Assignment 2

Team number: 12

Team members

|  |  |  |
| --- | --- | --- |
| **Name** | **Student Nr.** | **Email** |
| Andrei Dragomir | 2669304 | [a.dragomir@student.vu.nl](mailto:a.dragomir@student.vu.nl) |
| Francesco Marciani | 2677750 | [f.marciani@student.vu.nl](mailto:f.marciani@student.vu.nl) |
| Adam Ristov | 2659483 | [a.ristov@student.vu.nl](mailto:a.ristov@student.vu.nl) |
| Sander Vrzina | 2678335 | [s.vrzina@student.vu.nl](mailto:s.vrzina@student.vu.nl) |

## Implemented features

|  |  |  |
| --- | --- | --- |
| **ID** | **Short name** | **Description** |
| F1 | Extensible deck | The deck size and specific cards will be specified in a JSON file that the user can modify, and gets loaded by the server at start-up. |
| F2 | Client- Command Line interface | The client side of the game will solely act as the interface through which the user selects and requests a certain card in their hand to be played. On top of that, it will present a function which will act as a filter in order to prevent the user from trying to request a card to be played which would not be possible (because of its incompatibility with the rules of the game). |
| F3 | Server - Game state | The part of the server responsible for the game state. It tracks the status of all the decks and player hands, and updates such statuses when required (ex: a player plays a Shuffle card). This is basically the game itself.  At its most basic iteration it should be able to handle:   * Game setup * All types of cards * At least 2 players * Game ending |

**Used modeling tool**: diagrams.net - draw.io

## 

## Class diagram

*Author(s): Sander*

This is our UML Class Diagram that models the current implementation we have. In general terms it is capable of running a game between 2-4 players locally through the command line interface. Some cards have been semi-implemented, but most cards remain unimplemented. We also have the functionality of reading a deck from a .json file, this way custom decks can be added (F4 of implementation).

*The figure is at the end of the section*.

**Card:**

* This is one of the core classes in our implementation, it is the abstract class that is used to bundle up all the cards that exist in Exploding Kittens. This way we can put all the different types of cards in one ArrayList.
* *className* is used to identify the type of card that it is (e.g. defuse, favor etc) and it contains the name of the class.  
  *deckAmount* is used to show the amount of cards that are present when constructing the deck. This is only used during the construction and not at any other time.
* The most important function here is action(Deck, Hand) which will, once called, activate the ability of the card. Since this is an abstract function, calling it on a Card object will activate different abilities depending on what subclass that Card object is.  
  The equals() function is used to make it possible to compare an object to another object. This is normally not possible, which means that for example now we can do cardDrawn.equals(new exploding\_kitten) which gives us True if it is an Exploding Kitten. This function is used very often throughout our implementation so it helps a lot.
* aa

**Deck:**

* This represents the draw pile that players will draw from, it can provide information about the current state of the deck through several functions.
* *cardDeck* is an ArrayList that contains the Deck itself. The first index is the top of the deck, and the last index the bottom. We considered using a stack since that is “logical” in terms of a deck, where drawing from the deck is equal to popping off the stack. However this would quickly create problems with some card abilities like shuffle and placing the exploding kitten back. Hence why we ended up using an ArrayList which has useful helper functions like insert() and shuffle().

*cardDeck* is a HashMap which is used to convert the strings that are read from the .json file into their appropriate objects. It maps the card name to an object and that object gets added to the deck on construction. I was trying to use the java function forName since this creates an object that corresponds to a string, however after at least 10+ hours of debugging I was unable to make it work even a bit. Hence why I used this work around using the Gson library.

* Deck() is the constructor of the **Deck** class and is rightfully the function which constructs the deck from a json file. I have already explained how this would happen and the difficulties I faced in the *cardDeck*attribute part.  
  Other operations in **Deck** are helper functions which other Objects can use to construct stuff or manipulate the *cardDeck*, aka the deck itself. Some of these functions include:   
  draw() - Returns a **Card**, which is at the top of the *cardMap*

getDeckSize() - Returns amount of cards left in the deck

getStartCards() - Used by **Hand** to get 7 starting cards.

* The most important association for the **Deck** class is that it contains multiple **Card** objects in it’s *cardMap*. This is what makes the **Deck** class easy to integrate with the rest of the system.

**Hand:**

* This is a class used to handle a player’s hand, it constructs the Hand open creation and has a helper function that can be used to manipulate a specific player's hand.
* The only attribute in this class is *currentHand*, which is an **ArrayList** that contains all the current **Card** objects in that particular hand. This seemed like the most straightforward approach to making a hand, and it didn’t need anything more fancy. With well made helper functions it will be super simple to manipulate a player’s hand, which will keep the code clean and concise.
* Hand(Deck) is the constructor which uses the getStartCards() helper function from **Deck** to create the starting hand for a player. The reason it was designed this way was because a starting hand cannot contain an Exploding Kitten, so we had to draw() and return the card if it was a Exploding Kitten. Doing this all from within **Hand** would require a lot more unnecessary helper functions than simply doing all the processing within **Deck**.   
  Other functions in the **Hand** class are helper functions so that other classes can manipulate the player’s hand (e.g. get the whole hand of a particular player with getHand(), a player plays a card so play() is called).
* The most important association with this is that it contains the **Card** class for the **ArrayList** that is stored in *currentHand*. However this was already discussed in the **Deck** class and as per the assignment instructions will not be discussed further.

**DiscardDeck:**

* This class is used to represent the discard pile where cards are player upon. It is overall a fairly simple class. It is very similar to the **Deck** class, except that it requires less operations which means it has relatively few operations and attributes.
* The most important (and only) attribute is *d\_deck* which holds all the discarded decks in an **ArrayList** that contains **Card** objects. The latest discarded card is added at the front of the **ArrayList** (index 0).
* discardCard(**Card**)is the main operation, which is used to add a played **Card** to the discard pile.

top() returns the latest discarded **Card**, which is needed to make some **Card** abilities functional

* The most important association with this is (once again) that it contains the **Card** class for the **ArrayList** that is stored in *d\_deck*. This was already discussed in the **Deck** class and as per the assignment instructions will not be discussed further.

**Player:**

* This is the class used to represent a human or computer player. It will contain all the relevant information to that player, which is for now only the name and the **Hand**.
* *hand*is an attribute that holds a **Hand** object. This will be used by other functions to manipulate a particular player’s hand. Many **Game** events and **Card** abilities manipulate the hand of one or more players, hence why it’s useful to have a structure wherein the **Hand** of a specific **Player** can be manipulated. This will help tremendously in clean and clear coding.

*name* is an attribute that simply stores the name of the player as a **String**, which will be needed to identify the player when we implement rooms in a server-client implementation. It also shows who’s turn it is in local play.

* getHand() returns the **Hand** of that specific **Player**, so that it can be manipulated using helper functions from the **Hand** class.  
  initHand() will initialize the **Hand** object, which will in turn generate the starting hand.
* The **Player** class has a **Hand** attribute, which is the most important dependency. The reason was already explained in the attribute section (allows manipulation of specific player’s hands).

**GameManager:**

* The **GameManager** class is basically the helper class of an active game of Exploding Kittens. It keeps track of the players left and who’s turn it is. It has some mostly helper functions which will help with implementing **Card** abilities. For assignment 3 we will most likely have to add more attributes that keep track of the current game state, for example how many **Nope** cards were played (so that we can accurately get the end result of someone that nopes a nope that was already noped which noped an attack).
* *turns* is an attribute that keeps track of who’s turn it is. It uses a handmade circular linked list called **DoublyLinkedList**. This way we can easily go to the next player’s turn by simply going to the “next” element in the list. The turn can be manipulated with helper functions as well. *alivePlayer* is an attribute that keeps track of players which haven’t exploded yet, once one player is left in the **List** the game is over and the winner is determined.
* In this class only have helper functions which **Game** or a **Card** subclass will call once a certain event occurs. These functions directly impact specific **Player** objects, which can in turn manipulate the **Hand** of that **Player**. There is no method that is the “most important” since all of them serve different purposes depending on the situations. Some examples include:  
  setNextTurn(**Player**) which sets which player has the next turn, this can be used by a **Skip** or an **Attack** card for example.   
  addPlayers(**int**) adds players when starting the game, the amount of players is determined by the host (which for now is always the local computer since we have no server-client). So it is determined from within **Game** by input from the host.  
  killPlayer(**Player**) will, like the name suggests, eliminate a player from the game. This will most likely happen if the player draws an Exploding Kitten and has no **Defuse**, but it can also happen if the player disconnects in a server-client scenario.
* The **GameManager** class interacts very often with the **Player** class, it contains that class in a **List** within *alivePlayer.*

**Game:**

* This is the “main” class of our implementation. This is where you have to start the game up, and it will guide you through the process of starting a game with 2-4 players. In assignment 3 it will allow you to join different rooms and edit what kind of deck you will be using (or completely custom decks using the json file). It handles reading from input and controlling game events with the help of **Card** abilities and **GameManager** helper methods.
* The **Game** class doesn’t contain many unique attributes, it just creates the most of the classes discussed above, and will handle them accordingly. The only attribute that is unique is the *rand* and *scanner* attribute. The *rand* attribute will be used to program the computer opponent. However we will likely move it to a new class that is called something along the lines of “ComputerOpponent”. The *scanner* attribute is used to read from input and determine what the host / player wants to do. For assignment 3 we are planning to create a GUI (if time allows it), which will remove the need for most of the input.
* The only operation we have is start(**int**), which contains all the code to start and run a game. It will read from input, determine the amount of players, play cards, execute appropriate actions etc. We realize that this method is way too big and should be split in multiple methods. We have started doing this while we are slowly implementing the server-client and GUI, however the implementation we are handing in for assignment 2 is without any of these two. We didn’t change it already since we would have automatically been forced to change it while doing the GUI again, so it would just create double the work.
* The **Game** class has 3 important associations. It has the **Deck**, **DiscardDeck** and **GameManager** objects inside it. Considering this is the heart of our implementation, it will need access to pretty much all the edges of our classes, which can be done through (chaining) helper method calls.

Since the **Card** subclasses are mostly the same, we will only name the difference between the action(**Deck**, **Hand**) method between the different subclasses (since this is the only difference).

*TBI = To be implemented*

**Defuse:** Defuses the **Exploding\_kitten** if you have one in your hand.  
**Exploding\_Kitten:** The player will die if they don’t have a defuse card, otherwise they have to play the defuse card

**Favor:** *TBI,* you can ask a player for a card, they choose which card they give you.

**Attack:** *TBI,* you end your turn without drawing, and force the next player to take 2 turns

**Cat\_card:** *TBI,* does nothing except when played as a special combo

**Nope**: *TBI,* Stop any card except a **Defuse** or **Exploding\_kitten**. You can stop another **Nope** which in turn can also be stopped by another **Nope**  
**See\_future:** *TBI,* See the top 3 cards of the deck in private

**Skip:** *TBI,* End your turn without drawing a card

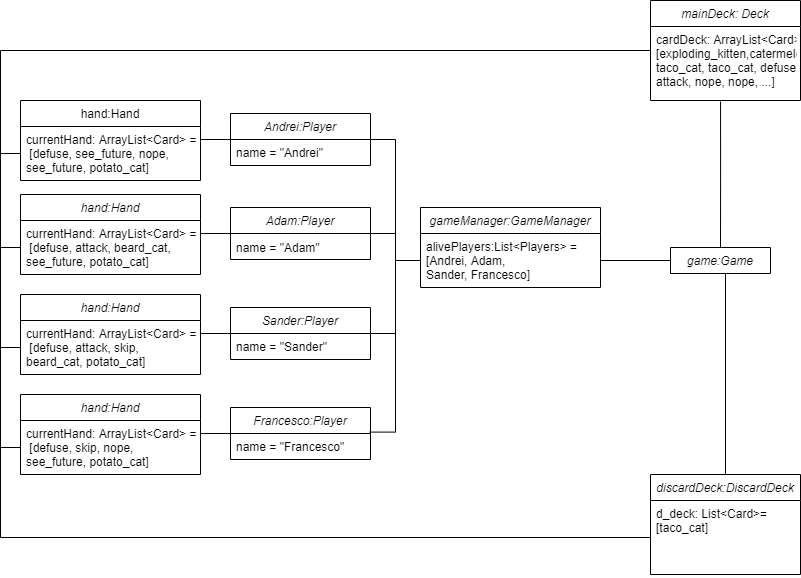
**Shuffle:** *TBI,* Shuffle the deck, useful if you think someone placed an Exploding\_kittenon top of the deck.

*Note: The* ***Cat\_card*** *has the cards:* ***Rainbow\_cat, Potato\_cat, Catermelon, Beard\_cat, Taco\_cat*** *as its children, the only difference is that they have different pictures.*Below is our UML Class Diagram  
  
*A note about our dependencies: a dependency in UML is often shown with a dashed line and an empty arrow (*[*example*](http://www.cs.sjsu.edu/~pearce/modules/lectures/uml2/relationships/Relationships_files/image003.jpg)*), however we thought that we should not make the lines dashed since it’s clear from the black triangles what kind of dependency one class has from another and it would be a tautology.*

## 

## Object diagram

*Author(s): Andrei*



Textual description: The diagram above shows the state of the game after the first players turn (Andrei). Starting from right to left the game object contains a *game manager* which contains in itself the *player* objects which in turn contain their specific *hand* objects.

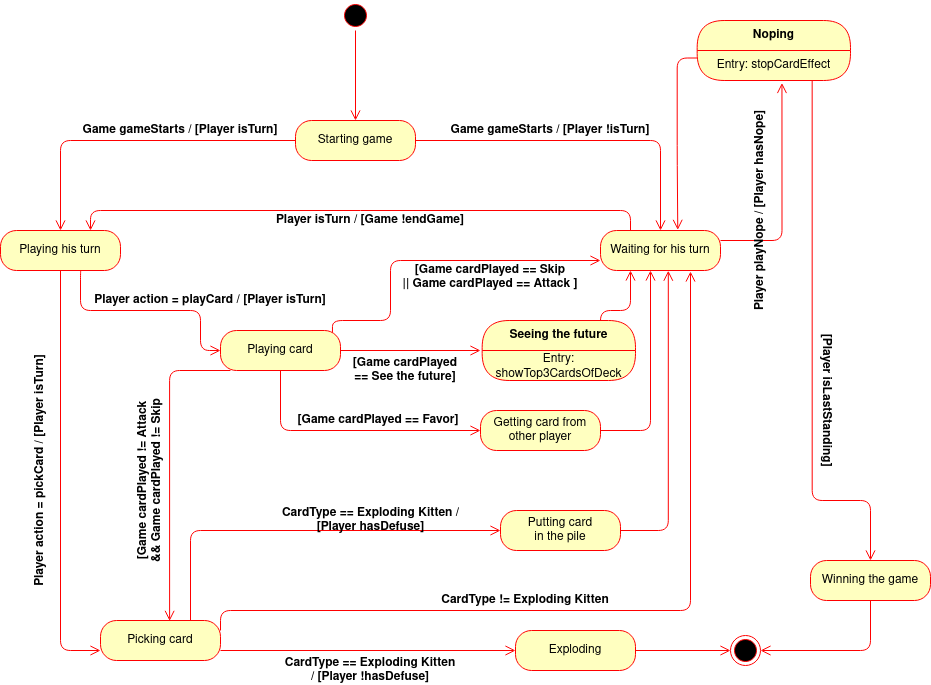
The relationship that the two decks have with the hands of the players are related to the ability of transferring the cards directly from the *mainDeck* or the *discard Deck* into the *hand* of a *player* and from the *hand* back into the *deck*.

## 

## State machine diagrams

*Author(s): Francesco*

**Player Class**



The **Player** class represents the possible states an individual player can find themselves in. When the game starts, it can either be the *Player’s* turn or someone else’s. If it isn’t the *Player’s* turn, he can wait or, if he has a **Nope Card**, play the **Nope Card**, which stops the action of the card another player just played (except for **Exploding Kitten** and **Defuse**). If there’s only two *players* left and the other explodes during their turn, the *Player* wins.

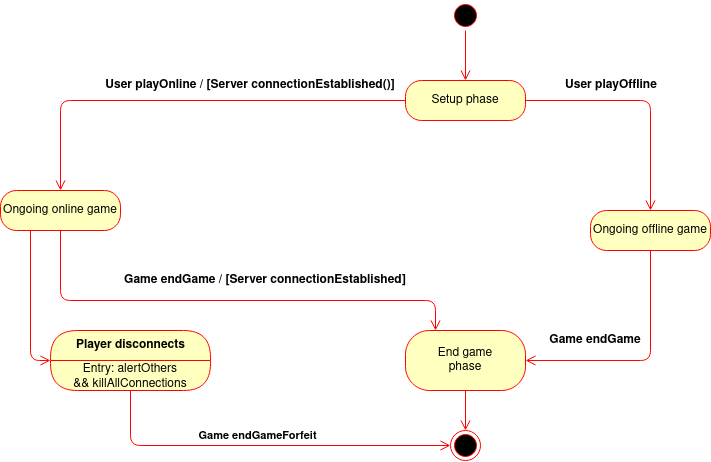
If it is the *Player’s* turn, they can either play a card or not. If they choose not to play a card, then they have to directly pick one from the pile and end their turn.

If they play a card, they have to then pick one from the pile, unless they played a **Skip** or **Attack** card. If they played a **See the future Card**, then they get to see the top 3 cards of the deck, and if they played the **Favor Card**, they receive one card from another player of their choosing.

When they pick a card, if it’s not an **Exploding Kitten** they go straight to waiting for their next turn. If it is an **Exploding Kitten** then they either have a **Defuse** or not in their hand. If they do, they play it, put the **Exploding kitten** back in the deck and then wait for their next turn. If they don’t, they explode and their *game* is over.

**Game Class**

The java class **Game** represents the state of our game, which is rather simple. When the game is started, it goes through the setup phase, which ends by either starting an online or an offline *game*. If the game started is offline, the *game* plays out and, once it’s concluded, it enters an end game phase and then terminates. If it is an online *game*, then it can either do the same, if all players’ connections are stable until the end; or if one connection drops, every player gets an alert and then gets disconnected, and the *game* terminates without a winner.



## 

## Sequence diagrams

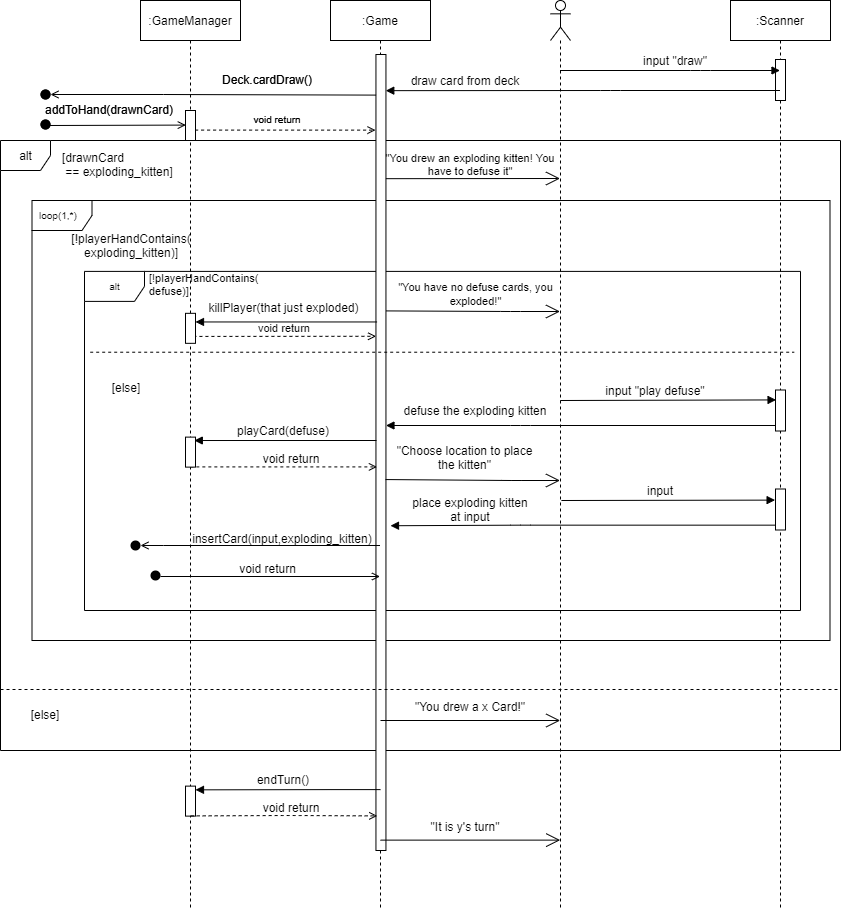
*Author(s): Adam*

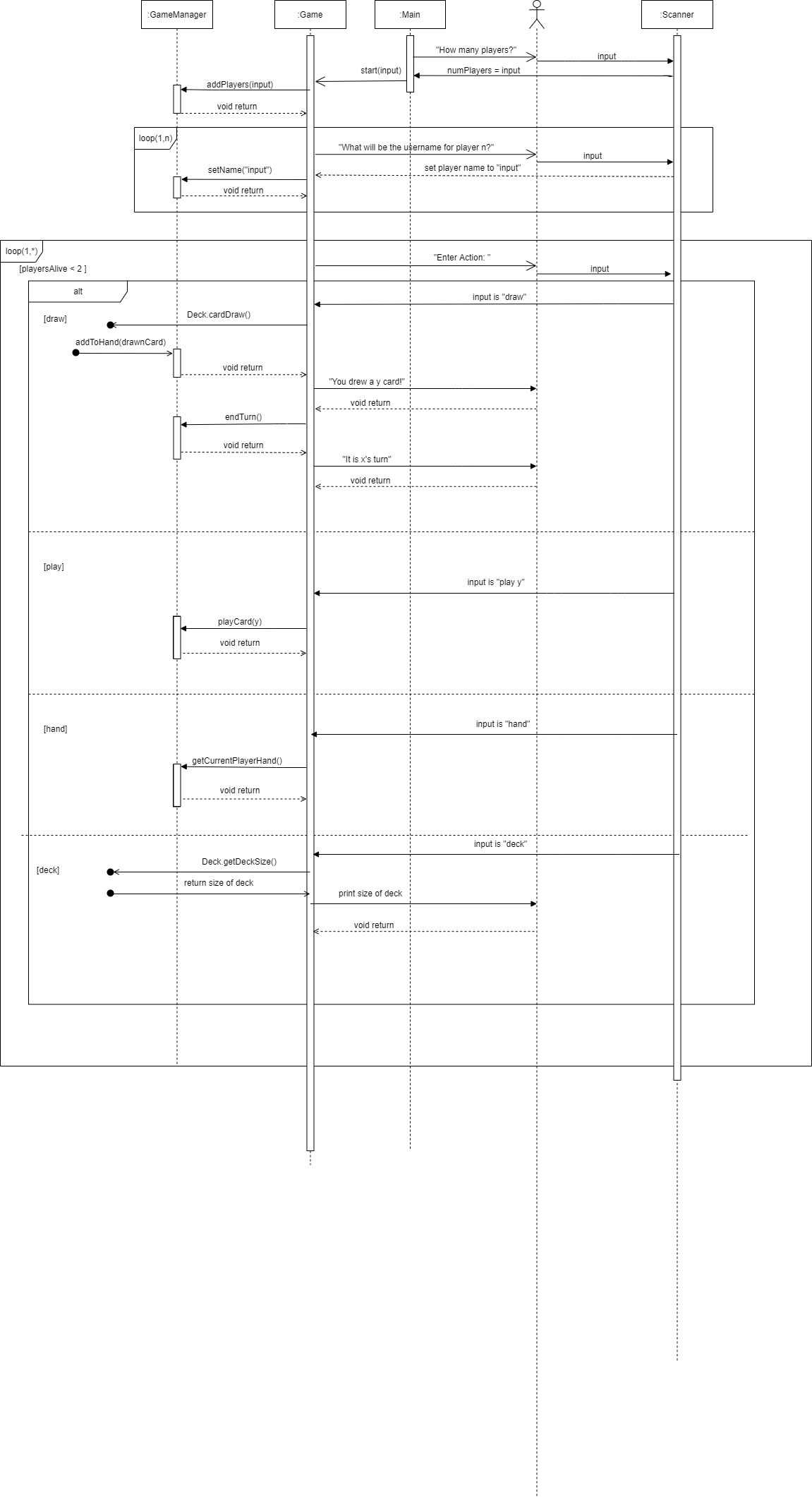
*(The diagrams and descriptions are in reverse order so that the diagrams could fit in 4 pages)*

**Initialization and typical running mode of the videogame (Figure 2)**  
  
In this diagram we’ve decided to describe the initialization process of the game, namely the initial interaction between the user and the game and the typical running mode of the game. More specifically, in the diagram, the user interacts with the **Scanner** and **Game** class, whilst the **Game** class, “under the hood” interacts with the **GameManager** class. The **Game** class of course interacts with more classes as well, however, for the sake of readability and to prevent a giant cluster of incomprehensible interactions, we limited this diagram to these 4 classes (as we deemed these to be the most important), along with the **Main** class which serves as a starting point for the game. The **Main** class initially asks the user to input the number of players which is done through the built in **Scanner** scanner. The input from the Scanner is then sent back to the **Main** which uses this input to start() the game. The **Game** class then tells the **GameManager** to add the input number of players into the game using the addPlayers() method, which returns void. Then, a loop begins, where the **Game** class asks the user(s) to input the desired usernames, which the user does through the **Scanner** class, and that information is sent back to the **Game** class, which forwards it to the **GameManager** class using the setName() method. Along with the names, other things are also initialized, such as the turn handler and the initial hands for the players, however for readability these have been omitted from the diagram. After the initialization, the main loop for the game starts, which runs until there are less than 2 players alive. At the top of the loop, there is always an output from the **Game** class to the user, asking for input. After the user gives input, there is an alt fragment which branches according to the user input. In the case the user inputs “draw”, then the **Game** goes to the **Deck** class to draw a card and the **Player** class tells the **GameManager** to add the drawn card to the player’s hand, using the addToHand() method. The **Deck** and **Players** classes are omitted as they are used only once/twice in this diagram and to prevent convolution. After the hand is added, the **Game** class calls the **GameManager** to end the turn, outputs whose turn is next and then the main loop starts again. In the case of the user inputting “play” along with the index of the card in his hand, the **Game** class executes the action of the card by calling playCard() through the **GameManager**, after this, the user returns to the main loop and can perform another action. In the case of the input being “hand”, the **Game** class calls getCurrentPlayerHand() through **GameManager**, which prints the player’s hand and returns void. Furthermore, in the case of the input being “deck”, the **Game** class calls the **Deck** class using getDeckSize(), which returns the size of the **Deck** to the **Game** class and then the **Game** prints the size of the deck.

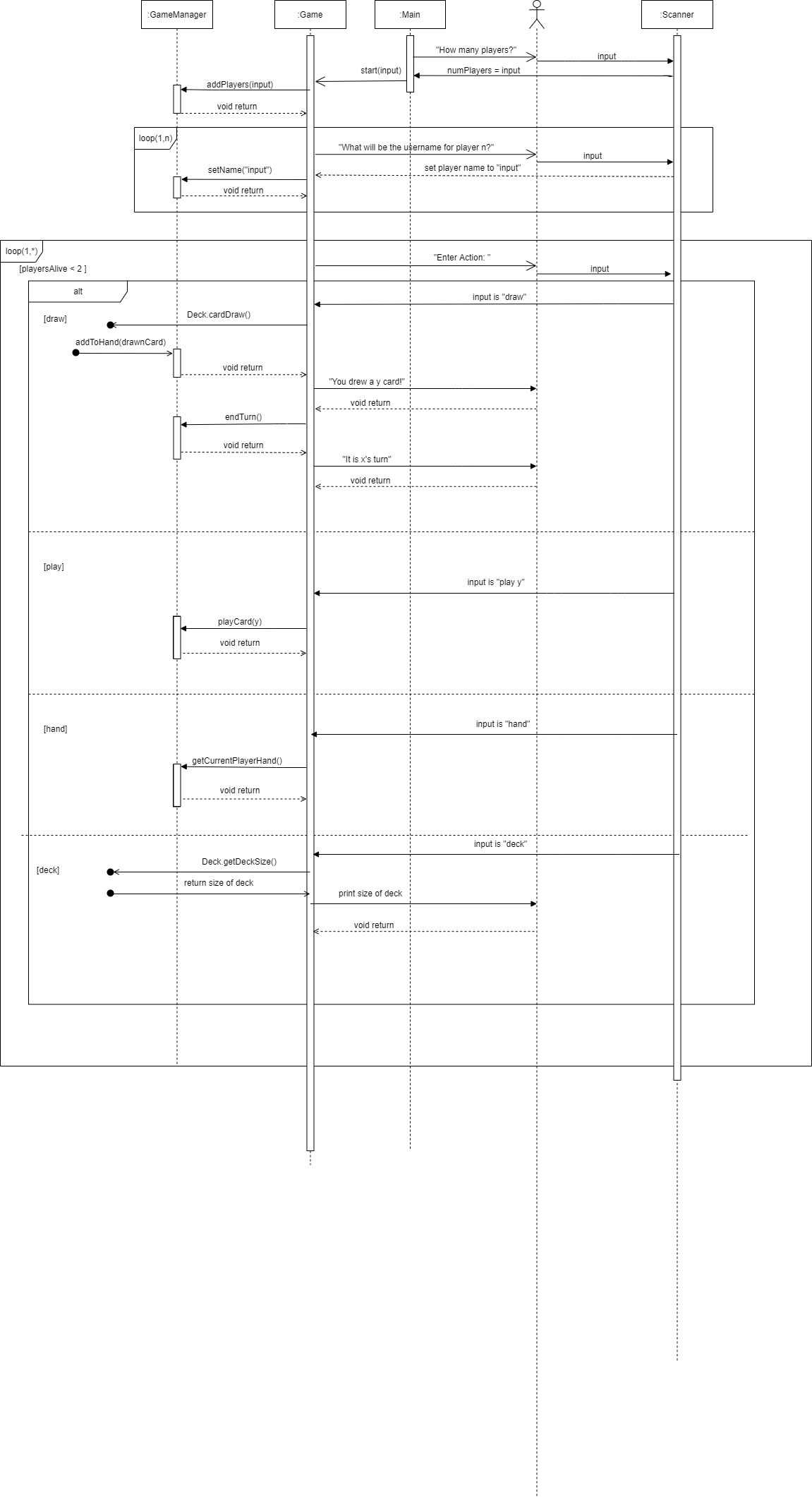
**Drawing regular cards and exploding kittens (Figure 1)**

In the second sequence diagram, the “draw” part of the alt fragment is examined further. In the previous diagram, the inner workings of the “draw” were hidden in order for the diagram to be more coherent. In this diagram, the same participants are included except the **Main**, as it is irrelevant here. Some overlap from the first diagram is expected, thus the first part of the diagram will not be explained here, as it is already explained in the previous diagram description. Instead, the focus here will start from the first alt fragment. In the case of the drawn card to not be an exploding kitten, the **Game** class outputs what kind of card the user drew, ending the [else]part of the fragment. However, if the user draws an exploding kitten, a loop starts which continues until the user has an exploding kitten in their hand and the **Game** class outputs that the user drew an exploding kitten. In the case of the user not having a defuse card, then the **Game** class outputs that information as well, right before calling the killPlayer() method on the player through the **GameManager**, the method returns void and the loop ends, as the player no longer has the card in his hand. In the case of the user having a defuse card, he is forced to play it by inputting “play x” where x is the index of the defuse card in his hand. In the diagram it is written “play defuse” for better understandability. After the user enters their input to the **Scanner**, the **Scanner** forwards that information to the **Game**, and then the **Game** calls the **GameManager** using the playCard() method, which returns void. After the kitten is defused, the user is given the choice of placing the kitten back into the deck at any location through an output from the **Game** class. After the user inputs their desired location, the **Scanner** relays that information and the **Game** class calls the insertCard() method from the **Deck** class, which returns void and successively exits the main loop, as the kitten is removed from the player’s hand. Moreover, the **Game** class calls endTurn() to the **GameManager** to end the current player’s turn, after the player successfully draws a card. The **Game** class also outputs whose turn it is next.

**Figure 1:**



**Figure 2^**



## Implementation

*Author(s): Andrei*

In this chapter you will describe the following aspects of your project:

* For this assignment, when we decided to switch from the UML to the implementation, we started with the implementation of the bigger classes and then started implementing their respective subclasses in order to not have to change the structure or implementation of an entire branch of subclasses. The reason why we could reverse engineer the game in this situation is because we were already able to set the limits and the capabilities of our program, knowing already what the final product of our project will be.
* Once the game.start() class function is called, the player goes through the initialization of his game, specifying the number of players he wants in the game and their names. Afterwards, the game starts and the first player is prompted to play his turn.

The key aspect of our implementation of the Exploding Kittens game is mostly based on the abstract class named **Card**. Up to this point in our implementation, we managed to handle all card interactions by using this abstract class out of which we created different instances with different actions based on the card’s name. In this way, we can use the name of the card to match the action wanted. At the moment, the cards can be played using an input command starting with the word “play” followed by the index of the card in hand. Based on that index, the card in *hand* is “taken out” of the *hand*, it gets placed inside the *discardDeck* and then its related **Card** class instance function called “.action()”.

* Path to executable java: Assignment 2/src/main/java/softwaredesign/Main.java
* The location of the JAR:

Assignment2/out/artifacts/software\_design\_vu\_2020\_jar/software\_design\_vu\_2020.jar

* The link to the demo of the game up to this point: <https://youtu.be/MWP_vKAANuA>

## Time logs

