Software Design - Assignment 2

Team number: Exploding Kittens 14

Team members:

Name	Student Nr.	Email
Sergei Agaronian	2661076	s.agaronian@student.vu.nl
Sofia Fraile Zandejas	2659410	s.m.fraile@vu.nl
Zahraa Salman	2659909	z.salman@student.vu.nl
Tenzin Yangdostang	2707824	c.yangdotsang@student.vu.nl
Marco Huijs	2705658	m.c.d.huijs@student.vu.nl

This document has 15 pages without this title page, as stated in the assignment.

Implemented features

The diagrams are not feature-specific, however, to set realistic goals, the implementation part is feature-specific. Because we planned to work in an agile manner, we also implemented the features vertically, as suggested by the TA. This also means that the implementation part is an ongoing process, thus not every feature is finished and perfectly construed.

ID	Short name	Description	Champion
F1	GUI	The players will be able to access the game through a simple graphical user interface. The interface consists of the following components: - Rules button provides the user with practical information about the game and its goals. - Card buttons (only active when it's the user's turn). - Card buttons appear when the user draws a card and disappear if they play it. - To play a card, the user has to press the corresponding card button. - Every card has a question mark (?) icon to see what it does. - Use Nope button (only active if the Nope card is in hand and can be played at all times). - Discard Pile card is not visible unless a combination of five different cards is played. - Talon is not visible unless See The Future card is played, then a window pops up with three cards on the top of the talon. - When a player draws an exploding kitten, they can use the defuse card button.	Sofia
F2	Turn-based system	The players should be able to take turns according to the rules of the game. For this reason, a **Game State class** is added to the implementation. The Game State will keep track of the order in which turns are taken. Besides that, the Game State also keeps track of the effects applied by a certain type of card to determine what is going to happen in the next turn. When a player ends their turn, this class examines the played cards and applies the corresponding effects to the game environment or the other players. When there is only one player left on the playing field, the Game State class determines the winner and ends the game.	Marco
F3	Deck	The featured game utilizes two main types of decks: - **Talon:** includes all the undealt cards that can be drawn during the game. - **Discard pile:** includes played cards and can only be used when the rules allow. The proposed **Deck class** is going to store all the cards from the playing pool. The main goal of this class is to make the interaction between the piles and the players possible. The players would be able to draw the cards from the piles, shuffle the talon, hide an exploding kitten, and check the top cards of the playing pile with See the Future cards. Furthermore, the cards inside the Deck will be of a different class, namely the **Card class**. This class will contain all the information about the card properties and their corresponding effects.	Zahraa
F4	Player	To make competition possible we need players inside our system. For this reason, the **Player class** is going to be implemented.	Tenzin

		This class treats two types of agents: human and computer players. Before the game starts, the user chooses the number of competitors they want to play against. Based on the chosen parameter a game is started where the user is the human player and all the other competitors are computer players. The computer players simply perform random actions each turn, meanwhile, the human player is allowed to make decisions for themselves.	
F5	Rules	Each player has to stick to the rules of the game. There will be a **Rules class** that contains all the possible moves according to the official manual of the game. For example, a player that has two identical cat cards in their hand is allowed to play them as a pair and steal a random card from one of the opponents. When a player declares an action, this class will check its validity. Furthermore, for the user's reference, there will also be a rules button that opens a window containing all the relevant information about the game. This rules class would be different from the game class as it would only contain the possible moves the players can do, while the game class would contain all the information that the GUI would use to show the game, e.g. whose turn it is, how many players are playing, which player has which card, etc.	Zahraa
F6	User Actions	To play the game, the human player has a fixed set of actions they can perform using the game UI. During a game, the player can click on the following interactive elements of the GUI: - By pressing a **card icon** a user chooses to play it during their turn - **End Turn** button draws the top-most card from the playing talon and ends the current turn unless an exploding kitten is drawn - **Use Nope** allows the user to play a special Nope card - **Question mark icon** is available for every card in the user's hand to consult them on what the card does - **Trick/combo** buttons allow the user to play special combinations of cards. For example, a button to play a pair of identical cat cards. - **Defuse** button allows the user to defuse an exploding kitten While setting up a new game, the player can click on the following interactive elements of the UI: - **Player Count** allows the user to select the number of players in the game. - **Start Game** allows the user to start a game with the current settings	Sergei

***Used modelling tool**: diagrams.net*

Class diagram

Author(s): Tenzin

This chapter contains the specification of the UML class diagram of our system, together with a textual description of all its elements, such as the relevant attributes, operations and associations.

<For readability purposes figure representing the UML class diagram can be found on the first page of the Appendix through this link:
https://drive.google.com/file/d/1XT0fbEWbZJBRF6QhJosiBAEu_xfbdn2i/view?usp=sharing>

GUI:
The **GUI** handles the presentation of information to the player. We will be implementing the **GUI** using JavaFX and SceneBuilder libraries.

GameState:
The **GameState** class is one of the most important classes in the featured implementation since it connects the **GUI** with the rest of the system. This class controls the state of the general system, as indicated by its name. The **GameState** class is the heart of the implementation that keeps all the relevant information about the necessary attributes of the game.

Attributes
Attributes are a significant piece of data containing values that describe each instance of that class.

- gui: GUI - initially, the player will be able to access the game and start it through a simple yet eye-pleasing user interface. This attribute will hold several components, such as, the “rules button”, which can be used to inform the player on the general rules of the game, the “card buttons”, that allow the user who is currently holding the turn to play a certain card from their hand. Moreover, this attribute will also hold a “play nope card” button, since this card holds a particular value in the game and has a different rule set according to which it can be played. The significant value is also true for the “play defuse card” button. Naturally, some of these components of the gui attribute will only be visible to the user in relevant situations.
- drawPile: TalonPile - Once the player has started the game, a new *TalonPile* object will be created. Thus this attribute holds a reference to the draw pile that is to be used during the currently running game. The *TalonPile* allows the players to access the playing deck, and draw cards.
- discardPile: DiscardPile - Once the player has started the game, a new *DiscardPile* object will be created. This attribute holds a reference to the discard pile that is to be used during the currently running game. The discard pile holds the information about the already played cards, and allows the users to access them when they meld a 5 card combination.
- turnBase: TurnBase - Once the player has started the game, a new *TurnBase* object will be created. This attribute holds a reference to the turn base structure that will be

construed during the currently running game. This attribute holds the information that is necessary to imitate the clockwise turn pass present in the original game.

- winner: Player - Since every game requires at least one winner and one loser, naturally this class contains an attribute that holds a Player object, that will eventually be the winner of the game, namely the last player who is left with cards in their hand. Thus, this attribute holds a reference to the winning player that will be decided through the currently running game. This attribute is an important detail that leads the state of the game to its ending point.

Operations

Operations, or also known as methods or functions, allow you to specify any behavioural feature of a class

- startGame(numPlayers: int): - This operation allows the user to initialize the game. After this method is called, all the attributes of the **GameState** will be updated to set the settings of the starting point of the game. This operation requires the parameter numPlayers that holds the number of players that will join this game. The number of players is determined by the user before the game starts. The original game is designed to be played by 2 to 5 players, and the featured implementation inherits this principle.
- nextTurn() - This method will be called when the player's turn has ended and the turn has to be passed on to the next player. To do so, the *GameState* object will be accessing the *TurnBase* object.
- isTalonEmpty(): boolean - This method checks whether the talon pile is empty or not. This method is crucial to keep the structure of the game and prevent the players from accessing an already empty draw pile. Naturally, the method returns true when the pile is empty, and false when it is not.
- isDiscardPileEmpty(): boolean - This method is used to indicate whether the discard pile is empty or not. This will return true at the start of the game since nothing has been put on the discard pile yet. Similar to the previous function, it helps in keeping the structure of the game and prevents erroneous actions.
- setPlayerHand(): Map<Player, List<Card>> - This method sets each of the player's cards in their hands respectively. A map is used to make pairs of the players and their hands. The method is called at the beginning of the game to set the playing hands.
- setActivePlayer(): Map<Player, boolean> - This method sets a player in an active status, meaning the boolean value is true when it is their turn. So the map contains a pair of the players and the boolean value that is dependent on their turn.
- getWinner(): Player - By accessing the **TurnBase** class and consequently the **PlayerStatus** this method determines the winner of the game. This happens through the list of all "active" and "dead" players.
- gameIsOver(): boolean - This boolean method checks whether the game has ended or not, meaning true, when the game is over and a winner is decided or false when the game is still ongoing. This method will make use of the **PlayerStatus** to determine how many players are left in the current game. If the **PlayerStatus** returns a list of one active player, then the game is over and the winner is the last player standing. If the game is over, and the method returns true value, then the getWinner() method is called to indicate the winner.

Associations
This indicates how the class is related to other classes.

- GUI - Associated. The **GUI** and the **GameState** need to communicate with each other, meaning there is a link/connection between the two. The arrow indicates the navigation direction to **GUI**. The multiplicity, that allows us to set numerical constraints on the relationships, shows here that a single **GameState** can be associated with a single **GUI**.
- DiscardPile - Associated. The **GameState** should be able to communicate with the **DiscardPile** to get all the relevant cards for interaction. Specifically, when a player plays a combination of 5 cards and can pick a card from the discarded cards. A single **GameState** can be associated with a single **DiscardPile**.
- TalonPile - Associated. Similar to the **DiscardPile**, the **TalonPile** should be able to communicate with the **GameState** to keep all the information about the playing cards relevant and to allow the players to interact with the cards.
- TurnBase - Associated. The **GameState** will be able to access the **TurnBase** by accessing its attributes, using its methods, and returning relevant information. A single **GameState** can be associated with a single **TurnBase**.

TurnBase:
The **TurnBase** class allows the players to take turns following the rules and set up of the game. It contains the following attributes, operations and associations with other classes.

Attributes
- playerStatus: PlayerStatus - Creates a new *PlayerStatus* object and holds a reference to the status of a player during the currently running game.
- activePlayer: Player - Creates a new *Player* object and holds a reference to the player who has the turn at the moment.
- turnsList: List<Player> - Creates a list of the turns the player must take accordingly based on the list of players. During the game, the **TurnBase** will iterate over this list to imitate a clockwise direction in which the original game is played.

Operations
- getPlayerController(): PlayerStatus - This method returns the status of a player.
- getCurrentPlayer(): Player - This method should return the current player, meaning the one holding the turn.
- endTurn() - This method allows the player to end their turn. By ending the turn a player should always draw a card from the talon pile.

Associations
- PlayerStatus - Composite. This implies a relationship where the child cannot exist independent from the parent. So **TurnBase** cannot exist without **PlayerStatus**. This is due to the fact that **PlayerStatus** holds the information about all the players currently in the game, and without the players, there cannot be a turn base structure.

PlayerStatus:
The PlayerStatus is important to keep track of the number of players in the current game, so the dead players and the active players must be determined.

Attributes

- activePlayers: List<Player> - This attribute holds a reference to the list of active players that are currently still participating in the game.
- deadPlayers: List<Player> - This attribute holds a reference to the list of dead players that are not participating in the game.

Operations

- getActivePlayers(): List<Player> - This operation makes it possible to get a list of active players.
- getPlayerHand(): Map<Player, ArrayList<Cards>> - This operation gets the hand of a specific player based on their key values in the map.
- addNewPlayers(numPlayers: int) - This operation adds new players to the game and has as a parameter the number of players that must be updated accordingly.

Associations

- Player - Composite. **PlayerStatus** cannot exist without the players. The multiplicity indicates that a single **PlayerStatus** contains a minimum of 2 players and a maximum of 5 players.

Player:
This class is an abstract superclass, meaning that its attributes and methods also apply to its subclasses, in this case, **HumanPlayer** and **ComputerPlayer**. If this class is changed in any way, then the subclasses will also be changed in the same way.

Attributes

- Name: String - the name of the *Player* object.
- playerHand: PlayerHand - the hand of the *Player* object that keeps the information about all the playing cards that a player possesses.

Operations

- getName(): String - method returns the name of the player.
- setName(name: String): void - sets the name of the player to a specified value.
- setPlayerHand(hand: HandCards) - sets the playing hand of a player.
- getPlayerHand(): HandCards - returns the playing hand of a player.
- addToPlayerHand(card: Card) - adds a card to the playing hand of a player.
- removePlayedCard(card: Card) - removes the played card from the player's hand.
- hasCard(card: Card): boolean - returns true if a player has the specified card in their hand. Returns false if the player does not have the card.
- drawCard(): Card - draws a single card from the talon pile and puts inside the player's hand.

Associations

- HumanPlayer - Inherited. **HumanPlayer** inherits the attributes and methods of **Player**.
- ComputerPlayer - Inherited. **ComputerPlayer** inherits the attributes and methods of **Player**.
- HandCards - Composite. Each player has exactly one playing hand. Hand cannot exist without a player attached to it.

HumanPlayer:
This subclass represents the user inside the featured implementation.

Operations

- passTurn() - passes the turn chosen by the player to the **GameState**.
- playNope() - the user can play the nope card at any point in time during the game, and this method allows the user to choose to either play the Nope card or not.

ComputerPlayer:
This subclass represents the computer opponents of the user.

Operations

- playRandom - plays a random card from the player's hand.
- playRandomNope - allows the computer player to play the Nope card at any point in time by asking the player whether they want to play the card or not. For the computer players, playing the Nope card has a random probability.

HandCards:
This class manages the hand of the player. It contains operations such as the boolean hasCard, which returns true if the player has a specific card in their hand. All these operations are there to manage and access the hand of a player.

Attributes

- hand: ArrayList<Card> - the list contains all the cards that the player has.
- selectedCards: ArrayList<Card> - indicated the list of cards selected by the player to be played.

Operations - The operations of this class are self explanatory and are used to manage the hand of the player.

Associations

- TalonPile - bidirectionally associated. The association indicates that the player hand gets populated through the talon pile.

Deck:
This class manages the deck of the game, which is why it's also connected to the **TalonPile** and the **DiscardPile** classes.

Associations

- TalonPile - Composite. The multiplicity makes it clear that one **Deck** holds exactly one **TalonPile** and only one **TalonPile** can exist in one **Deck**.

- DiscardPile - Composite. The multiplicity makes it clear that one **Deck** holds one **DiscardPile** and one **DiscardPile** can exist in one deck.
- Card - Aggregated. This is a type of relationship where a part can exist outside of a whole, the multiplicity shows that the **Deck** can hold 0 to many cards and that cards exist in exactly one **Deck**.

TalonPile:
This class manages the draw pile of the game through different methods, such as populating the draw pile and drawing the top card of the pile. The operation seeTop3Cards is directly connected to the **SeeTheFuture** subclass since that subclass allows a player to see the first three cards of the draw pile.

Associations
- Rules - bidirectionally associated.

DiscardPile:
This class manages the discard pile of the game through different methods, such as adding cards to the discard pile.

Card:
This class holds a reference to all the different cards in the game including the action cards. The bidirectional association with the enumeration class **CardType** indicates what kind of cards can exist in the game.

Rules:
This class is a superclass over its subclasses shown through the inheritance vectors. The operation in these subclasses is dependent on what kind of action card a player plays, for example, if the player plays a nope card, then the cardAction operation is different than when a player draws an exploding kitten card.

Object diagram
Author(s): Marco Huijs

<For readability purposes figure representing the UML object diagram can be found on the second page of the appendix file:
https://drive.google.com/file/d/1XT0fbEWbZJBRF6QhJosiBAEu_xfbdn2i/view?usp=sharing>

The Object diagram in the appendix represents a snapshot of a three-player game of Exploding Kittens.

"myGame" is an instance of the **GameState** class. It represents the state of a game of Exploding Kittens. It contains a *GUI* object which serves as an interface to the user, *TalonPile/DiscardPile* objects to keep track of the cards in the game, and a *TurnBase* object to keep track of the turn order. The winner attribute of the “myGame” object indicates that the game is ongoing and the winner is not yet determined.

In this snapshot, the *TalonPile* can be seen to contain five *Card* objects, while the *DiscardPile* contains four *Card* objects.

The *TurnBase* object contains a *PlayerStatus* object to keep track of all the *Player* objects in the game. This snapshot contains exactly three players: *Zahraa*, *Tenzin*, and *Marco*. *Zahraa* and *Marco* are computer players, *Tenzin* is a human player. *Marco* is already eliminated from the game, as can be seen in the deadPlayers list.

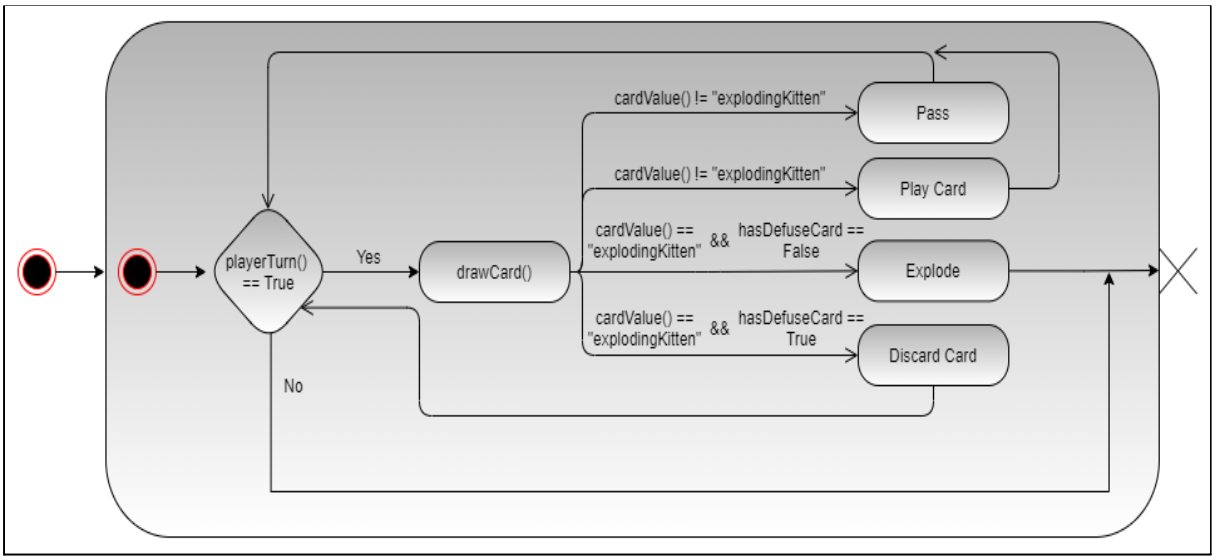
Each *Player* object has a *HandCards* object representing the cards they are holding. In this snapshot, *Zahraa* is holding two cards, while *Tenzin* is holding five. *Marco* has no cards, as he has already been eliminated.

The selectedCards list in Tenzin's *HandCards* object shows that she has selected three cat cards. She has selected these cards through the *GUI*, and can now decide to play them together.

State machine diagrams

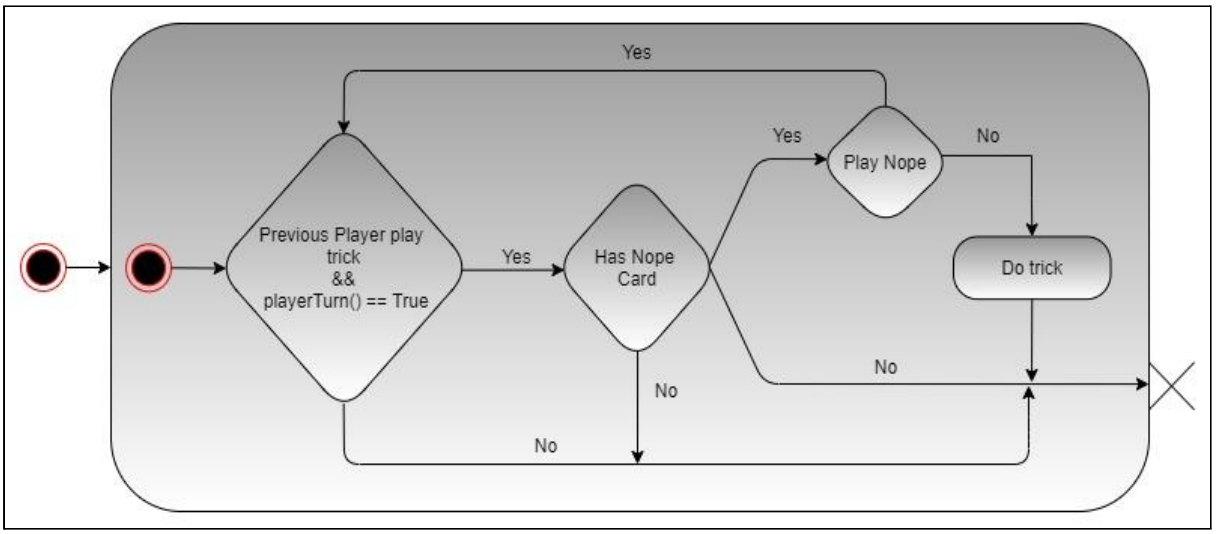
Author(s): Zahraa

First Diagram



The following state machine diagram represents the situation when a Player draws a card from the Talon. This is implemented in the Rules class. When it is the Player's turn and they draw a card there are a few possible scenarios that can happen. This diagram focuses on a single scenario when a Player either draws an Exploding Kitten or they do not draw it. If the card is indeed an Exploding Kitten, the Player will explode, unless they have a Defuse card that allows the Player to put the Exploding Kitten card back into the Talon and stay in the game. If the Player draws any other card then they can either pass or play the card. Once they choose an action they loop back to check if it is their turn, this is checked by the Rules class using the turn-based system, if it is their turn then they can draw a card again, otherwise, they exit the state. They also exit the state after they explode.

Second Diagram

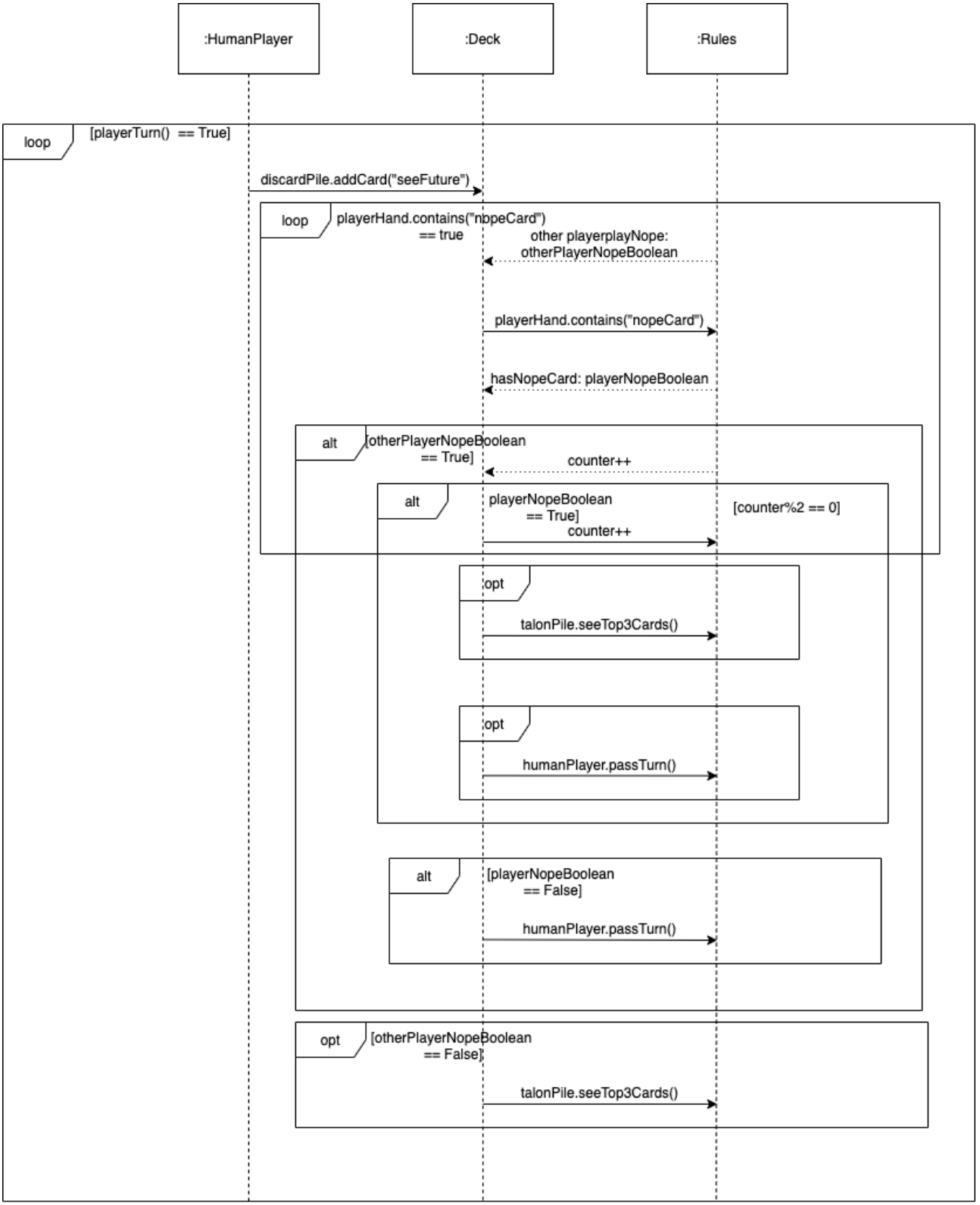


The second state machine diagram considers the Nope card loop. The game allows a loop of the current Player using a Nope card and another Player playing a Nope card that cancels the original Player's Nope so the original player has to do the trick. However, this can continue if the other players play another Nope and so on. The diagram describes the situation if a previous Player plays a trick and it's the current Player's turn, if this is not the case then the Player exits the state. If the Player has a Nope card they can choose to play it and that loops over another player playing Nope. If they choose to pass or they didn't have a Nope card in the beginning, they have to do the trick and exit the state.

Sequence diagrams

Author(s): Tenzin

HUMAN PLAYER PLAYS NOPE CARD



For this part of the document, we had to model two sequence diagrams of our system. These diagrams are supposed to be focusing on specific cases. We decided to model the first sequence diagram on a situation in which a Player plays a Nope card. This is shown in the diagram above. The interaction partners are as stated the Human player, Deck, and the Rules class. These three components interact with each other in a case the Human player plays a Nope card. The outer loop runs until it is no longer the current Player's turn. So, the diagram depicts the following situation:

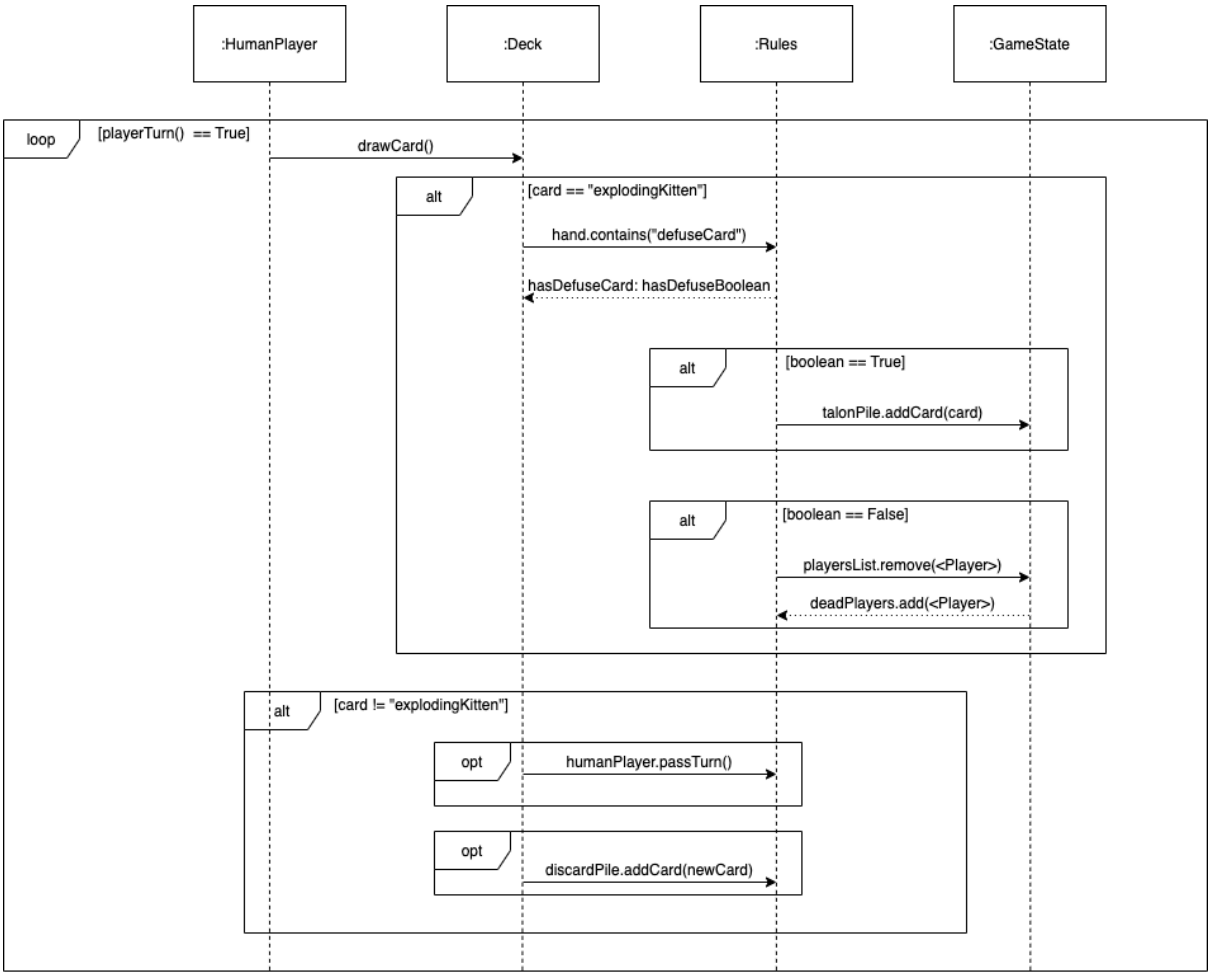
Player A plays the See The Future card, and that card is pushed onto the Discard pile. Player B has a Nope card and decides to play the Nope card and that is allowed, since the Nope card can be played throughout the entire game, this blocks Player A's opportunity to see the first three cards of the Draw pile. But it is naturally also plausible that player A has a nope card in their hand. Thus the inner loop iterates until one of the Players run out of their nope cards. This is checked with the loop's condition. If the Player has a Nope card, then a message is returned with a boolean value of "true" and the loop is exited when one of the players doesn't have a Nope card, meaning the returned message contains the boolean value "false".

Then we have an outer alt fragment that depicts the situation in which Player A has a Nope card in their hand, i.e. they can "nope" Player B's Nope card with emphasis on can. Player A can also choose to keep their Nope card for another occasion. This is depicted by the two inner opt fragments in the first inner alt fragment. If Player A does play their nope, then they are allowed to play their original See The Future card and can therefore see the top three cards of the draw deck. In case they don't play their Nope card, their turn ends naturally. The counter keeps track of how many Nope cards have been played, if the counter is an even number then the Nope card trial has been ended by Player A.

Nevertheless, if Player A doesn't even have a Nope card in the first place or if their Nope card got "noped" again by Player B, then their turn ends and the next Player can play their cards.

The last opt fragment depicts a situation in which player B doesn't play their Nope card, meaning Player A can play their See The Future card.

HUMAN PLAYER DRAWS AN EXPLODING KITTEN



The second sequence diagram depicts a situation in which a Player draws an Exploding Kitten from the Draw pile. We have chosen to model this scenario since it is a very crucial point in the game. This can decide the winner or the loser. The interaction actors are Human player, Deck, Rules, and Game state. These components interact with each other in this specific scenario. The outer loop checks whether the Player is even allowed to play since it needs to be their turn for that to be true. So as long as it is the current Player's turn, the loop iterates. The Human player then draws a card from the Deck, this is shown through the exchange message between these two actors.

There are two alternative scenarios, in this case, the Player either draws an Exploding Kitten card or not. The first alt fragment describes that an Exploding Kitten card has been drawn by the Player. Under the Rules, you can only stop this card with the use of a Defuse card. So the Player either has a Defuse card or not, this is returned through the exchange message from Rules to Deck. If this boolean turns out to be true, meaning the player has a Defuse card, then the Exploding Kitten card is inserted again into the Draw deck. If the boolean turns out false, meaning the player doesn't have a Defuse card, then the player automatically loses the game and is added to the Dead players' list. This is clearly shown through the exchange messages between the Rules and Game state. However, there is also the chance that the Player doesn't draw an Exploding Kitten card at all and this leads to two plausible scenarios. Either Player ends their turn or plays a card and that card is added to the Discard pile.

Implementation

Author(s): Sofia

For the start of our implementation, we had to look at our UML models and decide on a good starting point that would allow us to continue working horizontally and make sure we had a good strategy going forward. For this, we decided to start with the simple move of drawing a card as this can be easily represented in the GUI.
Following the implementation of this meant to create a scene with SceneBuilder and JavaFX that would contain a button that would make a call on the drawing a card function.
JavaFX and Java 11 have their challenges when working together on IntelliJ which were unexpected and have caused some delays, however, the implementation, as mentioned before had to be horizontal, therefore the classes Card, Deck, DiscardPile, GameState, HandCards, Launcher, Main, Player, PlayerStatus, Talon, and TurnBase were all created with the basics in them. The resources file was also created with the FXML file for the view of the decks and drawcard button.

The main solution we went for at the start of the implementation was to allow the Player to draw a card from the Deck by simply checking if the Deck is empty or not and then remove the top card.

Since our implementation was done in a horizontal manner we advanced in most classes in this way, this means that most of our time was spent creating each class with its basic functions to make sure that each step of the way was done in conjunction with the classes that were used in parallel.

Our main Java class can be located under src\main\java\softwaredesign\main.java

Our Jar file can be located under \out\assignment2.jar

As mentioned above, the execution of our system does not yet allow for GUI implementation to be shown, due to a few issues with JavaFX, however, we are confident that we have made enough progress within our classes and overall system to show our work. The intended implementation example can be seen with the file src\main\resources\softwaredesign\DeckView1.fxml

Time log

Team number:	14		
Member	**Activity**	**Week number**	**Hours**
Zahraa Salman	Implement Card Class	3	0.5
Zahraa Salman	Implement Deck Class	3	0.5
Zahraa Salman	State Machine Diagram	3	2
Zahraa Salman	Sequence Diagrams	4	2
Tenzin Yangdotsang	Sequence Diagrams	4	2
Marco Huijs	Implement GameState	4	2

Sofia Fraile	Class Diagrams	4	3
Sofia Fraile	UI Implementation	5	4
Tenzin Yangdotsang	Class Diagrams	4	3
Tenzin Yangdotsang	Implement Player class	5	1
Marco Huijs	Object Diagrams	5	2
Marco Huijs	Write report	5	1
Marco Huijs	Implement PlayerStatus and TurnBase	5	2
Sergei Agaronian	Class Diagrams	5	3
Sergei Agaronian	Object Diagrams	5	1
Sergei Agaronian	Revise Assignment 1	5	1
Zahraa Salman	Implement Card and Deck Class	5	4
Zahraa Salman	Write report	5	4
Tenzin Yangdotsang	Implementation	5	3
Tenzin Yangdotsang	Write report	5	4
Sofia Fraile	Write report	5	2
Sofia Fraile	Implementation	5	2
Sergei Agaronian	Write report	5	3
		TOTAL:	41

Evaluation

Firstly, we learned a lot about both the programming language Java as well as the IDE IntelliJ through this assignment. We also learned the importance of UML diagrams, as they are supposed to help your teammates and other stakeholders involved in this project to understand better what you are doing. We experienced that clear UML diagrams are necessary, because if your teammates can't even comprehend what your intentions were, then other stakeholders can not. What went well was the communication between us and the TA, whenever we had a question, we didn't hesitate and asked the question right away in Slack, which has been very useful on various occasions. Secondly, what can go better with the next assignment in communication with each other, since communication is the key to success in every team project. We think that we should check more often with each other through regular fixed meetings, which might bring our project even closer to a real "agile" development. We could also improve our organization because there wasn't a clear structure or organization in the way we worked. The planning and the organization had a big impact on our workload since we spent a lot of time figuring out the diagrams. This led to us falling behind on the schedule for the implementation of the diagrams. We believe that working with a logbook next time will help us avoid these mistakes.