Sokoban Project

Advanced object-oriented programming

William Olsson, Filip Vikström, Nicholas Sjögren

Report Contents

1. Introduction

This is the final project report for the Advanced Object-Oriented Programming course. The subject is taught to engineering students at Halmstads University.
We were tasked with developing a system that could handle simple 2D logic puzzle games like As. We had a few needs for functionality and design before we started the project. The framework should be simple to use and greatly assist you in developing a 2D logic puzzle game.

Going into the project, we got some requirements on the functionality and design, and we had the Model-View-Controller pattern in mind. It was chosen because of its ease of use, and the framework/design helped implement the game. The game is divided into several classes and in different packages, where the pattern primarily resides. This is done to organize and separate the functionality into logical components. Our pattern also separates specific concerns, making it easy to maintain and modify.

The report will include below sections:

Software Development: Includes the development of the program and descriptions of various classes and packages in the program.

Testing: Includes a description of what unit and integration testing were done throughout the program.

Interesting Changes: Section where we discuss the problems encountered and improvements made to the program.

Result: Includes a summary of the goals reached by the Pattern and design of the program.

Version Control (GitHub): Includes our experience with the version control system used during the project.

2. Software Development & Design

In this part, we will describe our project design. The project consists of the MVC-Pattern and the Sokoban program, which was built following the principles of that design.

2.1 Framework Choice

2.1.1 Objectives

We made several design choices to make the user interface more intuitive and improve the overall gameplay experience. Here are the main objectives we had in mind.

• Abstract JFrame handling to simplify the game window creation process.

• Implement a grid-based rendering system for map tiles and game entities.

• A 'Player' class to manage player character properties, grid-based movement, and collision handling.

• Facilitate updating and drawing player and tile entities within the game panel.

• Utilize separate classes to adhere to the Model-View-Controller (MVC) design pattern.

• Implement the Model-View-Controller (MVC) design pattern to manage the game's structure and functionality.

• Apply Object-Oriented Programming (OOP) principles by creating a 'GameObject' superclass to store shared attributes of game elements.

2.1.2 Development

The design principle for the game was to create a platform that could easily manage different game components, maps, players, and tiles, without the user directly interacting with the JFrame. For the layout, a grid is created containing components representing the game view to achieve this. We used various classes such as **‘GamePanel,’ ‘GameWindow,’ ‘MapView,’ ‘PlayerView,’ ‘TileView,’** and **‘ConsoleView’** to create the visual structure of our game. The **‘GamePanel’** class creates the central panel of our game window, managing the size and layout of the game view. It also holds the map and player views, critical game components.

The **'MapView'** class is in charge of displaying the map on the game panel. It iterates through the grid, drawing each tile at the appropriate position. The depiction of the player and tiles is handled by the **‘PlayerView’** and **‘TileView’** classes. The **‘ConsoleView’** class displays the game's current state in the console, which allows debugging and understanding the current game state without opening the game window. Controller classes is used to control the game, listening to inputs from the user, such as moving, toggling on or off the console/gui view, switching between ways to control the game, switching the level, or resetting the game. **‘KeyHandler’** and **‘ButtonControlHandler’** are the main classes used for controlling and updating the state of the game. There are also smaller control classes, **‘reset’**, **‘levelChooser’**, **and ‘CheckBoxHandler’** that listen for if events in the side menu happen, such as resetting the game, changing the level, or turning on/off the console/gui view.

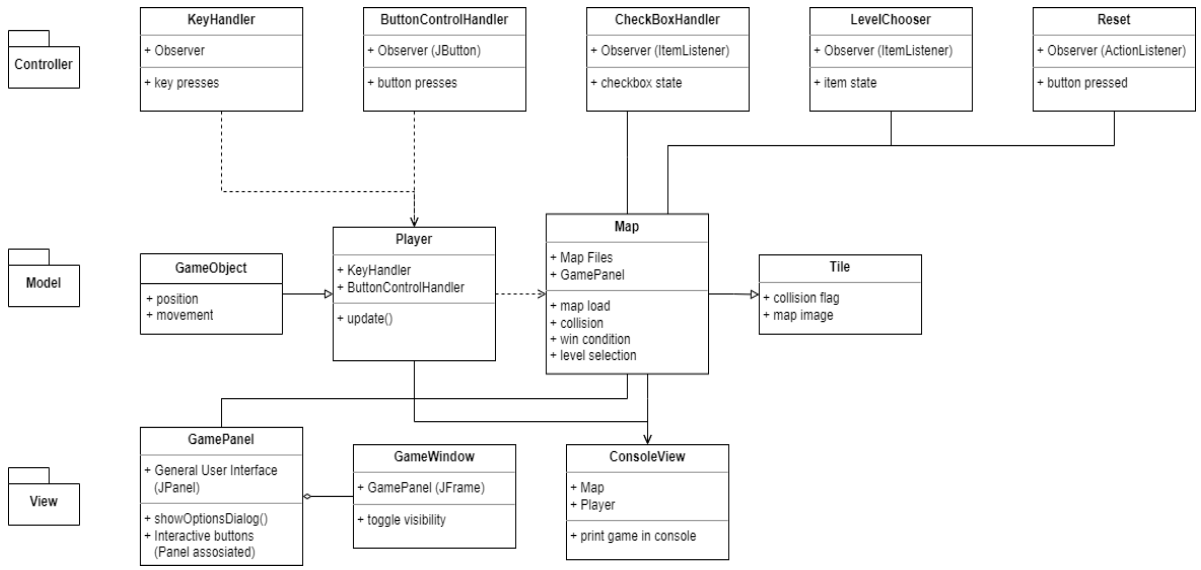
2.2 Sokoban Game

2.2.1 Objectives

• Structured code

• Modifiable and reusability on classes.

• Easy and understandable code by separation of packages.



2.2.2 Model

The game mechanics and logic reside here, for example, in the **“Player”** class. It handles both movement logic and how it should handle the walls and crates. First, we built it so that each tile is a number in a text document where we could quickly try out the first map and our functions. This made us build different kinds of tiles and see if they responded like we wanted them to and had different functionalities. We got four different tiles in our map class in our latest build. It uses a " Tile " class to tell if the collision is false or true. The **‘Map’** is constructed based on a two-dimensional array of integers that represent different tile types in the game world. To begin this procedure, the Map class is initialized with a **‘GamePanel’** object. An array of Tile objects is formed to hold the tile numbers, and a two-dimensional array is established. The **‘getTileImage()’** function retrieves the pictures for each type of tile, whereas the **‘mapLoad()’** method reads a text file containing the tile numbers for each tile in the map and converts it to a multidimensional array which is later used to change and read the state of the game. Implementing the map in this way makes it very easy to add more levels without changing the code. The **‘Map’** also contains the method of changing level **‘changeLevel(int level)’**, which loads a new map using **‘mapLoad()’** and other methods to set the new parameters of the new map. It changes the tile size and the game's width and height depending on how big or small the new map is. The Map class also loads the player's picture and sets its starting location, the number of levels, if the current map is finished, or how far the user come at the current level.

2.2.3 Controller

The Controller package is accountable for all the user's interactions with the program. This includes the keyboard button inputs, interactions with the user interface, etc. The classes included in the controller package are **‘KeyHandler’, ‘ButtonControlHandler’, ‘CheckBoxHandler’, ‘levelChooser’**, and **‘reset’**. These classes are designed based on the “Observer”- pattern, the classes are responsible for delivering inputs to the ‘model’ part of the project. The classes are designed as separate modular input receivers in such a manner that new observers could easily be added or removed without interfering with the rest of the project.

The class **‘KeyHandler’** is the primary way of controlling the game, handling user inputs, and moving the player tile accordingly. Registers which button was pressed and updates the **‘GamePanel’** accordingly. The smaller control classes **‘reset’, ‘levelChooser’, ‘CheckBoxHandler’,** and **‘ButtonControlHandler’** listens if either of the buttons in the menu is pressed. **‘reset’** resets the current map to its starting position. ‘**levelChooser’** is a drop-down menu where it is possible to choose the level you want to play. ‘**CheckboxHandler’** is where you can turn on or off the viewing of the game on the console. **‘ButtonControlHandler’** is an alternative way of controlling the game with mouse-interactive buttons provided by JButton from the swing API.

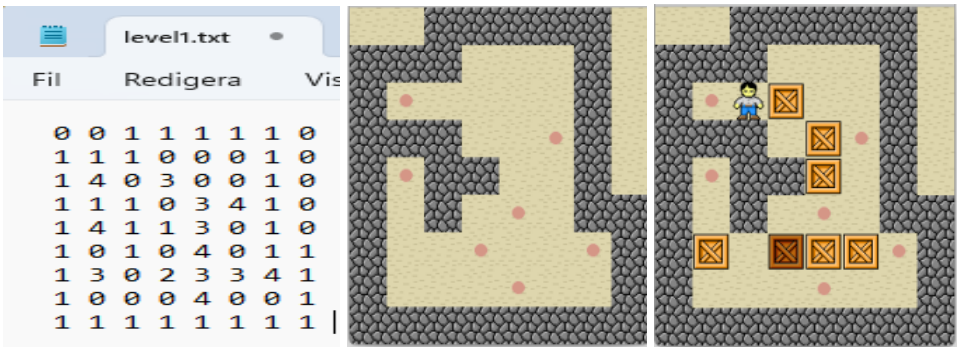
The controller package was designed to establish a clear separation of responsibilities as part of the Model-View-Controller (MVC) architectural pattern. The controller acts as an intermediate between the model (the game logic and data) and the view (the game's graphical user interface) in the MVC paradigm. This split keeps user interaction logic separate from game logic, making maintaining and modifying the code easier.

The class implements the **‘KeyListener’** interface, focusing primarily on the keyPressed method to capture keyboard events for player movement. A flag keyProcessed flag avoids multiple processing of a single key press, blocking further processing until the key is released. The keyPressed method identifies the ASCII value of the pressed key, setting a corresponding static direction variable to true. The Player class uses these variables (upPressed, downPressed, leftPressed, rightPressed) to decide the player's movement direction.

This controller design choice and the MVC pattern guarantee that our game is adaptable, manageable, and scalable. Future improvements or modifications to the game's control logic may be implemented in a single location, the ‘**Controller’** package, without altering the entire game logic or display components.

2.2.4 View

This contains two parts the view and the view. Entity. The view package contains all the classes related to the corresponding GUI and visual representation of the game. The main **'GamePanel'** is where the game is played and handles the game updates and drawing for both the player and the map and the **'GameWindow'** simply creates a new window of the game. **'ConsoleView'** is our second view, displaying the map in the console. The user can see our character coordinates and the character marked with an X on the map. When making a different class to view the game, we implemented a method that enables the user to toggle between the console view and the main game window view. The purpose of this implementation was to prove the modularity of the game sections, which means that the game project could include other different ways of viewing the game without being dependent on the other viewing method. This was also really helpful when we debugged. To adhere more to the MVC pattern, we created a sub-package to view called entity. View holds classes representing views of different entities of the game. It consists of three classes map view, player view, and tile view. The first one handles the map rendering, composed of the tiles, and the tile view handles the rendering of the individual tiles on the map. Lastly, we have the player view that simply handles the rendering of the player.



3. Tests

The application was evaluated in two stages: first, using a JUnit test that tested most of the methods independently, and then with an integration test that included extensive gameplay testing.

3.1 Integration testing

The integration test was carried out by playing the game on several maps and experimenting with different movement combinations. This was done to ensure that all classes interacted adequately and that the whole software and game mechanisms operated as planned.

3.2 Unit tests

For the unit testing, we used the JUnit framework, which we used to test some of our representative classes.
We started by testing our Player class because it holds most of our game logic, mechanics, and movement. So the method in it that was considered the most essential was our **'move()'** method. The test was to see if it behaved as we were expecting. This means that we verified the properties of the different tiles, a blank Tile, for example, is occupiable but not mobile. Therefore, these attributes were verified for each tile. We start by checking the position, followed by simulating the keypresses into if the player has moved to the right. Then likewise for a downpress and if the player moved.

The **'PlayerTest'** class is a JUnit test class that tests the functionality of the Player class. The test class contains three test methods: testPlayerInitialization, testSetStartValue, and testMovePlayer.
In the **'testPlayerInitialization()'** method, the test verifies that the player object is initialized correctly. It checks if the initial values of gridX, gridY, x, y, and speed are set as expected. Using assertions, the test ensures that the values match the expected values.

The **‘testSetStartValue’** method tests the setStartValue method of the Player class. It sets a start value for the player's position and then verifies if the gridX and gridY values are updated correctly by using assertions.
The **‘testMovePlayer’** method tests the movement functionality of the player. It sets an initial position for the player and then simulates the movement in different directions using the move method of the Player class. After each movement, assertions are used to check if the gridX and gridY values are updated correctly.

These tests cover the **‘Player’** class's initialization, establishing start values, and movement functions. They contribute to ensuring that the class acts as intended and can be used confidently in the application.

```
28    @Test
29    public void testPlayerInitialization() {
30        // Verify that the player object is initialized correctly
31        Assert.assertEquals(0, player.gridX);
32        Assert.assertEquals(0, player.gridY);
33        Assert.assertEquals(0, player.x);
34        Assert.assertEquals(0, player.y);
35        Assert.assertEquals(1, player.speed);
36    }
37
38    @Test
39    public void testSetStartValue() {
40        // Test setting the start value of the player's position
41        player.setStartValue(3, 2);
42        Assert.assertEquals(3, player.gridX);
43        Assert.assertEquals(2, player.gridY);
44    }
45
46    @Test
47    public void testMovePlayer() {
48        // Set up initial position
49        player.setStartValue(3, 2);
50
51        // Test moving the player up
52        player.move("UP");
53        Assert.assertEquals(3, player.gridX);
54        Assert.assertEquals(2, player.gridY);
55
56        // Test moving the player down
57        player.move("DOWN");
```

(**Note:** Code that checks if the right press moves the character to the correct tile)

We then decided to test the Map class, which was responsible for the whole game map, including collisions and tile images. We proceeded then to test **‘isCollidable(),’** also a quick movement test was performed to ensure that the index representation of a moveable tile's position was successfully updated following a move.

```java
public class MapTest {
    @Test
    public void testIsCollidable() {
        Main main = new Main();

        GamePanel gamePanel = new GamePanel(main);
        Map map = new Map(gamePanel);

        // Test that the map's collidable tiles behave as expected
        // (replace with the grid coordinates of a wall or crate in our map)
        assertTrue(map.isCollidable(4, 2));  // Wall or crate

        // (replace  with the grid coordinates of a blank space in our map)
        assertFalse(map.isCollidable(3, 2));  // Blank space
    }
}
```

(**Note:** Checks collision is working etc.)

4. Interesting solutions

4.1 Problems solved

One of the most concerning and threatening problems we faced was when we were almost done with all the code and got the core game to function. Was that our code had strayed away from the MVC-Pattern, which we had intended to use and fully follow. Over time, as features were added and the codebase grew, game logic, user input processing, and data rendering had become intertwined and scattered throughout the code, making it hard to understand, maintain, and extend. For instance, we had to draw functions in our “model package” that handled different views, which was a grey area, but a violation of the MVC-Pattern.
Similarly, the Player class was responsible for the character properties and the character's movement and collision detection, which mixed the responsibilities of model, view, and controller.

The solution to this was to sit down and take our time to refract the code base to be more in line with the MVC-Pattern. We did this by first creating packages to separate the view, model, and controller. We then restructured the model package only to include game logic and data. We moved everything rendering the game and its elements to the view package and separated them into various classes. We then moved all player user inputs to our KeyHandler instead of being half there and half in the player class. Lastly, we ensured one-way dependencies between the model, view, and controller, with the controller depending on the model and view and the view depending on the model. This restructuring made our code easier to understand, debug, test, and extend, improving long-term maintainability.

An example of a wrongly placed method was the **‘update().’** This is in the current player class before it resided in the GamePanel class in the view package. Furthermore, the player state update logic was unnecessarily complex, making it hard to maintain and modify in the future. As we can also see, the code is much more convoluted and redundant for that little shown part.

```
public void update() {
        // updating player
        if(KeyHandler.upPressed) {
            if(!player.isCollidable(player.gridX, player.gridY - 1)) {
                player.gridY--;
            }
        } else if(KeyHandler.downPressed) {
            if(!player.isCollidable(player.gridX, player.gridY + 1)) {
                player.gridY++;
            }
        } else if(KeyHandler.leftPressed) {
            if(!player.isCollidable(player.gridX - 1, player.gridY)) {
                player.gridX--;
            }
        } else if(KeyHandler.rightPressed) {
            if(!player.isCollidable(player.gridX + 1, player.gridY)) {
                player.gridX++;
            }
        }


        player.x = player.gridX * tileSize;
        player.y = player.gridY * tileSize;
        consoleView.update();
```

(**Note:** A none MVC-Pattern update() method)

Another change that had to be made was in our **'mapLoad()'** method, where before, it read each line without checking if the maximum rows had been reached, which sometimes led us to get **'ArrayIndexOutOfBoundsException.'** This was changed to our current version, which will always count and check.

```
    public void mapLoad() {
        try {
            InputStream is = getClass().getResourceAsStream("/maps/Map.txt");
            BufferedReader br = new BufferedReader(new InputStreamReader(is));
            String line;
            int row = 0;
            while((line = br.readLine()) != null) {
                String numbers [] = line.split(" ");
                for(int col = 0; col < gp.maxScreenCol; col++) {
                    int num = Integer.parseInt(numbers[col]);
                    mapTileNum[col][row] = num;
                }
                row++;
            }
            br.close();
        }catch(Exception e) {
            e.printStackTrace();
        }
    }
```

(**Note:** names and variables have changed since the shown version)

5. Results

The application was successfully implemented using the Model-View-Controller (MVC) - Pattern. All our goals were ultimately reached, especially around the pattern we decided on. The application also reached our goals, and everything worked as expected. The code is organized into three main categories: the Model, View, and Controller.

The pattern improved our code substantially. First of all, by following it, the codebase was a whole lot more modular. Each one of our classes has a clear purpose. The separation of all of the responsibilities leads to a cleaner and easier-to-read codebase. Further on, the code for the game is now easier to maintain and extend. For example, only the **'Player'** class must be modified if a new feature is introduced to the game logic. If it is not a substantial new feature, then a new class to the model package is required. This significantly limits the possibility of introducing flaws into other portions of our software.
Another highlight is how the **'Player'** class handles our movement. The method **'update()'** checks our key presses and will update the player accordingly. It also checks for possible collisions using another method called **'isCollidable()'** in the **'Map'** class, which ensures correct movement.

We are proud of how we produced the **Model** component that encapsulates the game's fundamental business logic and data state. This section contains the GameObject, Map, Player, and Tile classes. The GameObject class is a superclass for many elements in the game, such as the player and crates. The Map class maintains the game's map, loading it from a text file and handling the game tiles. The Player class is responsible for the player's movements and interactions in the game. The Tile class represents the various tiles in the game. We could easily add extensions and modifications to the game by adhering strictly to the MVC-Pattern at this part. It also allowed us to easily change or add different game mechanics with this design build-up approach.
Overall, the code adheres to the MVC pattern successfully. The model, view, and controller components are cleanly separated, making the code easier to comprehend

and maintain. Furthermore, the code is grouped into proper packages, which aids in understanding the organization and structure of the code.

6. Experience with GitHub

This section details our first-hand experience utilizing a version control system for our project. This was an initial endeavor for all group members, as this was the first time anyone had interacted with such a system. We selected **GitHub** as the platform for our project management. We generally utilized **GitHub** to distribute our work, enabling mutual access and collaborative usage of shared code. Additionally, we utilized the repository as a failsafe storage mechanism for our files. However, we encountered several complexities during the process, particularly when merging branches, often with the main branch. This resulted in numerous errors and unexpected modifications, rendering the code useless. Also, we tried several things in sharing the code through **GitHub** but have yet to get it to work. This was probably due to our lack of experience and knowledge of the program. Regardless of these early struggles and failed attempts to implement a version control system, we collectively recognize the potential advantages of incorporating such a system in future applications and projects. Going forward, we commit ourselves to enhancing our proficiency and learning the appropriate usage of version control systems.

7. References

[1] Sokoban game tile images. Format: .png Sponsor of source: Wojciech Mostowski, Available at The courses website at
https://bb.hh.se/
(Accessed: 26 May 2023)

[2] Wikipedia (2023) MVC-Pattern, Available at:
https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
(Accessed: 26 May 2023).

[3] Wikipedia (2023) Observer pattern, Available at:
https://en.wikipedia.org/wiki/Observer_pattern
(Accessed: 26 May 2023).