Software Engineering Project Documentation

Alexander Erzmann & Mikita Sidarenka

Winter semester 2024/2025

Sudoku Solver

Status:
October 28, 2024

Contents

1. Introduction

As part of the Software Engineering course of the M1 Informatique degree program at the Université Côte d'Azur, a Sudoku solver was to be implemented, taking into account software engineering aspects, as design patterns, or principles as the seperation of concerns. The following documentation presents the Sudoku solver from a development perspective. The main focus will be on the presentation of the solution and the justification of decisions made. After a brief requirements analysis, the architecture used and the design patterns employed will be explained using the code. An introduction to the deduction rules used within the Sudoku Solver provides an insight into the application's solution strategy. The tests carried out will then be listed and described. Finally, an outlook on possible next steps and a concluding conclusion are provided.

2. Analysis

Every software development should begin with a comprehensive requirements analysis. Although the stakeholders do not really exist in this case, some requirements have already been set due to the fact that this project is an assessed examination. These are listed below for the sake of completeness.
The requirements already provided include ...

- ... either Java or Python must be used as the programming language.
- ... the display of the SudokuSolver in the application is less relevant than the underlying software.
- ... at minimum four different Design Patterns must be used.
- ... at minimum three different Deduction Rules must be used.
- ... the following input file format must be used in a .txt file:

```
0,7,0,0,0,0,0,4,3
0,4,0,0,0,9,6,1,0
8,0,0,6,3,4,9,0,0
0,9,4,0,5,2,0,0,0
3,5,8,4,6,0,0,2,0
0,0,0,8,0,0,5,3,0
0,8,0,0,7,0,0,9,1
9,0,2,1,0,0,0,0,5
0,0,7,0,4,0,8,0,2
```

- ... if the solver has no more deduction to make and if the grid is not full, then the user must be asked to fill a cell.
- ... if there is an inconsistancy in the sudoku the user should receive a message to restart the solving.

3. Implementation

This section is the centerpiece of the documentation and explains the actual implementation of the application. First, the general structure and architecture of the software will be explained. In other words, what classes are there, how do they interact with each other and with what purpose. Subsequently, the design patterns used will be explained in more detail using the respective classes. Finally, the three required deduction rules are explained and the algorithm implemented here is illustrated[1].

3.1 Architecture

A layered architecture was used to implement the Sudoku solver. In this sense, the individual responsibilities should lie in different layers. The client module contains the presentation layer, which is responsible for displaying the Sudoku and interacting with the user. It also contains interfaces to the other layers. The Deduction Rules module then contains the actual business logic for solving the Sudokus. The actual data structure "Sudoku" is located in the module Sudoku, which bundles the data access to Sudokus and the manipulation of these.

3.1.1 Client

The client contains the classes SudokuPresenter, SudokuChecker, SceneManager and SudokuController. The class SudokuPresenter is the central class for controlling the user interface. This is where the interface is created and updated. The SceneManager is responsible for switching between different scenes. Currently there is only a StartScene and a GameScene. The SudokuChecker checks the Sudoku for errors that may result from the user entering values. Both the SudokuPresenter and the SudokuChecker implement the SudokuObserver interface to be notified by the Sudoku object when changes to the Sudoku occur (see Section 3.2.4). In order to control which observers are notified, two more specific SudokuObserverInterfaces are used to distinguish between DisplayObserver and CheckerObserver. For example, the Checker should only be activated when setting new values, but the Presenter should also be activated when changing the possible values, as these must also be displayed. The interface to the Sudoku and the DeductionRules is provided by the SudokuController, which is responsible for creating the Sudoku and for solving the Sudoku. The concrete Sudoku instance is also held here.

[1] Please note that ChatGPT was used to assist the implementation process and fix bugs faster.

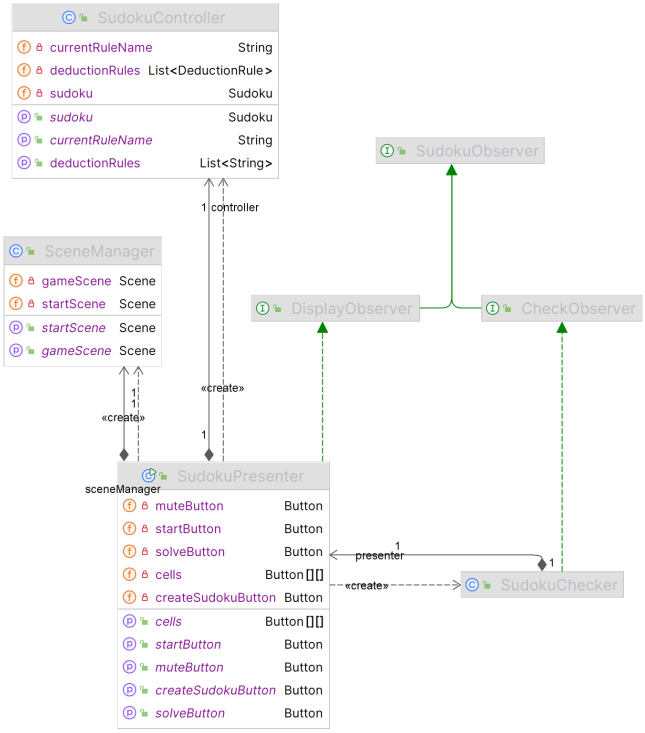

Figure 3.1: Module Client

3.1.2 Deduction Rules

The DeductionRules module contains a factory for producing a specific deduction rule or for producing a list of all currently existing rules. The respective rules (see Section 3.3) implement the DeductionRule interface and thereby implement the most important method, the run() method. This returns a boolean indicating whether the rule was successfully applied. The DeductionRuleType contains a specific enum value for each rule, such as NAKED_SINGLE.

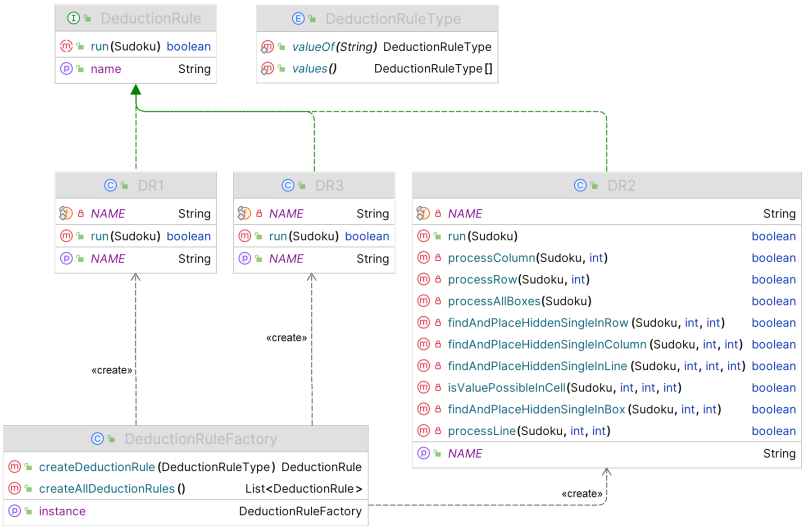

Figure 3.2: Module DeductionRules

3.1.3 Sudoku

The actual data structure used in this application can be found in the Sudoku module. The Sudoku class itself consists of rows, columns and boxes. All of these components implement the RowColumnBox interface to ensure a uniform approach. Components of Row, Column and Box are Cells, which may contain possible values in addition to a fixed value.

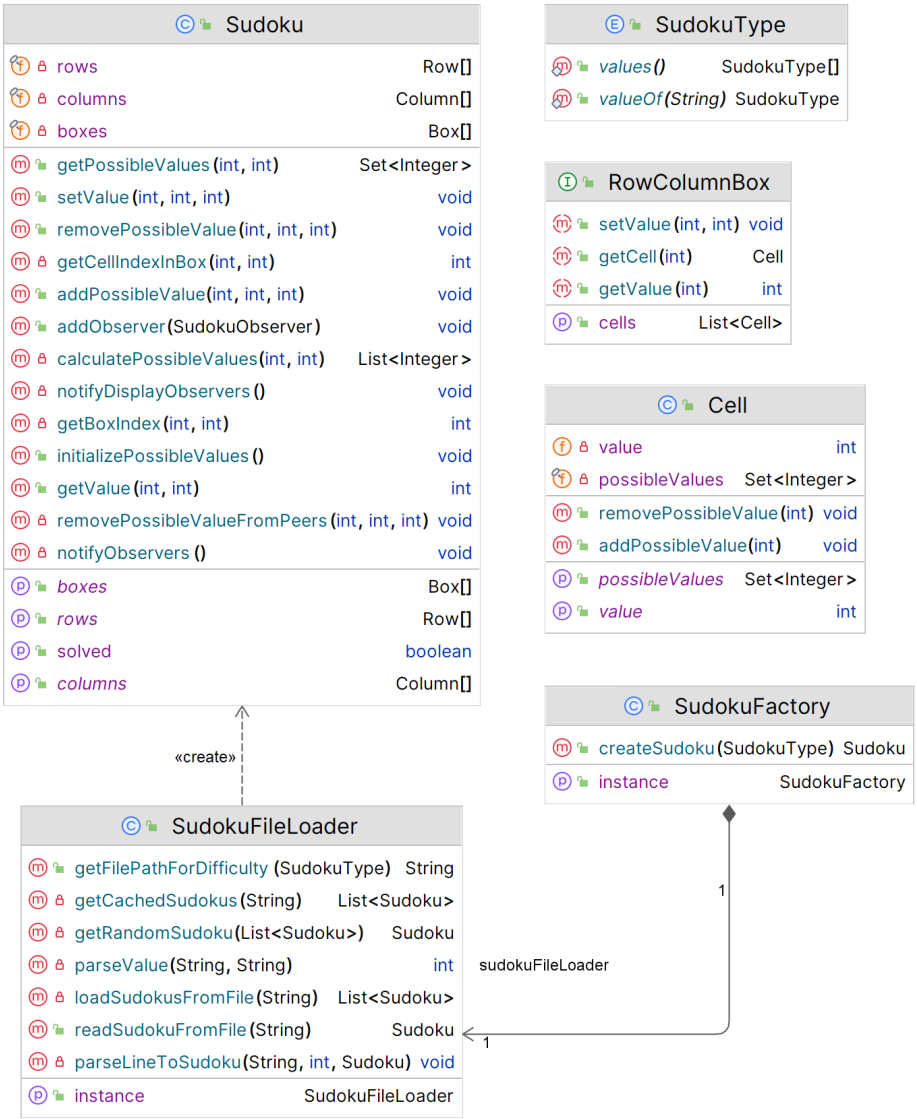

Figure 3.3: Module Sudoku

The SudokuFactory is analogous to the DeductionRuleFactory responsible for the creation of Sudokus, whereby in this case the SudokuFileLoader is used, which creates Sudokus using text files in which Sudokus are stored as in the required format (cf. chapter 2).

3.2 Design Pattern

In this section, the design patterns used in the project will be explained using the code to illustrate the usefulness of using these patterns. [2]

3.2.1 Singleton Pattern

This is a creation pattern that ensures that a class has only one instance and one global access point to that instance. This ensures that only one object of a class has been created, and that no object of that class will ever be created again. This is very useful if you want to control access to some shared resources, or if there are certain data or services that need to be the same throughout the programme.

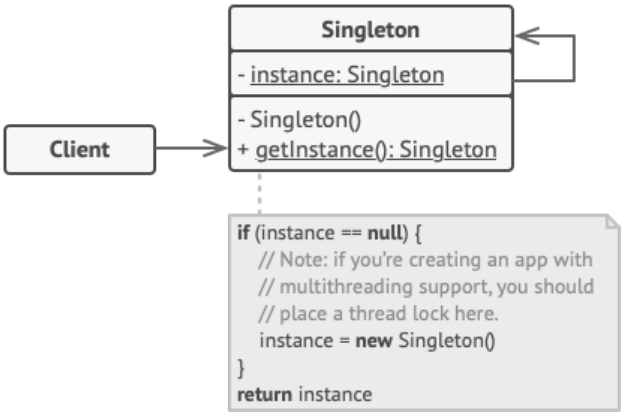

Figure 3.4: Class diagramm - Singleton Pattern
`https://refactoring.guru/design-patterns/singleton`

Figure 3.4 shows the class diagram for the singleton pattern with the following central elements of the singleton class:

1. instance: This attribute contains the single instance of the class.
2. Singleton(): This private constructor prohibits the creation of new instances of similar classes from outside.
3. getInstance(): Instead of using a constructor, this method gives access to a single instance. If the instance has not been created, it will create it.

The DeductionRuleFactory serves as an example of a class that does not require more than one instance. The getInstance() method has the same name as in figure 3.5 and performs the same function. In the illustration, the 'client' is the one that requests the singleton instance. In our case, the client would be any object that calls the method. If the client needs to access the DeductionRuleFactory, it uses getInstance() and ends up calling the factory that can generate

[2]To get an overview of the most popular patterns, and to understand their usage and implementation we made use of the Refactoring Guru website [Ref24].

deduction rules. When using the singleton pattern, attention must be paid to synchronisation, as several instances could be created if the getInstance() method is executed at the same time. In the case of the SudokuSolver, synchronisation is ensured by the static inner class 'InstanceHolder'. By using a static inner class, the JVM is implicitly responsible for synchronisation.

The classes SudokuFileLoader and SudokuFactory also represent the Singleton pattern in the same way.

3.2.2 Factory Pattern

This pattern is used to create objects. Instead of direct object creation via 'new className()', we use a factory method that decides which type of object is to be created.

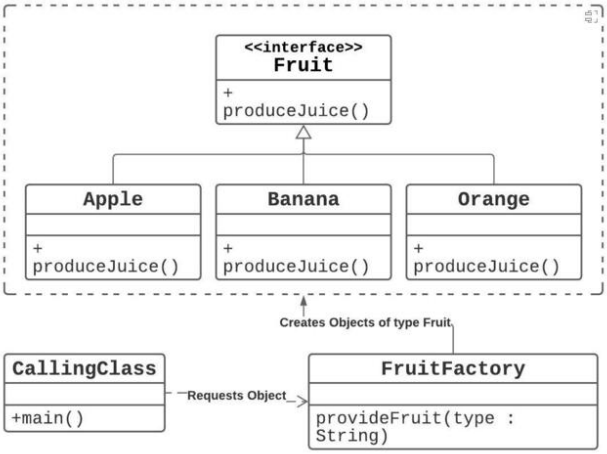

Figure 3.5: Class diagramm - Factory Pattern
`https://springhow.com/factory-pattern/`

In figure 3.5 you can see the main components using the factory pattern using a FruitFactory example to creates different kinds of fruit. In the case of SudokuSolver, there are two objects created by factories: Deduction Rules and Sudokus. We have different types of Sudokus like EasySudokus, MediumSudokus or HardSudokus and different types of Deduction Rules like DR1, DR2 or DR3. The classes DeductionRuleFactory and SudokuFactory (which are also singleton objects) are responsible for creating Sudoku objects depending on the passed SudokuType parameter or Deduction Rules depending on the passed DeductionRuleType parameter. In fact, the DeductionRuleFactory also has a method to create all existing deduction rules, because the single creation is not needed in the current state of the SudokuSolver.

3.2.3 Strategy Pattern

The strategy pattern is used to assign different solution strategies to a specific context according to the current situation. A context could for example be 'solving a sudoku' by using a strategy which could be a deduction rule.

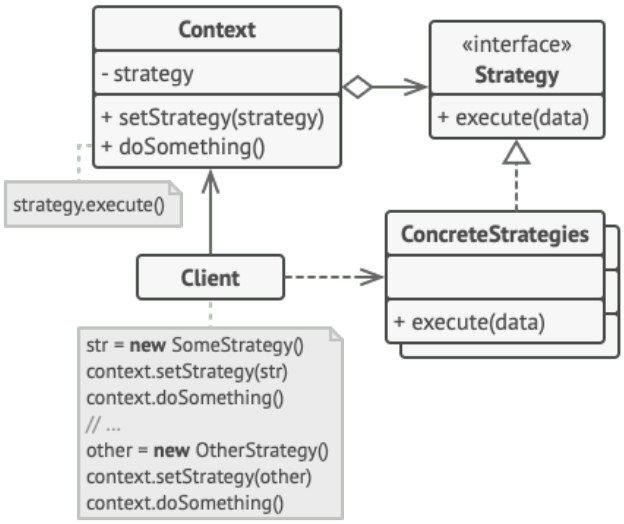

Figure 3.6: Class diagramm - Strategy Pattern
`https://refactoring.guru/design-patterns/strategy`

Regarding this concrete implementation of a SudokuSolver the 'ConcreteStrategies' seen in figure 3.6 are the Deduction Rules (DR1, DR2 and DR3). The current implementation does not exactly fit to the shown class diagramm in figure 3.6 because there is no way to set the strategy individually by the client. However by pressing the solveButton in the SudokuPresenter, the strategies are set considering all the available deduction rules to solve the Sudoku. In theorie it is possible to create a single Deduction Rule and if in future versions of the Sudoku solving using a set of chosen rules should be possible, the strategy pattern could be used exactly in the way presented in the class diagram.

3.2.4 Observer Pattern

If there is a context in which another object is interested in the changes to the state of another object, the observer pattern can be used. The observer pattern is used to inform subscribers about changes in the state of the publisher.

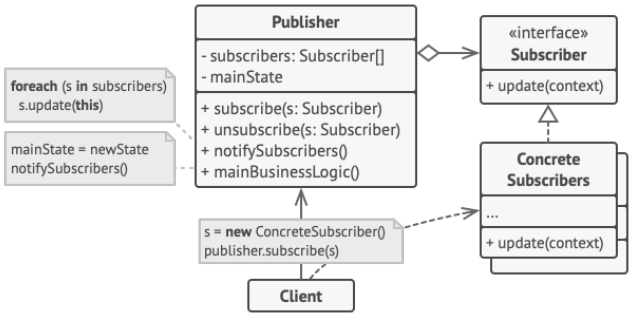

Figure 3.7: Class diagramm - Observer Pattern
`https://refactoring.guru/design-patterns/observer`

The Sudoku class plays the role of the publisher. It maintains a list of subscriber objects and sends them notifications about changes via its notifySubscribers() method. Similar to

the methods in figure 3.7, methods for adding and removing subscribers are available, such as addObserver() and removeObserver(), which allows flexible management of subscribers.

The SudokuChecker class realises an observer. Its method updateSudoku(Sudoku sudoku) is called by the Sudoku when a state change occurs. When the Sudoku is updated, the SudokuChecker checks the legitimacy of the Sudoku. The SudokuPresenter as a user interface is also an observer of the Sudoku, as it is interested in the state of the Sudoku for correct presentation to the user.

In this implementation, in addition to the SudokuObserver interface, which the two classes implicitly implement, two further interfaces were derived that can specify observers in CheckObserver or DisplayObserver. This was used in order to have the option of not notifying the SudokuChecker of changes to the PossibleValues, as a check would be superfluous.

3.2.5 Facade Pattern

This is a pattern that uses a 'simplified interface' to hide the user from interacting with a complex system. The facade shows the user only the information they need so they don't understand the complexity of the code or classes.

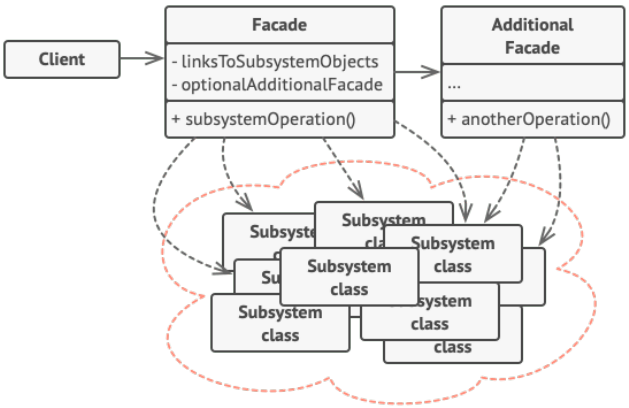

Figure 3.8: Class diagramm - Facade Pattern
`https://refactoring.guru/design-patterns/facade`

Our Sudoku code is implemented using a facade pattern, whose structure is similar to the one shown in figure 3.8. The client is the class SudokuPresenter, which represents the interaction with the user via an interface. The main element of the facade is the SudokuController class, which encapsulates the complications of creating and solving Sudokus, providing the client with only the simple createSudoku() and solve() methods. Subsystems could be SudokuFactory, DeductionRuleFactory and DeductionRule. These implement the internal logic and are hidden from the client. This would be consistent with the scheme that the client only interacts with the facade, not with the internal classes of the subsystems directly.

3.3 Deduction Rules

This section deals with the deduction rules implemented in SudokuSolver. In addition to the idea of the rule, the implementation of the algorithm for applying the rule will be discussed in detail. Each DeductionRule implements the DeductionRule interface, which specifies the run() method. This method can then be used independently of the specific rule to apply a rule.

3.3.1 Naked Single

This is one of the simple strategies that uses an analysis related to the original Sudoku rules. For all cells, all possible solutions that fulfil the standard Sudoku rules are determined (each digit can only be used once in a row, column and 3x3 block). The strategy now searches for cells that have only one possible solution and sets it.

The run() method executes exactly this rule using the algorithm shown below as pseudo code:

```
METHOD run(sudoku):
    wasApplied = false

    FOR each row:
        FOR each column:
            IF the cell is empty:
                SEARCH the possible values for that cell
                IF there is only one possible value:
                    set that value into the cell
                    wasApplied = true
                END IF
            END IF

    RETURN wasApplied
```

This method checks each cell of the Sudoku grid for an empty cell (value 0). For each empty cell, this method checks the number of possible values that could be inserted into this cell. If there is only one possible value, it is assigned to the cell. It uses the wasApplied flag to determine whether at least one change has been made to the grid. If changes have been made, the method returns the value true, indicating that the strategy has been successfully applied. If no changes have been made, it returns false.

The code differs from this algorithm in that the values that only have one possible value are stored temporarily in order to set exactly these values at the end. If the value is set immediately, new naked singles could appear in the next cells. This would be very intransparent for the user, which is why these cells should only appear at the next step.

3.3.2 Hidden Single

The idea of the hidden single is to find cells that, although they could theoretically have multiple possible values, only have one possible value in relation to the possible values of the cells in the row, column or box.

Lets again take a look at the 'run()'-Method shown as pseudo code:

```
METHOD run(sudoku):
    wasApplied = false

    FOR each row, column, and block:
        CALL processRow(), processColumn(), processBox()
        IF at least one hidden single was found processing row,
        column and box
            wasApplied = true
        END IF
    RETURN wasApplied

HELPER METHOD processRow/Column/Box:
    FOR each value from 1 to 9:
        apply method findAndPlaceHiddenSingleInRow/Column/Box
    RETURN the result

HELPER METHOD findAndPlaceHiddenSingleInRow/Column/Box:
    counter = 0
    FOR each cell in a row/column/box
        IF the value is possible in the cell:
            counter++
        END IF
        IF the value is only possible in one cell of a row:
            set this value in the cell
            RETURN true
        END IF
    RETURN false

HELPER METHOD isValuePossibleInCell:
    IF the value is possible in the cell
        RETURN true
    END IF
```

The run() method processes each row, column and block to find such hidden values by successively

calling the processRow(), processColumn() and processBox() methods. Each method will call a second helper method called findAndPlaceHiddenSingleInRow()/Column()/Box() for each possible value in the Sudoku (1 to 9). This second helper method will check for the given value if the value is only possible for one cell by using a third helper method isValuePossibleInCell(). If yes, the value is set in that cell, otherwise false is returned.

3.3.3 Box-Line Reduction

The following code implements a Sudoku solving strategy called "Box-Line Reduction". It works by eliminating the possible values of a number in the rows or columns outside the 3x3 block, if the number fits into only one row or column inside the block.

```
METHOD run(sudoku):
    FOR each block in sudoku
        FOR each number from 1 to 9
            Initialise the sets of rows and columns containing that number

            FOR each cell in a 3x3 block
                IF a cell is empty AND the number can be in that cell
                Add the row and column to the corresponding sets
                END IF

            IF the row/column set contains a single row/column
                Remove the number from all cells in this row outside
                the block
                RETURN true
            END IF
    RETURN false
```

The main logic is done in the run(Sudoku) method. This program iterates linearly over each block in the Sudoku, calculating the initial row and column indices in each block. For each number from 1 to 9, it keeps track in each block of the possible rows and columns where that number can be placed, collecting them in rowsWithNum and colsWithNum sets. If only one row or column is found with a particular number, the technique removes that number from all possible values for all other cells in that row that are outside the current block. It returns true if any changes have been made during execution, i.e. the possible values for the cells have been adjusted. Consequently, the box-line reduction strategy is applicable.

3.4 Solving procedure

The solving process of this application was implemented by using a button. This means that the Sudoku is solved step by step. An alternative to this would be a complete solution (at

least as long as the rules continue) without the user having to take any action in the meantime. The step-by-step implementation offers a comparatively transparent implementation with little effort. In this sense, the extensive and delayed display of the solution steps could be avoided.

When the 'solve' button is pressed, the following procedure is called:

```
private void solveStep() {
    String ruleName = controller.solve();
    if (ruleName != null) {
        ruleLabel.setText(RULE_HEADER + ruleName);
        currentRuleName = ruleName;
        updateRuleList();
        if (controller.getSudoku().isSolved()) {
            sceneManager.showInfo("Sudoku solved!");
            solveButton.setDisable(true);
            for (int row = 0; row < SIZE; row++) {
                for (int col = 0; col < SIZE; col++) {
                    cells[row][col].setDisable(true);
                }
            }
        }
    } else {
        sceneManager.showWarning("The Sudoku cannot be solved by
        the Deduction Rules. Please fill any cell.");
    }
}
```

If at least one of the existing deduction rules could be successfully applied, its name is passed. In this case, only the last used rule is updated. The update of the Sudoku has already been done during the manipulation. If the Sudoku is already solved, this is indicated by an info message and the solve button and all cells are disabled to prevent further interaction. If no rule could be applied, the user is prompted with a warning to fill a cell themselves.

The solving algorithm itself is fairly simple. It tries to apply the deduction rules in the controller in the order in which they appear. When a rule is successfully applied, the solving step is terminated and the name of the rule is passed on to the presenter.

```
public String solve() {
    currentRuleName = "None";
```

```
        if (sudoku == null) return null;

        for (DeductionRule rule : deductionRules) {
            if (rule.run(sudoku)) {
                currentRuleName = rule.getName();
                return currentRuleName;
            }
        }
        return null;
    }
```

4. Testing

As part of the implementation, tests were written for the most important classes. The tests are mainly integration tests and black box tests. This means that, for the most part, the classes were not tested in complete isolation, for example using Mockito. In addition, it was generally only tested whether the end result of executing a method under certain initial conditions corresponds to expectations. To demonstrate a unit test and white box tests, a test using Mockito was added to Deduction Rule 3 (DR3).

Since the SudokuFileReader loads a random Sudoku from the .txt documents, the TestSudoku-Factory class was added to carry out the tests, which loads a predefined two-dimensional array into a Sudoku object. Since only the method setValue(row, col, value) of the Sudoku is used here, the TestSudokuFactory itself was not tested. The following example shows how this class is creating test sudokus:

```
// Example for an easy Sudoku
public Sudoku createEasySudoku() {
    int[][] grid = {
            {5, 3, 0, 0, 7, 0, 0, 0, 0},
            {6, 0, 0, 1, 9, 5, 0, 0, 0},
            {0, 9, 8, 0, 0, 0, 0, 6, 0},
            {8, 0, 0, 0, 6, 0, 0, 0, 3},
            {4, 0, 0, 8, 0, 3, 0, 0, 1},
            {7, 0, 0, 0, 2, 0, 0, 0, 6},
            {0, 6, 0, 0, 0, 0, 2, 8, 0},
            {0, 0, 0, 4, 1, 9, 0, 0, 5},
            {0, 0, 0, 0, 8, 0, 0, 7, 9}
    };
    Sudoku sudoku = new Sudoku();
    fillSudokuFromGrid(grid, sudoku);
    sudoku.initializePossibleValues();
}
```

Using this class there are tests for the Sudoku class and for all the deduction rules.
The SudokuFileReader is tested using some test data files to not have random input in the file.

5. Conclusion

As part of this project, a Sudoku solver was implemented that uses prominent design patterns to provide an extensible structure. The interface allows the user to solve the Sudoku by hand or by using the implemented deduction rules. Black box and integration tests have been added for the most important classes.

In the future, the project could be expanded to include tests of other classes, such as the JavaFX-relevant classes, or other forms of testing, such as white-box tests and unit tests. Possible extensions to the application can be made in the area of the Sudoku database and the deduction rules used. Here are a few examples for more deduction rules:

1. Naked Pair
2. Hidden Pair
3. X-Wing
4. Swordfish
5. Jellyfish
6. XY-Wing
7. XYZ-Wing

With a little effort using the Model-View-Presenter pattern and an event bus, the application's architectural style could focus even more on the separation of concerns.

Regardless of the many expansion options, the current state of the project fulfils the requirements listed in chapter 2 and provides a solid foundation for a Sudoku solver application.

List of Figures

Bibliography

[Ref24] Refactoring.Guru. Creational design patterns, 9/17/2024.