Assignment 3

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) |

.

## Summary of changes of Assignment 2

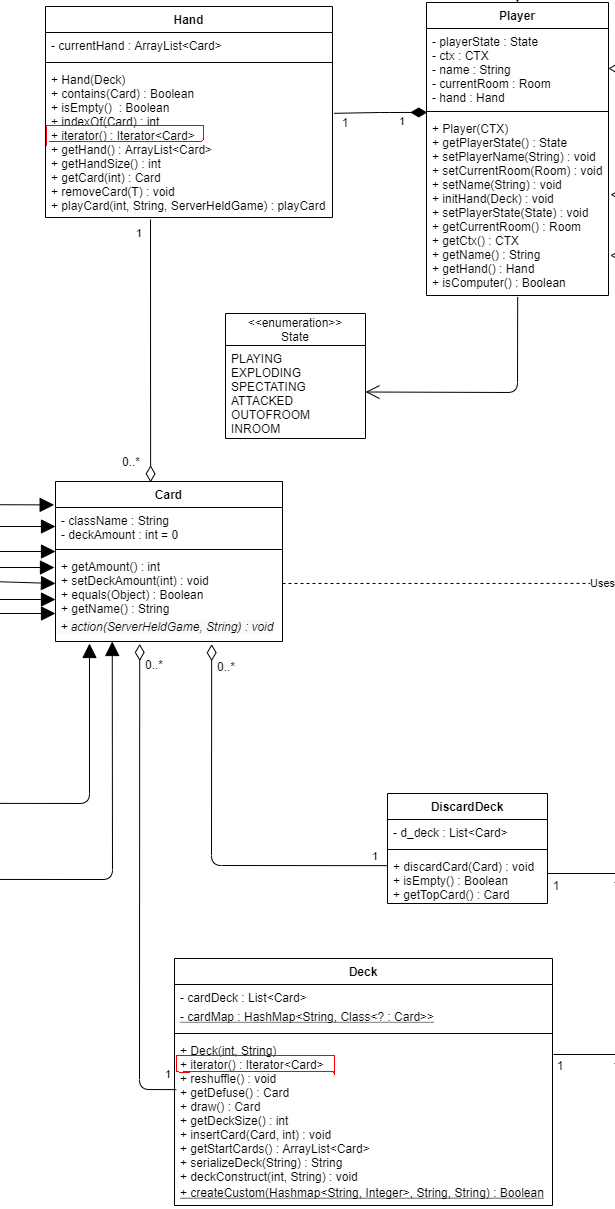
*Author(s): All members*

* In the sequence diagrams, the lost messages are the Deck and Player class and the decision for not including them in the diagram is explained in the description, thus nothing was done in relation to that comment. The ‘input’ loop typo was also fixed.
* In the object diagram, the card class instantiations are now provided for both the hand objects of each player, as well as the deck/discard deck. As for the doublyLinkedList that was missing, the implementation for this assignment no longer contains this structure.
* State machine diagrams align more with the implementation and less abstract
* Class diagrams are explained in more detail

## Application of design patterns

*Author(s): Adam*

|  |  |
| --- | --- |
|  | **DP1** |
| **Design pattern** | Iterator |
| **Problem** | Making the access to Hands/Decks easier |
| **Solution** | The Iterator design pattern solved this issue by having the player’s current hand transform to an iterator object to allow for easier looping when needed. It also helps prevent the unnecessary access to private variables and to large amounts of getters and setter calls. |
| **Intended use** | It will be used to iterate over a player’s hand when needed, such as displaying it through the GUI or when searching for a specific card in the Hand |
| **Constraints** | None |
| **Additional remarks** | None |



## Class diagram

*Author(s): Sander*

For the class diagram descriptions, an attempt was made to describe all the classes, however that went beyond the 20 page limit, thus we decided to describe only the most important classes.

**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 ExplodingKittenCard) which gives us True if it is an *Exploding Kitten Card*. This function is used very often throughout our implementation so it helps a lot.

**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 shuffling 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.
* iterator() - Returns the Iterator of the *cardDeck*
* 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 playCard() 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.
* Functions such as contains(), isEmpty(), indexOf() are used in order to find and return certain traits of the *currentHand* ArrayList.

**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.
* getTopCard() 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, that being 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.

* *currentRoom* contains a reference to the *Room* object to which the player is assigned in the lobby.
* *ctx* contains the netty ChannelHandlerContext details for the networking part of the game. It is assisted with a getter and a **Player** constructor which assigns the ctx.
* 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).

**ServerHeldGameManager:**

* The **ServerHeldGameManager** 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 help with implementing **Card** abilities.
* *alivePlayers* is an ArrayList 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.
* initGame() is the function which builds the game once the creator of the game starts the game. This function instantiates *Deck* and *DiscardDeck* objects and populates the *alivePlayers* ArrayList.
* In this class we only have helper functions (other than initGame()) 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:
* changeNextTurn() changes the turn to a specific player which is a requirement in the AttackCard action and the FavorCard action.
* endTurn() which ends the turn of the current player.
* 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 **ServerHeldGameManager** class interacts very often with the **Player** class, it contains that class in the *alivePlayers* ArrayList.*.*

**ServerHeldGame:**

* This is the game controller class of our game. This is what actually handles the actions of players and processes them. It uses the **ServerHeldGameManager** as a helper function to get the needed info.
* The **ServerHeldGame**class has only 2 attributes. *gameManager* is the object that refers to an instantiated **ServerHeldGameManager**, which it will use (like discussed above) as a helper function to retrieve info from the game.  
  *Room* is the other attribute that exists, it is kinda off of that **Room** has an attribute referring to the **ServerHeldGame** and that the **ServerHeldGame** also has an attribute to refer to **Room**. However after many different approaches, it seemed the easiest to do it this way because the hierarchy is still clear: **Room** creates the **ServerHeldGame**, which just refers to which **Room** this object belongs to. This creates it easier for us to use methods from **Room**, which are needed in quite diverse classes.
* The important operations here are handlePlayAction() and handleDrawAction(), these handle the 2 most common actions in the game.   
  handlePlayAction() plays the card of the current player, and sends the appropriate messages to the players. The actual removal and .action() of the **Card** is handled in the **Hand** class, because this way you don’t have to do many method calls to edit the hand of a player.

handleDrawAction() handles the drawing of cards. This includes ending the turn. In the case that a player draws a **ExplodingKittensCard** it will not end the turn, rather it will send a message “DREWEXP” to inform the player that they drew a kitten, or it will kill the player immediately if they don’t have a **DefuseCard**. In the case that a player dies we also have to check whether a player has won, we do this with checkWin(). This function simply checks if there is one player left, and if there is it takes care of changing the state to a Win/Lose screen.

* The most important association of the **ServerHeldGame** is the composition with the **ServerHeldGameManager**, as explained before it uses this class to read the core info of the game.

**Room:**

* A room is a lobby which can contain at max 4 players. In the room you can chat and see the other players. It is essentially a standard “waiting room”.
* **Room** contains a lot of important variables for managing players.  
  *roomPlayerList* contains all the players and you can look them up using their respective names (HashMap<String, Player>). This means if you have the name you have access to the **Player** class, which in turn gives you access to all of the information about a player.   
  *onlineGame* is the **ServerHeldGame** instance which will be used to start the game once the start signal has been received (and approved). From onlineGame the room has access to lower level game methods, which means it can call the appropriate method if a specific message has been received.
* The most important operation is the channelRespond() operation, this parses the received message from the client, and acts appropriately. This way if it receives “PLAY 5”, it will be play the 6th card (5th index) in the current players hand. A player cannot play a card when it’s not his turn (this is handled in the GUI).   
  2 other important operations are sendMsgToRoom() and sendMsgToPlayer(), which is used very often to communicate to players (even outside of the Room class, like in Card for example).
* The most important association is with the **ServerHeldGame** class, this allows it to call the right methods by parsing the messages.

**ServerHandler:**

* This class is used to keep track of all the players that are connected to the server using a HashMap called *playerMap* and also too keep track of the rooms in which the connected players are located, using another HashMap called *roomList.*
* On top of that tracking, the **ServerHandler** class is also used to handle actions and respond to messages incoming from all the clients that are connected. It does so by using the Overridden netty functions such as channelActive() and channelRead().
* channelActive() adds a client to the server once the respective client connects.
* channelRead() takes in a message sent through the pipeline and parses it through the channelRespond() function.
* channelRespond() is the most important function because it handles all distribution of the players through the 2 HashMaps mentioned. It is also used to create *Room* objects at the request of a client.
* Lastly, the class comes with small helper functions which are needed in order to either send back messages to clients, checking for availability of rooms, etc.

Example of such functions: constructDeckFromMessage(), checkNameInUse(), cleanRoomOfEntity(), etc.

**Computer:**

* This is the “A.I” of the game, as the user can choose to play against computers rather than other players. The **Computer** class has a boolean method called isComputer() which returns true and it is called for determining whether there is a computer among the alive players.
* The most important attribute of the **Computer** is the *rand* attribute which allows it to randomize its plays.
* The **Computer** class has a getRandCard() method which chooses which card to play which can be any card except the defuse card

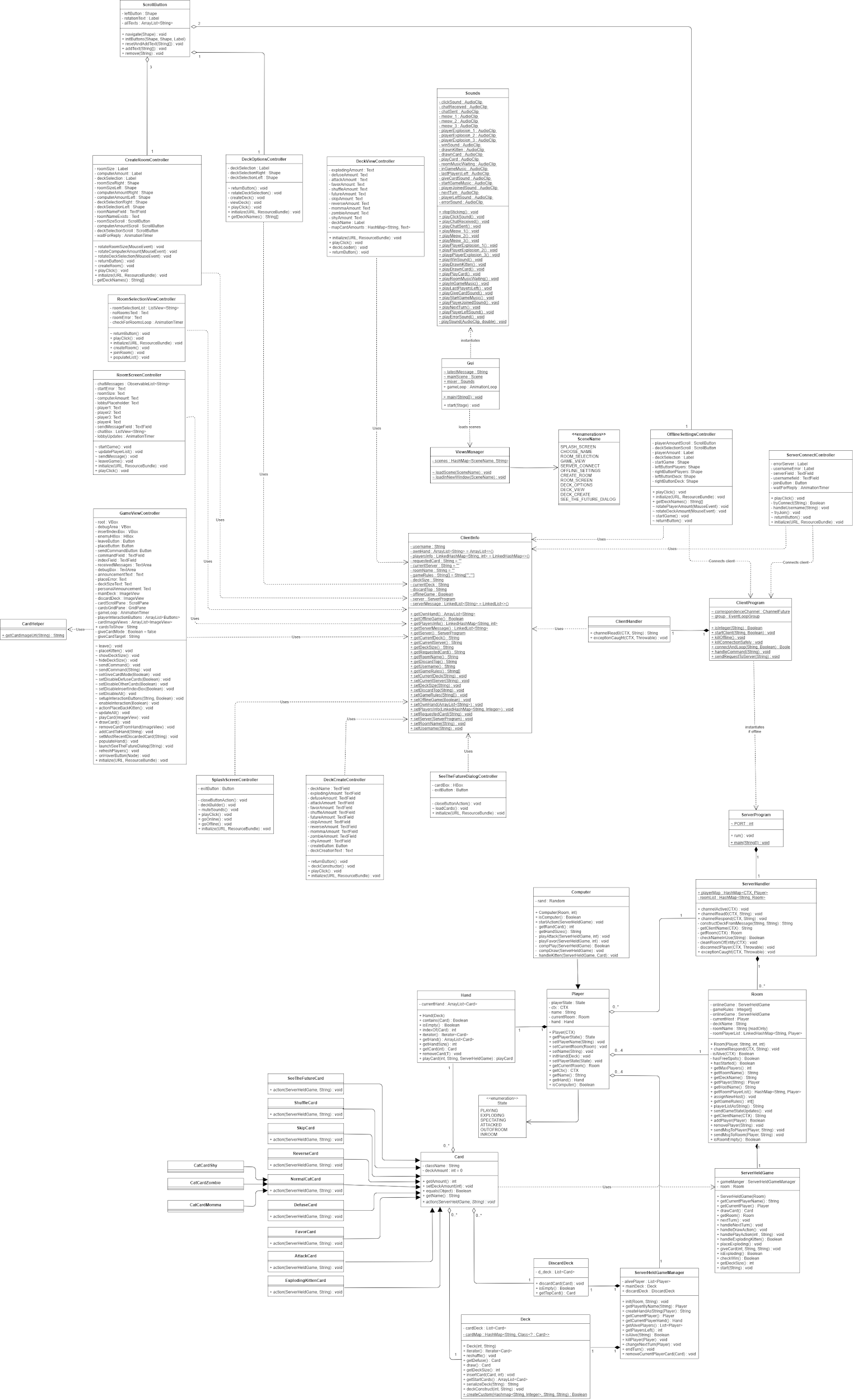
It also has a playAttack() which attacks the player with the least amount of cards, which is calculated using getHandSizes()

Moreover, it has a handleKitten() which handles the drawing of an Exploding Kitten, namely it defuses the card if it can, and places it at a random index.

* And the class also has comPlay() and compDraw() which chooses a card to play and draws a card for the computer, respectively.

**Card**:

* The **Card** class is the abstract class that most of the game cards extend. This means they have to implement the action() function. It also has a **NormalCatCard**  which is extended by 3 normal cat cards: **MommaCatCard, ShyCatCard, ZombieCatCard**. These are essentially the exact same cards so they extend the **NormalCatCard**.
* The most important attribute is *deckAmount,* which is used for the deck construction and deconstruction of the deck. This allows players to make custom decks and play with them (even with people who don't have the deck)
* The most important operation is the action() operation, it is an abstract operation which means all the subclasses have to implement it. This way you can just call action() on any **Card** drawn and it will play the appropriate card action (so it will skip if the **Card** is a **SkipCard**). One downside of our implementation is that we pass the **ServerHeldGame** instance to the action() operation. This gives the **Card** class access to the whole core game, which isn’t that good of an implementation. We have discussed alternatives to this but we couldn’t find a better way to implement it without changing the whole codebase. We decided this would be too time consuming so we left it the way it is.
* **Card** doesn’t have one important association, the **Deck,** **Hand, DiscardDeck** associations are equally important. They hold the **Card** and can call the actions when someone plays a card for example.

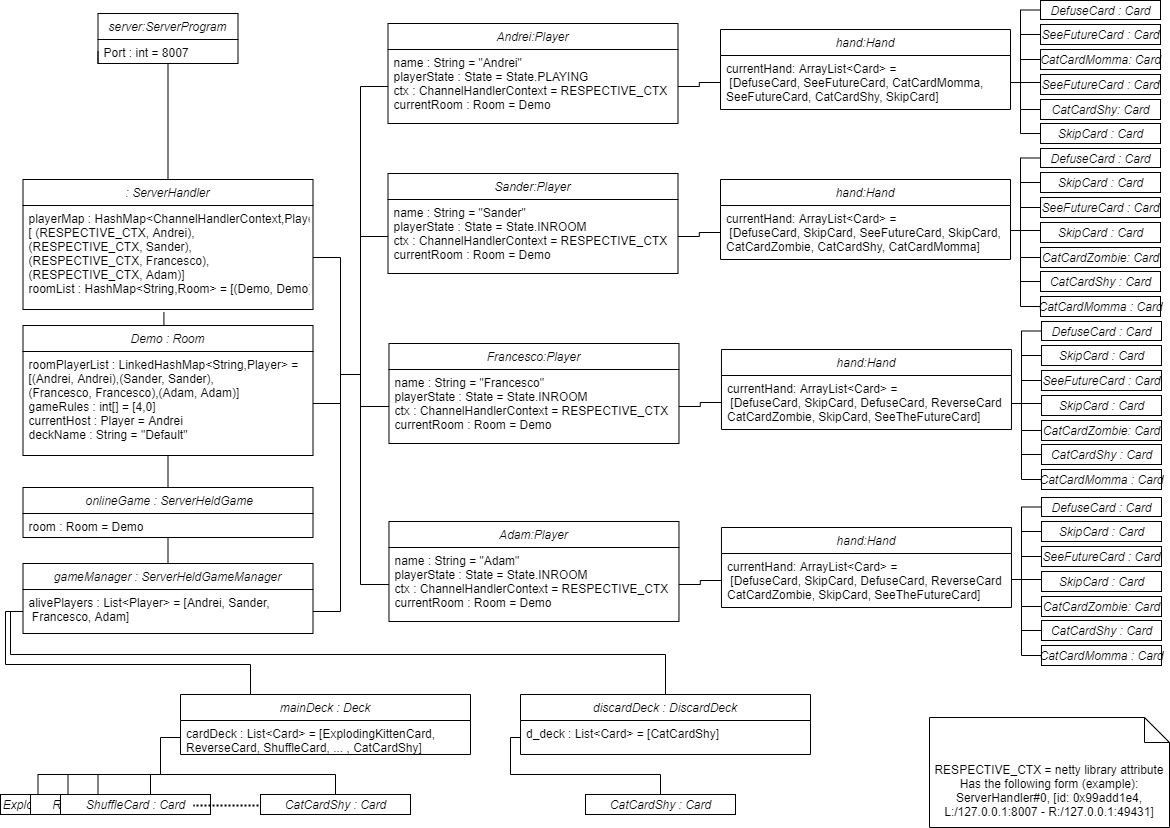


*(See github for full image size)*

## Object diagram

*Author(s): Andrei*

Object Diagram on the server side of an online game after first player has played one card



The above-presented diagram represents the object diagram of a specific snapshot of the system, that being the online game from the *server*’s perspective after the first *player* (*Andrei*) has played one *card* (*CatCardShy*).

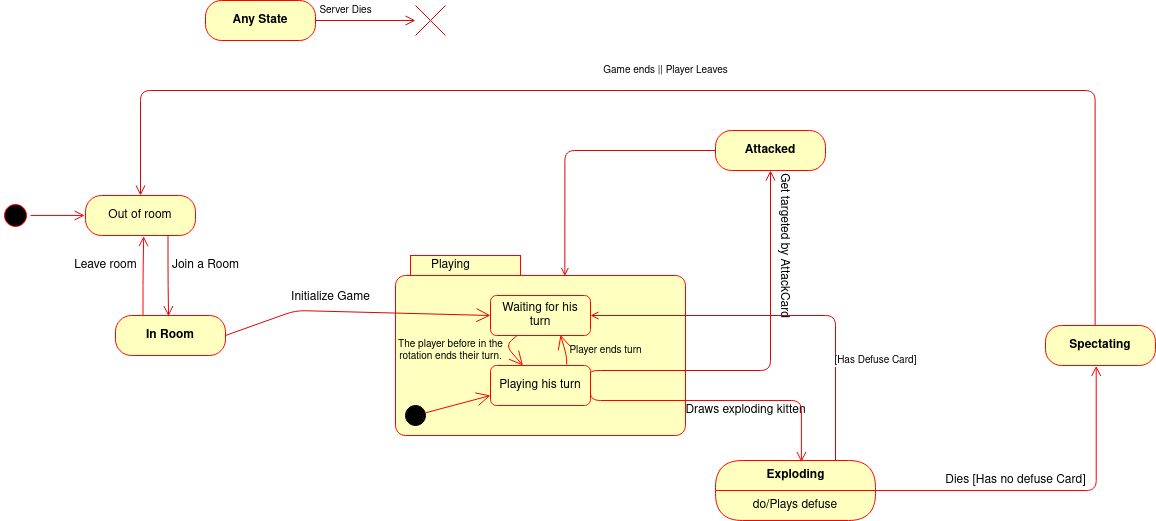
As observable in the diagram, the *server* contains a *ServerHandler* object which in turn contains a *Room* object. This *Room* object in turn, contains a *ServerHeldGame* object which in turn contains a *ServerHeldGameManager* which lastly, contains the *Deck* and  *DiscardDeck* objects.

The *Player* objects are accessed and updated through *ServerHandler, Room* and *ServerHeldGameManager* and these *Player* objects contain one *Hand* object each which in turn contain a varying number of *Card* objects.

## State machine diagrams

*Author(s): Francesco*

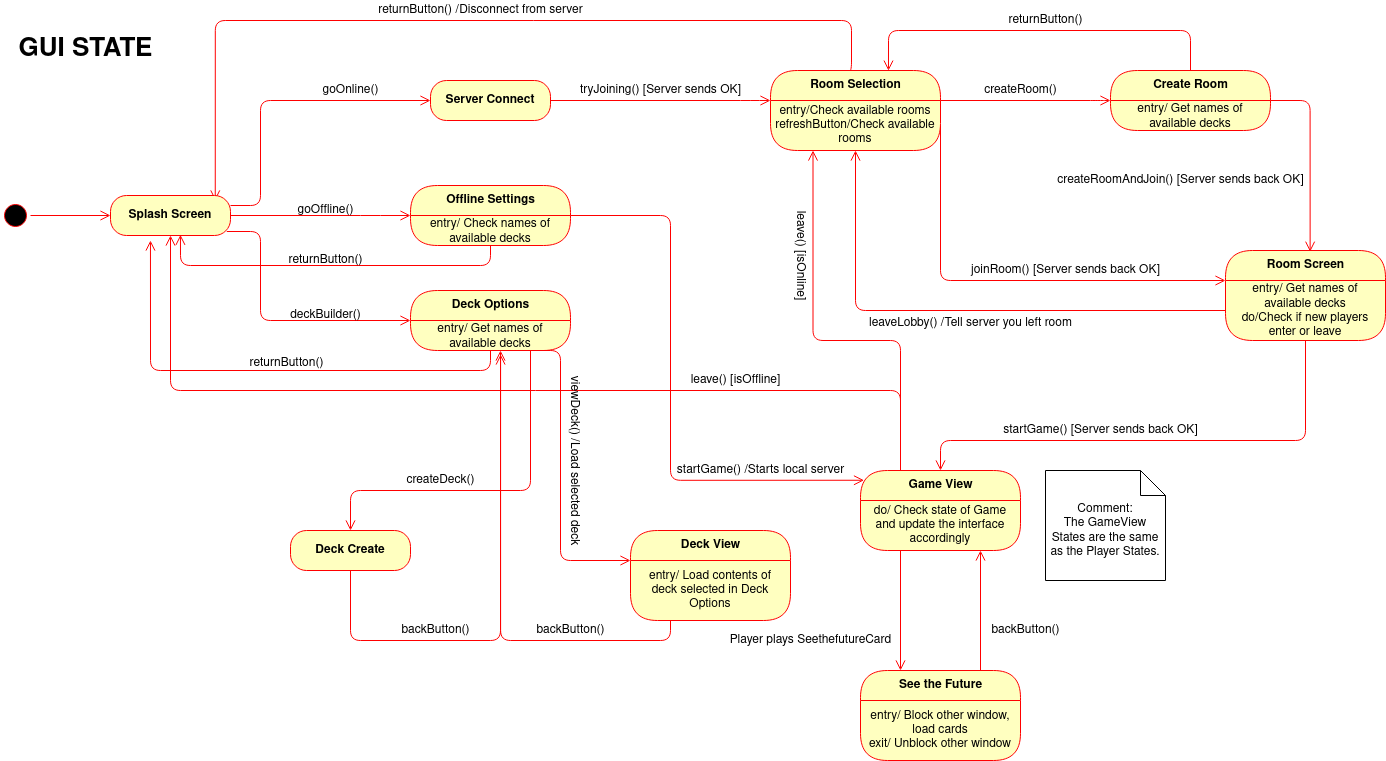
**Player state:**

**Description:**

This state represents the *Player*. When created, the *Player* object starts outside of any room (lobby). Upon joining a room, the player waits until the host starts the game. In this state the Player can at any time choose to leave the room and go back to the previous state.

When the game is initialized, the player starts playing. The Playing state is a composite state in which the player alternates between Playing his turn and Waiting for his turn. When playing his turn, if he draws an Exploding Kitten, he has to defuse it to go back to waiting for his turn. If he can't defuse it, he dies and goes into a Spectating state, where he can watch the game without interacting. He leaves when the game ends or he chooses to leave. In any of his states the *Player* object that is held by the *Server*, can terminate if the *Server* dies.

**GUI state:**



**Description:**

**Format**: the name of each class in bold, whereas the attributes, operations, and associations as underlined text, objects are in italic.

The GUI is actually composed of several **Classes** in the actual code, but that is due to the structure of JavaFX, the graphical library we use. States wise, all those *Scenes* (and relative Controllers) simply represent a State of the **GUI**.

When launched, the GUI first opens up on the SplashScreen, the State in which the user can select whether they want to play multiplayer (online), solo (offline), or they want to edit their *Decks*.

If the user selects Online, the State changes to Server Connect, where the user gets to choose what server to join and what username he wants. If the server is reachable and the username is available, the user gets to the Room selection State, where he can choose a room to join, if any exists, or create a new one. He can also update the room list while there. If the user chooses to create a room, the GUI changes to Create Room State, where the user can choose details about the *room*, including what deck among the available ones. Once selected the desired options, they can create the *room* and if the server gives the OK, they then join it. This gets us to Room Screen, where the user can wait for the game to start. This is the same State that the GUI reaches from Room Selection if the user chooses to join a Room instead of creating one.

Once here, when the host starts the game, the State goes to Game View, where the game is played. The GUI stays in this state until the user exits using the leave button (which goes to Splash Screen if playing offline and to Room Selection if playing online), and occasionally it opens a second window for the See The Future card, and then goes back to Game View upon closing it.

If from the Splash Screen the user selects offline, he goes to the Offline Settings State, where he chooses the settings for the game, which then gets launched, going straight to Game View.

If from the Splash Screen the user selects Deck, the GUI goes into Deck Options State, which allows to choose between creating a new Deck or viewing an existing one. If the user chooses to view one, the GUI switches to Deck View State, and if he chooses to create a new one, the GUI goes to Deck Create State. In both cases it’s possible to go back to Deck Options with the back button.

## Sequence diagrams

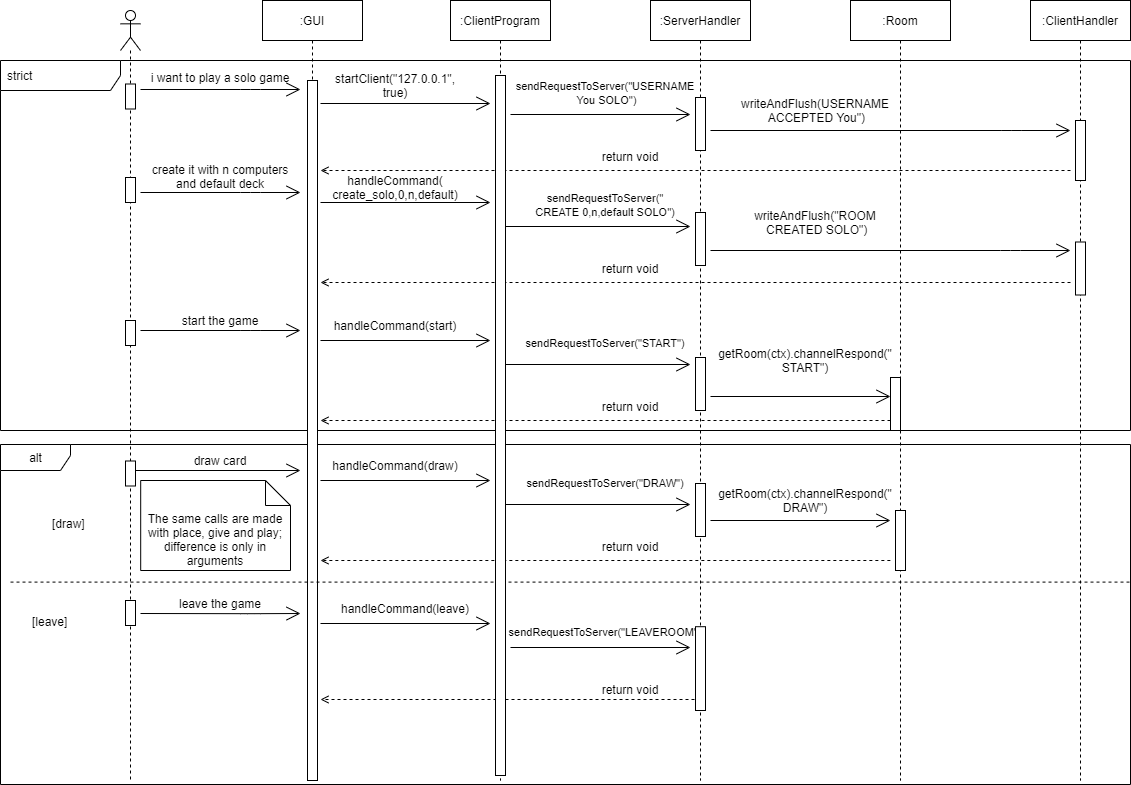
*Author(s): Adam*

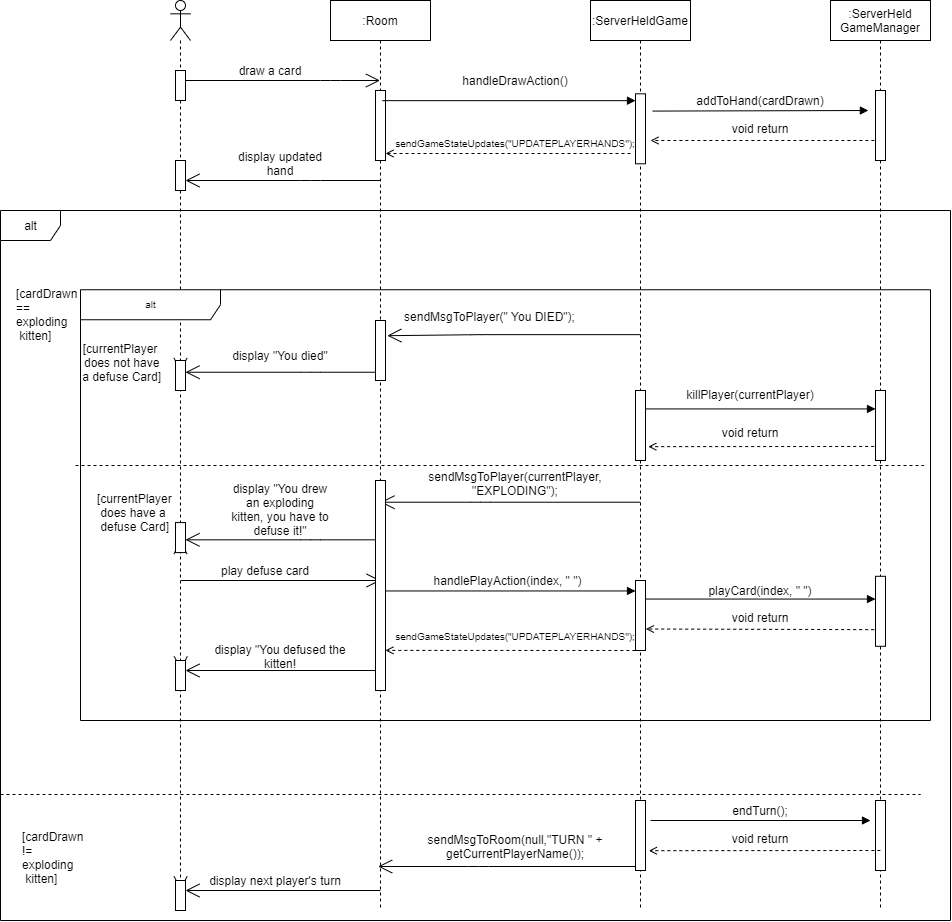
**Typical initialization and play of a solo game** *(First diagram)*

The diagram consists of 6 lifelines, namely **ClientHandler**, **Room**, **ServerHandler**, **ClientProgram**, user (Actor) and the **GUI**. The **GUI** lifeline represents all the classes/controllers that are related to the **GUI**, as using each separate class from the **GUI** would clutter the diagram. Initially, the player wants to play a solo game vs computers, to do this the user presses a button on the **GUI** labelled Play Local. This in turn calls startClient() function from **ClientProgram** with localhost as the IP and the local argument set to true. The **ClientProgram** then relays that information to the **ServerHandler** using sendRequestToServer("USERNAME You SOLO"), and the **ServerHandler** does some internal checks on the username before sending writeAndFlush("USERNAME ACCEPTED You") to the **ClientHandler**. The "You" inside the arguments is merely a placeholder for local mode on the game, whereas in the online version, the placeholder would be replaced by a username which the user would select. After the username is accepted, the user is sent to a screen in the **GUI** where the user selects the number of computers they wish to play against. That information is sent to the **ClientProgram** using handleCommand(create\_solo,0,n,default), where 0 is the number of real players, n is the number of computers and default is the deck choice (which can be modified). Then the information is sent to the server using sendRequestToServer("CREATE 0,n,default SOLO"), the server creates the room and sends a confirmation to the **ClientHandler**. The user then presses the Start Game button on the **GUI** and the **GUI** sends that information to the **ClientProgram** using handleCommand(start); the **ClientProgram** relays that information to the **ServerHandler** using sendRequestToServer("START") and since the starting of the game is the role of the Room, that information is relayed to the **Room** from the Server using channelRespond("START"). That is essentially the initialization of the game and the ordering of those operations is very important, as the server first must be started and then the room and the game must be the last thing that is started, which is why this part is under the 'strict' fragment. After the initialization phase, the user chooses his actions, as can be seen from the alt fragment. If the user chooses to draw/play or place an exploding kitten/give up a card (if he is prompted to do so), then his choice is relayed from the **GUI** to the **ClientProgram** through the handleCommand() function, which is then relayed to the server using the sendRequestToServer(), which is further relayed to the **Room**, which internally does the appropriate actions. However, if the user decides to leave, the same order of message sending applies, except the disconnection occurs in the **ServerHandler** rather than the **Room**.

**Handling of an exploding kitten** (*Second diagram)*

As can be understood from the aforementioned diagram and description, when the user does an action, it goes through a set of classes, such as the **GUI**, **ClientProgram**, **ServerHandler** and etc. In order to not obfuscate the diagram, those intermittent classes have been omitted. Initially, when the user draws a card, the action is received by the **Room**, which calls handleDrawAction() inside the **ServerHeldGame**. The **ServerHeldGame** then calls the **ServerHeldGameManager** to add the drawn card to the player's hand and then the **ServerHeldGame** returns a sendGameStatusUpdate("UPDATEPLAYERHANDS") to **Room** in order to sync the **GUI** and the rest of the code with the newly drawn card. The **GUI** then displays the new hand. If the drawn card is an exploding kitten and the player does not have a defuse card, then the **ServerHeldGame** kills the player using killPlayer(currentPlayer) in **ServerHeldGameManager** and also notifies the player of his unfortunate death using sendMsgToPlayer("You DIED") which is then displayed using the **GUI**. However, if the player does have a defuse card, then he is made aware by having a message displayed through the **GUI**, which forces him to play his defuse card. When the player plays the card, that information gets sent to the **Room**, which calls handlePlayAction() in **ServerHeldGame**, which then calls playCard() in ServerHeldGameManager. After this exchange of information, the player is notified that he has defused the card and the game status is also updated using sendGameStatusUpdates("UPDATEPLAYERHANDS"). However, if the drawn card is not an exploding kitten, then the player's turn merely ends and the **ServerHeldGame** calls endTurn() on the **ServerHeldGameManager** and then displays the following player's turn.





## 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.
* Upon running the JAR, the Gui.main() function of the Gui class is called through the Launcher.main() function of the Launcher class. When this happens, the Gui shows the main screen (splashScreenController) of the game. Because we did not host the game server online on any remote platform, we had to make both the local and online games connect to the generic localhost IP on PORT 8007.

In the GUI, when the game actually starts, we managed to handle all card interactions by using the abstract class **Card** 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()”.

In order to handle multiple rooms on one server, we implemented a list of rooms in the *server* object, which contains separate *room* objects which offer the option of clients playing on different servers at the same time.

* Running the java file:
  + For the online version:
    - Step 1: run src/main/java/softwaredesign/server/ServerProgram
    - Step 2: run src/main/java/softwaredesign/gui/Launcher

If you follow this path, both online and local can be played.

* For the local version: run src/main/java/softwaredesign/gui/Launcher

If you follow this path, only local can be played, otherwise, when attempting to play online, the connection to the server will not be realized.

* The location of the JAR:

For the game: Assignment3/out/assignment3.jar

For the server: Assignment3/out/server.jar

* The link to the demo of the final game: <https://youtu.be/L1nPZFVXA-A>

## Time logs

