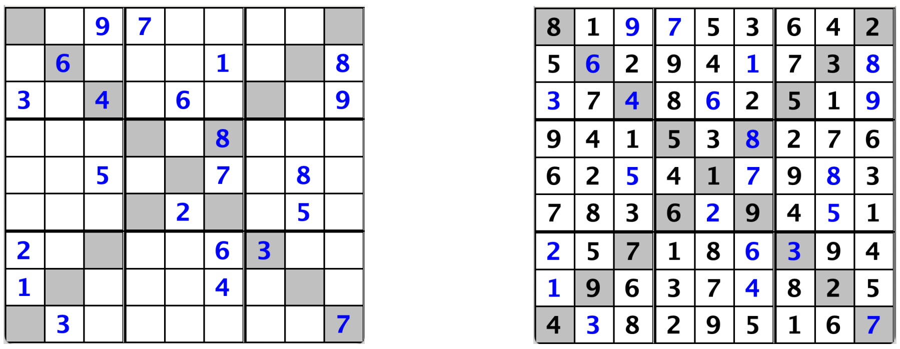
CSED232 Assignment 6

Due Saturday, May 31

Background: X-Sudoku

- *X-Sudoku* is a variant of Sudoku (`https://en.wikipedia.org/wiki/Sudoku`). The goal of an X-Sudoku puzzle is to fill numbers from 1 to 9 in empty squares of a 9×9 grid such that
  - The numbers 1 through 9 appear exactly once in each row, column and 3×3 box.
  - Each of the two main diagonals contains the numbers 1 through 9 only once.



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

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

Figure 1: An X-Sudoku puzzle and its solution.

Problem 1: Implementing X-Sudoku Using Observer

- An X-Sudoku puzzle can be implemented using the Observer pattern. The key classes are `Cell` and `Group`, where each group observes its member cells.
  - A cell has a set of possible numbers that it can contain and may eventually be assigned one of them. A cell changes when it gains or loses a value or possibility.
  - There is a group for each row, column, 3×3 box, and main diagonal. If a member of a group is assigned a particular value, none of its other members can have that value as a possibility.
  - For example, in the unsolved puzzle in Figure 1, the first row of the first column has no value and the set of possibilities $\{5, 8\}$.
- Figure 2 shows the class diagram for this problem. The goal is to implement the three classes in the diagram: `Cell`, `Group`, and `Board`, all located in the `edu.postech.csed232.model` package.
  - `Subject` and `Observer` are already implemented. A subject notifies *events* to its observers. An event is an instance of `Event` and provides additional information about changes.
  - A cell should notify its observers with appropriate events when certain changes occur. E.g., if a cell loses all of its possibilities, it should notify its observers with `ActivationEvent(false)`.
  - A group receives an event when the value of its cell is set or unset (`NumberEvent(`n`, `*isSet*`)`), and responds by updating the possibilities of the other cells in the group.
  - The interface `CellConstraint` is introduced to reduce the coupling between `Cell` and `Group`. The `Group` class implements both `Observer` and `CellConstraint`.
  - A board maintains 9 row groups, 9 column groups, 9 square groups, 2 diagonal groups, and 81 cells. The classes `Board`, `Group`, and `Cell` define the object-oriented model.

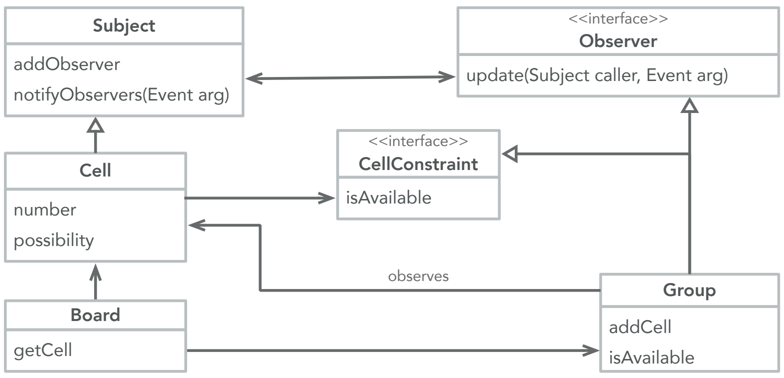
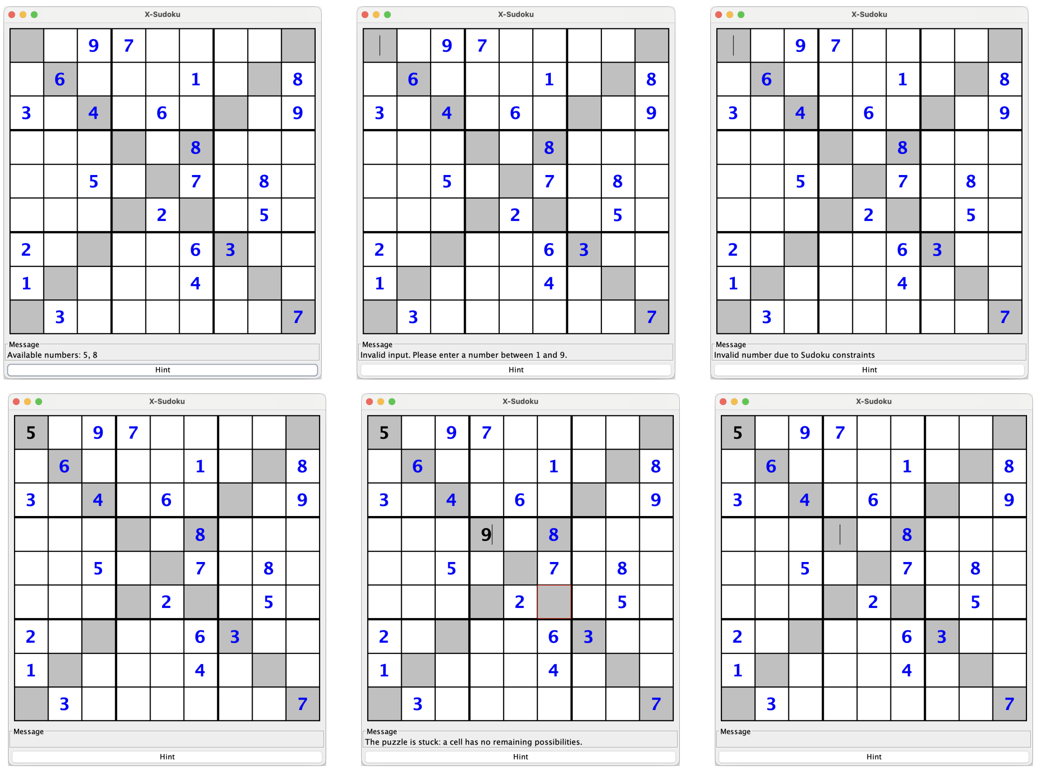


Figure 2: Class diagram for the X-Sudoku model.

The X-Sudoku Graphical User Interface

- Figure 3 shows the graphical user interface (GUI) of X-Sudoku. It consists of the board, a message box, and a hint button. Each cell UI is implemented as a single-line text field.
  1. When the hint button is pressed, the available numbers for the selected cell are displayed.
  2. When text is entered into an empty cell:
     - If the update fails (due to invalid input or an X-Sudoku constraint violation), an error message is shown in the message box, and the cell UI is cleared.
     - If the update succeeds, the number remains displayed in the cell UI.
  3. If a cell loses all of its possibilities, its UI is *deactivated* and visually marked with a red border.
  4. If a deactivated cell later regains a possibility, its UI is *activated* (with a normal border).



Available numbers: 5, 8

Invalid input. Please enter a number between 1 and 9.

Invalid number due to Sudoku constraints

The puzzle is stuck: a cell has no remaining possibilities.

Figure 3: X-Sudoku GUI

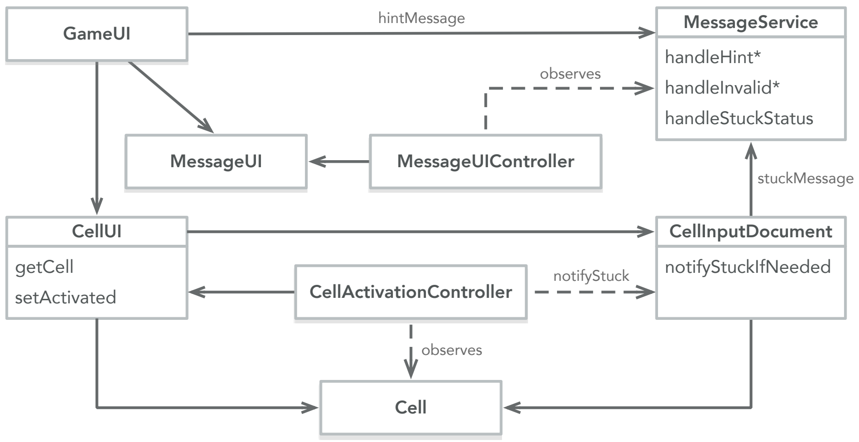


Figure 4: Class diagram for the X-Sudoku GUI.

Problem 2: Implementing the Controller of the X-Sudoku GUI

- The X-Sudoku GUI is designed based on the Model-View-Controller (MVC) architectural pattern.[1] The system is divided into three interconnected components:
  - **Model:** the internal representation of information and logic.
  - **View:** the interface that presents information to the user and accepts input.
  - **Controller:** the components that coordinate interactions between the model and the view.
- Figure 4 shows the class diagram for the X-Sudoku GUI.
  - The "model" consists of the classes `Board`, `Cell`, and `Group`. In Figure 4, only `Cell` is shown, as the other two classes are not directly related to the view and controller layers.
  - The class `GameUI` defines the top-level container, and the classes `CellUI` and `MessageUI` define its components. These three classes represent the "view" part (already implemented).
  - The "controller" part is implemented by the remaining classes in Figure 4, which handle input and coordinate interactions.
- The goal is to implement the three classes in the diagram: `MessageService`, `MessageUIController`, and `CellActivationController`, all located in the `edu.postech.csed232.control` package.
  - The class `MessageService` provides a centralized interface for communicating user feedback, such as hint messages, constraint violations, and stuck conditions.
  - The classes `CellActivationController` and `MessageUIController` observe model changes and orchestrate updates to `CellUI` and `MessageUI`.
  - The class `CellInputDocument`, which is already implemented, coordinates among the classes `Cell`, `CellUI`, and `MessageService`.
- In this problem, it is important to understand the existing design and implementation, and to write code that integrates well with the provided structure. Carefully read the Javadoc comments.

Problem 3: Writing JUnit Test Cases

- The goal is to achieve at least 95% **branch coverage** (measured by JaCoCo) for each class in the packages `edu.postech.csed232.model` and `edu.postech.csed232.control`.
- Each test case should cover one logical scenario and include meaningful assertions. Test both typical and edge cases, such as constants, zero expressions, and empty or singleton polynomials.
- After executing `gradle test`, you can generate reports for coverage information of your unit tests using `gradle jacocoTestReport`. The reports will be stored in `build/reports/jacoco/test`.

[1]https://en.wikipedia.org/wiki/Model-view-controller

General Instruction

- The `src/main` directory contains the skeleton code. You should implement all classes and methods marked with *TODO*. Before writing code, carefully read the descriptions in the source files.
- The `src/test` directory contains test cases. Use JaCoCo to find out the coverage achieved by your tests. Upload the JaCoCo report in CSV (`build/reports/jacoco/test/jacocoTestReport.csv`).
- The command `gradle jar` will create a jar file in the `build/libs` directory, which can be executed using the command: `java -jar homework6-1.0-SNAPSHOT.jar`.
- Do not modify the existing interface, class names, or the signatures of public methods. However, you may add additional private fields and methods if they are needed.

Turning in

1. We use Github Classroom to manage homework. Click on the following link, and accept the assignment: `https://classroom.github.com/a/DqSpVYN0`
2. Your assignment repository should now be created. Clone the repository, complete your homework (including the JaCoCo coverage report), and push the changes to GitHub before the deadline.
3. The JaCoCo coverage report in CSV should be uploaded to the root directory of your repository (i.e., the same directory where `build.gradle.kts` is located).

Reference

- Java Language Specification: `https://docs.oracle.com/javase/specs/`
- Learn Java: `https://dev.java/learn/`
- Core Java, Volume I: Fundamentals 13th by Cay S. Horstmann, Pearson, 2024 (available online at the POSTECH digital library `http://library.postech.ac.kr`)
- Using Swing Components: `https://docs.oracle.com/javase/tutorial/uiswing/components` (Understanding Java Swing is not strictly required for this assignment.)