D7034E Home Exam
Exploding Kittens

Author:
Tovah Parnes tovpar-5@student.ltu.se

Department of Computer Science, Electrical and Space Engineering

October 27, 2022

Contents

1 Unit testing

Which of the requirements 1-13 are currently not being fulfilled by the code?

For all of the tests the input is assumed to be correct, since inputting an incorrect option makes the program crash.

1: There can be between 2 and 5 players

There are no checks for number of players in the game. If the game is initialized with a total of 7 or more players and bots together the game exits with the error message *Index 0 out of bounds for length 0.* XXX: can this be tested?

3.a: 2 Defuse cards for 2-4 players, 1 Defuse card for 5 players

The game adds *6-numPlayers* defuse cards to the game, this is only correct if there are 4 or 5 players in the game. This is impossible to test since making the deck and dealing it to the players is done in one action, there is therefore no point that the whole deck can be tested.

10.b: An exploded Player discards the cards on hand, including the ExplodingKitten card, and may not take additional turns

If a player explodes with multiple turns left they will take the turns left but without having any cards in their hand. This is not testable without starting the game and running it.

11.a.i: The next Player takes 2 turns in a row unless that Player plays another Attack card, in which case the consecutive player has to take an additional 2 turns (e.g. 4 turns, then 6, etc.)

If a player has for example 6 turns, passes so that they only have 4 turns and then plays an attack, then the next player would take 8 turns instead of 6.

11.g: Playing three cards of any kind will allow the Player to name a card to receive from another player (if available)

Writing this option only gives the error message *Not a viable option, try again* and the player gets to go again.

2 Quality attributes, requirements and testability

Both *easy to modify and extend your software* and *easy to test and to some extent debug* are very vague instructions, leaving a lot of interpreting up to the developer. Easy in particular is up to the beholder since what is easy for a beginner and someone experienced are very different. Having the quality attributes modifiability, extensibility and testability is a good start but they need more context than only easy. These quality attributes can be implemented in many different ways that needs to be discussed further.

3 Software Architecture and code review

3.1 Extensibility

Adding functionality to a program with only one long class and very few functions is very hard to do without refactoring the whole program. Because of these few classes and functions the program has a high coupling and a lot of dependencies. To increase the insensibility the program would need to be divide into more classes, modules and functions as well as tie the behaviour of the class to the state of the program, increasing cohesion.

3.2 Modifiability

All of the code is currently very static with no variables or methods for anything. This means that to modify the code, ex change the function of a specific card, isn't straight forward. Formatting the code so that the functionality for each card is it's own method would increase the primitiveness of the program and increase the modifiability. Decreasing the cohesion of the program as well as increasing the coupling would also greatly increase the ease to modify the program.

3.3 Testability

Part of the current program could possibly be tested by observing the print outs in the terminal by playing the game but automating it would be impossible because of the random nature of the game. Using automated tests, for example JUnit here, would be impossible for the majority of the given requirements.

The only methods that does use *return* are *checkNrNope*, which returns the amount of *nope*s played, and *readMessage* that is used when playing *nope*s. Since these do use return they rcould be tested but this only corresponds to one out of the 13 requirements.

4 Software Architecture design and refactoring

4.1 Extensibility

The program uses a lot of inheritance, especially for the different card types. This is to have an abstract class Card and each individual card is implemented as an subclass with predetermined variables such as name. Adding more cards are implemented in the same way, meaning that it is easy to add cards that have a different functionality. This means that the program has a higher coupling but it means that it instead reuses a lot of code.

The game takes in an argument for which version of the game to run. Currently only *Original* is implemented but in the future this is how to choose between the original game or different expansions.

4.2 Modifiability

That the program is divided into separate classes and methods means that it is much easier to modify than the original program.

Modifying the variables or amount of cards for the game is very easy, since they are both in json files. When extensions for the game are implemented these would be implemented as individual json files.

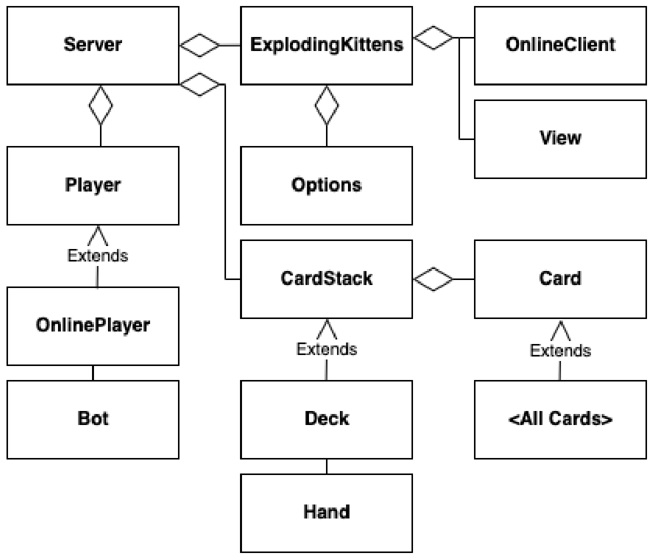
4.3 Testing

Unit testing have not been implemented in the code but the structure of the program is made so that testing is possible. For example server is what handles the game but the game in not started until the start game function is run, therefore the games individual methods can be run without starting the game loop.

The deck is initialized based on the given object file, this way the deck can be made and the number of each card can be tested for, checking off testing for requirement 3.

The only requirement that would not pass in testing is Nope, since the Nope functionality is was not completed in time.

4.4 Classes



4.4.1 ExplodingKitten

The main class that starts the program. Depending on the arguments it starts either a new server, arguments *numPlayers numBots version*, or an online client with the argument *IpAdress*.

numBots is for the number of bots in the game. Currently bots are not implemented and trying to start the game with any will throw an error.

version is a String for the version of the game that is to be started. Currently only *Options* is implemented but in the future this is how to choose between the original game or different expansions.

4.4.2 View

Contains all outputs to clients and server, as well as all player inputs.

4.4.3 Options

An object containing all the variables read from the given json file.

4.4.4 OnlineClient

Handles connecting to the server as an online clients.

4.4.5 Server

Server takes care of the game and what is happening in the game. For example, validates players inputs into

4.4.6 Card

Handles all functionaly for cards. Card is an abstract class where each different type of card is a subclass where the name and different variables are set. Each card has a *onPlay* or *onDraw* corresponding to the cards ability.

4.4.7 CardStack

Handles all types of cardStacks, where the cardStack is an ArrayList filled with Cards. Deck and Hand are subclasses of CardStack.

4.4.8 Player

Contains all the information around a player, for example their id, hand, if they are exploded and connection. OnlinePlayer and Bot are subclasses of Player.

4.5 Naming Conventions

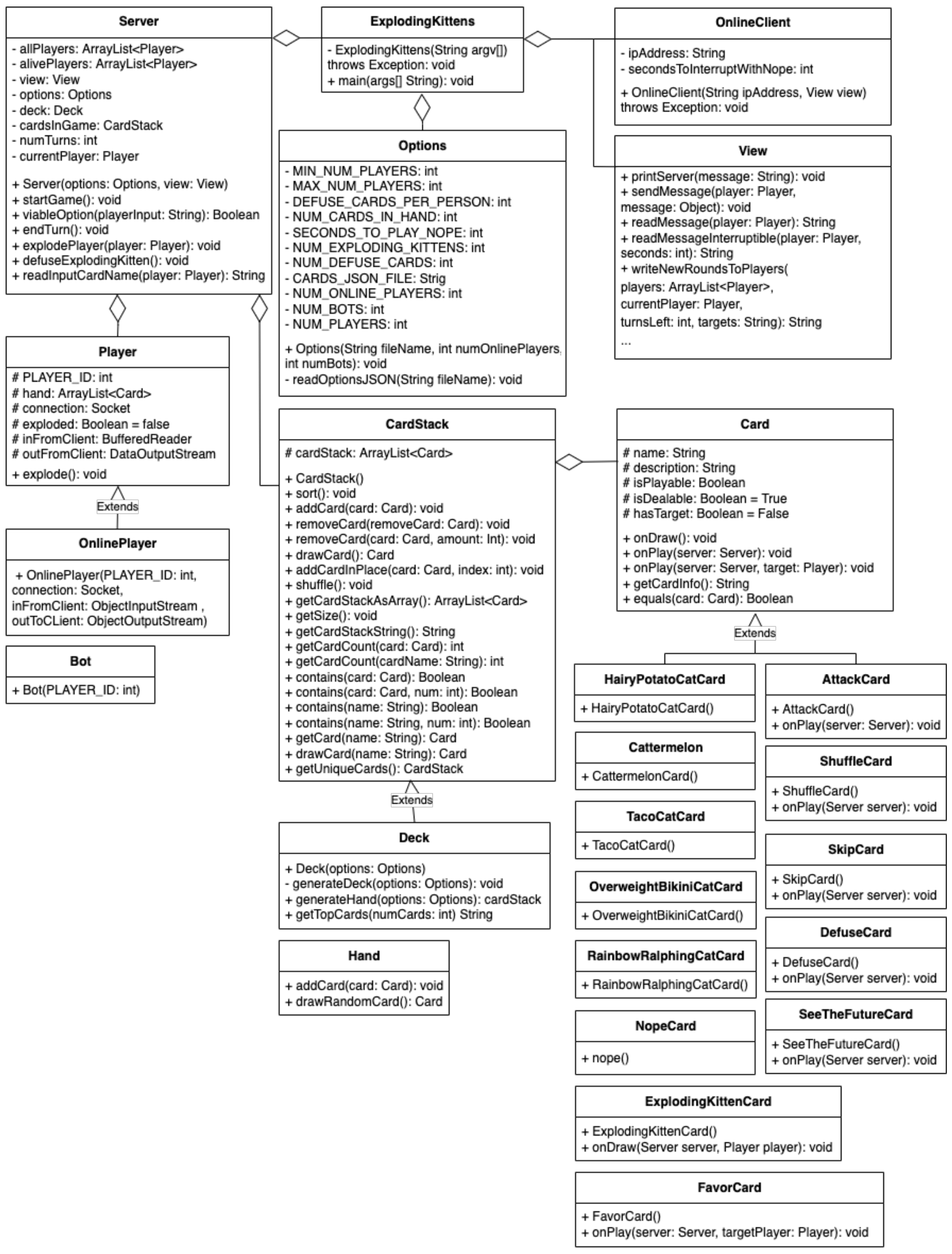
4.5.1 View

In the class view the naming convention is that methods starting with *write*, for example *writeNewRoundsToPlayers(players: ArrayList¡Player¿, currentPlayer: Player, turnsLeft: int, targets: String): String*, writes someting to one or multiple players as well as the server. Methods with the prefix *print*, for example *printServer(message: String): void* prints a message to the server and not any player.

4.5.2 CardStack

In the class cardStack the naming convention of *drawCard(name: String)* returns the card and removes it from the cardStack while *getCard(name: String)* only returns the card while not changing anything in the cardStack.

4.6 UML Diagram


Server
- allPlayers: ArrayList<Player>
- alivePlayers: ArrayList<Player>
- view: View
- options: Options
- deck: Deck
- cardsInGame: CardStack
- numTurns: int
- currentPlayer: Player
+ Server(options: Options, view: View)
+ startGame(): void
+ viableOption(playerInput: String): Boolean
+ endTurn(): void
+ explodePlayer(player: Player): void
+ defuseExplodingKitten(): void
+ readInputCardName(player: Player): String
ExplodingKittens
- ExplodingKittens(String argv[]) throws Exception: void
+ main(args[] String): void
OnlineClient
- ipAddress: String
- secondsToInterruptWithNope: int
+ OnlineClient(String ipAddress, View view) throws Exception: void
Options
- MIN_NUM_PLAYERS: int
- MAX_NUM_PLAYERS: int
- DEFUSE_CARDS_PER_PERSON: int
- NUM_CARDS_IN_HAND: int
- SECONDS_TO_PLAY_NOPE: int
- NUM_EXPLODING_KITTENS: int
- NUM_DEFUSE_CARDS: int
- CARDS_JSON_FILE: Strig
- NUM_ONLINE_PLAYERS: int
- NUM_BOTS: int
- NUM_PLAYERS: int
+ Options(String fileName, int numOnlinePlayers, int numBots): void
- readOptionsJSON(String fileName): void
View
+ printServer(message: String): void
+ sendMessage(player: Player, message: Object): void
+ readMessage(player: Player): String
+ readMessageInterruptible(player: Player, seconds: int): String
+ writeNewRoundsToPlayers(players: ArrayList<Player>, currentPlayer: Player, turnsLeft: int, targets: String): String
...
Player
# PLAYER_ID: int
# hand: ArrayList<Card>
# connection: Socket
# exploded: Boolean = false
# inFromClient: BufferedReader
# outFromClient: DataOutputStream
+ explode(): void
Extends
OnlinePlayer
+ OnlinePlayer(PLAYER_ID: int, connection: Socket, inFromClient: ObjectInputStream , outToCLient: ObjectOutputStream)
Bot
+ Bot(PLAYER_ID: int)
CardStack
# cardStack: ArrayList<Card>
+ CardStack()
+ sort(): void
+ addCard(card: Card): void
+ removeCard(removeCard: Card): void
+ removeCard(card: Card, amount: Int): void
+ drawCard(): Card
+ addCardInPlace(card: Card, index: int): void
+ shuffle(): void
+ getCardStackAsArray(): ArrayList<Card>
+ getSize(): void
+ getCardStackString(): String
+ getCardCount(card: Card): int
+ getCardCount(cardName: String): int
+ contains(card: Card): Boolean
+ contains(card: Card, num: int): Boolean
+ contains(name: String): Boolean
+ contains(name: String, num: int): Boolean
+ getCard(name: String): Card
+ drawCard(name: String): Card
+ getUniqueCards(): CardStack
Extends
Deck
+ Deck(options: Options)
- generateDeck(options: Options): void
+ generateHand(options: Options): cardStack
+ getTopCards(numCards: int) String
Hand
+ addCard(card: Card): void
+ drawRandomCard(): Card
Card
# name: String
# description: String
# isPlayable: Boolean
# isDealable: Boolean = True
# hasTarget: Boolean = False
+ onDraw(): void
+ onPlay(server: Server): void
+ onPlay(server: Server, target: Player): void
+ getCardInfo(): String
+ equals(card: Card): Boolean
Extends
HairyPotatoCatCard
+ HairyPotatoCatCard()
AttackCard
+ AttackCard()
+ onPlay(server: Server): void
Cattermelon
+ CattermelonCard()
ShuffleCard
+ ShuffleCard()
+ onPlay(Server server): void
TacoCatCard
+ TacoCatCard()
SkipCard
+ SkipCard()
+ onPlay(Server server): void
OverweightBikiniCatCard
+ OverweightBikiniCatCard()
DefuseCard
+ DefuseCard()
+ onPlay(Server server): void
RainbowRalphingCatCard
+ RainbowRalphingCatCard()
SeeTheFutureCard
+ SeeTheFutureCard()
+ onPlay(Server server): void
NopeCard
+ nope()
ExplodingKittenCard
+ ExplodingKittenCard()
+ onDraw(Server server, Player player): void
FavorCard
+ FavorCard()
+ onPlay(server: Server, targetPlayer: Player): void