HALMSTAD UNIVERSITY

# Sokoban project in Java

## Advanced Object-Oriented Programming

## Halmstad University

Written by: Sara Alterkawi, Rahaf Darouich and Nashwa Nanaa

HALMSTAD UNIVERSITY

# Abstract

This report documents the design and development of a tile-based puzzle game called Sokoban, implemented in Java. The game utilizes the Model-View-Controller design patterns, with two distinct views of the game state provided: a graphical view using Swing, and a console-based text view for debugging and logging. The game model keeps track of the current level, game state, and events while accepting input from the keyboard. The report also includes a testing section, where different parts of the program are tested using both manual and automated techniques. The report concludes with a discussion of interesting aspects of the program, its results, and experiences with version control.

HALMSTAD UNIVERSITY

# Table Of Contents

# 1. Introduction

The objective of this project is to design and develop a Java-based tile-based puzzle game, Sokoban, that incorporates the MVC design patterns. The game is intended to provide an enjoyable user experience using various repluggable methods of control, including keyboard through the provision of multiple views of the game. This report discusses the process of designing and implementing the game, including considerations of functionality, design, implementation, and testing. It also highlights the unique aspects of the program and how it differs from similar applications. Finally, the report provides an overview of the organization of the program, testing procedures, interesting coding solutions, and results, as well as a discussion of version control and sources of information.

# 2. Design

The Sokoban game project is designed using the Model-View-Controller (MVC) architecture pattern, which separates the software into three components responsible for managing data, rendering the user interface, and handling user input (Figure 1). To enhance the software's modularity, maintainability, and extensibility, the project implements design patterns such as the observer and strategy patterns.

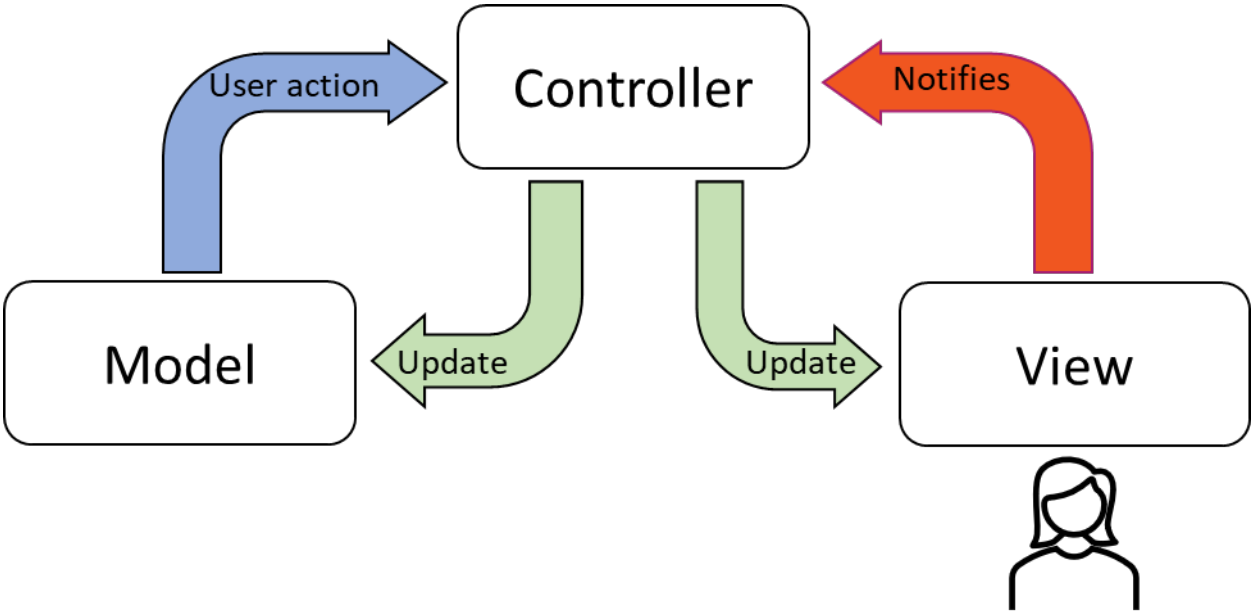


Figure 1: MVC pattern explanation

The project comprises five classes Main, Model, View, ConsolePrinter, and Levels and an interface called Controller. The Model class manages the game state and logic, while the View class handles the game's display. The Controller interface enables the View to interact with the Model, while the ConsolePrinter class outputs game-related information to the console. Finally, the Levels class provides methods for loading and changing levels, and together, these classes form the foundation of the Sokoban game, enabling users to play, interact with, and view information about the game (Figure 2).

## 2.1 Model

The Model class is responsible for managing the state of the game. The Model contains the game board, the position of the player, and the position of the boxes. The Model also loads and manages the images used in the game.

The game board is represented as a 2D array of Strings. Each element of the array represents a cell on the game board. The contents of a cell include wall, free space, box, goal place, box on goal, Sokoban, and Sokoban on goal. The Model provides methods for accessing and updating the contents of cells on the game board. The Model also contains a LinkedList of ChangeListeners. This allows other objects in the

program to be notified of changes to the game's state, which can be useful for coordinating different parts of the program or updating the user interface.

## 2.2 ConsolePrinter

The “ConsolePrinter” class represents the grid and provides methods to set and get the content of each cell in the grid. The ConsolePrinter class is responsible for printing the contents of the grid to the console. The ConsolePrinter class implements the ChangeListener interface, which allows it to be notified when changes are made to the Model class. The generateOutput() method in the ConsolePrinter class generates a String representation of the grid by iterating over each row and column in the grid and appending the content of each cell to a StringBuilder object.

## 2.3 View

The “View” class assumes responsibility for implementing the model component of a Sokoban game. The model component undertakes the crucial task of keeping track of the game state, encompassing the game board, player and box positions, and the prevailing state of the game, such as whether the player has won or lost. It also provides means of updating the game state, which includes facilitating player and box movement.

To enable the effective management of the game board, the class incorporates instance variables that can store the game board's width and height, the current level, and the positions of the player and boxes. Furthermore, it utilizes additional instance variables to store images for various game objects, like walls, boxes, and the player.

The class features diverse methods that aid in loading level data, setting the game board, and getting and setting the content of individual cells on the game board. It also offers methods that facilitate determining whether the player has won or lost the game.

Of particular interest is the class's use of the Java 8 Streams API to flatten the 2D game board array into a 1D stream of cells. Subsequently, this 1D stream of cells is leveraged to verify if all cells are marked with something other than a box to determine if the player has won the game.

## 2.4 Levels

The "Levels" class is designed to create levels of a game. It also contains a static 3D array called "gameLevels" that holds the data for several levels of a game. This project has three levels for the game Level1 (1), Level2 (2) and Level3 (3). Level class includes an array of strings that contains the data for the levels, where each level is represented as a two-dimensional array of strings. The code is designed in an object-oriented programming style and uses the Java Swing library for creating the user interface.

Levels class extends the Component class and implements the ChangeListener interface, which allows it to listen for changes in the state of the Model object. The class has an instance variable called "currentLevel" that keeps track of the current level. It also has an instance variable called "model" which is a reference to a "Model" object. The constructor initializes the instance variables and adds the Levels object as a listener to the Model object.The class also includes a static variable numMoves and three static

methods: moved(), getNumMoves(), and getCurrentLevel(). The moved() method increments the numMoves variable, the getNumMoves() method returns the current value of numMoves, and the getCurrentLevel() method returns the data for the current level.

The stateChanged() method is called when the state of the Model object changes. If the player has won the game and the game is still running, a message is displayed using the JOptionPane class, and the current level is updated. If the player has lost the game, the current level is reset, and the number of moves is set to zero. The updateLevelData() method updates the current level and calls the levelDataSetting() method of the Model object to set the data for the new level.

## 2.5 Controller

An interface named Controller that contains a single method signature getDataSaver(), which returns a String value. This interface is intended to be implemented by classes that act as controllers for a Model-View-Controller (MVC) architecture. In the MVC pattern, the Controller acts as an intermediary between the View and Model components, allowing the user to interact with the application and updating the Model accordingly.

The getDataSaver() method defined in the Controller interface is intended to retrieve the user input data from the View component, which can then be used to update the Model. The specific implementation of this method will depend on the requirements of the application and the specific interactions between the View and Model components. By defining this method in an interface, different implementations of the Controller can be created, allowing for flexibility and modularity in the design of the application.

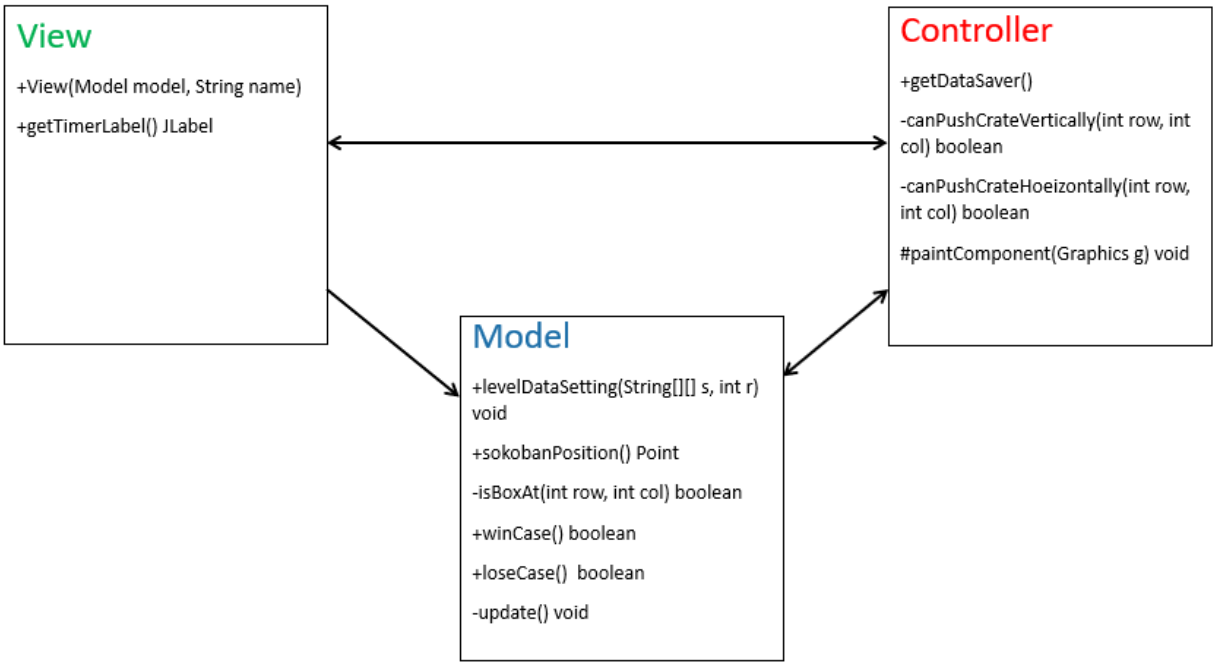


Figure 2: structure overview

# 3. Testing section

To ensure that our programs meet the project's expected requirements, the code underwent a thorough testing process that involved multiple steps. Initially, the Model-View-Controller (MVC) pattern was evaluated by transferring data without engaging the Graphical User Interface (GUI). The validation process for this phase was conducted over a considerable period using the console. During the test, conventional issues such as NullPointerException were detected. To address such issues, both the console and debugging tools integrated within Intellij were utilized for verification and resolution. In this regard, it is crucial to tackle one problem at a time and progress through the stages in a systematic and methodical manner.

To further evaluate the effectiveness of our project, we ran additional tests where we created multiple levels to explore different scenarios players might encounter during gameplay. With this in mind, we took a close look at the LoseCase, and WinCse functions to determine their accuracy and proper functioning.

# 4. Interesting parts

An intriguing and uncomplicated execution in game development is the utilization of player animation.

**loadImg():** This function loads images used in the Sokoban game, such as wall, box, and sokoban images. The interesting part of this method is that it uses the ImageIO class to read images from the file system and then assigns the loaded images to the corresponding BufferedImage fields in the class. This makes it easy to reference the images later in the program, such as when drawing the game board (Figure 3).

```
private void loadImg() {
    try {
        wallImg = ImageIO.read(new File( pathname: "sokoban_icons/wall.png"));
        freeSpaceImg = ImageIO.read(new File( pathname: "sokoban_icons/freeSpace.png"));
        boxImg = ImageIO.read(new File( pathname: "sokoban_icons/box.png"));
        goalPlaceImg = ImageIO.read(new File( pathname: "sokoban_icons/goalPlace.png"));
        boxOnGoalImg = ImageIO.read(new File( pathname: "sokoban_icons/boxOnGoal.png"));
        sokobanImg = ImageIO.read(new File( pathname: "sokoban_icons/sokoban.png"));
        sokobanOnGoalImg = ImageIO.read(new File( pathname: "sokoban_icons/sokobanOnGoal.png"));
        } catch (IOException e) {
        System.out.println("Image is not existed !!");
    }
}
```

Figure 3: “loadImg” method

**winCase():** This function checks if the player has won the game by checking if all the cells on the game board contain something other than a box. The interesting part of this method is that it uses Java 8 Streams API to flatten the 2D game board array into a 1D stream of cells and then checks if all cells are

marked with something other than a box. This is a concise and efficient way to check if all the cells contain something other than a box (Figure 4).

```
public boolean winCase() {
    // Use Java 8 Streams API to flatten the 2D game board array into a 1D stream of cells.
    // Check if all cells are marked with something other than box.
    return Arrays.stream(gameBoard) Stream<String[]>
            .flatMap(Arrays::stream) Stream<String>
            .allMatch(cell -> !cell.equals(box));
}
```

Figure 4: “winCase” function

**loseCase():** This function checks if the player has lost the game by checking if all the boxes are stuck and cannot be moved anymore. The interesting part of this method is that it first checks if the player has won the game to avoid false positives. Then it checks each box on the game board and determines if it can be pushed vertically or horizontally. If none of the boxes can be moved anymore, the game is considered lost. This is a smart and efficient way to check if the player has lost the game (Figure 5).

```
public boolean loseCase() {
    if (winCase()) {
        return false;
    }
    for (int row = 1; row < numRows - 1; row++) {
        for (int col = 0; col < numCols; col++) {
            if (isBoxAt(row, col) && canPushCrateVertically(row, col)) {
                return false;
            }
        }
    }
    for (int row = 0; row < numRows; row++) {
        for (int col = 1; col < numCols - 1; col++) {
            if (isBoxAt(row, col) && canPushCrateHorizontally(row, col)) {
                return false;
            }
        }
    }
    return true;
}
```

Figure 5: “loseCase” function

This part of the code is also interesting because it's a simple way to keep track of the number of moves the player has made. The moved() method is called every time the player moves, which increments the numMoves variable. The getNumMoves() method allows other parts of the code to retrieve the current number of moves (Figure 6).

```
public static int numMoves = 0;

4 usages
public static void moved() {
    numMoves++;
}

no usages
public static int getNumMoves() {
    return numMoves;
}
```

Figure 6: “moved” & “getNumMoves” functions.

# 5. Result

The program is well-structured and designed in a modular way, which makes it easy to add new functionality without affecting existing code. For example, if one wants to add new levels to the game, it can be done by simply adding new level files to the existing levels folder. This modular approach makes the program highly extensible. The program is also easy to modify if a better implementation of a part is possible. For instance, if a more efficient algorithm for the game AI is found, it can be easily incorporated into the program without affecting the existing code. The code is well-documented, which makes it easier for new developers to understand the codebase and make modifications as necessary.

Overall, the Sokoban project was a success, and the resulting program is of high quality. Its modularity and extensibility make it easy to modify and add new functionality, making it a robust foundation for future development.

# 6. Version control/ GIT

The project's development was managed through GitHub version control, which provided a robust and efficient system for code management, collaboration, and issue tracking. The use of Git enabled developers to work on the project independently and in parallel, while minimizing conflicts and making it easier to revert changes if needed. With version control in place, the team was able to collaborate seamlessly, track changes and history, and quickly identify and resolve issues, all of which streamlined the development process.

# 7. References

1. Sokoban Online."Easy Level" *Just for Fun.* https://www.sokobanonline.com/play/just-for-fun/25990_easy-level (accessed May 10, 2023).

2. Sokoban Online."Emerald". *Just for Fun.* https://www.sokobanonline.com/play/just-for-fun/122112_emerald (accessed May 10, 2023).

3. Math is Fun. "Sokoban" *Math is Fun*, https://www.mathsisfun.com/games/sokoban.html (accessed May 10, 2023).