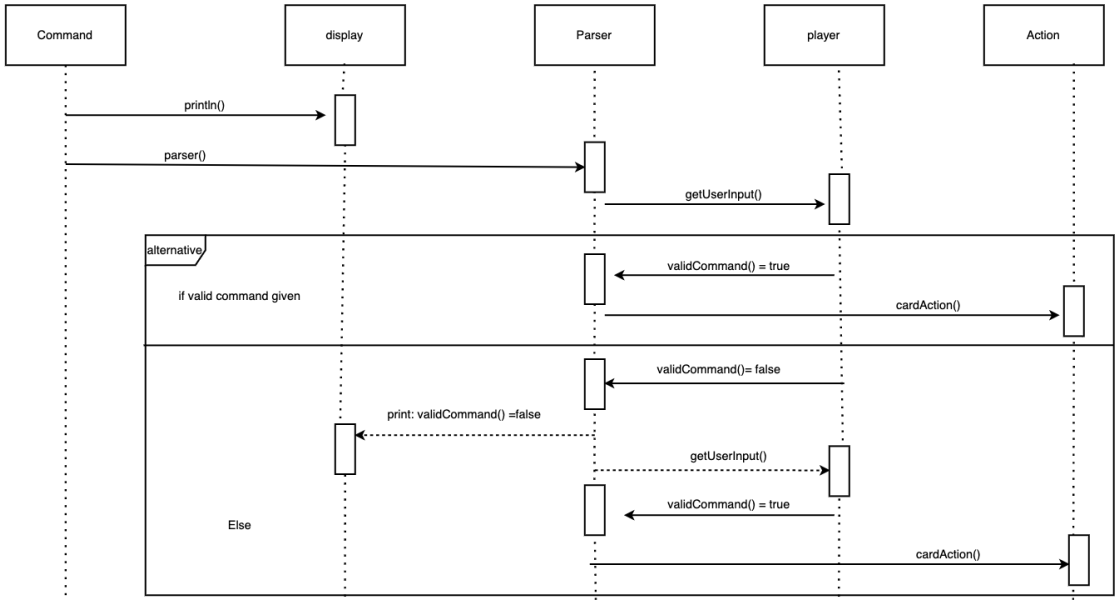


The sequence diagram is about two players playing the game and an exploding kitten card is drawn by one of the players during the game. We assume objects are created and the game has already been initialized except for dealing the cards to the players that is the starting point of the diagram. This diagram is not intended to cover the initialization of the game. The dealing of cards in the diagram is to show no turn has happened before.

At the start of the game cards are assigned to each player. This will be done by calling the *dealcard()* function at the beginning of the game by the gameSetup.
After dealing is done, the human player *Player1* does the first turn. He/she throws or draws a card by giving a command (command could be "play nope", "draw card" etc. Nope card (in case nope card is thrown) it is then added to the discardPile and removed from the

handCard by calling discardPile.add("input") and drawDeck.remove(playedcard). If the player gives a "draw card" as a command, then first card from the drawdeck will be added to the hand *handCard.add("drawcard")* and removed from the *drawDeck(drawDeck.remove[0]),* except when the first card from the drawdeck is explodingKitten. A signal will be sent to the gamelogic if a player has drawn a card and let the gameLogic know what type of card it is. In case the draw card is explodingKitten, then a message will be sent to the player to check for the "defuse" card. If he has a defuse card he lets the gameLogic know he has a defuse card and the player will be able to play a defuse card. After this The gamelogic calls a *turnMove()* function and will move the turn to the next player (we expect that player1 has not drawn an exploding kitting in this case). The move is turned to the next player. Now player two throws a card and after that draws a card. While drawing the card he unfortunately draws a explodingKitten. This message is then sent to the *gamelogic.* Gamelogic checks if the player has any defuse card in his hand. If the player has a defuseCard, then the gamelogic will send a signal and make it available for the player to play defuse and move the turn to the next player, but if he has not, then the player will be out of the game and the winner will be player 1. In the case of this diagram the move will be passed to player 1. If it was the case that there were more than 2 players, then the move would go to player +1. This is a loop and continues during the whole game.

(2) Sequence diagram from calling command until the action



This diagram shows the part in which the command class is called until the action is executed. When the game has started and command class is called. The very first thing that will happen is, it will check for the last card in the dispiledeck and return a message by *println()* with the instruction what the player should do and this instruction will be displayed in the terminal. An example of instruction could be "last player has thrown a normal card, please draw or play a card please. After the instruction has been shown in the terminal. the *parser()* method will be called. This method calls the starttimer (will be fully implemented in assignment 3) method which will start the timing of it and it will display the remaining time to enter a valid command on the terminal. we assume that the instruction is given and a command is given by the user. It will forward those commands to the *parser()*. Parser will

then guide further to the game. Everything that a player has to do will simply be displayed in the terminal. When the player gives an userInput, the *parser()* first checks if it is a valid command bij comparing all the available commands (*validCommand()*). If it is indeed the case that the given input is valid. The command will be forwarded to the to cardAction to perform the action. If invalid command is given(validCommand() = false) , a message is sent to the display that “the user has given a invalid input”. Display will then show the instruction for the player(“invalid command, please give a valid command or type quit to quit the game). The system will ask the user to input a valid command or the command “quit” which will delete the player from the game . When one of the commands is given this will be forwarded by *cardAction()* to the action method. Which will execute the given command

Implementation *Author(s): Ashish*

It took us more time than expected to start with the implementing part. First of all, our group was struggling making perfect class/state-machine/sequential diagrams. This has not been a very good move in my opinion. We should have made a draft and started programming and encountered problems and updated the class diagram during the programming phase accordingly. However at that moment we thought, it would be much easier to make a good class diagram first and start implementing. While programming I encounter few problems. We were asked to at least implement the command and the timer class for the second assignment, which we described in assignment 1.

Since it was a card game we would not be able to produce any videos for assignment 2 just with only command and timer classes. This was a great obstacle, because we wanted to show some product. We then came up with the solution and concluded to build a prototype of the whole game, including the main features such as starting the game with a manual handCard, card in the deck and a discardPile, asking the userCommand and parsing it and doing the action. In this way we were able to make the game running and show how the game would be working in general. This will leave us less work for the next assignment. For this assignment all the methods are included in the command class, however for assignment 3 we will be splitsing the method and implement all the different methods separately in different classes, which will make the code more readable and adjustable and we will update the uml diagram accordingly. As discussed in the lectures about the agile method, we tried to implement that and therefore made a first prototype of the game instead of only implementing 2 features for the second assignment. This has given us more knowledge about the obstacles that occur during the programming and how we can solve it.

Talking about the game, this prototype begins with a list of handCard, a drawdeck containing a few cards and a discardPile containing a few cards. When the programs run we assume it is someone’s turn. It will then check what the last player has thrown. According to the lastCard, an instruction will be shown on the terminal. If we assume the last thrown card on the discardPile was a normal card. It will ask the player to play or draw a card from the drawDeck. if the player chooses to play something it can just give the command of his choice, for example, “play skip”, “play normal”, “draw card”. The Player can also decide to play more cards, this instruction will also be shown into the screen. To make clear when the lastCard is checked by the method “cards" and the instruction is shown. cards method will call the parse function. This will ask for input and execute the action as well, furthermore it

will start the timer and give every 2 print every 2 minutes a message it has passed 2 minutes since the last message. This way it will be clear how long the game is taking. We initially wanted to implement the command action in a separate class for the last assignment. But for now the actions are also executed in the parser method. When a card is played or drawn by the player. The play card will be added to the discardPile and removed from the handCard. Or if the player chooses to draw a card, the draw card will be removed from the drawdeck and added to the hand. Except when the draw card is an exploding kitten. Then it will put the explodingKitten back in the deck at position index 3. And the player is asked to play defuse or quit the game. The parser function also checks if the played card is available in the player's hand. If so it will be allowed to play the card. Else a message will occur in the display and the game will end to discuss the rules again. We have made use of if else statements during programming. This was the easiest way to go through the logic of the game and it was the most logical thing to do, because we had to compare the userCommand/userInput every time receiving an input.

The location of the main Java class is:
src/main/java/softwaredesign/Main.java

location of the jarfile:
out/artifacts/vu-2021-Expldingkitten/assignment2.jar

An example of the prototype can be found by clicking on the following link.

https://youtu.be/YaBwLLpcEAY

Time logs

Exploding kittens	3							
Ashish	**Activity**	**Week number**	**Hours**		**Björn**	**Activity**	**Week number**	**Hours**
	Class diagram	3_4_5	6			Class diagram	3_4_5	10
	Object diagram	3_4_5	2			Object diagram	3_4_5	10
	State machine diagram	3_4_5	1			State machine diagram	3_4_5	2
	Sequence diagram	3_4_5	8			Sequence diagram	3_4_5	3
	Implementation	3_4_5	13			Implementation	3_4_5	2
	Report?							
	Total		**30**			**Total**		**27**
Hao	**Activity**	**Week number**	**Hours**		**Muhammad**	**Activity**	**Week number**	**Hours**
	Class diagram	3_4_5	10			Class diagram	3_4_5	12
	Object diagram	3_4_5	4			Object diagram	3_4_5	6
	State machine diagram	3_4_5	10			State machine diagram	3_4_5	8
	Sequence diagram	3_4_5	4			Sequence diagram	3_4_5	6
	Implementation	3_4_5	2			Implementation	3_4_5	2
	Total		**30**			**Total**		**34**