Mittuniversitetet

MID SWEDEN UNIVERSITY

DT180G, Object Oriented Programming I

Laboration 1

The purpose of the laboration is to provide practical experience in regards to…

- … conforming to specifications as stated in class diagrams!
- … working with objects within class hierarchies!
- … Strategy Design Pattern!

Table of Contents

Laboration 1

Before starting to work on the laboration you should have studied the module's belonging lecture material and read relevant chapters in the course literature.

The laboration consists of a single assignment and you'll need to present the solution and its corresponding document in your student repository. You find some inherent files for this assignment under `Laboration_1/src/main/com/dt180g/laboration_1/` and will need to add an additional three documents:

Inherent Documents	Needed Documents
`Lab1.java`	`ToolRock.java`
`Game.java`	`ToolPaper.java`
`Player.java`	`ToolScissors.java`
`Tool.java`	
`README.md`	

The use of proper file structure will be enforced and you may not change the names of folders or files. Your code implementation needs to be placed within designated source file and the `README` needs to contain information about how the solution has been implemented as well as the author's personal reflections regarding the assignment. Further details regarding the structure of this document can be seen in the study guidance.

Before doing anything else make sure you have installed **Java, Git** and **Maven** on your system, as well as a suitable work environment as described in **Module 1**. Once you are ready to begin work on this assignment, start by **cloning** your designated student repository to your local system. There are different ways of working with *git repositories*, but we'll assume that you use a command line / terminal for these purposes. Depending on whether you have set up **Secure Shell Authentication** for your Bitbucket account, you can either use `SSH` or `HTTPS`:

```
# clone using SSH
git clone git@bitbucket.org:miun_dt180g/studid_assignments_vt23.git

# clone using HTTPS
git clone https://USERNAME@bitbucket.org/miun_dt180g/studid_assignments_vt23.git
```

You replace `studid` with your own student id and, in case of `HTTPS`, `USERNAME` with your own Bitbucket credentials. You can also find both of these clone commands, specific for your account, by visiting the repository from a browser. You'll find a `Clone` button at the top right corner of the window, where you can shift between `SSH` and `HTTPS` to see corresponding git-command for cloning.

Assignment Description

The game **Rock, Paper, Scissors** is a classic method for determining superiority between two individuals. The rules are quite straightforward: two players face off in a competitive battle, selecting one of three possible options each round. **Rock** triumphs over **Scissors**, which in turn defeats **Paper**, which subsequently prevails against **Rock**. Typically, the victor is determined in a best-of-three series, or as we will implement in this assignment, the first player to achieve three victories.

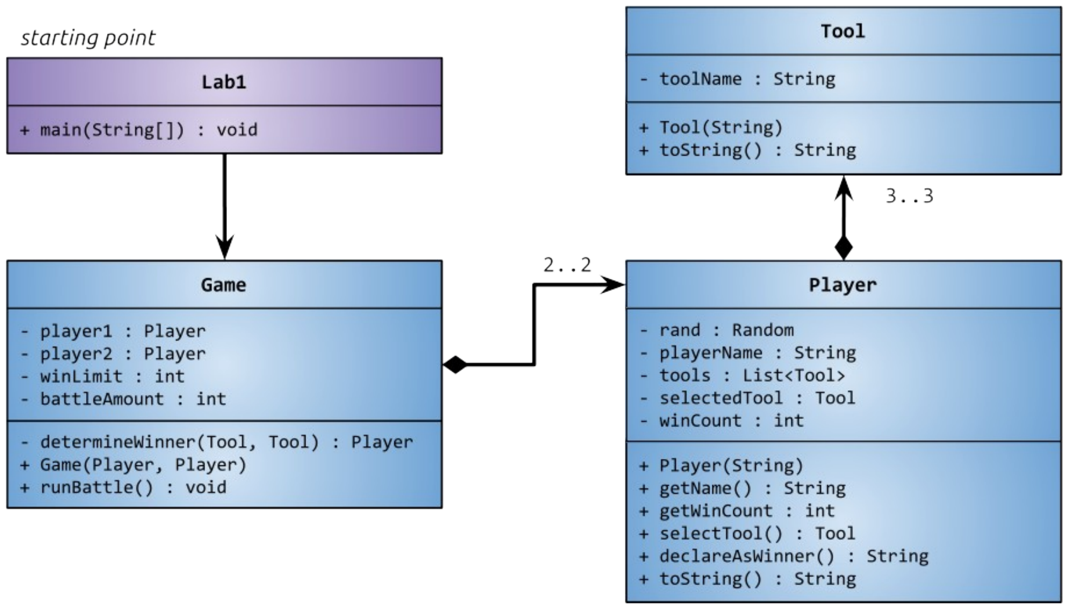

Diagram 1: inherent classes

The above diagram depicts the class structure and relationships in the current solution, with relevant documents available in your repository. `Lab1` serves as the starting point, creating an object instance of the `Game` class, which includes two `Player` entities, and initiates the battle simulation. `Game::runBattle()` calls itself recursively until one of the players reaches the win limit, which is set to 3 by default. In each round, both players select one of the three available options: **Rock**, **Paper**, or **Scissors**. This selection is randomized, as demonstrated in `Player::selectTool()`. If both players choose the same option, a draw occurs, and they must repeat the selection process. Once valid choices have been made, the winner is determined by `Game::determineWinner(..)`.

To comprehend the execution flow, you will need to examine the underlying code base. Additionally, ensure that you have reviewed the material in Module 2 regarding the Strategy Design Pattern, as we will employ it to enhance the cohesiveness and scalability of our solution.

Problem Issues

The current implementation functions, but suffers from ***low cohesion*** and ***tight coupling***. The `Tool` entity primarily serves as a container for the different tools and is closely linked with the `Player` class, relying on it for specific details. As evident in the `Player` constructor, we populate a list of tool objects by specifying their names as parameters during `Tool` construction.

```
public Player(final String nameOfPlayer) {
  this.tools = new ArrayList<Tool>(
    Arrays.asList(new Tool("Rock"), new Tool("Paper"),
            new Tool("Scissors"))
  );
  this.playerName = nameOfPlayer;
}
```

This approach introduces several issues, with the most significant being the potential to violate the game's core rules. Only Rock, Paper, and Scissors are permitted in this game; however, our implementation allows for the creation of additional tools, such as `new Tool("Knife")`. Therefore, it is crucial to ensure adherence to this fundamental rule in order to maintain the game's integrity.

The next issue arises with the method `Game::determineWinner(..)`, which appears to be excessively concerned with the specifics of each tool. The current implementation uses a `switch` / `case` construct to determine the winner by comparing the chosen tools against a ruleset and returning the outcome.

```
private Player determineWinner(Tool firstPlayerTool, Tool secondPlayerTool) {
  return switch (firstPlayerTool.toString()) {
    case "Rock" -> secondPlayerTool.toString().equals("Scissors") ? player1 : player2;
    case "Paper" -> secondPlayerTool.toString().equals("Rock") ? player1 : player2;
    case "Scissors" -> secondPlayerTool.toString().equals("Paper") ? player1 : player2;
    default -> player1;  // why??
  };
}
```

This implementation is susceptible to errors. Firstly, we repeatedly use tool names, which can easily lead to misspellings. The names in the switch statement are thus coupled with the `Player` constructor, without any means of validating accuracy. The primary concern, however, is evident in the last `default` case, which returns **player1** for some reason. Since the method's specification requires a `Player` object to be returned, all possible execution paths must guarantee this outcome. If none of the other cases are valid (e.g., due to the presence of a knife or misspelling), **player1** is declared the winner. In addition to these issues, one might question whether it is appropriate for the `Game` class to possess knowledge about each tool's weakness.

Solving The Problems

Your task is to address the aforementioned issues in the current solution by employing the **Strategy Design Pattern**. We already know that the `Player` will serve as our **context**, and the various tools will function as our **strategies**, given their intended variability. Examine the following diagram and modify the current solution to achieve a corresponding implementation.

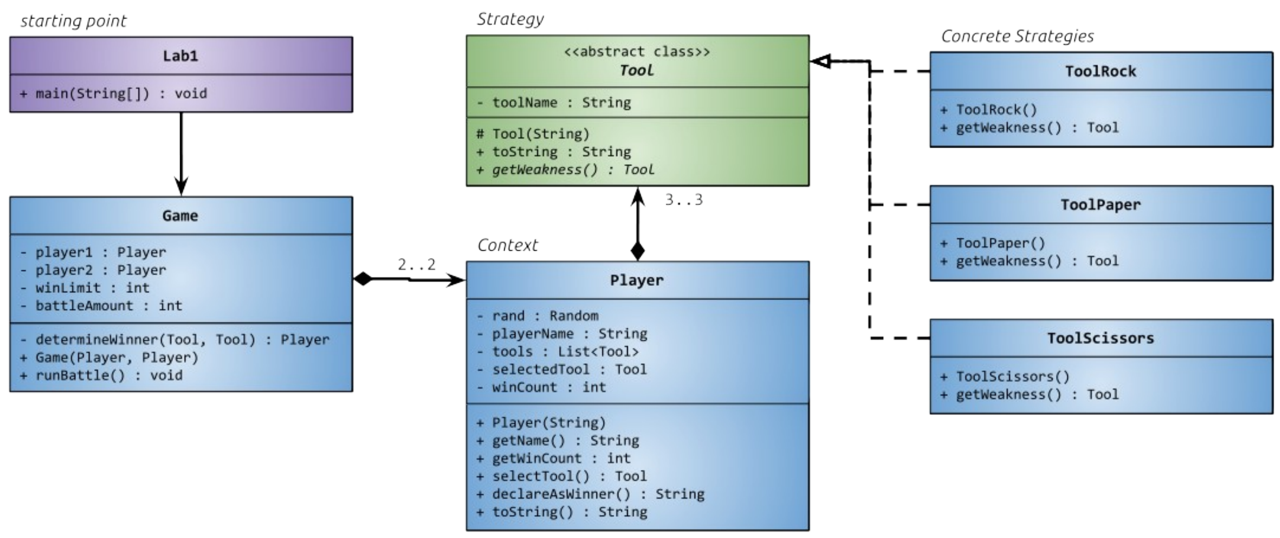

Diagram 2: strategic solution

The diagram conveys all the necessary information, clearly indicating that we want `Tool` to serve as an abstract base class from which concrete tools are derived. Apart from the `Tool` hierarchy, the rest of the specifications remain unchanged; however, you will need to modify the implementations for the `Player` constructor and `Game::determineWinner(..)`.

Note that part of this assignment is for you to read and understand class diagrams, so you need to conform to the stated specifications!

Tip! In order to find out the type a particular object you can use `Object.getClass()`, which may also be used together with `equals(..)` for comparison!

Report Purpose

Below you'll find the purpose statement for the report, available in both English and Swedish. Choose the language you prefer for your report, and be sure to include the necessary headings as outlined in the study guidance. Please keep in mind that consistency is key when writing your report, so ensure that you use the same language throughout the document.

English Version

The aim of this lab is to enhance the cohesiveness and scalability of a **Rock, Paper, Scissors** game implementation by applying the **Strategy** Design Pattern, while addressing the issues of low cohesion, tight coupling, and potential violation of the game's core rules.

Concrete Goals:

- Refactor the current implementation to utilize the **Strategy** Design Pattern by introducing an abstract `Tool` class and deriving concrete tool classes from it.
- Modify the `Player` constructor to correctly instantiate and use the new concrete tool classes.
- Refactor the `Game::determineWinner(..)` method to eliminate the use of switch / case statements based on tool names and to ensure the game's integrity by adhering to the core rules.
- Ensure that the final implementation adheres to the provided class diagram and specifications.

Swedish Version

Syftet med denna labb är att förbättra sammanhållningen och skalbarheten hos implementationen av spelet **Sten, Sax, Påse** genom att använda designmönstret **Strategy**, samtidigt som problem med låg sammanhållning, tät koppling och potentiell överträdelse av spelets grundläggande regler hanteras.

Konkreta Målsättningar:

- Refaktorisera nuvarande implementation för att använda designmönstret **Strategy** genom att införa en abstrakt `Tool`-klass och härleda konkreta verktygsklasser från den.
- Modifiera konstruktorn för klassen `Player` så att den korrekt instansierar och använder de nya konkreta verktygsklasserna.
- Refaktorisera metoden `Game::determineWinner(..)` för att eliminera användningen av switch / case-satser baserade på verktygsnamn och säkerställa spelets integritet genom att följa de grundläggande reglerna.
- Säkerställa att den slutliga implementationen följer det angivna klassdiagrammet och specifikationerna.

Requirements

- You may not change the names of folders or files relevant to the assignment!
- All implementations needs to be placed within the source package for `laboration_1` in your student repository!
- `laboration_1/README.md` needs to conform to requirements stated in the study guidance. This refers to both structure and contents.
- The solution needs to conform to stated specifications as conveyed from Diagram 2!
- Any entity you have changed / modified needs to be documented together with the name of author as annotaion: `@author firstname surname`.
- The solution, in its entirety, needs to be available in **master** branch of your student repo at the time of submission. Be sure to synchronize local changes with ***remote origin*** upon submission!

Tip! In order to find out which versions of Java, Maven and Git are installed on your system, you can use the following commands in terminal / command line:

```
java –version
mvn --version
git --version
```

Submission

All local changes should continuously be committed with suitable messages and it’s important that these local changes are synchronized with the repository’s ***remote origin*** before submitting the solution for evaluation. When ready, you need to formally submit the work in Moodle by using the dedicated submission box. You should not attach any files to this formal submission, as the solution state will be pulled directly from your student repository.

All work needs to be conducted individually by each student, but you are encouraged to both discuss and support each other using various platforms for communication!