Informatics 1: Object Oriented Programming

Tutorial 07

Week 8: 11/03 - 15/03

Volker Seeker (`volker.seeker@ed.ac.uk`)
Naums Mogers (`naums.mogers@ed.ac.uk`)

1 Introduction

In this tutorial you will make use of inheritance to extend your existing Sudoku GameGrid class with additional functionality for a variation of the game called **X-Sudoku**. In an X-Sudoku game the usual three rules like in the regular Sudoku game apply. In addition, you now have a fourth rule where numbers are not allowed to be repeated on any of the two diagonals[1] in the grid. Figure 1 illustrates this for an example.

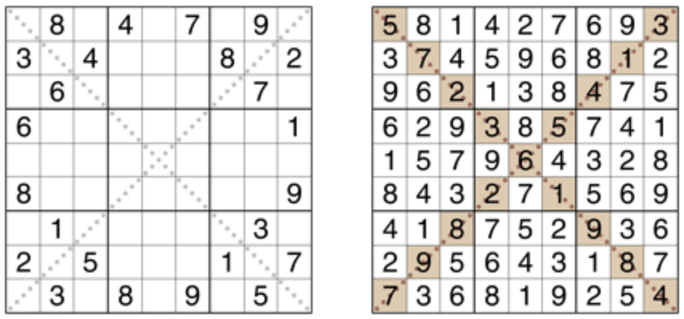

Figure 1: X-Sudoku rules for placing numbers in the two diagonals.
source: https://www.conceptispuzzles.com/index.aspx?uri=puzzle/sudoku/rules

2 Exercises

The tasks of this exercise are broken down into three parts. The first part introduces the new class hierarchy into the existing code and implements the new subclasses. The second part implements added functionality for the X-Sudoku game and the third part integrates the new game mode into the main menu.

[1] top left to bottom right and bottom left to top right.

Creating the Class Hierarchy

To extend the current GameGrid with an additional rule, you will implement two subclasses for the existing GameGrid class called RGameGrid for a regular Sudoku game and XGameGrid for an X-Sudoku game. Figure 2 illustrates this hierarchy using UML notation. The italic font in the GameGrid superclass indicates that this class will be **abstract**.

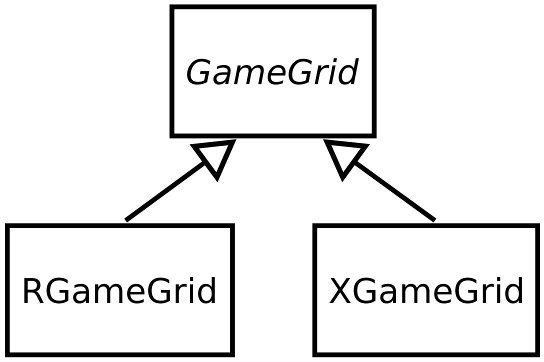

Figure 2: GameGrid class diagram.

Task 1 - RGameGrid and XGameGrid ◁ Task

Create two new classes RGameGrid and XGameGrid which both extend the GameGrid class. To satisfy the compiler, you will have to create at least a single constructor which calls one of the superclass' constructors with the required parameters. For practice, create a constructor for each of the three available constructors in each new subclass and pass the required parameters to the superclass using the super keyword. No additional work needs to be done in the constructors.

Lastly, add the abstract keyword to the class header of the original GameGrid class.

Task 2 - Correct Constructor Usage ◁ Task

If you compile your code base now, you will get compiler errors for every usage of the GameGrid constructor. This is because abstract classes cannot be instantiated, as you have learned. To fix this, you will have to choose either the RGameGrid or XGameGrid constructor as a replacement.

To keep the original codebase usable for both types of games, we will make use of **Polymorphism**. In the main function where you create the first instance of your GameGrid, replace only the constructor call with a call to a concrete subclass. Leave the variable type as GameGrid, for example:

```
GameGrid game = new RGameGrid(args[0])
```

This also needs to be done every time you use the copy constructor. For example, before calling Solver or Ranker functions. You now need to make sure that you use the correct copy constructor depending on the concrete subclass instance you want to copy. To simplify this, create a public static function copyGameGrid in your main class which gets a GameGrid parameter and returns a GameGrid.

Again using polymorphism, the actual arguments for this function will be either RGameGrid or XGameGrid instances. In the function, you will need to find out which instance you get by using the **instanceof** keyword and corresponding if clauses. Once you know which instance you have, you can use the correct copy constructor to create and return a copy.

With copyGameGrid in place, you can now replace every occurrence of a copy constructor call with a static call to your new function. For example:

```
GameGrid gameToSolve = Sudoku06.copyGameGrid(game);
```

Adding Extra Rules

Now you have to add an extra rule to your original `isValid` method to check the diagonals of the game grid. Also, a new `toString` implementation should be used for an X-Sudoku.

Task 3 - `toString` for X-Sudoku ◁ Task

Override the `toString` method from `GameGrid` in `XGameGrid`. Based on the original implementation, add functionality so that your generated string indicates the diagonals of the printed grid with additional symbols. For example as demonstrated here:

```
[0]  0   0    0   8   9    1   0  [0]
 2  [0]  0    0   0   0    0  [0]  0
 0   9  [0]   0   0   6   [4]  0   8

 0   0   3   [7]  0  [8]   0   0   0
 7   0   0    0  [1]  0    0   0   0
 8   0   1   [5]  0  [0]   6   0   0

 0   0  [0]   0   0   5   [0]  0   0
 0  [0]  0    0   0   0    0  [6]  0
[6]  0   8    3   4   0    0   0  [0]
```

NB: You will have to access data from the `grid` member of the `GameGrid` class. You can do this by changing the accessibility of `grid` to `protected` or by using the `getField` method to access field values.

Task 4 - Checking the Diagonals ◁ Task

To add the additional rule, you can reuse most of the functionality of a regular Sudoku which already exists. Override the `isValid` method in `XGameGrid`. Consider how best to change the access modifier of the original function to be able to do that. In your implementation, first call the superclass version of this method using the `super` keyword and save the resulting `boolean`. Now you have already checked the given `value` against the first three rules.

Secondly, combine this result with a method call to a new `private` method `checkDiagonal` which has the same parameter list and return type as `isValid`. This method checks for the given row and column coordinates if the new value is allowed to be set, i.e. if this value does not yet exist in the corresponding diagonal. The check only needs to be done if row and column indicate a diagonal and only for the diagonal specified by row and column unless it is the middle value in the grid.

Integration

Task 5 - Integrating the new Game Mode ◁ Task

As a last task, you should allow the user in your `main` method to decide whether he wants to play a regular Sudoku or an X-Sudoku. You can do this by adding an initial small menu which asks for the game mode.

Based on the user's answer, you will create an `RGameGrid` instance or an `XGameGrid`. This is what a menu could look like:

```
Select game mode:
1. Regular Sudoku
2. X Sudoku

Select an action [1-2]:
```

Alternatively, you could create a concrete instance based on the given Sudoku filename. With this exercise you are given three additional Sudoku game files which all start with an 'x'. You can use this fact in the beginning of the `main` function to decide which concrete `GameGrid` instance to use.

3 Optional Extra Tasks

If you have finished all of the above tasks and look to do more, please consider the following suggestions or come up with your own ideas:

The Command Pattern The handling of game menu actions is currently done in a way which is not particularly conform to object oriented design. At this point one would usually use an object oriented design pattern called the **Command Pattern**.

This pattern would encapsulate all menu handler code into its own *Command* object which provides an `execute` method to execute the handler code. The command pattern has four components: the Command, the Receiver, the Invoker, and the Client.

For our purposes, the Receiver would be the concrete `GameGrid` instance you are using, an Invoker won't be necessary, the Client would be the `main` method and a Command class would be necessary for each handler code, e.g. `SetFieldCommand`, `ClearFieldCommand`, `SolveGameCommand`, etc.

For this task, refactor the current menu selection handler code in your `main` function to use the Command Pattern. You can use the example on the following page as guidance:

https://www.baeldung.com/java-command-pattern