Exploding kittens

A home exam in the course d7032e

Magnus Stenfelt, magset-8@student.ltu.se

[2022-10-26]

1. Unit testing

Requirements that are faulty within requirement 1-13 are:

Requirement 1:

- You can have too many players and bots. But when playing the first turn the game becomes index out of bound and ends the game. This can happen because there is no check for the amount of players created.
- It is not possible to test the requirement without changing the code. This is because there is no method with a return value to use assertEquals with. There would have to be a method returning the number of players, a message of succeeding or a list that can be used for testing the size.

Requirement 3a:

- There are the wrong amount of defuse cards and the amount is based in a faulty way on the players.
- There is no function that returns the deck and therefore there is no way to test how many of each card there is in the deck. It has the same problems in testing as requirement 1. A get method of the deck and hands would have helped.

Requirement 10b:

- If you have extra turns in play and a player explodes their hand gets emptied but they can keep playing their turns and when their turns are done the game ends.
- Tricky to test normally but even harder when the parameters for an exploded player are within one function without return functionality. The big block of code does not allow for testing.

Requirement 11:

A: Attack to player 1 gives 2 turns, attack to player 0 gives 2 more and equals 4 turns. If then player 1 passes once and plays attack again it should be 5 turns but is 6 turns. So it adds one too many in that case.

b: works

c: works

d: works

e: works

f: works

g: prints out index 3 out of bounds for length 3 and then stops the turns for all players.

- These are not able to be tested without changing the code. This is because they are all if-else cases nestled into one big method. Not having separate methods to test the different values given by the game or having it modularized enough to test them separately makes it impossible.

2. Quality attributes, requirements, testability

Requirements 17,18:

Both of these requirements say that it should be *“easy”* and *easy* is a very bad word to use while describing a requirement. It is not specific at all and is subjective depending on the person who reads it. There is a big lack of *specificity* in general for the two requirements and there is nothing describing how it will be tested or how it will be modified/extended in the future. For example, saying “every requirement should be testable” would be more productive. With the requirements 17 and 18 it is also not possible to test them without having the customer themselves give input. Without clear parameters as given in the example above the developer will not be able to meet the requirements of the customer and generally have a hard time developing the game.

3. Software Architecture design and code review

Extensibility:

The key factors in a system that has extensibility is high ***primitiveness***, high ***modularity***, ***cohesion***,and low ***coupling***. This is because having the parts of the software be in a few big modules, poorly separated and highly connected does not allow for easy extensiveness.

Almost the entire game's functionalities are within the main method. They are not reachable separately and a developer would be forced to input the code directly into the ***one main module***. The high ***coupling*** it has also makes it so that if you change one thing you will most likely have to change more parts that are nestled within that part. There is also no ***cohesion*** in most of the methods which means that if you want to use one part in your new code you would have to use all of the different parts of the method. In the code more small methods would help introduce new cards.

Modifiability:

For modifiability there is a need for the software to have low ***coupling***, ***primitiveness***, ***cohesion*** and ***completeness***. This is because it helps to have the parts in the software be small and have on function in one method, most methods should do one thing and be easy to reuse.

This code however fails this in several ways. It has few if any small ***primitive*** methods with most of its functionality nestled in if-else cases within one main method. This is very hard to modify and change due to one change needing to assess the bigger scope of the method. The ***cohesion*** has a similar problem due to not everything being cohesive means that the developer who wants to modify has to change the main method a lot. There is no ***completeness*** within the software which makes it impossible to reuse code. This all together makes it very difficult to change the software.

Testability:

Sufficiency, primitiveness and ***cohesion*** are the main qualities that make up a softwares testability. If the softwares methods are ***sufficient*** enough in each use case, *primitive* enough to test the different parts individually, hold one function each and also is ***unittestable*** with return variables then it can be easily tested.

This code that is given has 3 return statements with 2 in readMessage and one in checkNrNope. These are obviously not enough to test the entire softwares functionality. With almost all code within one function there are no separate methods to test for the different values that are needed. This means that there is no way to have ***sufficiency*** in the software because there are almost no methods to be ***sufficient***. There is no ***primitiveness*** due to most of the code being in one single method making it hard to test separate parts. The ***cohesion*** is not there in the functions which makes it impossible to test separate parts.

4. Software Architecture design and refactoring

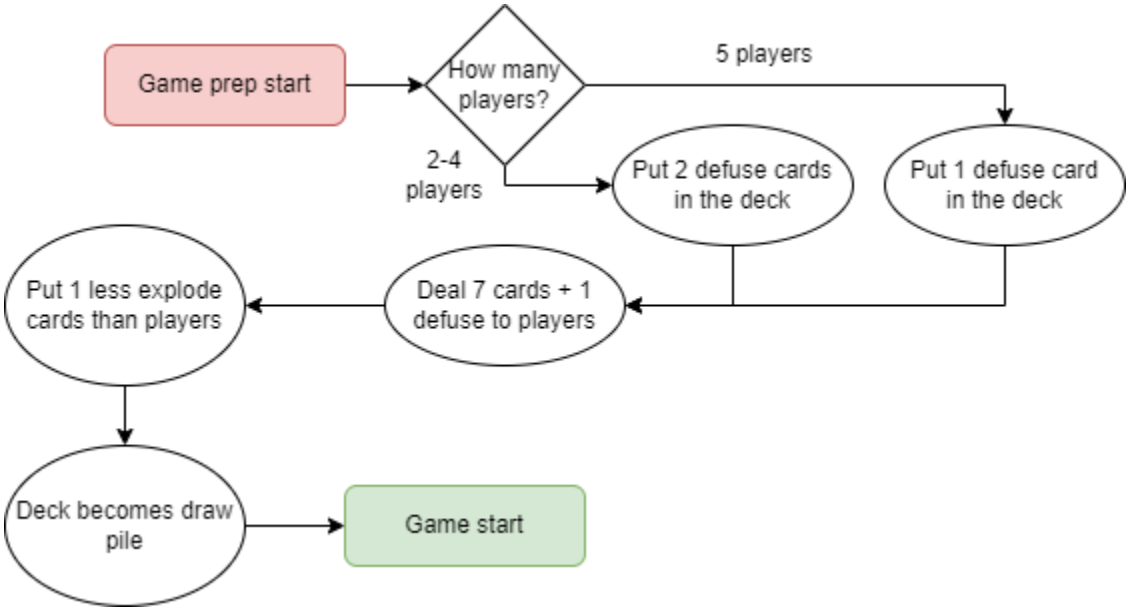

Figure 1: Pre game state diagram

Figure 1 pre game:
Before the game even starts there are a few things that need to happen. When the game is started the amount of player slots is chosen. That amount of slots is the amount of clients and bots that can be connected to the game session. Based on the amount of players+bots that are connected to the game the cards in the deck vary. Then every player gets 7 cards from the deck and 1 defuse card. After that the explode cards are added so that players don't immediately pull it.

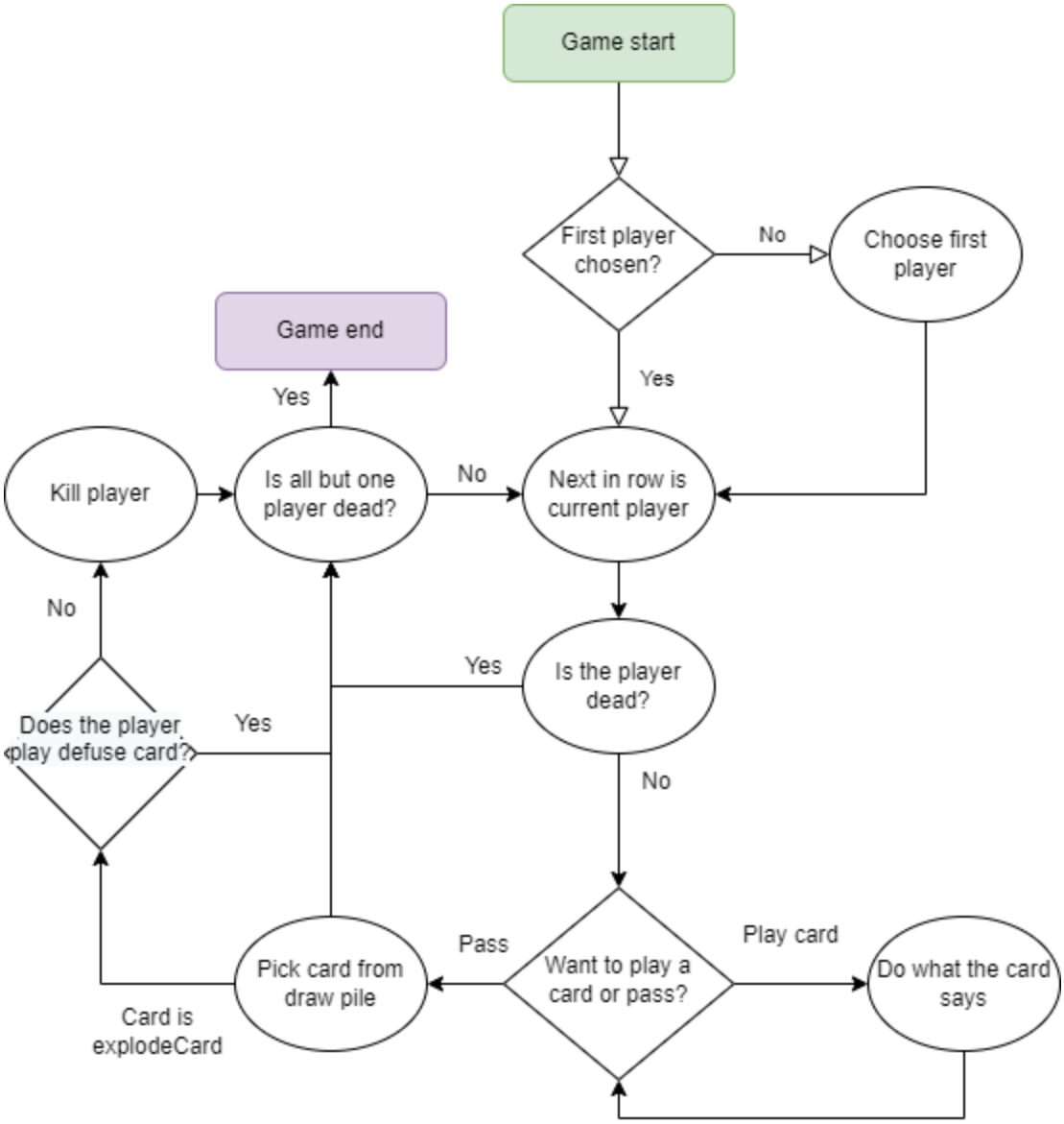

Figure 2: Game state during play

Figure 2 Game state during play:

The first player that is chosen is supposed to be random but I think the dev should be able to easily change that to a specific player. From that point on the game will be in a loop. The game checks if the current player is dead and if not it lets them play as many cards as they want. Most cards have an effect or a combo effect that affects the other players or where in the loop you are. For example, the skip card skips the “pick card” part in the diagram above. There is also continual checking if there is a winner from all players being dead. The card effect and small minutia is not included due to creating a too complex diagram.

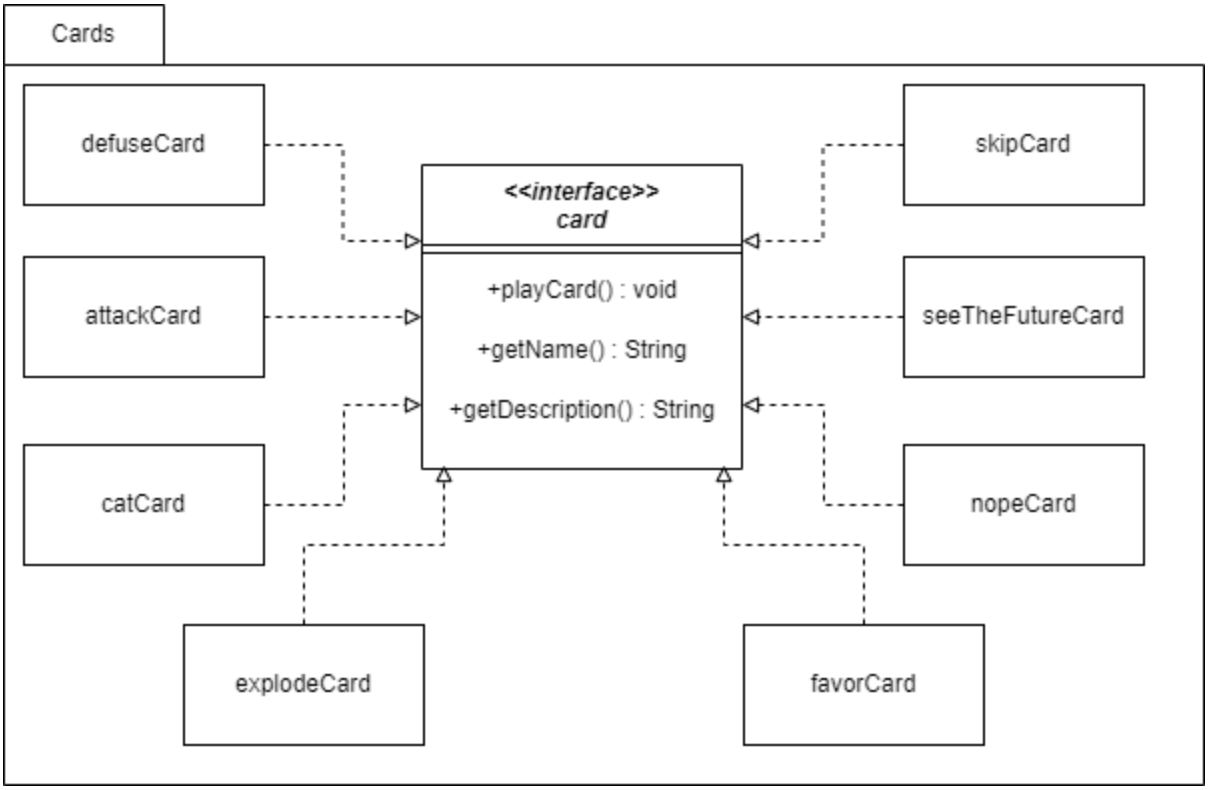

Figure 3: UML over the Cards

Figure 3 The card structure:
For the card structure used in the software I have chosen to go with an ***interface*** style after discussing the best way to go about it with my classmates. It works through having a very general structure with name, description and play card. Name and description are just the strings with information about the card. The card plays itself and therefore no other part of the game is needed to know the card functionality. This helps with future ***extensibility and modifiability*** because its able to separate the parts and make it easy to change or implement features in the future. The cards are objects and the amount of cards is done with a key - int hashmap with which we are able to create new expansions more easily. The cards use help functions in gameState and other classes that can be easily tested.

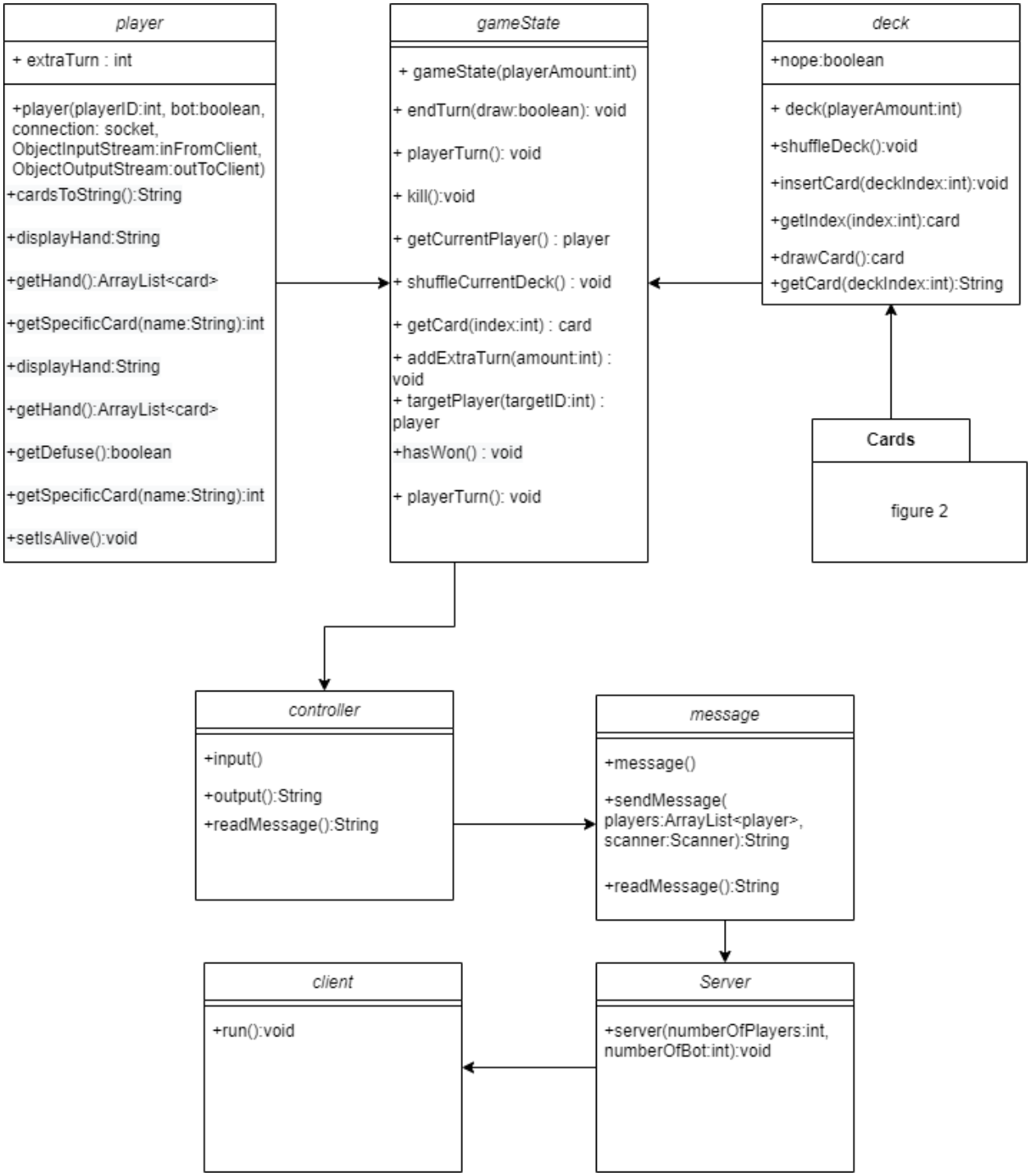

Figure 4: UML over entire exploding kittens

Figure 4 UML of everything:
This is the general UML diagram of the entire software however there are a few smaller parts left out. The main three parts are the player class, gameState class and the deck. These build up the actual game. The deck holds the functions to change the deck and is called often from gameState. It also initializes the deck with cards from the card interface explained in Figure 3. The players are initialized and created for each client that is created when the server starts. Most of the functions used by the cards are made to be general and ***easy to use and change*** in future implementations. The ***primitiveness*** of the methods help with it being understandable and the ***cohesion*** within them.

The connection for the players in the game is done with clients that connect to a server. The server has clients that connect to it but the server is the main hub of functionality for the communication between clients. The messages are sent through message class to the server and received from the server to message and shown through the controller.

Generally the whole philosophy of the structure is ***low coupling, high primitiveness*** and keeping the methods ***cohesive*** with themselves. Making sure it is ***modularized*** enough so that it is easy to extend and modify.

How will I make sure that the project works with the given quality attributes? Especially 17 & 18?
Before even starting coding the game there was much time set aside to design and decide how to test all the requirements which I hope will help with unit testing later. Through having that in mind while coding I will be able to write code that will be easily tested for third party software like jUnit. This will also likely end up separating the code into smaller parts and functions and therefore be more understandable code.

Separating the code will also help with making it ***modifiable*** and ***extensible*** through making it more ***modularized***. By making sure that there is high ***cohesion*** within the different parts there will be easy to change without changing unintended features. The ***extensibility*** will be reached by having the different function be able to slot in or slot out of the game. ***Completeness*** and ***sufficiency*** of the game will be reached by making sure that all of the parts fill their use case. Small ***modules*** will help with this.

What design patterns have been used in my design?
I am not familiar enough with design patterns enough to intentionally and competently use them in my game. I do understand that they are made to simplify development but for an unseasoned coder they have a tendency to overcomplicate things.