Lab 4 - Object-Oriented Software Development / ... with Design Patterns TOMK18 / TOUK18 - H22

Design with GRASP, implementation in Java
last presentation opportunity: Thursday, 13 October
submission deadline: Friday, 14 October, 23:59

Jönköping University, School of Engineering
Tuesday, 20 September 2022

General instructions

In this lab we develop a design for board games, starting with an easy tic tac toe. We let us guide by design principles (mainly GRASP) and then implement the design in concrete Java code. It will give us a beautiful tic tac toe game that is extendable, easy to maintain, reuse and change. Relevant background for this lab is lecture 3,4,5 and the corresponding readings. If you want some information about the tic tac toe game or board games, please consult wikipedia.

1 A terrible tic tac toe design

Somebody has already programmed the tic tac toe game. Check out the file *TerribleTicTacToe.java* on Canvas (*Canvas→Modules→Labs*). You can also import the eclipse project *TerribleTicTacToe.zip* into eclipse (*File->Import...->General->Existing Projects into Workspace, Select archive file*). You may run the game and verify that it works as tic tac toe should work. But the code is obviously terrible.

Task 1: find at least four design principles or patterns that this code design lies at odds with.

2 Redesign tic tac toe

Design principles such as *Controller* (GRASP) suggest us we should divide and de-couple the UI from the inner working of our system. A good staring point to do this is the Model-View-Controller (MVC) pattern (see previous lab, lecture 5 and literature). Our system lies in Model, the GUI in View and the Controller mediates between them. The question is: does this solve all our problems, are we done with the redesign?

Think in the categories of extendability, maintainability, reusability, change. What if we want to provide a different GUI for each player? What if we want to support players on different devices (via a network connection)? What if we want to extend the size of the board? What about variants of tic tac toe (confer wikipedia)? What about different rules

to place pieces or symbols on the board? Multiple boards? What about different winning conditions (other than three-in-a-straight-line)? What about more than two players? What if we want to provide better information, instructions and feedback to the players? What about information exchange between players (chat)? What about introducing a score or points?

From all these questions we realize: a design that allows for easy and quick implementation of all these variants and extensions can not consist of one single Model class. Morover, the desired design would pretty straight forward allow to implement any kind of board game (we anyway allow to change the rules, number of players, ...).

How do we subdivide the Model? A look at the domain tells us about the two different concepts of Player and a Board. It makes sense in terms of responsibilities: Players do moves on the Board, the Board provides the infrastructure and represents (at least a part of) the game's state. As we think about different types of rules and changing rules: who should take care of the rules in terms of knowing the rules and making sure they are followed/executed? *Information Expert* does not help us a lot, who has the information? Not really anybody. What about the player? But which player? All together? A MasterPlayer? All nonsense. What about the board? No good choice in terms of *High Cohesion* and *SRP*. We turn to *Pure Fabrication* and fabricate us a RuleEngine.

Are we done? Let us analyze what the different classes' responsibilities are and how the essential process of making a move works. The View should graphically present the board, so it should know a little about the board (e.g. dimensions, how to display pieces). The View should also register clicks on units of the Board and forward those to the Controller. The Controller should hand it to a Player and it should become a Move (another class? or are we exaggerating?). The Player hands a move request to the RuleEngine. If valid, the RuleEngine executes the move and does the necessary changes on the Board. The RuleEngine determines whose turn it is next and if there is a winner (and mabye computes a score and updates the Player's score). The RuleEngine should probably also inform the Player(s) about changes on the Board, so that the View can be updated.

That does not look too bad. But we might not be done, yet. We come therefore to your next tasks.

3 Main Task

Task 2: Draw a class diagram for this design suggestion.

Task 3: State each classes' responsibilities clearly.

Task 4: Does it make sense? Refine the design so you can start implementing it (this can involve adjusting responsibilities, but also adding/removing classes).

Task 5: Implement it!

The requirements on your results from task 4 and 5 are the following:

1. The design is compatible with GRASP and OO patterns and principles.
2. Have a correctly working tic tac toe game.
3. It is easy to switch between one GUI for all players or one GUI for each player.

4. It is easy to extend the board size.

5. It is easy to change the rules so that more than three in a line are required to win.

6. It is easy to have more than two players.

By "*it is easy to...*" we mean that only a few lines of code need to be changed, not the whole design.

4 Hints

- There are a number of choices still to make, e.g. who is responsible for the transformation from a click on a unit to a move? Do we need a Move class? Do we need a (Board-) Unit class? Do we need a Player class? Make your choices in accordance with the design principles.

- Another important question is: who creates all these classes? And who creates the links, so they can use each other's functions in the way they need to? Create only links that are absolutely necessary (keep coupling low!). Regarding the creation, many classes need links to exactly two other classes. Having a class responsible for all the creation doing things like e.g. `controller = new Controller(Player, View)` does therefore not work, because also the Player and the View each need links to the controller. It is the *chicken or egg dilemma*. If you figure out one object that starts from inside its constructor the creation chain of others, you might still find an elegant solution, avoiding rather ugly things like `controller = new Controller(); view = new View(); controller.setView(view); view.setController(controller);`

- Don't overdesign. At some point start implementing. On the way you will detect problems or aspects that you have not thought about. You may need to revise or adjust your design. This is completely normal.

- In the View class you could draw the board by buttons with something like this.

```
for(int r = 0; r < rows; r++) {
    for(int c = 0; c < cols; c++) {
        final int _r = r;
        final int _c = c;
        JButton button = buttons[r][c] = new JButton();
        button.setPreferredSize(new Dimension(50, 50));
        button.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                controller.unitClicked(_r, _c);
            }
        });
        myButtonPanel.add(button);
    }
}
```

5 Deliverables and presentation

You shall deliver *one* pdf document that contains answers to tasks 1-3. Your final design from task 2,3,4 (class diagram and responsibilities) needs to match your code from task 5. Task 5 shall be submitted electronically. Either in a zip-file containing several text files, or as a zip-file generated by exporting your project from eclipse (*File->Export...->General->Archive File*). Before you submit your two files (pdf and zip) on Canvas under the assignment "Lab 4", you need to orally present it to your lab assistant. Be prepared to answer any question regarding your solutions, in particular be prepared to justify your design choices with the principles. If you want to draw by hand, this is accepted, as long as it is **well-readable** and submitted within the pdf (scann it).