Assignment 2

Team number: 2
Team members

Name	Student Nr.	Email
Sebastian Fredrik	2669832	s.r.fredrik@student.vu.nl
Federico Giaj Levra	2674188	f.giajlevra@student.vu.nl
Kevin Koeks	2680522	k.a.b.koeks@student.vu.nl
Jelena Masic	2645593	j.masic@student.vu.nl

Conventions

Style

We applied the following formatting conventions in the text of this document:

- Class names are written in **bold**, e.g., **ExplodingKittens**
- File names and shell commands are written with a `monospace` font
- Object instances are written in *italic*, e.g., *deck* (instance of the **Deck** class)
- Attributes (class fields), operations (class methods), packages, and associations (relationships between classes) are underlined
- States and events in state-machine diagrams are underlined
- Sequence diagrams components (e.g., `alt` fragments) are written with a `monospace` font

For the diagrams, we use the following colouring conventions:

- White: descriptive
- Blue: prescriptive
- Orange: external (e.g., library component)

Diagrams

When not mentioned, composition and shared aggregation between classes have a 1:1 multiplicity.

Implemented features

We fully implemented the following features.

ID	Short name	Description
F2	Deck Functionalities	The original Deck is composed by 56 cards, divided as explained in the introduction (here), we would like to implemented so that it can be instantiated with any combination of the cards and it will have the following functionalities which allow for the use of it: 1. Pick a card 2. Shuffle the deck 3. Insert a card in the deck in a given position
F3	Discard Deck	When a card is played it is inserted in the discard deck. The discard deck allows for the following functionalities: 1. Add card(s) to the top. 2. Take a card from the discard deck
F4	Combos	The following combos are available: ● Two card combo, three card combo, five card combo. There is a detailed description in the introduction of Assignment 1
F5	Game mechanics	Different phases of the game have to be developed, such as: ● Game start, switching turns, attack turns, playing a card, keeping track of a players hand, Game end

Used modeling tool: diagrams.net

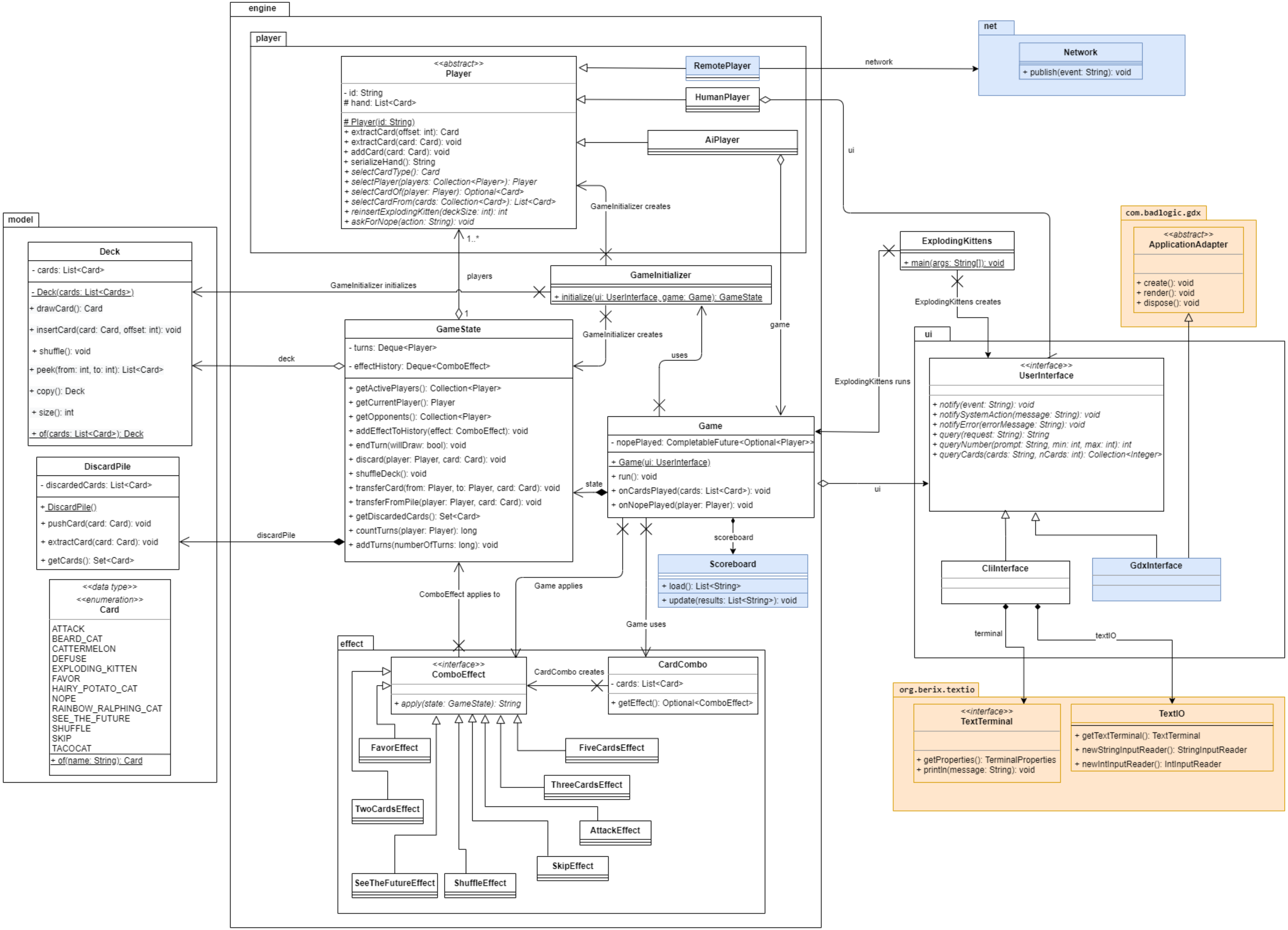
Class diagram

Sebastian Fredrik, Federico Giaj Levra, Jelena Masic

In the figure below our system's class diagram is depicted. It illustrates the general structure of our game's codebase, how it's separated into parts with each their own responsibilities and how each part is connected to create the whole.

The graph starts off at the **ExplodingKittens** class, which creates a type of interface implemented from **UserInterface** interface. This type of interface is for now a command line interface, or **CliInterface** object, and drives the user interface of the game. The **ExplodingKittens** class also creates a **Game** object with a reference to the aforementioned ui, as the driving force of the engine, which is built on top of the model. The model holds specifications for the deck that you draw cards from, the deck where you discard your cards on and the cards themselves. The engine specifies what the players can do, both humans and AI, specifies what effect the cards can have on the game and implements the **GameState**, operated by the **Game** object.

For better readability and for the sake of space, we decided to describe our classes in prose so that we would not exceed (too much) the page limit for the current section.


engine
player
<<abstract>>
Player
- id: String
hand: List<Card>
Player(id: String)
+ extractCard(offset: int): Card
+ extractCard(card: Card): void
+ addCard(card: Card): void
+ serializeHand(): String
+ selectCardType(): Card
+ selectPlayer(players: Collection<Player>): Player
+ selectCardOf(player: Player): Optional<Card>
+ selectCardFrom(cards: Collection<Card>): List<Card>
+ reinsertExplodingKitten(deckSize: int): int
+ askForNope(action: String): void
RemotePlayer
HumanPlayer
AiPlayer
network
net
Network
+ publish(event: String): void
ui
GameInitializer creates
1..*
players
game
GameInitializer
+ initialize(ui: UserInterface, game: Game): GameState
GameInitializer initializes
GameInitializer creates
uses
1
model
Deck
- cards: List<Card>
- Deck(cards: List<Cards>)
+ drawCard(): Card
+ insertCard(card: Card, offset: int): void
+ shuffle(): void
+ peek(from: int, to: int): List<Card>
+ copy(): Deck
+ size(): int
+ of(cards: List<Card>): Deck
deck
DiscardPile
- discardedCards: List<Card>
+ DiscardPile()
+ pushCard(card: Card): void
+ extractCard(card: Card): void
+ getCards(): Set<Card>
discardPile
<<data type>>
<<enumeration>>
Card
ATTACK
BEARD_CAT
CATTERMELON
DEFUSE
EXPLODING_KITTEN
FAVOR
HAIRY_POTATO_CAT
NOPE
RAINBOW_RALPHING_CAT
SEE_THE_FUTURE
SHUFFLE
SKIP
TACOCAT
+ of(name: String): Card
GameState
- turns: Deque<Player>
- effectHistory: Deque<ComboEffect>
+ getActivePlayers(): Collection<Player>
+ getCurrentPlayer(): Player
+ getOpponents(): Collection<Player>
+ addEffectToHistory(effect: ComboEffect): void
+ endTurn(willDraw: bool): void
+ discard(player: Player, card: Card): void
+ shuffleDeck(): void
+ transferCard(from: Player, to: Player, card: Card): void
+ transferFromPile(player: Player, card: Card): void
+ getDiscardedCards(): Set<Card>
+ countTurns(player: Player): long
+ addTurns(numberOfTurns: long): void
state
Game
- nopePlayed: CompletableFuture<Optional<Player>>
+ Game(ui: UserInterface)
+ run(): void
+ onCardsPlayed(cards: List<Card>): void
+ onNopePlayed(player: Player): void
scoreboard
Scoreboard
+ load(): List<String>
+ update(results: List<String>): void
ComboEffect applies to
Game applies
Game uses
effect
<<interface>>
ComboEffect
+ apply(state: GameState): String
CardCombo creates
CardCombo
- cards: List<Card>
+ getEffect(): Optional<ComboEffect>
FavorEffect
FiveCardsEffect
ThreeCardsEffect
TwoCardsEffect
AttackEffect
SkipEffect
SeeTheFutureEffect
ShuffleEffect
ExplodingKittens
+ main(args: String[]): void
ExplodingKittens creates
ExplodingKittens runs
com.badlogic.gdx
<<abstract>>
ApplicationAdapter
+ create(): void
+ render(): void
+ dispose(): void
ui
<<interface>>
UserInterface
+ notify(event: String): void
+ notifySystemAction(message: String): void
+ notifyError(errorMessage: String): void
+ query(request: String): String
+ queryNumber(prompt: String, min: int, max: int): int
+ queryCards(cards: String, nCards: int): Collection<Integer>
ui
CliInterface
GdxInterface
terminal
textIO
org.beryx.textio
<<interface>>
TextTerminal
+ getProperties(): TerminalProperties
+ println(message: String): void
TextIO
+ getTextTerminal(): TextTerminal
+ newStringInputReader(): StringInputReader
+ newIntInputReader(): IntInputReader

The model

In the model package you can find the foundation of our game, starting with the **Card** enumeration of possible cards in our **ExplodingKittens** game. Since this enumeration is widely used in all classes as a type-safe way of handling cards, we omitted all of the arrows to this class and just treated it as a (non-primitive) type[1], in the same way we did for **String**s.
Since **Card** is an enumeration, you can easily get all types of cards by calling values(). We also made an of(name: String) factory method for **Card**s, which checks if the given string matches a value of **Card** and otherwise returns null, so it also can be used to check if a card written in the CLI is an actual card.

These cards then are held in two types of collections, both built using a cards list, the **Deck** and **DiscardPile**. The **Deck** has the cards the players take after a turn, so all the actions that can happen to the **Deck** have methods e.g. shuffle() and peek(from: int, to: int). The class also contains a copy constructor and a factory method that takes in a list of cards and makes a **Deck** from that.
The **DiscardPile** has fewer actions attached to it but does have some differences from the **Deck**. The **Deck** had an insertCard(card: Card, offset: int) method because after defusing an *EXPLODING_KITTEN* card, a player can put it back into any position in the deck, whereas with the **DiscardPile** you can only put your card on the top so it has a pushCard(card: Card) method. The **Deck** would allow you to draw the top card with drawCard(), while with the **DiscardPile** you can draw any card with extractCard(card: Card) as long as you've used a five card combo.
Most of these methods are used by the **ComboEffect**s, not by any other code directly.

The engine

The engine as said before builds on the foundation made by the model, with at the heart being the **Game**, the controller of the program. It has a **UserInterface** ui from which it gets the inputs and to which it signals the events of the game and it has a **GameState** state which holds information about the status quo, i.e. whose turn it is and what **ComboEffect**s are currently active.
When it's first made by **ExplodingKittens**' main() function, the constructor makes the passed ui the **GameState**'s ui and then gets a state from the initialize(ui: UserInterface, game: Game) method in **GameInitializer** and then **Game**'s run() method is used to run turns until only one player is left. When a player plays a card or selection of cards, it will get handled by onCardPlayed(cards: List<Card>) which will turn the cards into a **CardCombo** and then try to turn that **CardCombo** into a **ComboEffect** that will be applied to the **Game**'s state. An **AiPlayer** can also play a *NOPE* card using onNopePlayed(player: Player) and the status of this effect will be stored in nopePlayed. It's planned for any **Player** to be able to play nope cards, not just **AiPlayer**s.

Then **GameState** has many methods and fields to keep track of the state. It has a list of players, both alive and exploded, and a queue of turns which holds the order of players' turns where multiple of the same **Player** means that they have to play more turns, i.e. when an attack card is played. It has a deck and discardPile but also an effectHistory that holds a sequence of **ComboEffect**s, for now used to determine how an attack card influences turns. It has a self-explanatory getCurrentPlayer() method and a getOpponents(), which returns only the alive players that aren't the current player.
There are also some methods that get called from **Game** and in **ComboEffect**s to change the game state, like endTurn(willDraw: bool) which will change the order of turns and, if willDraw is true (if the turn didn't end because of a skip or attack card being played), will draw a card that might remove a player from the game if they happen to die from an exploding kitten. Lastly there's the discard(player: Player, card: Card) method for moving a card from one of the players to the discardPile and the

[1] this is also why this class has the "`<<data type>>`" stereotype above it in the diagram

transferCard(from: Player, to: Player, card: Card) method used in e.g. a three card combo for moving a card between players.

As mentioned before, the **GameInitializer** class has the job of initializing the state of an **ExplodingKittens** game. To do that it uses initialize(ui: UserInterface, game: Game), which signals the ui to prompt the user to pick a number of players which then gets used to set everything up in the back. Later on we plan to also prompt the user to choose between using the standard deck or using a custom deck for which the user will have to provide a path to a JSON file.

Each **Player** in **GameState** can be either a **HumanPlayer**, an **AiPlayer** or later when we get to implementing networking, a **RemotePlayer**. All these **Players** are children of the **Player** parent class. The parent class holds a **String**-based id and a hand of **Card**s. The hand is a **List** and can have duplicates of one type of **Card**. To remove a card from this hand you can use extract(card: Card). Furthermore, there are abstract methods for certain actions, since the artificial players and human players do the corresponding actions differently. selectCardType() asks a **Player** to select a type of **Card**, either using an algorithm for the AI or through an interface for a human. selectPlayer(players: Collection<Player>) does the same for selecting a player out of a collection of players. selectCardOf(player: Player) does something similar to selectCardType() but from the hand of another **Player**. When an exploding kitten is drawn, the abstract method reinsertExplodingKitten(deckSize: int) is called to figure out where to insert it back. Lastly, if the player has an action they can veto, they get the chance to do so when askForNope(action: String) is called. A **CompletableFuture** is planned to be added as a parameter to askForNope so **HumanPlayers** can also use the *NOPE* card.

AiPlayers are managed by no user and their moves are instead automatic. For certain interactions it uses a game instance, e.g. playing a *NOPE* card. We plan to remove this game reference from the implementation once askForNope is properly implemented for all kinds of **Players**.

HumanPlayers are controlled by a local user and thus need a ui assigned so they can choose what to do in their turns and see what the other players have done.

RemotePlayers will not be present on the local computer and instead will be a reference to other players reachable via the network . In our vision, they will act as a proxy to **HumanPlayer**s hosted on other instances of our application.

The last currently implemented parts of the engine are the **ComboEffect** and the **CardCombo** class. The **CardCombo** class is a factory class for **ComboEffect**s. It's instantiated with a **List** of cards and then when the getEffect() method is called it's checked if those cards make a **ComboEffect**. If it does, the corresponding **ComboEffect** is given, if not, nothing is given.
The **ComboEffect** class is an interface used to represent card effects. It has a method called apply(state: GameState) which applies the effect represented by the **ComboEffect** to the given **GameState**.
FavorEffect causes a player to demand a card from an opponent. **TwoCardsEffect** causes a player to steal a card from an opponent. **ThreeCardsEffect** does the same, but the player can choose which type of card to steal and gets nothing if the opponent doesn't have that card. **FiveCardsEffect** causes a player to take a specific **Card** from the **DiscardPile**. **SeeTheFutureEffect** causes the top 3 cards of the **Deck** to be visible to the current player. **ShuffleEffect** shuffles the **Card**s in the **Deck**. **SkipEffect** ends the turn of the current player without them having to draw a **Card**. And lastly, **AttackEffect** causes a player to end all their turns, even if they have multiple left to go, without having to draw a **Card** and makes the next player play two more turns than the current player would still have had to play if they didn't play an **AttackEffect**.

Lastly we are planning on making a **Scoreboard** which would manage the scores of past games played on this machine. It would have a load() method to load the scoreboard from disk and an update(results: List<String>) method to update that scoreboard back on disk.

The UI

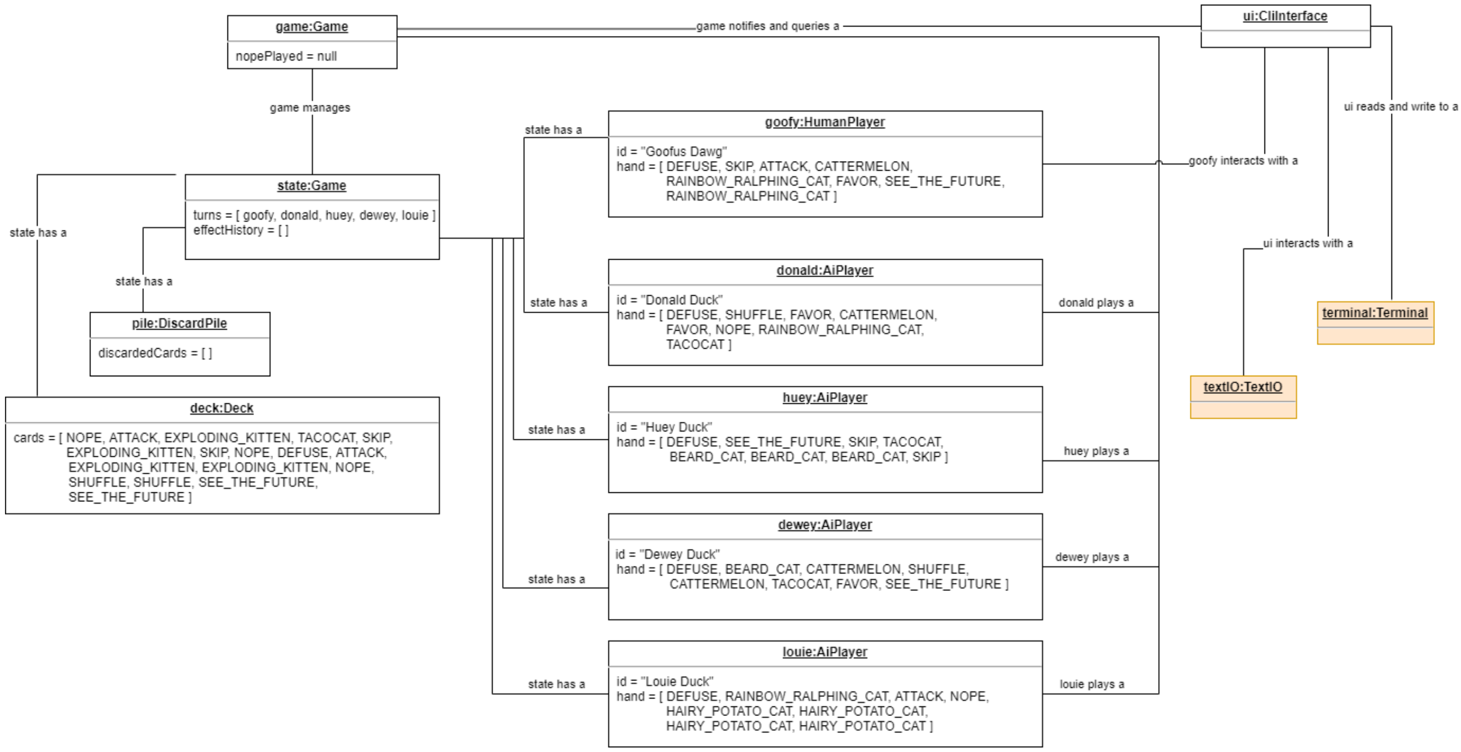
Lastly, we have a package for interfacing with the user. This package contains a generic interface **UserInterface** which is made up of methods to interact with the user, namely notify(event: **String**) operations, which outputs events towards the user and several queries, e.g., queryNumber(prompt: **String**, min: int, max: int), to prompt the user for input, possibly by specifying the expected return type and possible constraints on the input.
We foresee two different implementations for **UserInterface**: first, we implemented a command-line based interface **CliInterface** which interacts with the user by means of a reference to a terminal. This class uses the TextIO library to implement the methods of the interface. Then, we kept our architecture open to possibly implement a **GdxInterface** class, that is, a rich graphical interface which extends LibGdx's **ApplicationAdapter** parent class. By overriding methods of the parent class, instances of **GdxInterface** can, for example, render a screen with cards depicted as images.

Network

Since we're not completely sure on how to implement networking, we wanted to leave our architecture open to accommodate future changes. This is why we decided to encapsulate the logic to communicate over the network inside a Network class, which exposes an interface to publish an event to the other instances of our program.

Object diagram

Kevin Koeks, Jelena Masic



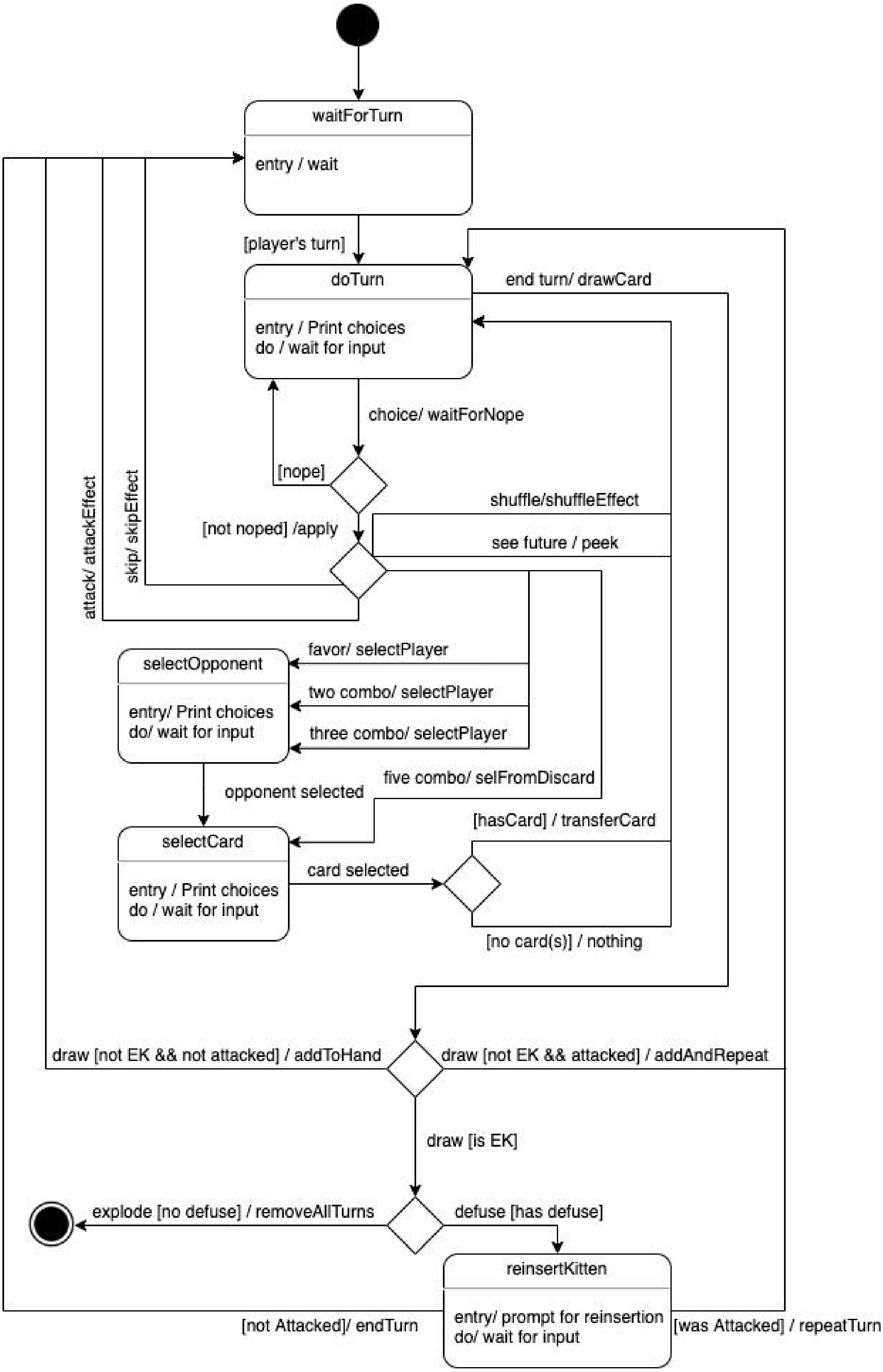
This section contains the description of a "snapshot" of the initial state of a local match. With the UML object diagram we will represent the system in a state in which no cards have been played yet. This diagram will help with the understanding on how some of the different classes interact with each other. We will describe the objects in our system in the following points:

- *game*: The *game* object regulates access to the game state. The <u>nopePlayed</u> attribute keeps track whether a nope-card has been used by a player against an effect.
- *state*: The *state* object is responsible for holding the state of the game. The <u>turns</u> attribute will be responsible for keeping track of whose turn it is to play next after the current player is done playing his/her turn. The <u>effectHistory</u> attribute will be holding the different types of effects that have been played in the game.
- *ui*: The *ui* object will make it possible for the players to interact with the game in an easy and clear manner (e.g., via a CLI) such that the current player knows what action one can take, with the current cards that one possesses in hand.
- *deck*: The *deck* object will be holding all the cards that players can draw to finish their turn. The <u>cards</u> attribute holds at this "snapshot" all the cards that are left after each player has received their even amount of cards to start the match with.
- *pile*: The *pile* object will be available for players to discard their used cards and for players that can access it with the "5-card-combo" effect. The <u>discardedCards</u> attribute is responsible for holding all the cards that have been played throughout the game.
- *goofy*: The *goofy* object represents the human player in this "snapshot". The <u>id</u> attribute will hold the name of the player. The <u>hand</u> attribute will hold at the beginning of the game 8 types of cards that belong to the player, which can increase or reduce throughout the game until the player "explodes" or wins the game. Furthermore, at the start of the game the <u>hand</u> will always contain at least one *DEFUSE* card.
- *donald, huey, dewey, louie*: The *donald*, *huey*, *dewey* and *louie* objects represent the artificial intelligent (AI) players. The AI-players also have the same types of attributes as the human player (*goofy* object), which are <u>id</u> and <u>hand</u>, because they inherit these attributes from the same abstract superclass (**Player**).

State machine diagrams

Federico Giaj Levra

Card effects



Introduction

In this state machine diagram, we represented the effects which the cards have on the game. During the game, the player will wait for its turn and, when the player's turn comes, a transition is fired in order to change the state from waitForTurn to doTurn.
In the doTurn state, the user will be immediately prompted with the choices which can be made, and the system will wait for the input; there are two main possibilities, draw or play a card.
In order to avoid cluttering the diagram with more redundant transitions, the individual cards have been omitted before the nope choice, as every one of the possible played cards can be noped except for the defuse, which is only played when a EK is drawn, and not directly from the hand.

Playing a card

We would like to note that discarding cards after making the choice (first decision node after doTurn) is implicit for each of the moves.
At this moment other players can play the *NOPE* card in order to cancel any effect of the card and force the user to return to the initial doTurn state.
If the card is not noped, then the actual effects are applied to the **GameState**. The second decision node models all the possible choices of cards and combos:

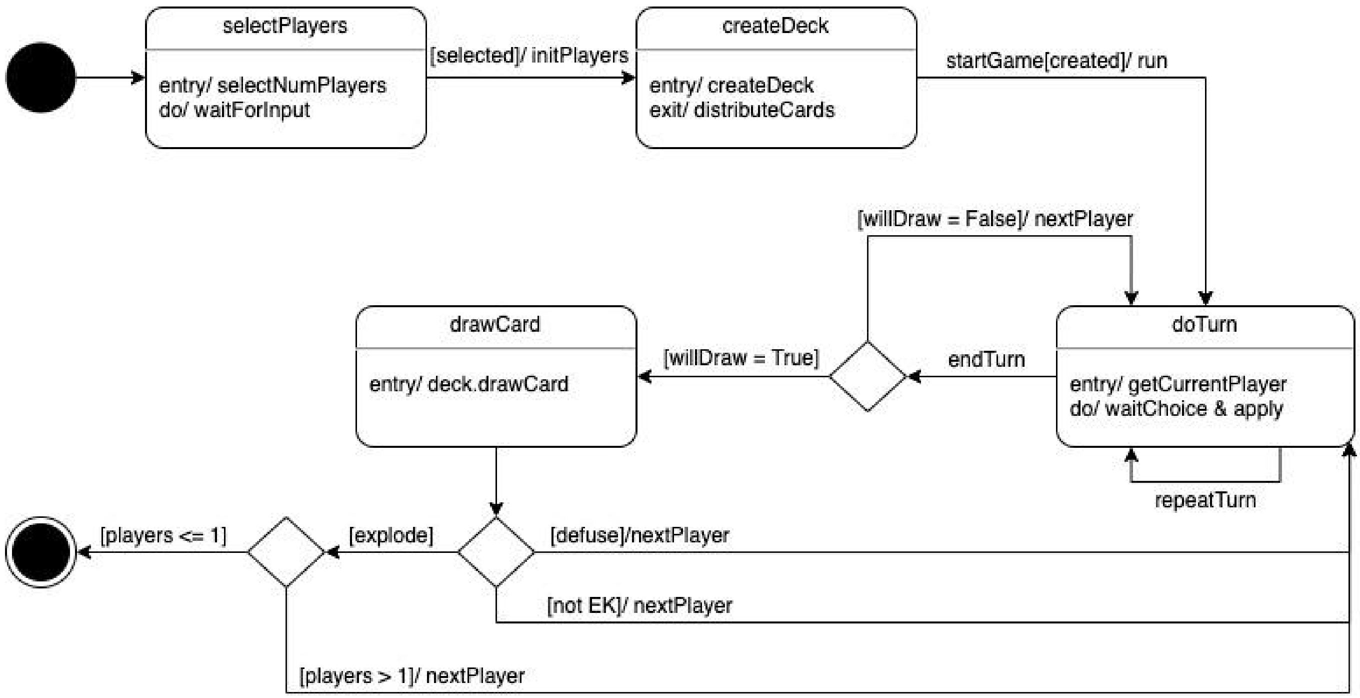
- For some of these (attack, skip, shuffle, see the future) the modelling is straightforward. Each of these choices trigger the related action, and then they either change the state to waitForTurn (attack, skip) or to doTurn (shuffle, see the future).
- Other selections require additional steps in order to affect the state of the game. They will be described in the following paragraph.

The two states (selectOpponent, selectCard) prompt the user with a list of choices and the system waits for an input. After inputting the requested card, if it is present it is transferred, if it's not then nothing happens. Ultimately both the transitions will return to the doTurn state.
It is important to note that, we are describing a system in which the user is Human, therefore the interaction is through prompts, AiPlayers are able to make the choice without the prompt.

Drawing a card

If the user decides to end the turn, it will be forced to draw a card. In this case a player might draw an exploding kitten. To model this without adding other decision nodes, we simply added a second condition to the guards. If the player was attacked its next state will be doTurn, if not it will be waitForTurn. In case it draws an exploding kitten, then more processing is required to determine the next player's state.
Once the exploding kitten is drawn, our system automatically plays a defuse card for the player, if it is present in its hand. Therefore, if it does not have one, the player will explode, and all the turns in that player's hand will be discarded. This is a terminal state for the player.
On the other hand, if the player has a defuse card, then it will be used to save the player from exploding and the user will be prompted to reinsert the exploding kitten in the deck. Once it has chosen where to insert it the following state is determined by evaluating the guards, if the player was attacked, it will repeat the turn, if not it will be reinserted in the queue.

Game class



Introduction

This diagram is a representation of the **Game** class, with some added states which are not strictly of this class. This is done because they are mostly blocking states, which are required in order for the game to progress.
Validation of the inputs has been omitted, as each of the inputs is controlled from the methods defined in the **UserInterface**.

Game initialization

In the current version of the game, when it starts, the amount of players has to be defined by the only **HumanPlayer**. Once a value is inputted, the **GameInitializer** class will take care of creating the deck and distributing the cards to each of the players. While doing this a game state is created and other important elements of the system are added, such as the linked list which holds the turns and the discard pile.

Game start and execution

Once the game is successfully created, from the doTurn state, the current player will be extracted from the turns list, and the **Game** waits for either one or more cards to play, or for a command to end the turn. The **AiPlayer** and the **HumanPlayer** have different ways of making said choices, but the **Game class** does not worry about this distinction. Once the choice is made, and the effect is applied, there are two possibilities, either the player ends its turn or it continues it (*maybe because one played a card or maybe because of an attack effect*).
If the player decides to end the turn, the state of the **Game** will change based on the willDraw guard. This will determine if the state has to transition to the drawCard state, or if it has to go back to doTurn and switch to the next player. The latter repeats the process just described, while the former enters the next state, in which the user draws a card from the deck.

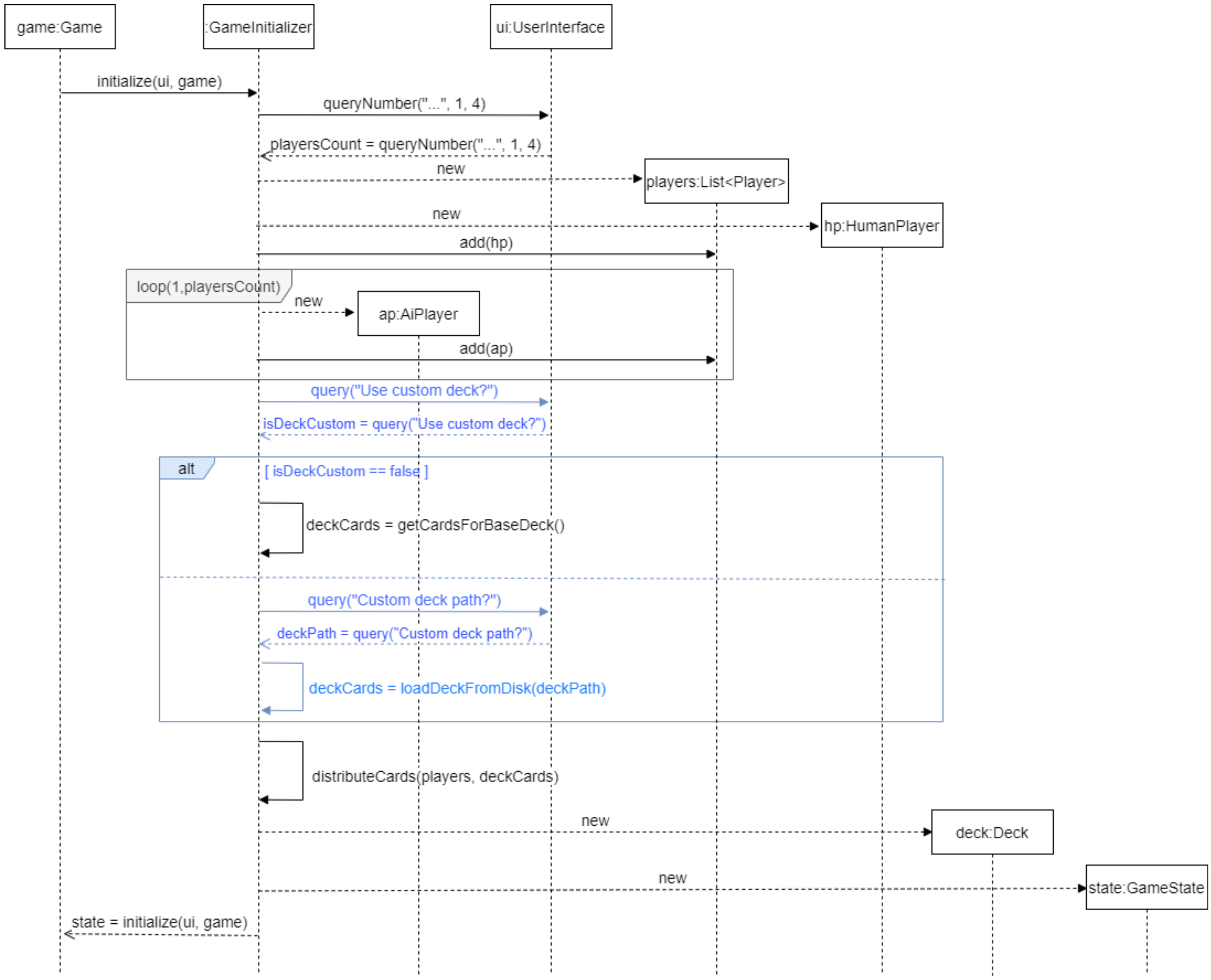
Terminal state

Drawing a card can have three outcomes, of which two have the same effect. When an exploding kitten is drawn and the player has a defuse card, the player is still alive, and the current player is switched with the next. The same happens if a non-exploding kitten card is extracted from the deck. The last case happens when an exploding kitten is drawn but there is no possibility of defusing it. In this case the player explodes, and if the number of remaining players is equal to 1, the transition will bring the state to a terminal state. On the other hand, if the remaining players are more than 1 then the player is switched and the state transitions to doTurn.

Sequence diagrams

Kevin Koeks, Jelena Masic

Local Game Initialization



The initialization of a **Game** instance is depicted in the above sequence diagram. More specifically, the above describes what happens during the construction of a *game* instance.
Game, as a class, is responsible for managing the lifecycle of Exploding Kittens matches. This is why, in its constructor, it calls a method of **GameInitializer** to get the initial state of a match.

Players Creation

First, the user is prompted to select a number of opponents between 1 and 4. After correctly selecting a number inside this range, players are added to a list. The first player which is added to a list is a **HumanPlayer**, that corresponds to the user who entered the number of opponents. Then, we used a `loop`-fragment which ranges from 1-to-*playersCount* to generate the requested amount of opponents. Each opponent is generated as an **AiPlayer** and then added to *players*, which is the list of players that will be eventually used for the **GameState**.

Deck composition

Then, our diagram features a prescriptive section, where the user is prompted to choose to play with a custom deck. Since we haven't implemented this feature yet, the corresponding parts in the diagrams are colored in blue. After a response from the user (Y/N, which is translated to a boolean value), we decided to employ an `alt` fragment to model a conditional branching in the flow.
If the user doesn't want to use a custom deck, then the base (standard) deck is used. This part is colored in black since we already implemented it and in our current program it will happen automatically (hence why the `alt`-fragment is also prescriptive).
If the user wants instead to use a custom deck, then one is prompted with the path of the deck on the filesystem. After receiving the *deckPath* as an input, the **GameInitializer** calls a method to load the list of cards from a file.

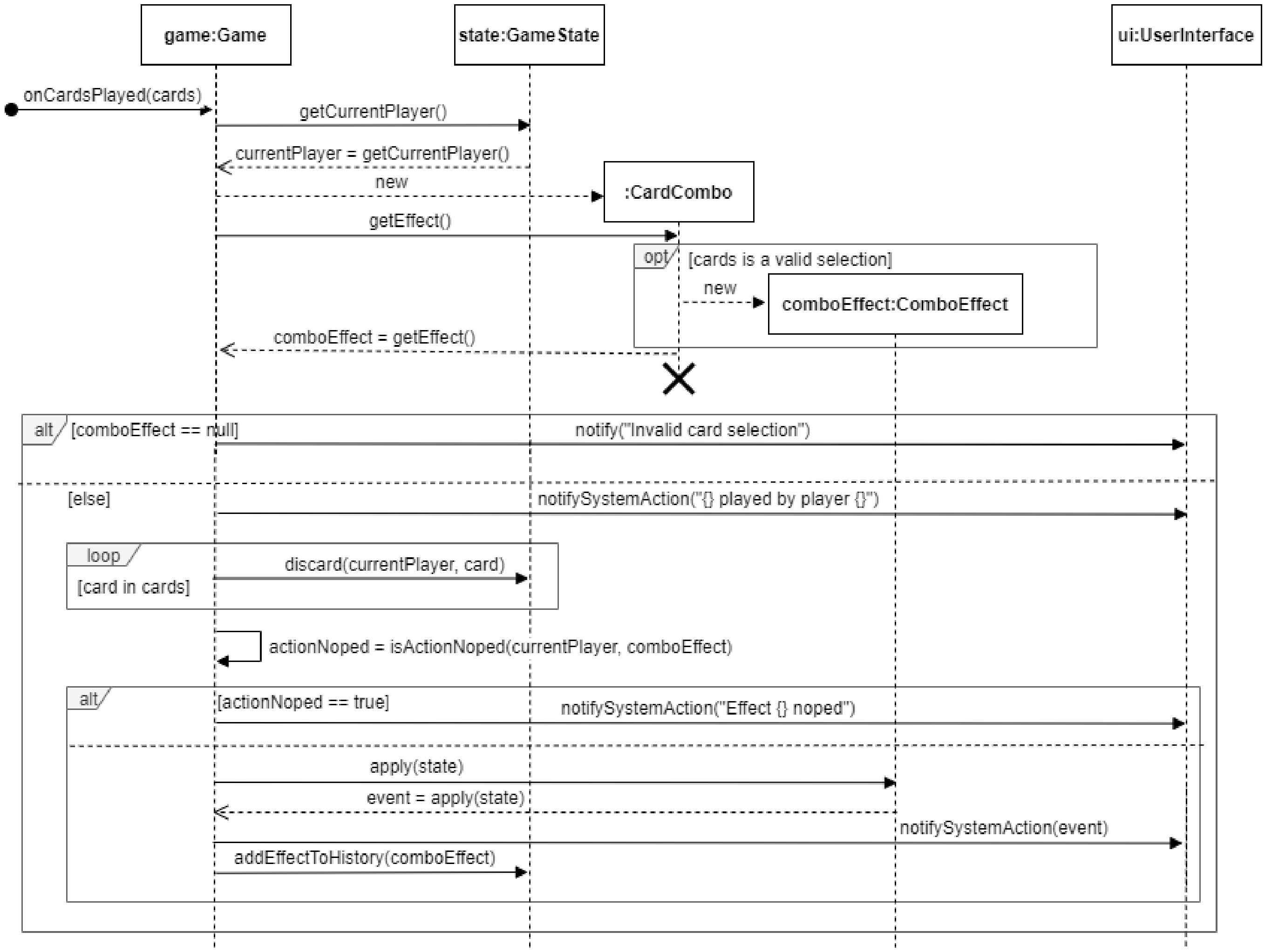
Dealing Cards

After the list of cards for the deck *deckCards* has been decided, they are distributed among the players. This operation happens at a lower level of abstraction, so we decided to exclude it from our sequence diagram. However, by distributing the cards to the players we mean applying the procedure described in the introduction of Assignment 1 and, while this is being applied, remove from *deckCards* all the cards being handed out.

State Instantiation

Finally, the *deck* instance is created with the cards which were not handed out, an instance of **GameState** is created with the list of *players* and the *deck*, and then the newly created state is returned to the game.

Playing a Combination of Cards



In this sequence diagram, we modelled the flow of execution for playing a combination of cards. First of all, the *game* instance determines who is the current player, that is, the player who is playing the selection of cards called "*cards*" in the diagram.

Transforming Card Selections into Effects

Then, a **CardCombo** is instantiated with *cards* to compute the effect of the given card selection. After the instantiation, getEffect() is called to possibly return a **ComboEffect** object based on the values inside *cards*. Since we wanted to keep this diagram at a high-level, we decided to represent the switch statement as an `opt` fragment that may or may not return an object. The logic for deciding whether or not "*cards*" results in a valid combination of **Card**s resides in **CardCombo**, hence we opted for a high-level guard such as "*cards* is a valid selection".

At this point, we used an `alt` fragment to represent an if-else decision. If the cards selection resulted in an invalid combination, e.g. playing *DEFUSE* or playing *TACOCAT* and *SKIP* together as a combo, *comboEffect* is null[2], the user is notified via the interface and the whole flow terminates without discarding the cards from the user's hand or applying any effect.

If *"cards*" is instead a valid combination (e.g., *SKIP* played alone), then a message is printed to the user interface to notify that a given selection of cards is being played, and by which player.

---

[2] to have a null-safe implementation, we decided to use Java's **Optional** type for storing *comboEffect*. Hence, when we say *comboEffect* is null, we actually mean *comboEffect* is equal to *Optional.empty()*

Discarding Played Cards

Since *cards* as a list of **Card**s ends up in the discard pile regardless of the action being countered by a *NOPE*, we immediately discard *cards* from the *currentPlayer*'s hand. To represent this, we used a `loop` fragment which iterates over all cards. Since the exit condition of this `loop` was not intuitive to express with the identifiers already present in the diagram, we decided to mark the fragment as “`loop foreach`” and to put a guard to express the iterative pattern of the foreach, that is, “`[ card in cards ]`”.

Waiting for a NOPE and Applying an Effect

After discarding *cards*, all active players are prompted for vetoing the action by playing a *NOPE*. The logic for this operation is encapsulated inside the isActionNoped method and, since it expresses logic at a different abstraction level, we decided not to expand it in this diagram.
The method isActionNoped returns a boolean value, which is used in an if-else construct represented with an `alt` fragment. If the action is “noped”, i.e., the previous invocation returned true, then all users are notified that the effect which was about to be played has been “noped”. If instead the action is not “noped”, the effect is applied by means of the apply() method, all the users are notified of the event caused by *comboEffect* being played, and finally the effect is added to the effectHistory in the **GameState** instance.

Notes

As a final remark, we put termination for the instance which is not “long-living”, that is, an anonymous **CardCombo**. Differently from *game*, *state*, and *ui*, this object serves its purpose only during the timespan of the execution of this flow, whereas *comboEffect* might be stored inside the effectHistory of *state* in case the action is not “noped”.

Implementation

Jelena Masic

UML To Code

Since we split the features vertically and none of us had much prior experience in designing software, it was difficult for us to start composing a complete UML diagram from scratch.
This is why we implemented prototypes for each of our respective features and then iteratively built up and refined UML diagrams which reflected our design.
Therefore, trying to adhere to the agile methodology of this project, we worked in short iterations by showcasing our ideas with minimal working examples and then translating them into UML in order to discuss them among us.
This approach allowed us to work quite vertically on our respective features, yet providing us a way to concretely explain why we would require a specific API/operation from another component present in another feature.
For example, while implementing prototypes for the effects for card combinations (that is, the **ComboEffect** hierarchy), we noticed which were the methods we needed in other classes such as **GameState** or **Player**.
Then, we collectively discussed our ideas and we were able to make progress consistently and steadily over the three weeks.

Implementation Details

The core value which drove our design and implementation is flexibility: we tried to identify the moving parts of our system and encapsulate their logic into separate classes under a common interface in order to enable extensibility for the future. The most interesting implementation challenges we faced are:

- GUI: in Assignment 1, we planned to develop a graphical user interface for a rich gaming experience. Nevertheless, we wanted to deliver a working demo with a user interface for the Assignment 2 submission without giving up the chance of implementing a graphical interface later on. This is why we extracted some common operations in a generic interface called **UserInterface** with methods like <u>notify(event)</u>, so that we are free to decide whether to replace the current CLI-based implementation with a rich graphical user interface for the final delivery. At the same time, this design allows us to abstract away the specific view we are using for our system
- MVC: even though we may not have adhered strictly to this architectural pattern, we used it as a soft guideline to divide our software into packages, where <u>ui</u> contains the View, <u>engine</u> contains the Controller (whose core logic is inside the **Game** class), and finally <u>model</u> contains classes which represent some passive entities of our system.
- Deck extensibility: we wanted to design our system around a lightweight system of cards and effects. We achieved this by:
    1. using an enumeration **Card** for the cards, which brought several benefits: no need to handle the life-cycle of an instance, type-safety instead of string literals being passed around the system, much clarity in the resulting code
    2. having all the card effects implementing a common **ComboEffect** interface and delegating the responsibility of their instantiation to a dedicated **CardCombo** class. This means that, if we want to add a new effect in the future, we only need to define its behaviour (as long as it implements the **ComboEffect** interface) and to handle its creation in **CardCombo**
- “Noping” effects: we wanted to have a unique and future-proof way to veto an action from a player. This is why we added a mutable field to the **Game** class which has type **CompletableFuture**. By doing this, **AiPlayer**s can (possibly asynchronously) “nope” an action

and, prescriptively, human players can asynchronously "nope" actions both locally and (possibly) over the network

- Initialization: while implementing **Game** and **GameState**, we started seeing these two classes growing further and further. When we noticed it, we pulled out the initialization logic to a dedicated **GameInitializer** class, which will also allow us (prescriptively) to isolate the logic for custom decks in a proper place
- Turns: we decided to have two data structures to store players in the **GameState**. This choice paid off since, by implementing turns as a **Deque**, we can add turns by simply putting more times the same **Player** instance and this resulted in an easier implementation for **ComboEffect**s such as **Skip** or **Attack**
- Effect history: by representing card effects with a common interface, it becomes possible to store them in a stack[3] inside the **GameState**. Thanks to this, we were able to implement a relatively complex set of rules when playing multiple *ATTACK*s (instance of **Card**), while keeping the *apply* method logic relatively simple
- Testing our system: since we implemented first the classes for our model and controller, we wanted to have an automated way to gain confidence about our system and prove among ourselves some properties. This is why, for some of our classes, we implemented unit tests, which were also useful to, later on, draw UML diagrams

Run Instructions

The runnable main for our project is located in the **nl.vu.group2.kittens.ExplodingKittens** class, that is in the "`src/main/java/nl/vu/group2/kittens/`" folder from the root of the repository.
When running the application from IntelliJ, logs are redirected to an "`exploding-kittens.log`" file in the root of the repository.
To run the application with IntelliJ, there is a small guide in the `README.md` file on how to set up Lombok as an annotation processor for this project.
The fat JAR we built to run our system directly is instead located at "`out/artifacts/software_design_vu_2020_jar/software-design-vu-2020.jar`" and takes no arguments to be run, hence one can simply run our game with the following:

```
$ java -cp software-design-vu-2020.jar nl.vu.group2.kittens.ExplodingKittens
```

Demo
demo_ek_2.mov

[3] the stack is actually implemented as a **Deque** since it is more flexible in this way

Time logs

Link to Time Table

Team number	2			
Member		**Activity**	**Week number**	**Hours**
All		Group meeting, discussing about class diagrams.	3	1.5
All		Documentation: Address feedback from Assignment 1	3	2
Jelena Masic		Implementation: skeleton and card/effects prototypes	3	10
Jelena Masic		UML: Class diagram	3	2.5
Federico Giaj Levra		Document changes for Assignment 1	3	1
Federico Giaj Levra		UML: Class diagram	3	3
All		Group meeting	4	1.5
Sebastian Fredrik		Trying to figure out if Skin Composer/Scene Composer will be useful	4	3
Sebastian Fredrik		Setting up LibGDX	4	2
Jelena Masic		Implementation: card, effects, API definition of classes interfacing with effects	4	10
Jelena Masic		UML: Class Diagram, Object Diagram	4	4
Kevin Koeks		Start coding deck (1/4), try to understand other coded part of project	4	4
Kevin Koeks		Made a draft version of object diagram	4	3
Kevin Koeks		Finished coding deck & discardDeck	4	3
Federico Giaj Levra		Implementation: Game, GameState, Player	4	12
Federico Giaj Levra		UML: Class Diagram, State machine diagram	4	3
All		Group meeting, divide other diagrams parts amoung each other	5	2
All		Documentation: Address feedback from Assignment 1	5	4
Jelena Masic		Implementation: integrate branches, Sonarlint issues, warnings, game initializer	5	10
Jelena Masic		UML: Class Diagram, Object Diagram, Sequence Diagrams	5	5
Jelena Masic		Documentation: Implementation, Class Diagram, Sequence Diagrams	5	9
Kevin Koeks		Tried to make sequance diagram based on class diagram	5	3
Kevin Koeks		Draft sequence diagram (initialization part)	5	4
Kevin Koeks		Added object diagram description	5	3
Federico Giaj Levra		Implementation: HumanPlayer, AiPlayer, CLI	5	15
Federico Giaj Levra		UML: State machine diagram 2	5	3

Federico Giaj Levra		Document for assignment 2	5	6
Sebastian Fredrik		Implementation: prototypes for GUI (**not** merged to Assignment2 branch)	5	4
Sebastian Fredrik		Documentation: Class Diagram	5	5
			TOTAL:	133