Assignment 3

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

Summary of changes of Assignment 2

Update the class diagram and text based on the implement during assignment 3.
Fixed the issue for the first state machine diagram based on the feedback from assignment 2, one is the typo, and another is add an event. And also write more explanations for the Normal card, and the design idea of the Card system.

Changed the **first sequence diagram** as per feedback from assignment 2 to a more concrete diagram by deleting the setup part and changed the text part accordingly.

Updated the **second sequence diagram** as per feedback from assignment 2. The name was not specifiek so I changed the name to a more specifiek one and also added a loop extra which was missing in assignment 2. Now the program will be in a loop for 60 sec if a invalid command is given. The text is also changed accordingly.

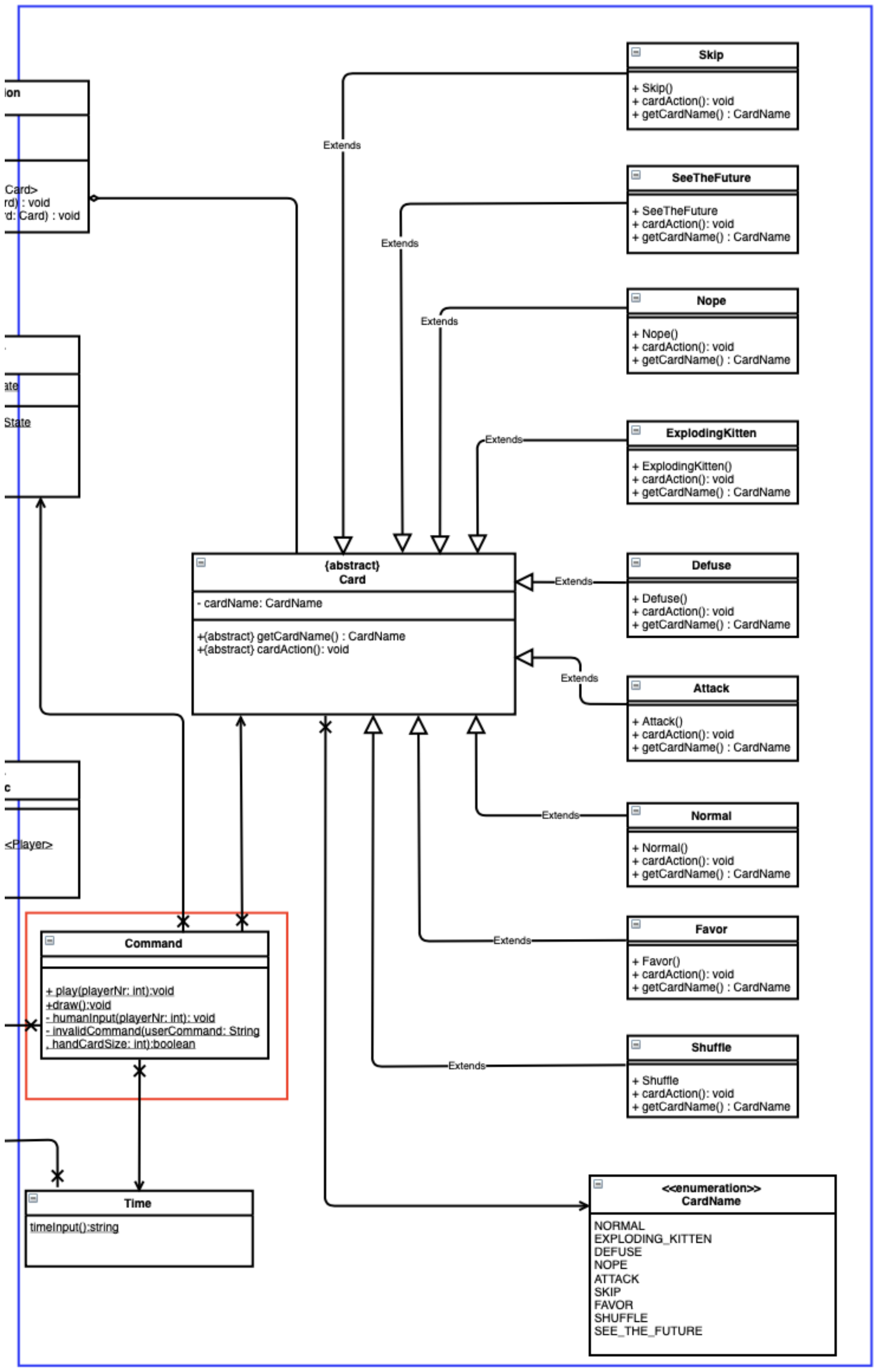
Explanation about using a **single abstract class** for display.

Used modeling tool
For this assignment we are using draw.io

Application of design patterns

Author(s): ashish, Bjorn, Hao, Muhammad

<Figure representing the UML class diagram in which all the applied design patterns are highlighted graphically (for example with a red rectangle/circle with a reference to the ID of the applied design pattern>

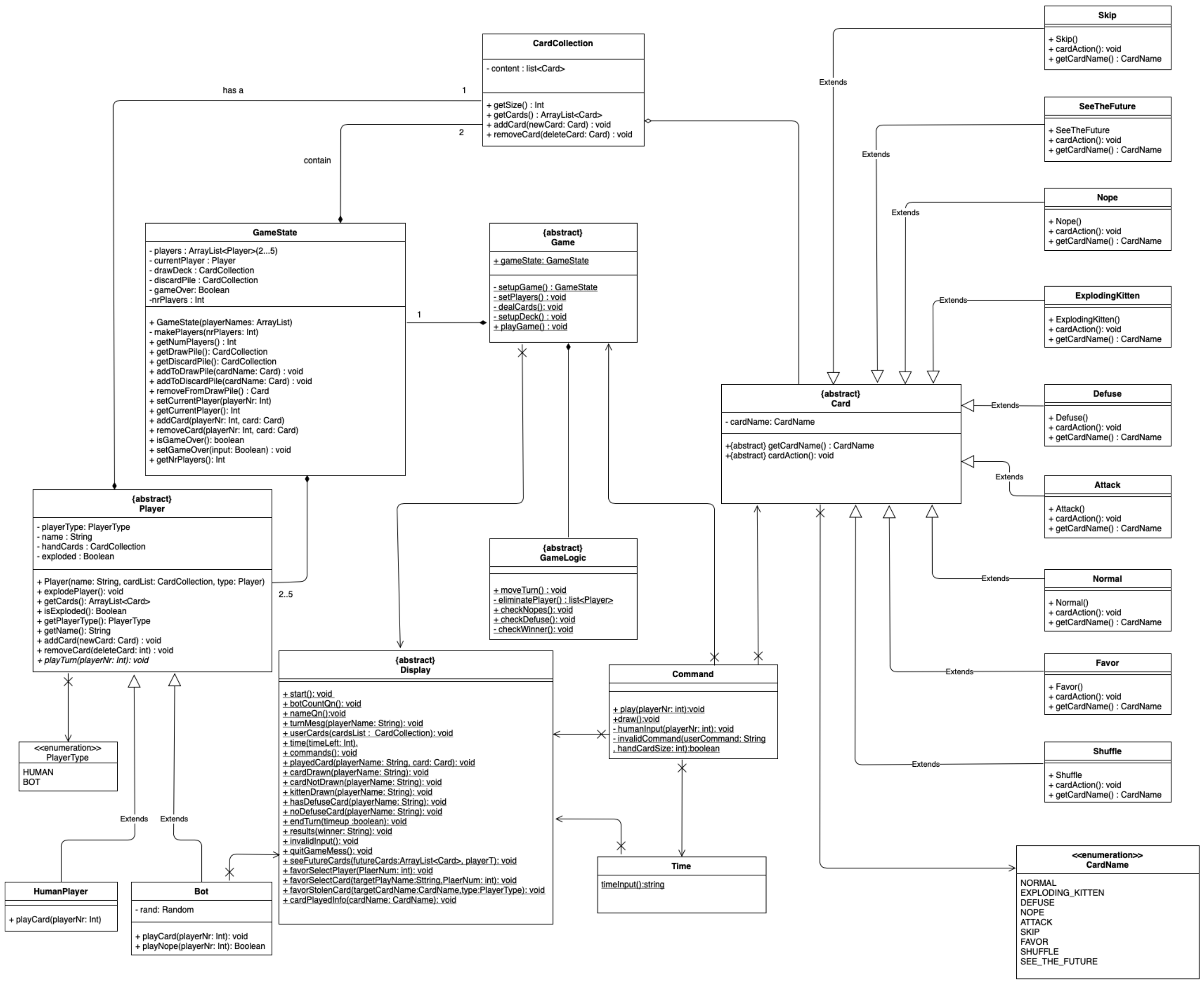


	DP1 (red)
Design pattern	Null object
Problem	strings and integers are part of the command and it is very easy to make mistakes while typing a command. So, it is important to handle these errors.
Solution	When an user gives an input. The command will be checked by the method humanInput() and if it is not correct it will ask the user to give the command one more time. In assignment 2 we removed the player after giving an invalid command. This gives a much better game experience then having to start the game again and again if you give an invalid input.
Intended use	Instead of just quitting the game after an invalid command is given. It will now handle the error and ask the user to enter a command again. This gives a much better game experience then having to start the game again.
Constraints	N/A
Additional remarks	N/A

	DP2 (blue)
Design pattern	Decorator pattern
Problem	The game consists of human and botplayers. They will have to be able to communicate with each other. For example, when an explodingcard is drawn a defuse, card need to be thrown to survive. The command class consists of valid input which will be call for execution if they are valid. For now, this program consists only of 3 commands: play draw and quit. It is possible that another developer wants to modify the commands. We should therefore provide a solution which will make the next developer easier to add new valid command, without needing to change a lot from the original code and perform a action accordingly.
Solution	A solution could be to add a method within the command class for the humanInput, which makes it easy to add possible new valid commands in the future.
Intended use	A use case could be to a new valid command like “eliminate player 3” with a special card. Player 3 would then be eliminated from the game. This is maybe not desireable for this game but next developer may think that it is an great idea.
Constraints	N/A

Additional remarks	N/A

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


CardCollection
- content : list<Card>
+ getSize() : Int
+ getCards() : ArrayList<Card>
+ addCard(newCard: Card) : void
+ removeCard(deleteCard: Card) : void
has a
1
2
contain
GameState
- players : ArrayList<Player>(2...5)
- currentPlayer : Player
- drawDeck : CardCollection
- discardPile : CardCollection
- gameOver: Boolean
-nrPlayers : Int
+ GameState(playerNames: ArrayList)
- makePlayers(nrPlayers: Int)
+ getNumPlayers() : Int
+ getDrawPile(): CardCollection
+ getDiscardPile(): CardCollection
+ addToDrawPile(cardName: Card) : void
+ addToDiscardPile(cardName: Card) : void
+ removeFromDrawPile() : Card
+ setCurrentPlayer(playerNr: Int)
+ getCurrentPlayer(): Int
+ addCard(playerNr: Int, card: Card)
+ removeCard(playerNr: Int, card: Card)
+ isGameOver(): boolean
+ setGameOver(input: Boolean) : void
+ getNrPlayers(): Int
1
{abstract} Game
+ gameState: GameState
- setupGame() : GameState
- setPlayers() : void
- dealCards(): void
- setupDeck() : void
+ playGame() : void
{abstract} Player
- playerType: PlayerType
- name : String
- handCards : CardCollection
- exploded : Boolean
+ Player(name: String, cardList: CardCollection, type: Player)
+ explodePlayer(): void
+ getCards(): ArrayList<Card>
+ isExploded(): Boolean
+ getPlayerType(): PlayerType
+ getName(): String
+ addCard(newCard: Card) : void
+ removeCard(deleteCard: int) : void
+ playTurn(playerNr: Int): void
2..5
<<enumeration>> PlayerType
HUMAN
BOT
Extends
Extends
HumanPlayer
+ playCard(playerNr: Int)
Bot
- rand: Random
+ playCard(playerNr: Int): void
+ playNope(playerNr: Int): Boolean
{abstract} GameLogic
+ moveTurn() : void
- eliminatePlayer() : list<Player>
+ checkNopes(): void
+ checkDefuse(): void
- checkWinner(): void
{abstract} Display
+ start(): void
+ botCountQn(): void
+ nameQn():void
+ turnMesg(playerName: String): void
+ userCards(cardsList : CardCollection): void
+ time(timeLeft: Int).
+ commands(): void
+ playedCard(playerName: String, card: Card): void
+ cardDrawn(playerName: String): void
+ cardNotDrawn(playerName: String): void
+ kittenDrawn(playerName: String): void
+ hasDefuseCard(playerName: String): void
+ noDefuseCard(playerName: String): void
+ endTurn(timeup :boolean): void
+ results(winner: String): void
+ invalidInput(): void
+ quitGameMess(): void
+ seeFutureCards(futureCards:ArrayList<Card>, playerT): void
+ favorSelectPlayer(PlaerNum: int): void
+ favorSelectCard(targetPlayName:Sttring,PlaerNum: int): void
+ favorStolenCard(targetCardName:CardName,type:PlayerType): void
+ cardPlayedInfo(cardName: CardName): void
Command
+ play(playerNr: int):void
+draw():void
- humanInput(playerNr: int): void
- invalidCommand(userCommand: String , handCardSize: int):boolean
Time
timeInput():string
{abstract} Card
- cardName: CardName
+{abstract} getCardName() : CardName
+{abstract} cardAction(): void
Skip
+ Skip()
+ cardAction(): void
+ getCardName() : CardName
SeeTheFuture
+ SeeTheFuture
+ cardAction(): void
+ getCardName() : CardName
Nope
+ Nope()
+ cardAction(): void
+ getCardName() : CardName
ExplodingKitten
+ ExplodingKitten()
+ cardAction(): void
+ getCardName() : CardName
Defuse
+ Defuse()
+ cardAction(): void
+ getCardName() : CardName
Attack
+ Attack()
+ cardAction(): void
+ getCardName() : CardName
Normal
+ Normal()
+ cardAction(): void
+ getCardName() : CardName
Favor
+ Favor()
+ cardAction(): void
+ getCardName() : CardName
Shuffle
+ Shuffle
+ cardAction(): void
+ getCardName() : CardName
Extends
Extends
Extends
Extends
Extends
Extends
Extends
Extends
Extends
<<enumeration>> CardName
NORMAL
EXPLODING_KITTEN
DEFUSE
NOPE
ATTACK
SKIP
FAVOR
SHUFFLE
SEE_THE_FUTURE

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, and in this super class, this method will be an abstract method here, which will make sure each subclass of **Card** shall implement a getCardName method to return the current card name.
cardAction():void, this method will return nothing. During the one player's turn, after this player plays 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. This method is an abstract method here, only its subclasses need to implement this method, each card will have a *cardAction()* method, because some cards have special card action for the game.

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 a subclass inherited from the **Card** class.

Instead of having different cards like: hairy potato cat, tacocat etc. We have made one Normal card for it. Normal cards don't have any specific function like skip or defuse. The implementation is realized following the same subclass pattern of the **Card** class. The purpose of having Normal cards in this game, first is to confuse other players how many useful cards the player holds during the game, second is if the player plays a specific quantity of exact same normal cards together, the cards together can be played as a Favor card, but this is not one of our features to be implement in this assignment.

This **Favor** card is a subclass inherited from the **Card** class.
cardAction():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.

In summary, under this Card design pattern, this part is very flexible for maintenance and extension, all we need to do is add a CardName in the numeration, and implement a subclass for this new Card.

Command
The command class is there to make the game understandable and stable to play. command class will come in action during the human player turns and when a card needs to be drawn by any of the other players. The command class also takes care of the timer which will force a player to play within 60 sec else a card will automatically drawn from the deck and the turn will move to the next player.

play(playerNr: int):void start the game for human

humanInput(playerNr: int): void, this method will ask for an input and also check if the given input is valid. if it is the case that the given command is valid and within the 60 sec framework a action will be called for the played card. If not a card will be automatically drawn as already mentioned above. The available valid command will be play n(n = number), draw and quit. quit will automatically. other commands are obvious.

invalidCommand(userCommand:String, handCardsize:int):Boolean, this will return true if the given command is invalid
draw():void, this method allows the user to draw a card form the deck

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. It calls the other initializing methods.
setPlayers():void sets the number of players and records the user's name.
dealCards():void gives (or "deals") each player the initial handCards.
setupDeck(): void sets the initial deck.
playGame(): void is the method that enables the actual playing and calls *setupGame()*.
delay():void is used to have some time between prints whenever appropriate, since the game is text based and it is not preferable to have all info printed instantly.

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:
turnType:TurnType keeps track of whether the turn is normal or attacked
numTurns:int counts the number of turns to be taken, which can grow through attack cards
increaseNrTurns():int will increase numTurns when an attackcard is played
moveTurn(): void will end the turn of a player and enable the clockwise next player. It will call upon *kittenDrawn()* and *checkDefuse()* to check if an exploding kitten card has been drawn and appropriate action has to be taken.
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, after an exploding kitten card has been drawn.
kittenDrawn():boolean Will check whether the last player drew an exploding kitten
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 most relevant attributes/methods are:
players: ArrayList<Player>(2...5) will store the players that are participating in an ArrayList.
nrPlayers:int holds the number of players that participate in the game
currentPlayer:int is the index of the Player in the *players* array, 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.
GameState(playerNames:ArrayList) will be the constructor for the GameState class.
makePlayers(playerNames:ArrayList)) will instantiate the ArrayList of players.

getNrPlayers(): *Int* will return the number of players currently in the game.
setCurrentPlayer(playerNr: Int): void will change the currentPlayer attribute when necessary.
removePlayer(int):void removes a Player from the *players* ArrayList when it is eliminated.
getCurrentPlayerNr():int returns the index of the *currentPlayer*
getPlayer(int):Player returns the current Player
getDrawDeck(): CardCollection will return a copy of the draw deck.
getDiscardPile(): CardCollection will return a copy of the discard pile.
addToDrawDeck(cardName: Card): void will add a card to the top of the draw deck.
insertInDrawDeck(int):void will insert a (exploding kitten) card in the given index of the draw deck.
getDrawDeckSize():int returns the size of the draw deck.
addToDiscardPile(cardName: Card): void will add a card to the *discardPile*
removeFromDrawDeck(int): Card will remove and return a specific card from the draw deck
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.
getCards(int):ArrayList<Card> returns the list of cards in a CardCollection object.

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.
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(card): void will add a card to a card collection when needed, e.g. after a draw.
removeCard(int): Card will remove a specific card from a card collection, e.g. a played card, and return it.
insertCard(int, Card):void will enable the player to insert the exploding kitten card after it is defused.

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.
Display class a single and abstract class with static methods. The reasons for such display class design are: 1. Single responsibility principle, 2. Having a single class helps in maintainability and it is easier to make changes to display code for example changing output statements can be easily done as all are in single class, 3. Enable coe reuse because output of bot playing and human playing are similar, 5. more than two classes do output, if we make print class and have objects then we would need to make 3 or 4 print classes for example a print class for setup, a different print class for player, a different print class for command class, etc, therefore we do not use this approach and have a single class accessible to all other classes, although this method have its draw back that is the class grows bigger so it is that each method has its pros and cons and after discussing and studying we decided the single display class method was better approach.

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. Reason for having this to enable comparing player type in code to execute code as per the player type.

Object diagram

Author(s): Björn

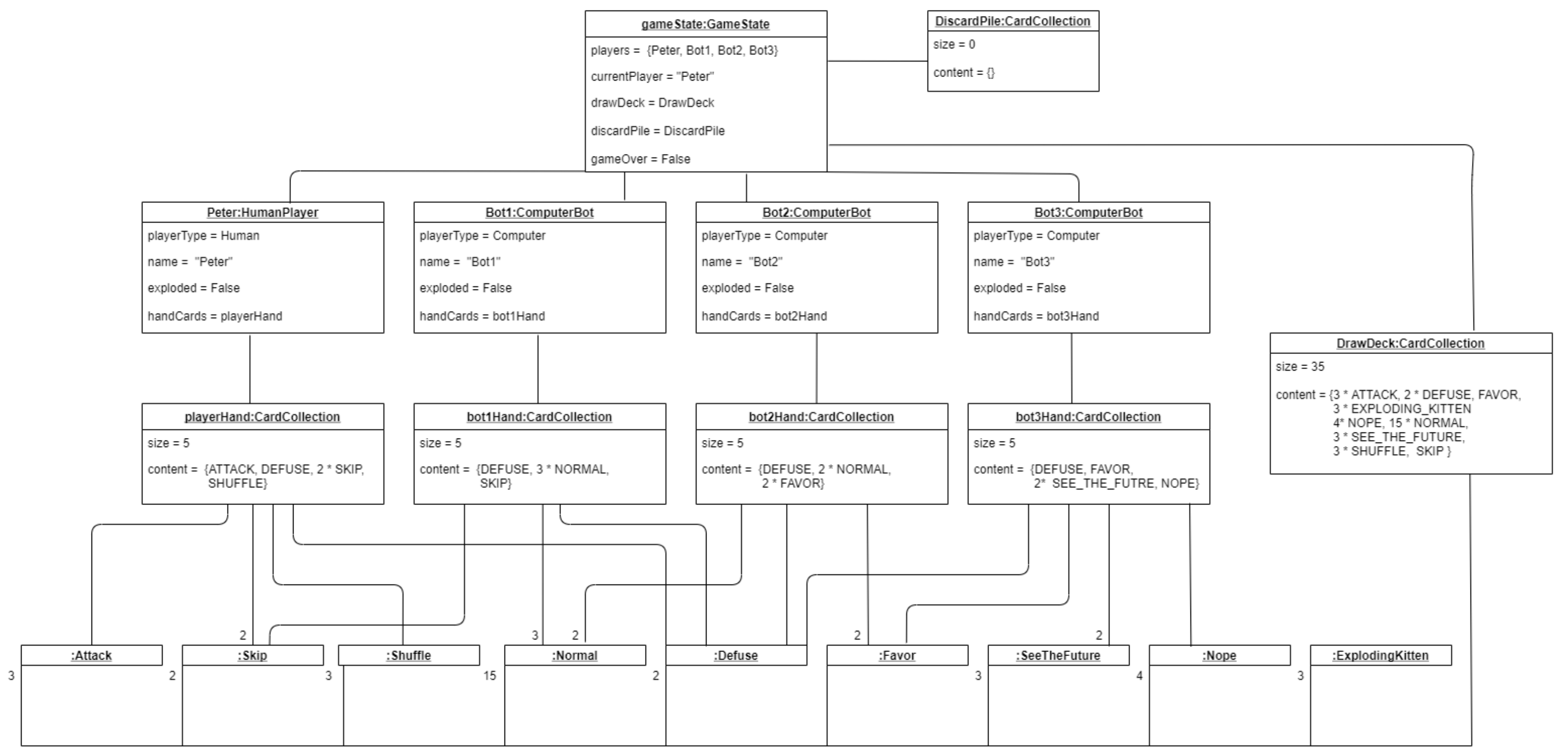


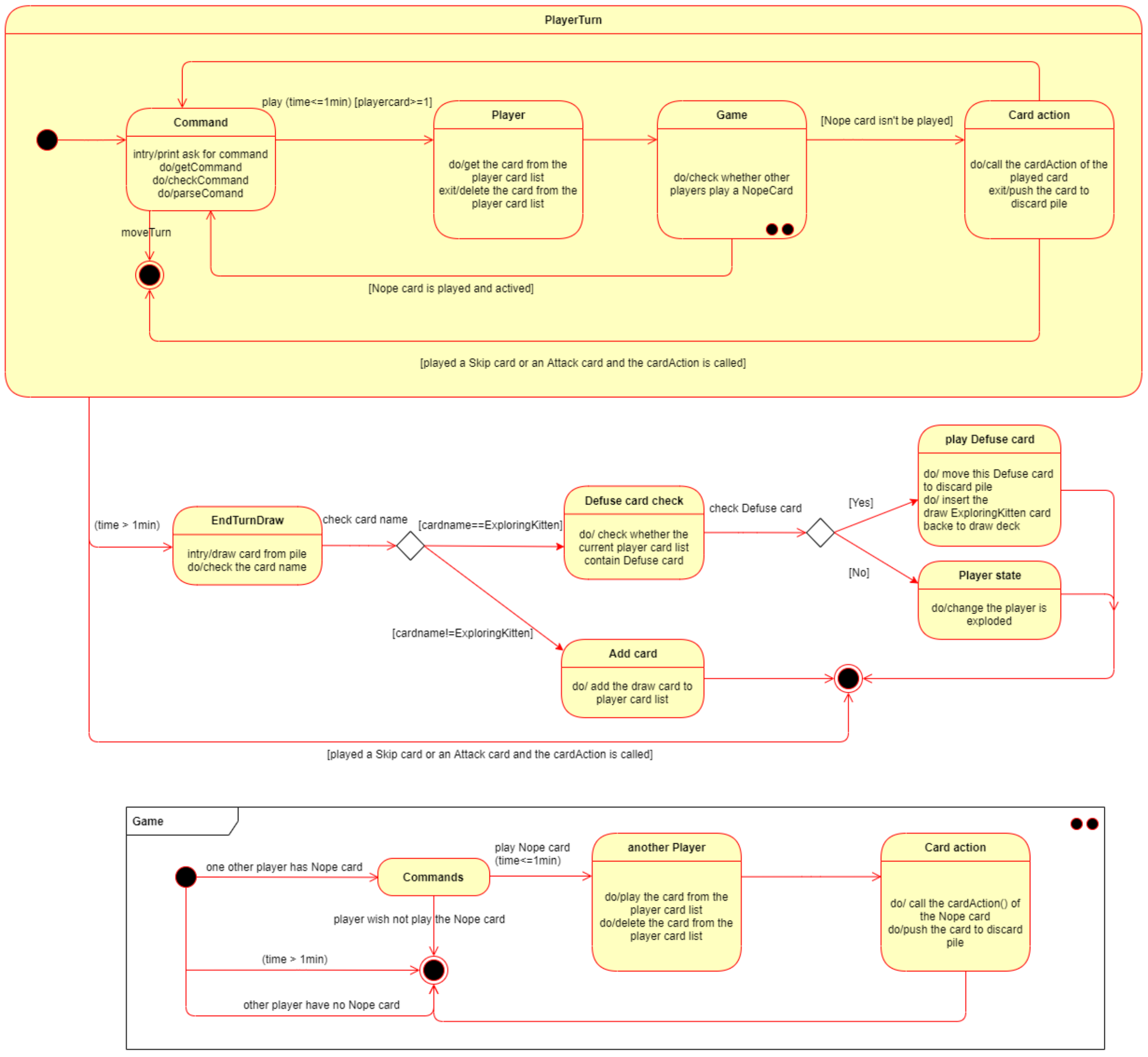
Figure representing the initial state of a game composed of 1 human and 3 bots (Diagram 2)

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


PlayerTurn
Command
intry/print ask for command
do/getCommand
do/checkCommand
do/parseComand
play (time<=1min) [playercard>=1]
Player
do/get the card from the player card list
exit/delete the card from the player card list
Game
do/check whether other players play a NopeCard
[Nope card isn't be played]
Card action
do/call the cardAction of the played card
exit/push the card to discard pile
moveTurn
[Nope card is played and actived]
[played a Skip card or an Attack card and the cardAction is called]
(time > 1min)
EndTurnDraw
intry/draw card from pile
do/check the card name
check card name
[cardname==ExploringKitten]
Defuse card check
do/ check whether the current player card list contain Defuse card
check Defuse card
[Yes]
[No]
play Defuse card
do/ move this Defuse card to discard pile
do/ insert the draw ExploringKitten card backe to draw deck
Player state
do/change the player is exploded
[cardname!=ExploringKitten]
Add card
do/ add the draw card to player card list
[played a Skip card or an Attack card and the cardAction is called]
Game
one other player has Nope card
Commands
play Nope card (time<=1min)
another Player
do/play the card from the player card list
do/delete the card from the player card list
Card action
do/ call the cardAction() of the Nope card
do/push the card to discard pile
player wish not play the Nope card
(time > 1min)
other player have no Nope card

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.

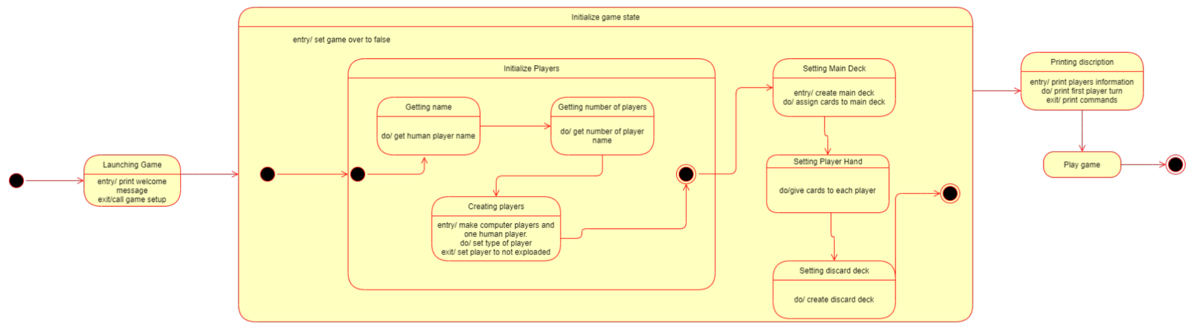
If the command is valid after the checkCommand(0 method, the parseCommand() will be called and parse the command. For example, if the player input to play a card, the state will go to the Player class to get the card from the card list, and then the Game will check with other players if they will play 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:


Launching Game
entry/ print welcome message
exit/call game setup
Initialize game state
entry/ set game over to false
Initialize Players
Getting name
do/ get human player name
Getting number of players
do/ get number of player name
Creating players
entry/ make computer players and one human player.
do/ set type of player
exit/ set player to not exploaded
Setting Main Deck
entry/ create main deck
do/ assign cards to main deck
Setting Player Hand
do/give cards to each player
Setting discard deck
do/ create discard deck
Printing discription
entry/ print players information
do/ print first player turn
exit/ print commands
Play game

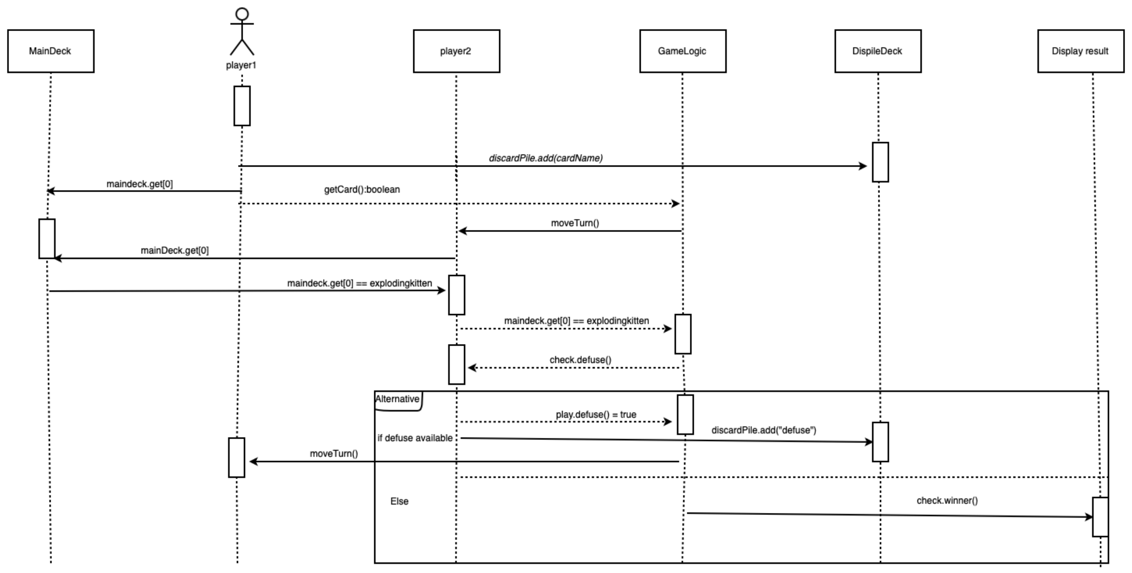
The state machine belongs to *Game* class and starts at the *setupGame* method. The game state is initialized. This happens in a specific order and different states are involved that are highlighted in the state machine. Throughout the state machine no guards are used because they are not needed. Next 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. After Initialising the players the main deck is set up and using the player objects the cards are given to each player. Then the discard deck is initialized empty. The decks cannot be initialised before the players because the main deck is set up according to the number of players in the game. 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

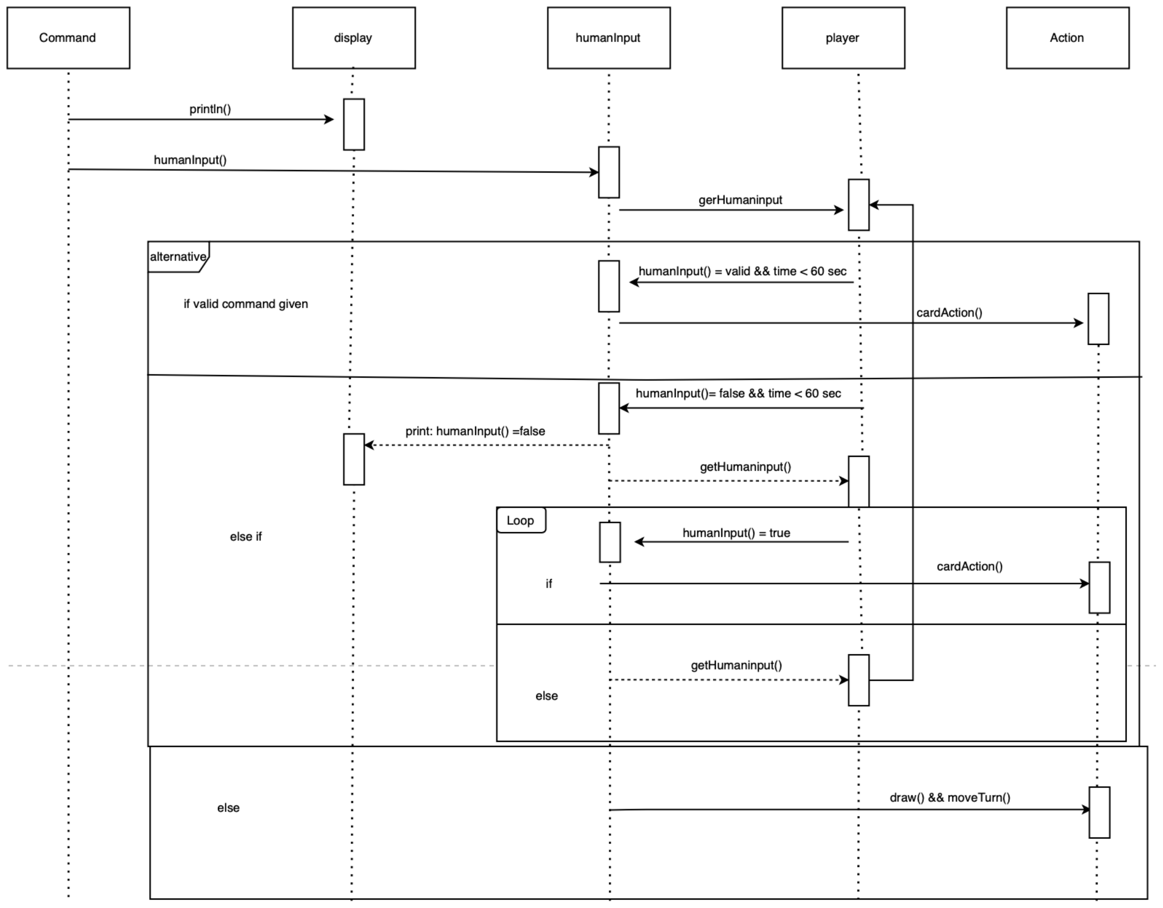
Descriptive



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 . This diagram is not intended to cover the initialization of the game.

We assume player1 is human and start the game. 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 human class calling command until the action method is called



This diagram shows the part in which the command class is called by the human class until an 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 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 *humanInput()* method will be called. We assume that the instruction is given and a command is given by the user. humanInput() 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 Input, the *humanInput() method will* first check if it is a valid command bij comparing all the available commands. If it is indeed the case that the given input is valid and within a timeframe of 60 sec. 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 (in a loop) or the command “quit” which will delete the player from the game and end the game. If the user gives a false command within the timeframe then a card will be drawn from the deck and the turn will be moved.

Implementation *Author(s): Bjorn, Hao, Muhammad, Ashish*

After making a good class diagram we started to implement the game. For assignment 2 we had made a prototype of the game, however this was slightly inconsistent with the class diagram and not fully according to the OOP standards. Therefore, taking the feedback into consideration, we started to implement the game in assignment 3 by OOP standards and in accordance with the UML Diagram. During the implementation phase we encountered a few issues which we handled differently than we originally intended, and therefore updated a few things in the UML diagram. Our strategy was to be finished and satisfied with the class diagram first before starting to implement. In assignment 2 I thought it would have been better to start implementing earlier. But I have come to realize that it is much easier to implement if all of the diagrams are profoundly thought through.

The greatest challenges of creating the explodingKitten game for us was to find the logic to implement all the actions of the cards so they all work perfectly together. One of the most difficult actions to implement was the attack card action. An Attack card ends your turn(s) without drawing from the deck and forces the next player to take an extra turn. But what happens when the next player plays a skip card? We read the rules and came to the conclusion that the next player then has 3 turns. This was a complex problem to handle, because we were moving the turns 1 by 1 and normally the player has to draw only one card and has 1 turn. Now we had to implement that the player had 3 or more turns. But we managed to fix this in our implementation (see our code), but it was a big hurdle. Another challenge occurred during the implementation of the command class. In the first instance we tried to check the last card which was thrown in the discard pile and act accordingly but this was not a nice approach since it seemed very “hard-coded” and didn't look efficient. Initially we had a parser class which we wanted to implement to check if the given input is correct. But later we realized that it was not needed to build a seperate class and just made a humanInput() method to get input from the human and check and call for action in one method in the command class. The command class is completely changed from the one which was implemented in assignment 2.

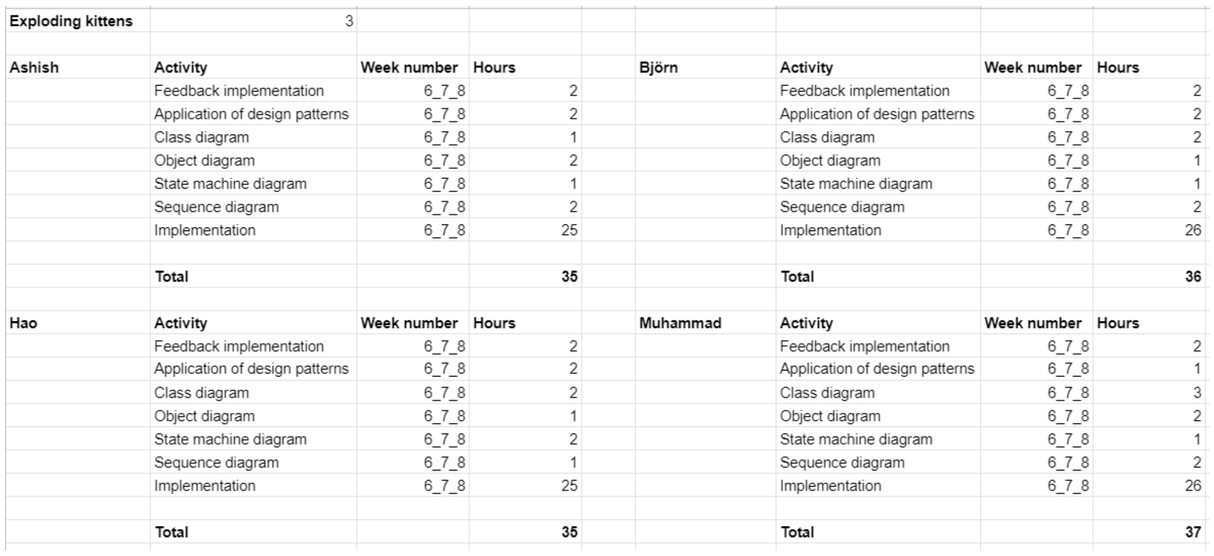
Besides the mentioned issues, there were of course more minor problems which interrupted the development, but that is normal while developing a system. If there are more questions or inclarity about the program, we suggest to look at the UML description, which can provide more explanation. It will give you a nice overview of what is going on. From this project we have learned and experienced how it is to work in the same project with different people in . It is more complex than you would expect, but if you have a good UML diagram that is built logically, it saves a lot of time and headache during the implementation. This course has also been a good practice to experience how it will be going while entering a programming job in the near future. Overall we liked the structure of the course and it was a very educational and important course in our opinion.

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

location of the jarfile:
out/artifacts/SoftwareDesignAssign3/SoftwareDesignAssign3.jar

An example of the game can be found by clicking on the following link.
https://youtu.be/cbVE1Z8v5Go

Time logs



Exploding kittens	3							
Ashish	**Activity**	**Week number**	**Hours**		**Björn**	**Activity**	**Week number**	**Hours**
	Feedback implementation	6_7_8	2			Feedback implementation	6_7_8	2
	Application of design patterns	6_7_8	2			Application of design patterns	6_7_8	2
	Class diagram	6_7_8	1			Class diagram	6_7_8	2
	Object diagram	6_7_8	2			Object diagram	6_7_8	1
	State machine diagram	6_7_8	1			State machine diagram	6_7_8	1
	Sequence diagram	6_7_8	2			Sequence diagram	6_7_8	2
	Implementation	6_7_8	25			Implementation	6_7_8	26
	Total		**35**			**Total**		**36**
Hao	**Activity**	**Week number**	**Hours**		**Muhammad**	**Activity**	**Week number**	**Hours**
	Feedback implementation	6_7_8	2			Feedback implementation	6_7_8	2
	Application of design patterns	6_7_8	2			Application of design patterns	6_7_8	1
	Class diagram	6_7_8	2			Class diagram	6_7_8	3
	Object diagram	6_7_8	1			Object diagram	6_7_8	2
	State machine diagram	6_7_8	2			State machine diagram	6_7_8	1
	Sequence diagram	6_7_8	1			Sequence diagram	6_7_8	2
	Implementation	6_7_8	25			Implementation	6_7_8	26
	Total		**35**			**Total**		**37**

https://docs.google.com/spreadsheets/d/1ns3TrgcDFuQzGHgjH_QyoATcjcJga-FC9y_NBiiWCy0/edit?usp=sharing