# Assignment 3

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
- Important changes from Assignment 2 are colored in blue

For the diagrams, we use the following colouring conventions:

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

### Diagrams

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

# Summary of changes of Assignment 2

*Jelena Masic*

## Pull Request / Code

- "*This piece of code and above seems* a bit of *repetition*" [malco96, Edward2k, Fouad-AJ, eas840]
    - We did improve the code were possible (**CardCombo**) and put comments for ignoring the code style on test classes
- "*Add some comments to the code*" [malco96, Fouad-AJ, AbramovMA, eas840]
    - We tried to put more comments to explain functionality, without exceeding common sense
- "*Shouldn't it return some feedback to the player about what happened?*" [malco96, AbramovMA]
    - We tried to improve timing and messages for a better gaming experience
- "*I do not fully understand the [...] division of the Game and GameState classes*" [AbramovMA]
    - There is now a cleaner separation between the two classes
- "*Give your methods good names*" [AbramovMA]
    - We addressed small technical debt here and there based on the comments

## Documentation

- "*Object diagram [...] it would be good if you would describe the pros/cons [...]*" [TA]
    - We expanded the section with a few more comments on the design repercussions

## Application of design patterns

*Jelena Masic*

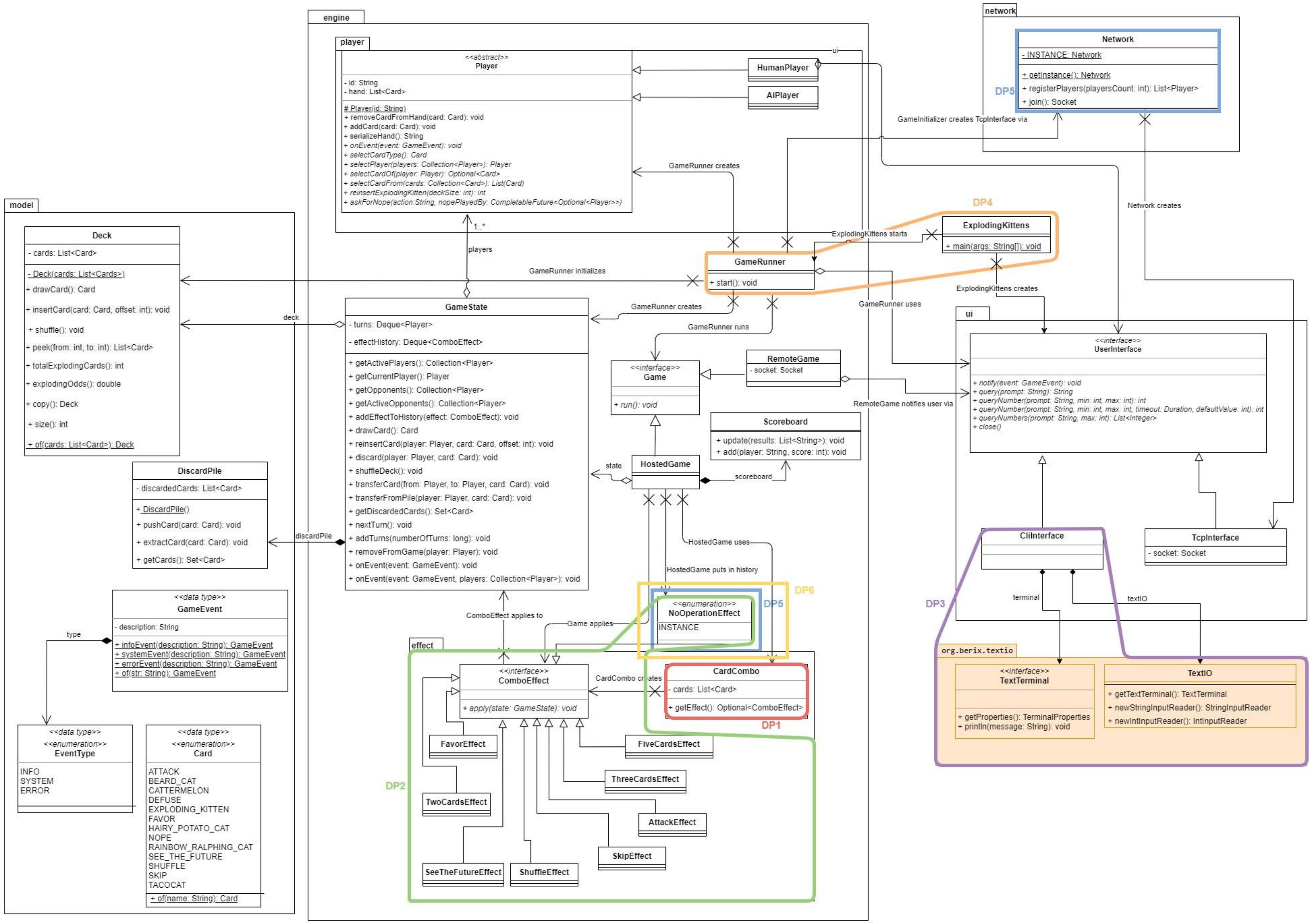
Below is the class diagram of our system, with some shapes drawn on top of it.

Each of the shapes corresponds to a design pattern, except for one case where two shapes correspond to the same pattern (DP5).

Design patterns can be distinguished by their colours and numerical abbreviation.

# Application of design patterns

*Jelena Masic*



engine
player
<<abstract>> Player
- id: String
- hand: List<Card>
# Player(id: String)
+ removeCardFromHand(card: Card): void
+ addCard(card: Card): void
+ serializeHand(): String
+ onEvent(event: GameEvent): void
+ 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, nopePlayedBy: CompletableFuture<Optional<Player>>)
HumanPlayer
AiPlayer
ui
GameRunner creates
1..*
players
network
Network
- INSTANCE: Network
+ getInstance(): Network
+ registerPlayers(playersCount: int): List<Player>
+ join(): Socket
DP5
GameInitializer creates TcpInterface via
Network creates
DP4
ExplodingKittens
+ main(args: String[]): void
ExplodingKittens starts
GameRunner
+ start(): void
ExplodingKittens creates
GameRunner uses
GameRunner initializes
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>
+ totalExplodingCards(): int
+ explodingOdds(): double
+ copy(): Deck
+ size(): int
+ of(cards: List<Card>): Deck
deck
GameState
- turns: Deque<Player>
- effectHistory: Deque<ComboEffect>
+ getActivePlayers(): Collection<Player>
+ getCurrentPlayer(): Player
+ getOpponents(): Collection<Player>
+ getActiveOpponents(): Collection<Player>
+ addEffectToHistory(effect: ComboEffect): void
+ drawCard(): Card
+ reinsertCard(player: Player, card: Card, offset: int): 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>
+ nextTurn(): void
+ addTurns(numberOfTurns: long): void
+ removeFromGame(player: Player): void
+ onEvent(event: GameEvent): void
+ onEvent(event: GameEvent, players: Collection<Player>): void
GameRunner creates
GameRunner runs
<<interface>> Game
+ run(): void
RemoteGame
- socket: Socket
RemoteGame notifies user via
Scoreboard
+ update(results: List<String>): void
+ add(player: String, score: int): void
HostedGame
state
scoreboard
HostedGame uses
HostedGame puts in history
<<interface>> UserInterface
+ notify(event: GameEvent): void
+ query(prompt: String): String
+ queryNumber(prompt: String, min: int, max: int): int
+ queryNumber(prompt: String, min: int, max: int, timeout: Duration, defaultValue: int): int
+ queryNumbers(prompt: String, max: int): List<Integer>
+ close()
CliInterface
TcpInterface
- socket: Socket
DP3
terminal
textIO
org.berix.textio
<<interface>> TextTerminal
+ getProperties(): TerminalProperties
+ println(message: String): void
TextIO
+ getTextTerminal(): TextTerminal
+ newStringInputReader(): StringInputReader
+ newIntInputReader(): IntInputReader
DiscardPile
- discardedCards: List<Card>
+ DiscardPile()
+ pushCard(card: Card): void
+ extractCard(card: Card): void
+ getCards(): Set<Card>
discardPile
<<data type>> GameEvent
- description: String
+ infoEvent(description: String): GameEvent
+ systemEvent(description: String): GameEvent
+ errorEvent(description: String): GameEvent
+ of(str: String): GameEvent
type
<<data type>> <<enumeration>> EventType
INFO
SYSTEM
ERROR
<<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
ComboEffect applies to
Game applies
effect
DP6
DP5
<<enumeration>> NoOperationEffect
INSTANCE
<<interface>> ComboEffect
+ apply(state: GameState): void
CardCombo creates
CardCombo
- cards: List<Card>
+ getEffect(): Optional<ComboEffect>
DP1
DP2
FavorEffect
FiveCardsEffect
ThreeCardsEffect
TwoCardsEffect
AttackEffect
SkipEffect
SeeTheFutureEffect
ShuffleEffect

| | **DP1** |
|---|---|
| **Design pattern** | Factory Method |
| **Problem** | The logic to decide which combo effect should be applied is non-trivial and we noticed that inserting it directly in the **Game** class was bloating its responsibilities with no or little cohesive value.<br>At the same time, our design should be extensible enough to accommodate new cards and corresponding effects, but we would like to have **Game** stable, i.e., not affected by such changes in the requirements, as much as possible since it contains the core of our system's logic. |
| **Solution** | We isolated the logic to instantiate a **ComboEffect** inside a **CardCombo** class. This class is responsible for creating the correct combo effect to be applied given a list of cards. Therefore, our **Game** class became much lighter and alleviated from this responsibility. As an additional benefit, Game is not aware of any concrete implementation of **ComboEffect**. |
| **Intended use** | This class is intended to be used by **Game** to possibly get an effect.<br>In case the combination of **Card**s is invalid, **CardCombo** returns *Optional.empty()* for null-safety.<br>See also the sequence diagram "Playing a Combination of Cards". |
| **Constraints** | N/A |
| **Additional remarks** | None |

| | **DP2** |
|---|---|
| **Design pattern** | Command |
| **Problem** | In the ExplodingKittens game, there is a relatively high number of effects which can be applied after playing a combination of cards.<br>At the same time, the game designers themselves wanted to keep the game open for the addition of new cards and effects, but closed for the basic set of rules.<br>Therefore, we would like to have the same properties for our design, thus avoiding to bloat classes like **Game** with too much logic to handle effects. |
| **Solution** | We isolated the logic to apply an effect of a combination of cards in classes which implement a **ComboEffect** interface.<br>This interface exposes only one method, which is basically the logic to implement the effect applied to the **GameState**.<br>This resulted in removing further logic from the **Game** class and neatly organizing the very same logic in several classes.<br>Finally, we are able also to keep a history of which effects were applied, since they are now encapsulated in distinct classes. |
| **Intended use** | This class is intended to be used by **Game** to apply an effect to the **GameState**.<br>If one wants to add a new effect to our program, it just needs to provide an additional implementation of **ComboEffect**.<br>See also the sequence diagram "Playing a Combination of Cards". |
| **Constraints** | **GameState** should offer an API suitable for applying the desired effect. |

| | |
|---|---|
| **Additional remarks** | This class brought us to add DP1 to our design and resulted in many classes in the effect package.<br>At the same time, we were able to design unit tests for effects pretty easily. |

| | **DP3** |
|---|---|
| **Design pattern** | (Object) Adapter |
| **Problem** | We wanted to keep our program open for the addition of new kinds of UIs.<br>As a result of that, we added the **UserInterface** interface to our design.<br>Then, when we wanted to implement an UI based on a CLI, we chose TextIO as a library.<br>This library (clearly) doesn't implement the interface present in our code, therefore it wasn't possible to immediately plug it into our application. |
| **Solution** | A small class which implements the **UserInterface** interface and embodies references to TextIO components (**TextIO** and **TextTerminal**).<br>This "**CliInterface**" class forwards the method calls of the **UserInterface** to the TextIO components by means of some glue logic. |
| **Intended use** | The **CliInterface** class is instantiated at the start of an Exploding Kittens game and passed around as a **UserInterface** reference, so that the clients of this class can use the TextIO library transparently. |
| **Constraints** | N/A |
| **Additional remarks** | In case in the future we would like to use another library for a CLI-based interface of our game, changes would be required only inside the **CliInterface** class. |

| | **DP4** |
|---|---|
| **Design pattern** | Façade |
| **Problem** | When bootstrapping our application, we wished for an easy-to-understand main method.<br>Yet, in our early structures, the main method would need to be aware of some classes inside the engine package. |
| **Solution** | Have a class which acts as an entrypoint for the whole game, that is, **GameRunner**.<br>This class is responsible for initializing and then running an Exploding Kittens match. |
| **Intended use** | From the main method, instantiate a **GameRunner** by passing the **UserInterface** (i.e., by injecting this dependency) which will be used throughout this game and then call start() on it. |
| **Constraints** | Some classes, for example **HostedGame**, have been given package-level visibility for a cleaner design and making the application of this design pattern more explicit.<br>Nevertheless, this could be a limitation in case in the future somebody would like to run a match in a different manner and use those classes directly from the outside. |
| **Additional remarks** | By applying this design, it was slightly easier to implement the networking bonus feature. |

| | **DP5** |
|---|---|
| **Design pattern** | Singleton |
| **Problem** | In the current implementation, we first wanted to have a centralized instance that manages inbound and outbound connectivity.<br>Later on, in case the networking functionality will be improved, we want to have a single class managing the network sockets.<br>In other parts of the system, we wanted instead to (minimally) improve the memory footprint and make the code a bit more readable. |
| **Solution** | We applied the singleton pattern in two places.<br>First, **Network** is a singleton so that later on we can apply performance improvements as needed. As a side note, in the current implementation we didn't notice performance degradation or a pressing need to do this, hence we didn't act on it.<br>Secondly, **NoOperationEffect** (inner class of **HostedGame**) is an enum-based singleton for better readability and (minor) performance gains. |
| **Intended use** | **Network** is used whenever a user wants to host or join a game and its instance is intended to be accessed via the getInstance() method.<br>**NoOperationEffect** is a special **ComboEffect** which is used as a dummy placeholder inside the effectHistory. In this case, the instance is accessed via its name "*INSTANCE*". |
| **Constraints** | In the future, we might want to swap out the current **Network** implementation and replace it with a different logic.<br>When using singletons, this is slightly hard to achieve because often (also in our case) the reference to the specific instance is statically determined. |
| **Additional remarks** | None |

| | **DP6** |
|---|---|
| **Design pattern** | Null Object |
| **Problem** | Null-pointers have been historically a source of headaches and bugs.<br>In our effect history, we would like to keep track of "no effect" being applied when a player ends its turn because the application of **ComboEffect**s depends on this.<br>For example, all the remaining turns of a player are added to the next player only if the **AttackEffect** is played immediately after another **AttackEffect**.<br>Instead, in the case a player draws a card before replying with another **AttackEffect**, then the next player should have only 2 turns.<br>Therefore, we would like to have an object in our effect history which represents "no operation", but we want to remain null-safe. |
| **Solution** | A class **NoOperationEffect** which implements the **ComboEffect** interface, with an empty body for the apply() method. |
| **Intended use** | When a player finishes his/her turn by drawing a card, then this effect should be placed on top of the effectHistory. |
| **Constraints** | N/A |
| **Additional remarks** | This could have been achieved with a simple lambda as well, but then it would be hard to perform equality checks if needed in the future. |

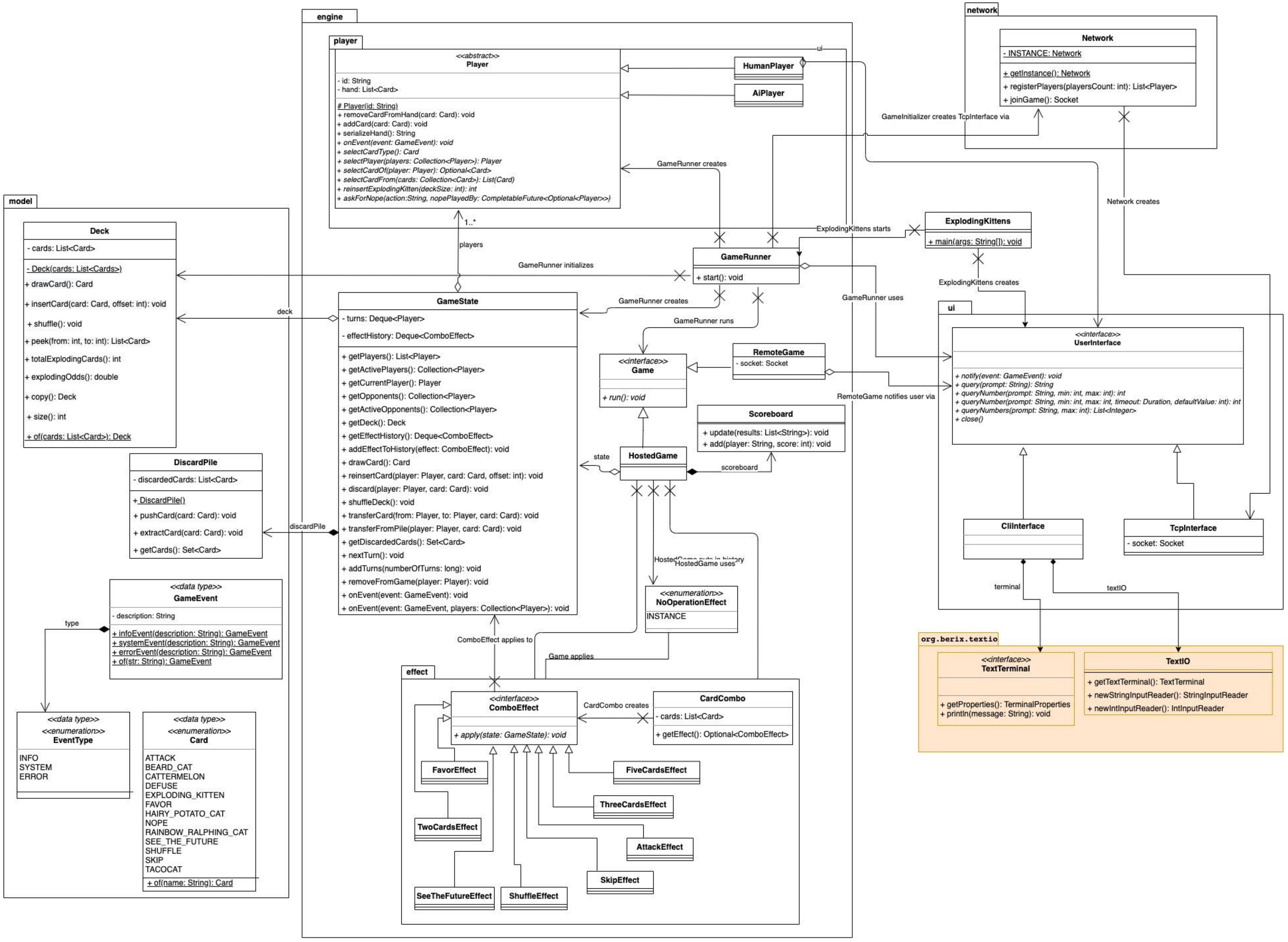
## Class diagram

*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. In the current implementation, this type of interface is a command line interface, or **CliInterface** object, and drives the user interface of the game. The **ExplodingKittens** class also creates a **GameRunner** object passing it a reference to the aforementioned ui, so that it can bootstrap a **Game**, which in turn will be built on top of the model classes.
The model holds specifications for the deck that you draw cards from, the pile where you discard your cards on, the cards themselves, and the game events that an interface may receive.
The engine package includes what the players (both humans and AI) can do, 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)
+ removeCardFromHand(card: Card): void
+ addCard(card: Card): void
+ serializeHand(): String
+ onEvent(event: GameEvent): void
+ 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, nopePlayedBy: CompletableFuture<Optional<Player>>)
HumanPlayer
AiPlayer
ui
GameRunner creates
1..*
players
GameRunner
+ start(): void
ExplodingKittens starts
ExplodingKittens
+ main(args: String[]): void
ExplodingKittens creates
GameRunner initializes
GameRunner creates
GameRunner runs
GameRunner uses
GameState
- turns: Deque<Player>
- effectHistory: Deque<ComboEffect>
+ getPlayers(): List<Player>
+ getActivePlayers(): Collection<Player>
+ getCurrentPlayer(): Player
+ getOpponents(): Collection<Player>
+ getActiveOpponents(): Collection<Player>
+ getDeck(): Deck
+ getEffectHistory(): Deque<ComboEffect>
+ addEffectToHistory(effect: ComboEffect): void
+ drawCard(): Card
+ reinsertCard(player: Player, card: Card, offset: int): 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>
+ nextTurn(): void
+ addTurns(numberOfTurns: long): void
+ removeFromGame(player: Player): void
+ onEvent(event: GameEvent): void
+ onEvent(event: GameEvent, players: Collection<Player>): void
<<interface>>
Game
+ run(): void
RemoteGame
- socket: Socket
RemoteGame notifies user via
Scoreboard
+ update(results: List<String>): void
+ add(player: String, score: int): void
state
HostedGame
scoreboard
HostedGame puts in history
HostedGame uses
<<enumeration>>
NoOperationEffect
INSTANCE
ComboEffect applies to
Game applies
effect
<<interface>>
ComboEffect
+ apply(state: GameState): void
CardCombo creates
CardCombo
- cards: List<Card>
+ getEffect(): Optional<ComboEffect>
FavorEffect
FiveCardsEffect
ThreeCardsEffect
TwoCardsEffect
AttackEffect
SkipEffect
SeeTheFutureEffect
ShuffleEffect
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>
+ totalExplodingCards(): int
+ explodingOdds(): double
+ 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>>
GameEvent
- description: String
+ infoEvent(description: String): GameEvent
+ systemEvent(description: String): GameEvent
+ errorEvent(description: String): GameEvent
+ of(str: String): GameEvent
type
<<data type>>
<<enumeration>>
EventType
INFO
SYSTEM
ERROR
<<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
network
Network
- INSTANCE: Network
+ getInstance(): Network
+ registerPlayers(playersCount: int): List<Player>
+ joinGame(): Socket
GameInitializer creates TcpInterface via
Network creates
ui
<<interface>>
UserInterface
+ notify(event: GameEvent): void
+ query(prompt: String): String
+ queryNumber(prompt: String, min: int, max: int): int
+ queryNumber(prompt: String, min: int, max: int, timeout: Duration, defaultValue: int): int
+ queryNumbers(prompt: String, max: int): List<Integer>
+ close()
CliInterface
TcpInterface
- socket: Socket
terminal
textIO
org.berix.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 a factory method of(name: String) 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.

In the model package we now find the **EventType** and **GameEvent** classes as well. **EventType** corresponds to the possible types of event we identified in our system (*INFO*, *SYSTEM*, *ERROR*) and **GameEvent** is a type-safe wrapper for a pair <**String**, **EventType**> that is sent from the game engine towards the **UserInterface**. Since these classes are widely used in our system and are little more than plain old Java objects, we marked them as a (non-primitive) type too and omitted the arrows towards them in the final class diagram.

Then, cards are held in two different types of collections, both built using a list of **Card**s, 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. Depending on what the user chooses after the start of the program, either a **HostedGame** or a **RemoteGame** gets instantiated.
The **ExplodingKittens**' main() function starts a match by using a **GameRunner**. **Game**s are indeed instantiated by this class, which interacts with the ui to get input from the user on what type of game one wants to play (local vs. remote, host or join, choose players number, select deck from a JSON file, ...).
After initializing a **Game**, the **GameRunner** runs it by invoking the run() method on it.
If a player is hosting a match, then **GameRunner** returns an instance of **HostedGame** that 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 a player plays a card or selection of cards, it will get handled by the internal method onCardsPlayed(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 **HostedGame**'s state.

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

In case of an event, a **HostedGame** instance will notify the "interested" **Player**s via the onEvent() methods of **GameState**.
When playing an effect, a **HostedGame** will loop on the other **Player**s to ask whether they want to "nope" with the askForNope(action: String, nopePlayedBy: CompletableFuture<Optional<Player>>). If a **Player** wants indeed to nope, then the **CompletableFuture** passed as a parameter will be completed with the reference to itself.
The **HostedGame**'s run() method will stop only after one player is left, then the **ExplodingKittens** main() method will prompt the user for a rematch.
If a user instead joins a game over the network, a **RemoteGame** will be instantiated by the **GameRunner**. This class has a simple behaviour, as it continuously waits for input from a socket until it receives a message which identifies the end of a match. When this class receives a message over the network, it "decodes" it and then interacts with the ui to handle it as a notification event or as a query.

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 nextTurn(), which will trigger the end of the current turn, or drawCard(). Lastly there's the discard(player: Player, card: Card) method for moving a card from one of the players to the discardPile and the transferCard(from: Player, to: Player, card: Card) method used in e.g. a three-card combo for moving a card between players.
Finally, **GameState** has also a couple of overloaded methods onEvent() which were introduced to remove the dependency between **HostedGame** and **UserInterface**. These methods iteratively notify players of a given event happening.

Each **Player** in **GameState** can be either a **HumanPlayer** or an **AiPlayer**. All these **Player**s 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 removeCardFromHand(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 a ui 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, nopePlayedBy: CompletableFuture<Optional<Player>>) is called as described above.

**AiPlayer**s are managed by no user and their moves are instead automatic. For better gaming experience, we introduced a delay to have a "real-life" effect when this kind of user plays. Furthermore, most of their choices are random, with a small degree of reasoning employed to choose which cards to play.

**HumanPlayer**s are controlled by a user and therefore they need a ui, so they can choose what to do in their turns and see what the other players have done. Differently from what we anticipated in Assignment

2, **HumanPlayers** are also remote players. In this case, the only difference lies in the dynamic type of UserInterface they use (**CliInterface** vs. **TcpInterface**).

The last currently implemented parts of the engine are the **ComboEffect** and the **CardCombo** class. The **CardCombo** class is a factory class for **ComboEffects**. 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 **Cards** 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**.
Finally, a special mention goes to **NoOperationEffect**, a type of effect defined inside the **HostedGame** class and used as a placeholder (null object) for the end of the turns.

Lastly, **HostedGame**, which holds the reference to the current **GameState** regardless of having a local or remote game, has a **Scoreboard**. During the course of a match, every time a player loses is added to the scoreboard. Then, at the end of a match, the results are dumped into a JSON file with the update() method and **HostedGame** sends the scoreboard as an output to all the **Players**.

## The UI

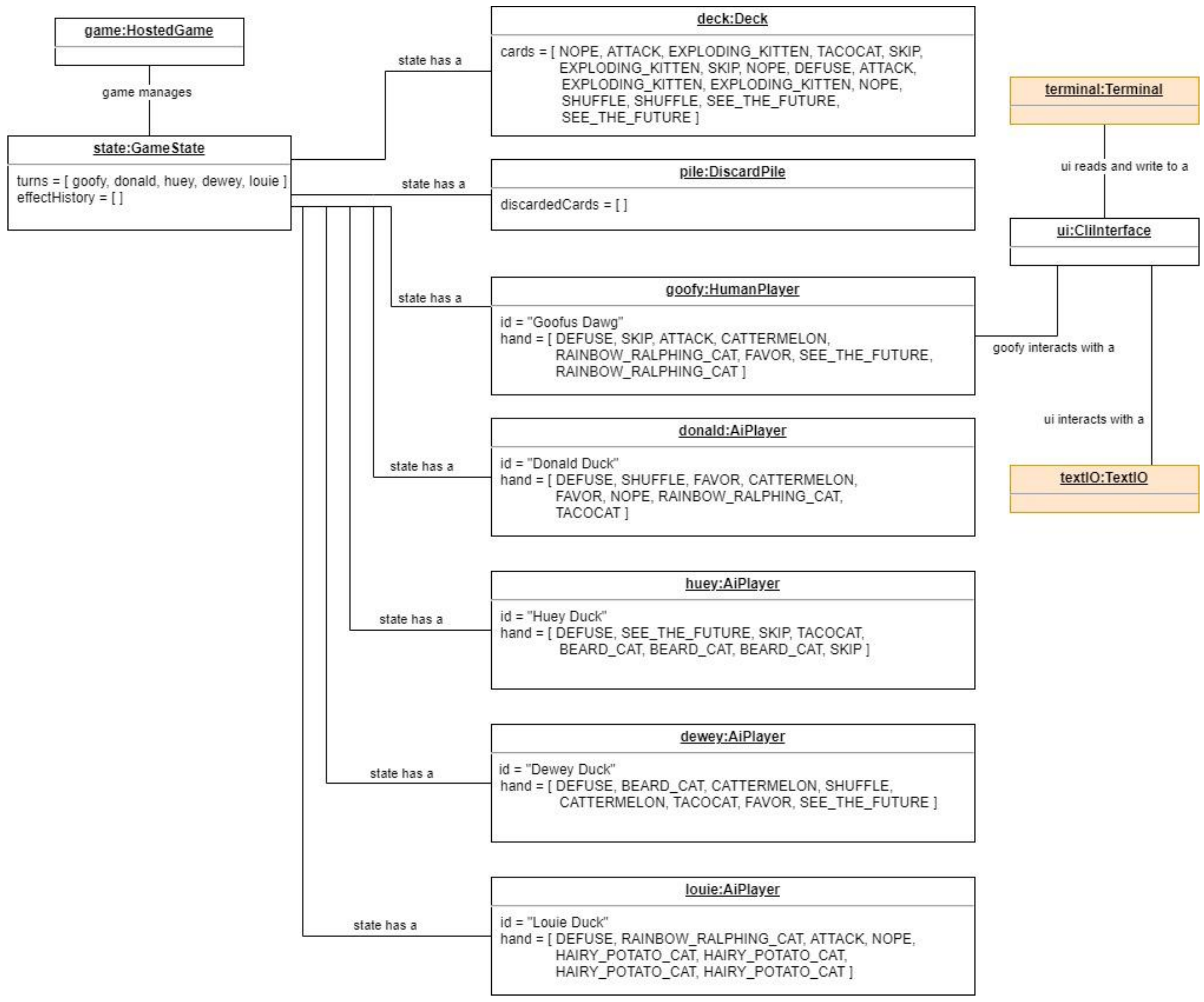
Lastly, we have a package for interfacing with the user. This package contains a generic interface **UserInterface** which is made up of methods to interact with the user, namely notify(event: **String**) operations, which outputs events towards the user and several queries, e.g., queryNumber(prompt: **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 have 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.
Secondly, we have a **TcpInterface** which interacts with the network by means of a socket. Here, we have communication over the network to either write to the socket or read from it.

## Network

The network package is used either to host or join a networked match. It has a single class, **Network**, which is a singleton that manages the inbound and outbound connections.
Then, a user can host a match via the registerPlayers(playersCount: int) method that returns the list of **Players** who joined over the network.
Alternatively, a user joins a match via the join() method, which returns a Java **Socket** to the server hosting the match.

## Object diagram

*Kevin Koeks, Jelena Masic*


game:HostedGame
game manages
state:GameState
turns = [ goofy, donald, huey, dewey, louie ]
effectHistory = [ ]
state has a
deck:Deck
cards = [ NOPE, ATTACK, EXPLODING_KITTEN, TACOCAT, SKIP, EXPLODING_KITTEN, SKIP, NOPE, DEFUSE, ATTACK, EXPLODING_KITTEN, EXPLODING_KITTEN, NOPE, SHUFFLE, SHUFFLE, SEE_THE_FUTURE, SEE_THE_FUTURE ]
state has a
pile:DiscardPile
discardedCards = [ ]
terminal:Terminal
ui reads and write to a
ui:CliInterface
state has a
goofy:HumanPlayer
id = "Goofus Dawg"
hand = [ DEFUSE, SKIP, ATTACK, CATTERMELON, RAINBOW_RALPHING_CAT, FAVOR, SEE_THE_FUTURE, RAINBOW_RALPHING_CAT ]
goofy interacts with a
ui interacts with a
textIO:TextIO
state has a
donald:AiPlayer
id = "Donald Duck"
hand = [ DEFUSE, SHUFFLE, FAVOR, CATTERMELON, FAVOR, NOPE, RAINBOW_RALPHING_CAT, TACOCAT ]
state has a
huey:AiPlayer
id = "Huey Duck"
hand = [ DEFUSE, SEE_THE_FUTURE, SKIP, TACOCAT, BEARD_CAT, BEARD_CAT, BEARD_CAT, SKIP ]
state has a
dewey:AiPlayer
id = "Dewey Duck"
hand = [ DEFUSE, BEARD_CAT, CATTERMELON, SHUFFLE, CATTERMELON, TACOCAT, FAVOR, SEE_THE_FUTURE ]
state has a
louie:AiPlayer
id = "Louie Duck"
hand = [ DEFUSE, RAINBOW_RALPHING_CAT, ATTACK, NOPE, HAIRY_POTATO_CAT, HAIRY_POTATO_CAT, HAIRY_POTATO_CAT, HAIRY_POTATO_CAT ]

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 manages the game state and the actions a **Player** has to take during his/her turn.
- *state*: The *state* object is responsible for holding the state of the game. The turns 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 effectHistory 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 (in this scenario, 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 cards 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 discardedCards attribute is responsible for

holding all the cards that have been played throughout the game. At the beginning of the game, the *pile* is empty.

- *goofy*: The *goofy* object represents the human player in this “snapshot”. The id attribute will hold the name of the player. The hand 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 hand 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*), which are id and hand, because they inherit these attributes from the same abstract superclass (**Player**).

In Assignment 2, we got (as a feedback) that we could elaborate a bit more on the pros and cons of the design we decided to go with. This is what we changed compared to Assignment 2:

- **AiPlayer**s don’t have a reference to *game* anymore
- *game* doesn’t have a reference to the **UserInterface** anymore
- nopePlayed was removed from **Game**

These are the “pros” of our choices:

- The changes led to a more clear and easy-to-understand design
- By keeping the reference to the **UserInterface** in **HumanPlayer**s, it was easier to implement networking
- The removal of the reference to *game* in the **AiPlayer**s allowed us to get rid of a (indirect) circular dependency
- By putting the list of **Player**s inside **GameState**, it’s easier to write cohesive code inside **Game** and give it responsibility to orchestrate the logic flow of a match

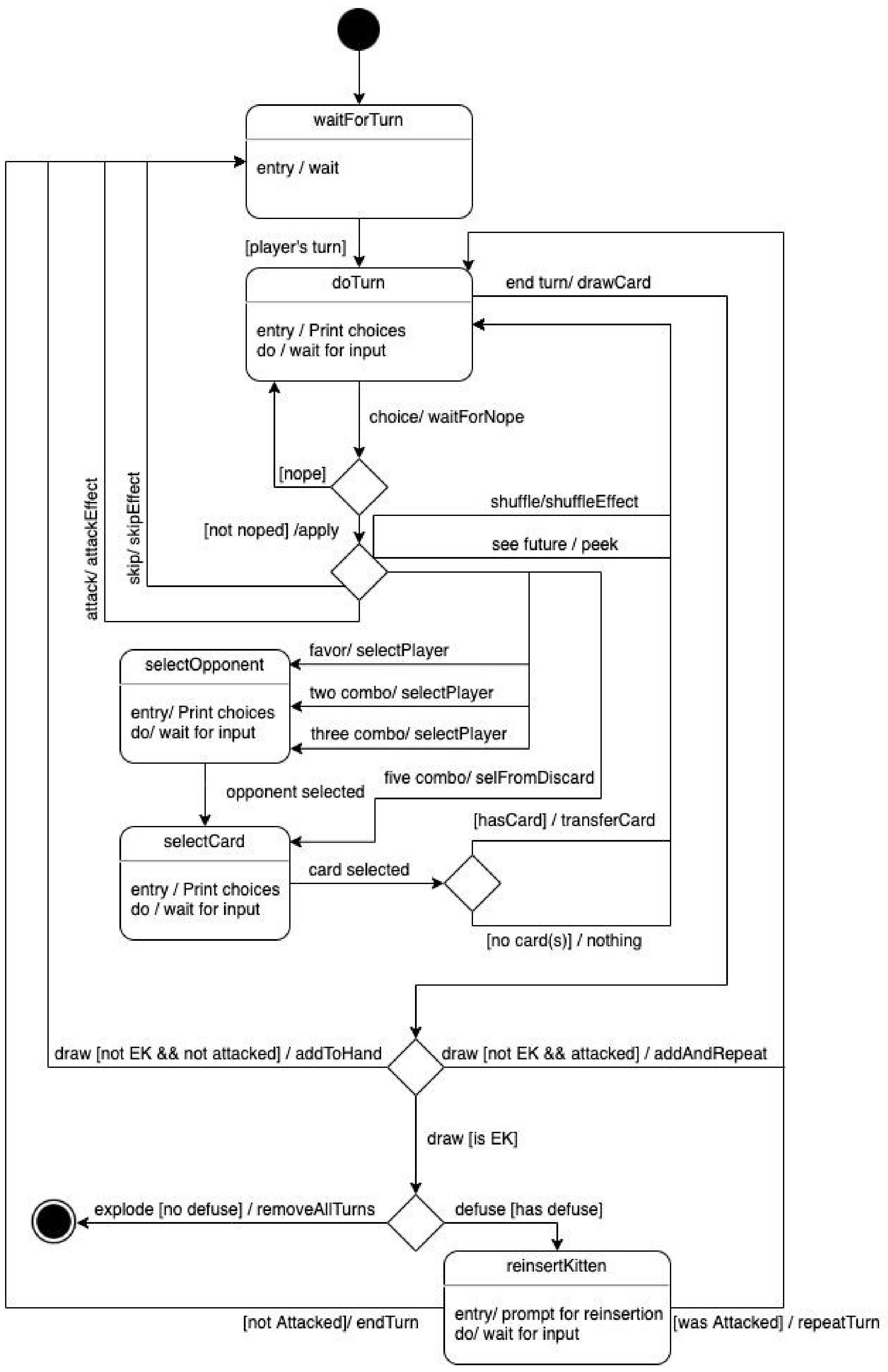
These are the “cons” of our choices:

- There are two lists of **Player**s inside **GameState**, which leads to better encapsulation at the expense of a slightly increased complexity in managing turns

## 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:

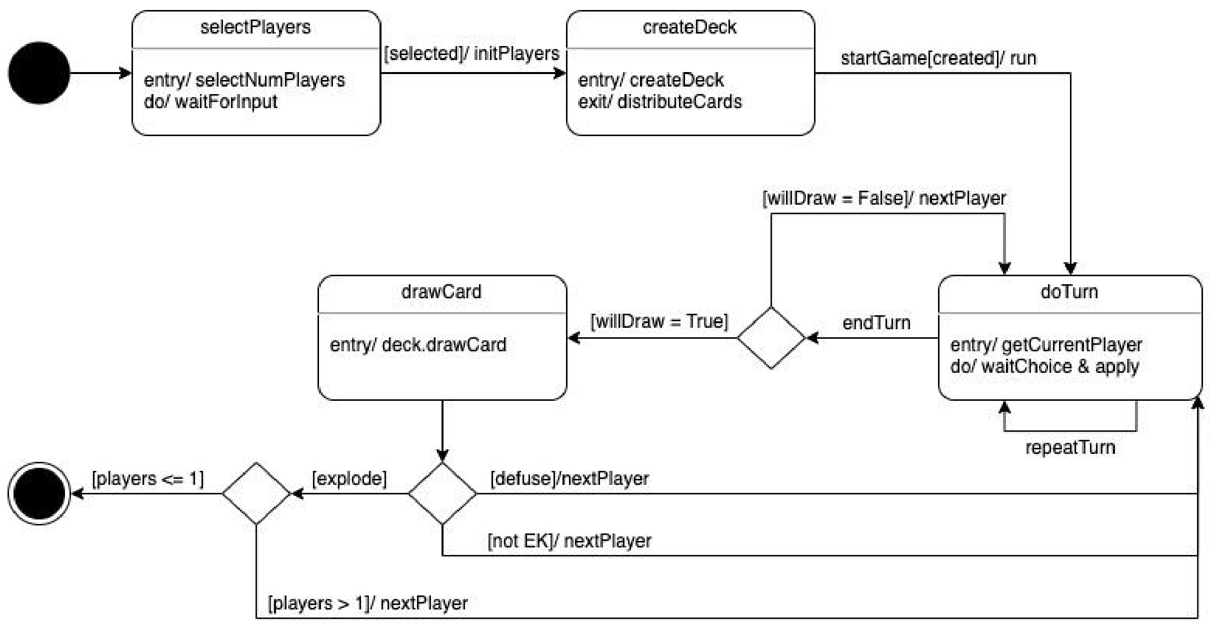
- For some of these (attack, skip, shuffle, see the future) the modelling is straightforward. Each of these choices trigger the related action, and then they either change the state to waitForTurn (attack, skip) or to doTurn (shuffle, see the future).
- Other selections require additional steps in order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

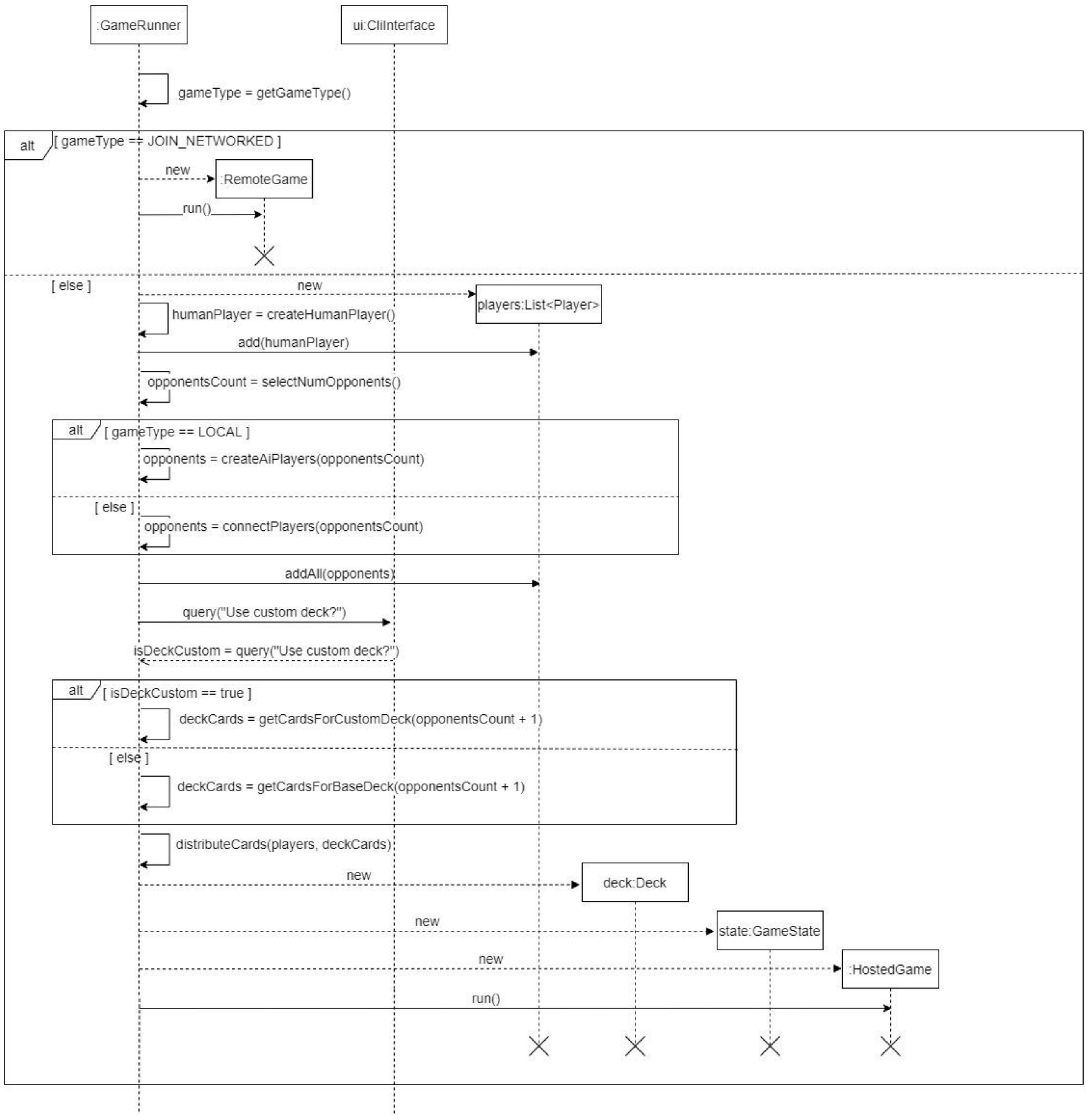
### Terminal state

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*

### Game Initialization


:GameRunner
ui:CliInterface
gameType = getGameType()
alt
[ gameType == JOIN_NETWORKED ]
new
:RemoteGame
run()
[ else ]
new
players:List<Player>
humanPlayer = createHumanPlayer()
add(humanPlayer)
opponentsCount = selectNumOpponents()
alt
[ gameType == LOCAL ]
opponents = createAiPlayers(opponentsCount)
[ else ]
opponents = connectPlayers(opponentsCount)
addAll(opponents)
query("Use custom deck?")
isDeckCustom = query("Use custom deck?")
alt
[ isDeckCustom == true ]
deckCards = getCardsForCustomDeck(opponentsCount + 1)
[ else ]
deckCards = getCardsForBaseDeck(opponentsCount + 1)
distributeCards(players, deckCards)
new
deck:Deck
new
state:GameState
new
:HostedGame
run()

The initialization process inside a **GameRunner** instance is depicted in the above sequence diagram. More specifically, the above describes what happens during the construction of a **Game** instance. **GameRunner**, as a class, is responsible for initializing an instance of a game and in the picture above we can see the steps taken to achieve this goal.

## Game Type Selection

Firstly, **GameRunner** asks the user what kind of match one wants to play: this happens in the method getGameType(). The possible outcomes of this method invocation are three: *LOCAL*, *HOST_NETWORKED*, and *JOIN_NETWORKED*.
*LOCAL* matches happen solely on the user machine, with the human player competing against **AiPlayer**s. Then, *HOST_NETWORKED* matches are games in which the user awaits for other players to join, and then this application instance manages the match. Finally, *JOIN_NETWORKED* matches are games in which the user joins a hosted match.

## Joining a Networked Match

In case the user chooses to join a networked match, then the **GameRunner** just needs to create and run a **RemoteGame** instance. Inside this concrete implementation of **Game**, there is all the logic to join a hosted match and we didn't expand this logic further in the sequence diagram in order to keep the abstraction level consistent.
We modeled this with an `alt` fragment since there is a clear logic split between the case in which a user joins a match over the network or decides to host one.

## Players Creation

If the user is not joining a match over the network, then the match will be hosted on his application instance (please notice that this also applies to *LOCAL* matches). Then, a **HumanPlayer** is always created (this corresponds to the user playing on this instance).
After this, there is a method to determine the number of opponents and, depending on whether the game is local or not, there is different logic to instantiate the opponents.
In the code, we implemented it with a `switch` statement for safety, but it could have just been an if-else.
In the sequence diagram, we decided to keep things simple and understandable, therefore we modeled it with an `alt` fragment as if we implemented it with an if-else statement.
If the game is *LOCAL*, then the opponents creation is straightforward as **GameRunner** just needs to create a given number of **AiPlayer**s. Otherwise, if the game is *HOST_NETWORKED*, then there is a method which hangs the **GameRunner** until the desired number of opponents have joined the match over the network.
Regardless of the way opponents are created, they and the **HumanPlayer** are added to the *players* list.

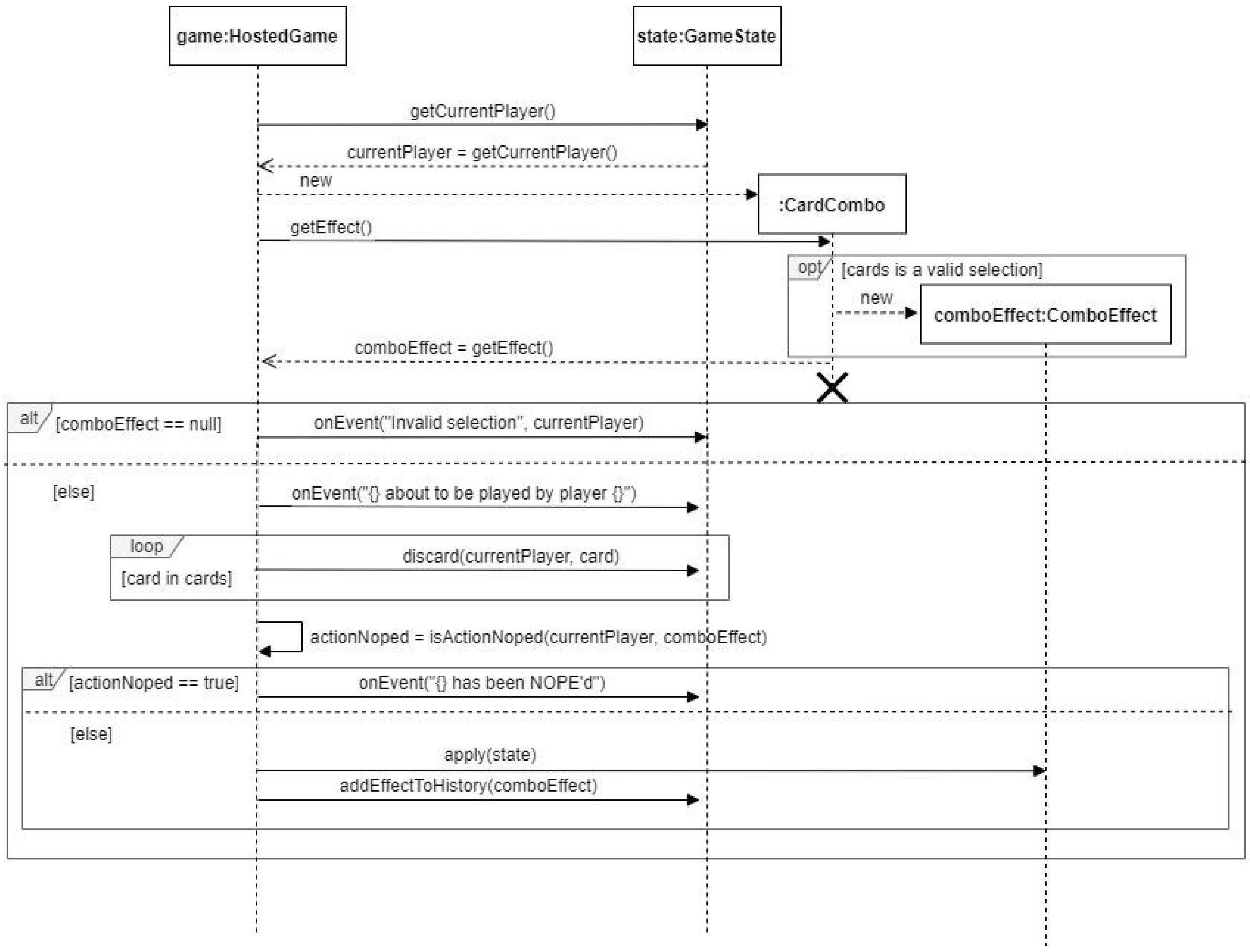
## Deck Composition

After all players joined the match, the user hosting the game is prompted to choose whether to use a standard deck or a custom one.
Since this choice is binary (Y/N), we modeled how the *deckCards* are determined with an `alt` fragment.
If the user wants to use a custom deck, then this is loaded from a JSON file through the invocation of the getCardsForCustomDeck method. Alternatively, the base deck of cards will be used.

## Creating a HostedGame

Finally, after the *players* and the *deckCards* are determined for a hosted match, the cards are handed out with the players via the distributeCards method. Then, a **Deck** is created, which is used to create a **GameState**, which is used to create and run a **HostedGame**.
The instances of **RemoteGame**, **Deck**, **GameState**, **HostedGame**, and *players* cease to exist as soon as a match is completed and therefore we indicated this with a termination for the respective lifelines.

## Playing a Combination of Cards


game:HostedGame
state:GameState
getCurrentPlayer()
currentPlayer = getCurrentPlayer()
new
:CardCombo
getEffect()
opt
[cards is a valid selection]
new
comboEffect:ComboEffect
comboEffect = getEffect()
alt
[comboEffect == null]
onEvent("Invalid selection", currentPlayer)
[else]
onEvent("{} about to be played by player {}")
loop
[card in cards]
discard(currentPlayer, card)
actionNoped = isActionNoped(currentPlayer, comboEffect)
alt
[actionNoped == true]
onEvent("{} has been NOPE'd")
[else]
apply(state)
addEffectToHistory(comboEffect)

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.

[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()*

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 about to be played, and by which player.

### 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 and finally the effect is added to the effectHistory in the **GameState** instance.

### Notes

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".
Furthermore, some minor things changed in this diagram:

- Since the reference to the **UserInterface** was removed from **HostedGame**, events are sent to the UI via the onEvent method of **GameState**[3] and the **UserInterface** object was removed from the diagram as it became non-relevant
- **ComboEffect**s send themselves their outcome to the respective players by interacting with **GameState** via the onEvent(...) methods instead of returning something to print as result of the apply() invocation. This is not shown in the sequence diagram since this logic varies from effect to effect

[3] The inner iteration on the players for onEvent(...) has been omitted for the diagram because it would have introduced quite some complexity for little gain in understandability

# Implementation

*Jelena Masic*

## UML To Code

For the third part of this project, we basically continued as we did until Assignment 2.
Nevertheless, there was little work left to do to finalize the project, that is, implementing the bonus features and addressing the comments from the pull request.
Therefore, we implemented prototypes for the bonus features and then iteratively built up and refined UML diagrams which reflected our design. When doing so, we found out that some of our early design choices were an obstacle towards the implementation of the bonus features and therefore we adapted our system where needed.
Again, this approach allowed us to work quite vertically on bonus features and occasionally synchronizing among us to decide how to proceed with the implementation.
Thanks to the above, we arrived at the final submission with all bonus and non-bonus features implemented and with consistent UML diagrams.

## Implementation Details

What we wrote for Assignment 2 still holds, since we didn't have to put much effort to implement the rest of the system compared to our previous submission. This can be somehow considered a good indicator of the flexibility of our system. Nevertheless, we faced interesting implementation challenges also in these weeks.

### Network

In Assignment 2, we sketched a design for the networking part, but we later found out that it wasn't fully usable and clean from a design perspective. This is why we tried to investigate alternative ways to integrate networking in our application.
One option we considered is Java RMI (Remote Method Invocation): this API, present in the standard JDK distributions, allows developers to transparently perform invocations on objects over the network.
After some prototyping, though, we decided to discard RMI because of some drawbacks:

(1) it wasn't straightforward to integrate within our system;
(2) it required us to start a local RMI registry, which would have made starting our application a bit cumbersome; and
(3) we were not sure that non-Java applications would have connected easily to our program.

Therefore, we turned ourselves to raw TCP sockets wrapped in some classes for better encapsulation and, in this case:

(1) it was relatively easy to integrate it in our system;
(2) starting our application doesn't require additional commands; and
(3) we were able to play our game both with our own program, but also with `netcat` (the CLI tool).

### UserInterface

After our decision on the networking design, we noticed that the operations we were planning to implement for that did somehow fit the interface we already had for the CLI or the GUI. At the end of the day, it was all about I/O (read/write) with an external connection. Therefore, we made our **UserInterface** more generic in order to easily integrate the **TcpInterface** and then we provided a socket-based implementation for it.

Part of the change was switching to “**GameEvent**” for the notifyX(...) methods we used to have in order to lighten up the API of **UserInterface**.

### Notification System

When supporting networked games, we noticed that we needed to send events over the network as well. We had some options to do it (e.g., apply the Decorator or Composite patterns on the **UserInterface**) but then we resorted to a slightly altered version of the Observer pattern.
We channeled all the notifications via **GameState** and provided an overloaded API “onEvent(...)” to notify some or all **Player**s. In this way, the **Game** didn’t have to know whether the match was networked or not and we can still easily adapt our program to different UIs without paying the price of using patterns such as Decorator or Composite, which put more constraints on the eventual structure when compared to Observer.
As a closing remark on this, we didn’t put Observer in the design patterns’ list since our implementation doesn’t strictly “go by the book”, but we found it was more effective and flexible if done this way.

### Clean “noping”

We refactored the way we were noping by passing down the **CompletableFuture** to **Player**’s askForNope(...). This design choice, already foreseen in Assignment 2, allowed us to make our design even more flexible and resulted in a way simpler implementation, with less worries about concurrency issues.

### Better UX

Quite some comments on Assignment 2 were aiming at improving the user experience and we put several fixes for this in place. For example, (1) **AiPlayer** moves are slowed down, (2) **AiPlayers**’ failing attempts to play a combination of cards are hidden to the **HumanPlayer**, and (3) all **Players** can see their cards at the start of the game.

### Custom Deck

In this part, we had to read from files and we tried to cover quite some corner cases in validating the JSON file. In hindsight, we could have pulled out this functionality in a separate class, but we thought it was still fine to keep it inside **GameRunner**.

### Scoreboard

Implementing the scoreboard was almost trivial, but we got stuck at some point when serializing/deserializing with FastJSON (and the documentation of the library helped poorly). We had to debug FastJSON to notice that it probably uses Java’s **Reflection** under the hood to deserialize objects and thanks to this we were able to fix a **NullPointerException** in our code.

## 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_ek2_a3 |

## Time logs

Time Table Assignment 3

| **Team number:** | **2** | | | |
|---|---|---|---|---|
| | | | | |
| **Member** | | **Activity** | **Week number** | **Hours** |
| | | | | |
| All | | Groupmeeting | 6 | 2 |
| Kevin Koeks | | Research Networking stuff, give feedback | 6 | 7 |
| Sebas Fredrik | | Research Networking stuff, read feedback | 6 | 7 |
| Federico Giaj Levra | | Research Networking stuff, address feedback | 6 | 6 |
| Federico Giaj Levra | | CLI updates, testing features | 6 | 3 |
| Jelena Masic | | Research Networking stuff, address feedback | 6 | 7 |
| Jelena Masic | | Custom Deck, timeout on CLI interface, and improved NOPE | 6 | 5 |
| | | | | |
| All | | Groupmeeting | 7 | 2 |
| Kevin Koeks | | Started implementing exploding-kitten probability | 7 | 2 |
| Sebas Fredrik | | Implemented Scoreboard and Result class | 7 | 4 |
| Federico Giaj Levra | | Network Prototype | 7 | 6 |
| Jelena Masic | | Network Prototype (both RMI and TCP sockets) | 7 | 8 |
| Jelena Masic | | Refactor GameEvent and UserInterface | 7 | 4 |
| | | | | |
| All | | Groupmeeting | 8 | 2 |
| Kevin Koeks | | Finishing probability method, add new methods to classdiagram | 8 | 4 |
| Federico Giaj Levra | | Editing Scoreboard to fit implementation, wiring probability, more testing | 8 | 6 |
| Federico Giaj Levra | | Working on document | 8 | 3 |
| Jelena Masic | | Finish networking | 8 | 4 |
| Jelena Masic | | Documentation (feedback, DPs, class, sequence, implementation section) | 8 | 10 |
| Jelena Masic | | Polish up code | 8 | 2 |
| | | | 8 | |
| | | | **TOTAL:** | 69 |