World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:4, No:6, 2010

An Idea About How to Teach OO-Programming to Students

Irene Rothe
Bonn-Rhein-Sieg University of Applied Sciences
St. Augustin, Germany
irene.rothe@h-brs.de

Abstract—Object-oriented programming is a wonderful way to make programming of huge real life tasks much easier than by using procedural languages. In order to teach those ideas to students, it is important to find a good task that shows the advantages of OO-programming very naturally. This paper gives an example, the game *Battleship*, which seems to work excellent for teaching the OO ideas (using Java, [1], [2], [3], [4]).

A three-step task is presented for how to teach OO-programming using just one example suitable to convey many of the OO ideas. Observations are given at the end and conclusions about how the whole teaching course worked out.

Keywords—OO ideas, Java, teaching, engineering students.

I. Motivation

Somehow it seems not easy to teach the beauty of object-oriented (OO) programming to students. However, it should be easy because it is so much more related to our daily life than procedural programming.

The goal of my programming class was to teach the main concepts of object-oriented thinking, and how to create useful objects and using inheritance.

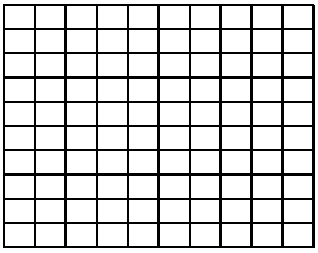
For my OO-programming teaching class I was looking for an interesting task to be implemented by the students which is totally easy to get and where during the programming the OO-idea comes into everyones mind kind of naturally. Explaining then the OO-principles to the students at the right minute makes them feel almost happy and relieved. Getting so wonderful programming tools makes it very easy for them.

The class described here covers one block in a Computer Science module for engineering students. The progamming language Java is used. In general, most of the students had already attented a class about procedural programming. For all three steps of my task I gave the students handouts including the detailed tasks and some hints.

II. The Task: Battleship

Almost everyone remembers the game *Battleship* (usually played with pencil and paper) from boring school days. So, there is no need to waste much time explaining the rules. On the other hand, the game *Battleship* is not trivial to implement and bears wonderful possibilities to apply the OO ideas.

The game Battleship is a guessing game played by two people, say player A and player B. Each player starts with a map (here called the *sea map*) consisting of 10 by 10 squares which build a grid. At the beginning each player sets ships on this map, for example one 3er ship, one 2er ship and two 1er ships (a ship consists of 3, 2 or 1 squares next to each other arranged either horizontally or vertically filled for example with the letter S like ship), which the other player is not able to see. The ships do not overlap. Further, each player has another map of the same size (here called the *enemy map*) where he records all results of his shooting against the enemy (the other player).



The game works as follows:

1) Both players get their maps settled and the ships arranged. For example, player A's *sea map* could look like:

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3				S	S	S			S	
4										
5										
6				S						
7				S				S		
8										
9										

2) One player, say A, shoots on the ships of player B by typing some (x, y)-coordinates into the computer program, for example x-coordinate 6 and y-coordinate 8, which means that the square on position $(6, 8)$ on the *sea map* of player B gets a hit.
3) The program gives then an output on the screen about this shot: success or failure. Further, in the *enemy map* of player A a cross X (ship got hit) or a tilde ˜ (water got hit) is drawn, correspondently.
4) The current *enemy map* of player A is shown on the screen.
5) Now, its player B's turn.
6) The game is over if one player has successfully eliminated all ships of the other player. This player is the winner.

III. Implementation steps

To comprehend the OO ideas step by step, the students have to program *three versions* of the game re-using everything they have done in the steps before. The educational hope behind this approach is to create wishes in the heads of the students what the programming language should make possible and then tell them SURPRISE! object-oriented languages offer everything you wish for.

First, the students are asked to implement the game of *Battleship* in a way where one human player fights against a

hidden fixed ship constellation (all ships are situated on fixed positions inside the programming code) in front of the screen. This seems not like much fun but students can very easily do it in whatever procedural programming language they already know. Further, they can test their code and at least have a first running version available. Later, they easily can exchange the programming key words they used with Java programming key words, which are very similar in most programming languages, then and compile the code again using the Java compiler `javac`.

The second step is to move over to a game with two players (both human) who shoot against each others ship armies. In order to do this, the students in general want to use functions. But while thinking about functions they also end up with a lot of variables (especially finding names for the two maps of the two players), which is kind of stressful to organize. Further, they have to pay a lot of attention to which player uses which map and not mixing up any ownerships. That is the minute, where the teacher can help and tell them to think about defining a new *object* (similar to already existing data types, for example `int` or `float`) which represents a general *Battleship-player*. So they have to make lists of what a Battleship-player owns and is allowed/capable to do. Then he tells them how to implement their own class for a Battleship-player with class variables for the maps a player owns and methods for all the business the player is allowed to do during the game like shooting and recording the successes and failures of the shots and so on. The real player will then be built as an instance of this new defined class when the main-programm is executed.

At last, the students implement a Battleship game version where one human player plays against the *computer* for which it is so wonderful to use the Battleship-player class and inherit all the variables and methods over to a new *Computer* class without listing all the belongings and capabilities of the player class in the code again. The students only have to implement everything a computer does differently than the human player specified in the player class.

IV. Getting started

The *sea map* and *enemy map* can easily be implemented in Java as two-dimensional arrays of characters:

```
char [10][10] seamap;
```

where the positions for the ships run from `seamap[0][0], seamap[0][1], ..., seamap[0][9],`.

The *enemy map* should be implemented analogously:

```
char [10][10] enemymap;
```

V. Game with one player against a fixed constellation of ships

A fixed constellation of ships might look as follows:

```
//a fixed constellation of ships
//first ship
seamap[3][3] = 'S';
seamap[4][3] = 'S';
seamap[5][3] = 'S';
//second ship
seamap[3][6] = 'S';
seamap[3][7] = 'S';
//two more small ships
seamap[8][3] = 'S';
seamap[7][7] = 'S';
```

A simple version of the Java program might look as follows:

```
class BattleshipGame{
    public static void main(String args[]){
      //declarations and initialization
      ...
      //positioning of the ships
      ...
      //start of the game
      while (...){
           //there are still ships over sea level
           //shooting of the player:(x,y)-coordinates
           ...
           //evaluation of the shooting:
           //success or failure
           ...
      }
      //output: player won
      System.out.println ("All ships are destroyed!");
   }
}
```

The students can do this code implementation with whatever programming language they already know.

It should be easy to switch all the programming key words over to Java programming key words and translate the new Java code via the Java compiler `javac`.

VI. Two players version

Now, the students are asked to change their code in a way that the game runs with two human players who fight against each other sitting in front of the same computer screen.

A good student remembers from his procedural programming language class that it is a fine thing to use functions to structure their code. But after thinking a while about all this they may realize that they end up having to deal with a lot of variables, for example four maps for the two players, and they have to pay attention to which map belongs to which player and which player is allowed to see or write into which map.

Now, the teacher can hand over the idea about what an *object* is and what it can be used for, and how to define one in a new *class*.

Definition of a player class

The teacher lets the students collect all actions a player is allowed to do and all belongings a player owns. Further, he tells them to look for nice words which define the above actions clearly and shortly.

They should come up with a list similar to this:

- Belongings of a player:
  - `seamap`,
  - `enemymap`.
- Actions of a player:
  - `placeships`: The player places his ships on the `seamap`.
  - `shooting`: The player shoots at the ships of the other player by typing the x- and y-coordinates in the keyboard of the computer.
  - `evaluation`: The player tells the other player if a ship or water was hit by his shot.
  - `recording`: Success or failure of the shot is recorded in the `enemymap`.
  - `printmap`: The `enemymap` of the player who shot is printed.

Nice Question for the students: Why is `evaluation` and `recording` not in just one method?

With the help of the list of all actions a player is allowed to do and all belongings a player owns, the students are able to create a new class called `Player`. The definition of the maps a player owns will be the *instance variables* and the actions a player is capable to do will be defined via *methods* with input and output parameters. The new class is implemented in a file called `Player.java`. This will be a non-executable class.

To implement all the methods, we have to decide about the input and output parameters, which are the things a player says and a player hears in the real world.

For example, while using the method `recording` a player hears the result of his shooting at position `(xcoordinate, ycoordinate)` which is a success or a failure and says nothing. That means the method might look as follows: `recording(result, xcoordinate, ycoordinate)` with no output paramater.

So, the `Player` class might look at the end similar to this:

```
class Player{
      //declaration of the maps
      private char [][] seamap = new char [10][10];
      private char [][] enemymap = new char [10][10];
      String name;
      //Default-constructor
      Player(){}
      //constructor
      Player (String name, int numberofships){
         //local variables and others
         ...
         this.name=name;
         this.numberofships=numberofships;
         ...
       }
       void placeships(){...}
       ... shooting(){...}
       int evaluation(...){...}
       void recording(int result, ...){...}
       void printmap(){...}
}
```

Now, also Java key words like `private` make perfect sense because it is clear that each of the two maps are owned only by the corresponding player.

Note that there are many possibilities for how to deal with the coordinates of the shots which is the reason for not giving the complete Java code here.

Using the `Player` class in the game by generation of two real players

In the main program two objects for the two players will be created, for example `playerA` and `playerB`. A player is generated in Java as follows:

```
<data type> <name of variable> =
       new <name of the constructor of the class>
             (<variables of the constructor>);
```

That means for our players:

```
Player playerA = new Player("PlayerA",numberofships);
Player playerB = new Player("PlayerB",numberofships);
```

Methods are used as follows:

```
<name of player variable>.<name of method>
```

Game with two players

Now, it is much easier for the students to build up the game loop for two players in the main program by using the advantages of the new implemented class `Player`. They do not have to struggle with map names and which map belongs to which player or who can write in which map or putting player names in the heads of functions and so on. The new class `Player` defined as described above will take care of all this. If player A records something in his map, it will neatly go into the right map.

The game loop might look as follows:

1) Player A puts his ships on his map: `playerA.placeships();`
2) Player B puts his ships on his map: `playerB.placeships();`
3) Player A shoots by typing the coordinates of his shot: `playerA.shooting();`
4) Player B sends the result of the shooting: `playerB.evaluation(...);`
5) Player A puts the result in his map: `playerA.recording(...);`
6) The `enemymap` of player A is printed on the screen: `playerA.printmap();`
7) Now, its player B's turn who starts with shooting at player A's ships.

VII. Player Class is Extended to a New Computer Class

To have more fun with the game, the students are now asked to replace one human player by a computer player so that they can play against the computer. A computer player is similar to a human player, so the students do not want to implement everything from scratch. That means, the new `Computer` class should somehow use parts of the `Player` class, and it would be useful if no one has to copy all the methods and variables they need from the `Player` class into the new `Computer` class. It is now time for the students to learn about *inheritance* in OO programming languages which will make their code implementing life much easier. OO-programming allows to use (without retyping) already implemented code by *extending* it through *inheritance*.

So, first the students have to think about what a computer player does *like* a human player and what it does *differently*. They should come up with the result that a computer player shoots differently and places its ships differently, for example randomly. Everything else the Computer does as it is already implemented in the `Player` class.

So, the computer class could look as follows:

```
class Computer extends Player  {
      //generating random numbers
      ...
      Computer(...){super(...);}
       void placeships(){...}
        ...shooting(){...}
}
```

In the above class two methods are *overwritten* from the `Player` class. Almost nothing else has to be changed or added to the `Player` class, only the key word `private` has to be replaced by the key word `protected` (meaning that all subclasses can use those maps as well). So the `Computer` class can be constructed without touching the `Player` class and still be able to use the code from the `Player` class.

At last, the students have to put in the computer player into the main program by exchanging one player with a computer player:

```
Computer playerB = new Computer(...);
```

VIII. Observation

The students were kind of heterogenous, some students already knew some ideas of OO-programming and others never heard anything about it.

Because the game was divided into small steps to motivate the students to get ideas for the next step, almost all of the students made it to the end and finally were happy to play the game by themselves against the computer. Because in our implemented game the computer places its ships randomly and shots randomly the students also got a real feeling for what for a big task testing can be. This gave the students a feeling of how the life of a programmer can be.

Very interesting discussions with the students came up during the programming phase. The teacher tried not to force the students to implement code in the way my own solution looked like. He always wanted them to create their own game even if this meant more flexibility and work from my side. At the end the students felt like they did exactly what they wanted to implement which gave them more self-confidence.

IX. Conclusions

It was a great idea to start with a task which was clear from the beginning. So, there was no time wasted for explaining *what* the students are supposed to implement. For students who already knew something about OO-programming, it was kind of confusing to start with a procedural version. But then they even got a deeper understanding of how helpful object-oriented programming can be.

It was possible to help the students understand why it is wonderful to create their own objects, how useful inheritance can be, what instance and class variables are, how methods are defined, what specific Java key words mean, what overwriting is, and more.

References

[1] D.J. Barnes and M. Klling, *Objektorientierte Programmierung mit Java*, Pearson Studium, 2003.
[2] C.S. Horstmann, G. Cornell, *core Java, Band 1 - Grundlagen*, Addision-Wesley, 2005.
[3] J. Bishop, *Java lernen*, Pearson Studium, 2005.
[4] R. Schiedermeier, *Programmieren mit Java*, Pearson Studium, 2005.

Irene Rothe Irene Rothe works at the Applied University of Bonn-Rhein-Sieg in Germany and teaches computer sciences to engineering students and journalist students.