Report — Anne-Greeth van Herwijnen

Introduction

This report summarises the development of the GO game for the Nedap University module 1. The code can be found here. This report first discusses some of coding problems and design choices, then it discusses the process in general and concludes with some future work and some what if I could do it again.

Coding problems & Design decisions

Axis

I've had some issues figuring out the way the axis worked and how my array was filled. The calculation for the translation between indices and coordinates was correct, however it did not seem to work correctly since the array was filled left to right instead of top to bottom. I wrote some hardcoded moves to finally find out en check the way the coordinates and indices worked.

Stone: Class or Enum

The object Stone can be implemented in different ways and depending on your implementation your stone can be either smart or dumb. I started with a "dumb" stone, being just an Enum! An Enum allows for easy access on the board. You can also implement functions in an Enum, however if you implement a lot of functions in a enum you might want to make it a class, or at least a really smart enum. I tried that approach, because my board got extremely smart and I thought it might make sense for a stone to know it's liberties and know when to remove itself. Partially due to the recursion problems (as mentioned below) and the fact that it didn't really solved any problems I decided to revert this decision, since it's not really relevant for a stone to know all these things, it's more relevant for the game or the board.

Board vs Game

As mentioned above which class knows what is the tricky part of designing an application like this. One major choice was regarding the classes Game and Board. In the beginning my Board-class was extremely smart, and I still think that the Board-class could use a good review to decide if all the functions should stay the way the are, especially functions like: `getLiberties` and `getChain` . Since the function `getTerritoryScore` is a part of the Game class, you could argue that the previous two should also be part of the Game class instead of the Board-class. I decided that the Game-class needed that last function because the game needs to calculate scores etc. and a board does not need to concern itself with those kind of game mechanics. However the board needs to remove the stones and set stones if that is valid, so in my opinion the board should be able to calculate chains, neighbours and liberties.

Storing data

One of the difficulties with programming a game like this was for me storing the data, such as boards (for the Ko-rule), people who want to play games etc. Especially since storing data in a certain way, for example an LinkedList makes sometimes senses, but sometimes a normal array is perfectly fine. I decided to store the games as well as the people searching for a game in two places, because of easy accessibility.

```
public void addToGameLobby(Integer dimention, ClientHandler t) {
        gameLobby.put(t.getClientName(), dimention);
        if (!dimMap.containsKey(dimention)) {
            dimMap.put(dimention, new LinkedList<>());
        }
        dimMap.get(dimention).add(t.getClientName());
        t.sendMessage(Keyword.WAITING.toString());
        int key = checkForPair();
        if (key >= 0) {
            startGame(key);
        }
    }
```

I store the players with there dimension in a simple HashMap, but I used a <Integer, Set> HashMap to store the players that want to play on certain dimension. This might be a bit overkill since there should never be more then two players in that set, however this makes it convenient to check if there are two players with the same dimension.

Recursion errors

There are multiple ways to solve the problems considered with Go and especially considering the liberties and chains. As I started thinking about the problem of liberties and chains I immediately thought of a recursive solution. In my opinion the way the problem is repeated if a stone with the same color is found just looked a lot like recursion for me. However, the challenge with recursion is to make sure there are no infinite loops. Even though I thought I made a smart decision to keep track of the field where the chain came from I still got into trouble. I decided to keep the chains in a set because sets only contain unique parts, so it's easy to check whether a function already had been where it wanted to go again. In the beginning I made a starting function and a more elaborated function for the recursive part, looking something like this in code:

```
public Set<Integer> getLiberties(int x, int y, Stone s) {..}   private Set<Integer>
getChainLiberties(int x, int y, Stone s, int prevX, int prevY) {..}
```

The second function would be called from the within the first. I reduced this to one function by merging the two into `getLiberties(int x, int y, Stone s, Set<Integer> liberties)` in which the already collected liberties could be passed into the next call and I did the same for the chain option. I simplified this when I noticed that the liberties functioned the same as chain and liberties are just a check of the neighbours of a chain, so I removed the recursion from the `getLiberties`

and made a function that as able to check all the neighbours of a chain and use this in the liberties-function.

Process evaluation

My own process

I started by building a board and a random player, this allowed me to test the board and certain behaviour easy. However this also made the need for tests lesser which lead to problems later on. Also by starting with the board my board became rather smart, is discussed above. The first major point I learned from this process was to commit regularly, especially after implementing and testing a certain feature. Committing ensures that you always have a working version on your GitHub and makes it easier to see why it suddenly stopped working. The second thing I learned, partly related to what was mentioned before is that you should always be careful when you (un-)comment code because this can lead to strange bugs if you (un)comment too much or too less. The third thing, was the fact that you should write tests. Of course this was something that was thought during the module and something you keep in mind, but sometimes it seems easier to hardcode certain steps or cases into your main function, however, this can lead to the problems previously mentioned. Even though I started to use more and more tests, this also did not always go the way I wanted it, due to some poor copy and pasting. Since the tests often required multiple moves I copy-pasted those, which let to copy paste errors on multiple occasions.

The protocol

One of the most tedious things of this project was the protocol (can be found here). I really liked the fact that there was a special student assistent to do the university protocol session with us. However, due to circumstances Martijn en Mark could not attend this meeting, so we scheduled a meeting on the first day of the project to go over the protocol and maybe make some minor changes based on the requirements of the project. This meeting was relatively difficult because most of us agreed on the protocol as discussed with the student assistent but Martijn and Mark had some remarks. We changed the protocol but during the project we needed to change some more and discuss some more and this was sometimes difficult because people started implementing the protocol and changes meant that they needed to revert work. Luckily, because I was sick, that wasn't me, but it made the discussion sometimes tricky. In the end there is a protocol everybody did more or less agree too.

The guide

Something that created some problems during the project was the guide provided by the coaches. This guide was partly the same as the guide that was used by the university, which created some confusion. Some of the requirements were university-requirements and not Nedap requirements. Besides this, some rules and requirements were open for interpretation, such as the Ko-rule and the pass rule. When we asked the coaches about these things they had to discuss or think of a

solution even though most of the questions should have been decided before we started the project. Overall I got the impression that the coaches not always knew why they wrote (or copy pasted) parts of the guide.

Future work

In general, I'm happy with the end result however there were some moments along the way I wished I would have done things different. I think that I might have been useful to reduce the legacy from the TicTacToe game and start by creating a kind of class diagram to get an overview of the whole system, since it was sometimes hard to keep track of everything. I think the overall structure of application makes sense: GUI, communication, game, player and test in different packages. Game and player used to be in the same package up till the end, but I decided to put the players into their own package because they are different from the board or the stones. If I had one more week (or maybe two weeks more) I would review all the code, see if there is a lot of redundancy and inconsistency and improve those. The same with the used data structures, I mostly chose those that seemed convenient however there might be better, more efficient ways to store certain objects. I think I would love to mirror my work with that of my fellow students to learn from their ideas, because I think you can learn a lot from this process on your own, but it's hard to think outside of the box when you have thought of the most convenient solution for yourself.

Artificial Agent

Unfortunately I did not have the time to implement a better AI. I would have loved to make the AI a bit smarted by using the getLiberties function to be at least better in defending my stones. I also would have considered refactoring the way the players now interact, because I'm not really using the players at the moment and am by passing them with the doMove function.