Sokobon

Ali Abedrabbo
Abd Alrahman Atieh
The King Abdulfattah Morad

November 24, 2023

Abstract

This report describes the difficult logic-based warehouse puzzle game Sokobon's development process. A group of students developed a trustworthy and entertaining gaming experience using the Model View Controller (MVC) design pattern. Manipulating containers to specific goal locations within the warehouse is the game's purpose.

The project's goals, which were to provide a playable and dependable Sokobon game that captures players' attention, are outlined in the report. Throughout the development process, the adoption of design patterns, including MVC, Observer, Strategy, and Template aided code structure and problem solving.

The implementation process is covered, emphasizing any issues encountered and how they were successfully resolved. The resultant game runs without a hitch and gives users a fun gaming experience. The team's findings, which highlight the Sokobon game's effective implementation, are presented as the report's conclusion.

The team's ability to create an engaging gameplay experience through the successful implementation of the Sokobon game is demonstrated by this report, which also acts as a testimonial to their problem-solving abilities, efficient coding techniques, and aptitude to do so.

Contents

1 Introduction

Sokoban, a popular logic game, challenges players to navigate a warehouse by strategically moving crates into designated goal positions. This project focused on creating and implementing a Sokoban game to deliver a reliable and enjoyable gaming experience for players.

Led by a group of talented students, the project drew inspiration from the Sokoban video game. To ensure a well-structured and organized codebase, the team leveraged various design patterns, including Model View Controller (MVC), Observer, Strategy, and Template. These patterns facilitated the separation of concerns and provided a flexible foundation for future enhancements and customizations.

The team's commitment to delivering a seamless gaming experience led to rigorous testing and debugging procedures. By identifying and addressing any issues or bugs, they ensured that the game ran smoothly and provided players with an engaging and satisfying gameplay environment.

This report encompasses a comprehensive overview of the Sokoban game, covering various aspects such as the application's specifications, layout, and the implementation of key components such as the model, view, controller, and game engine. Additionally, the report delves into the testing and debugging procedures employed to maintain a high level of quality. Noteworthy features of the framework's application are also highlighted, showcasing the team's innovation and creativity.

The research also examines the team's use of crucial tools for collaboration, such as JavaDoc documentation and version control (GIT). Throughout the course of the project, these technologies were essential for enabling effective teamwork and code management.

By successfully implementing the Sokoban game, and the team demonstrated their problem-solving skills, proficient coding practices, and the ability to create an entertaining gameplay experience. The report concludes with reflections on the project's accomplishments and provides insights into potential future enhancements that could further enrich the Sokoban game.

Overall, this project showcases the team's prowess in developing a well-designed and captivating Sokoban game using the provided framework, underscoring their ability to deliver a top-quality gaming experience.

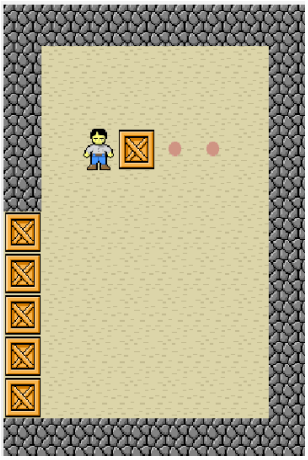

Figure 1: Sokobon implementation with the framework

2 Descriptions of the Class

2.1 Short descriptions of the class

Class: Sokobon
The Sokobon class represents the main entry point of the Sokobon game. It contains the main() method, which initializes the game and starts the game loop. The game loop continuously updates the game state, handles user input, and renders the game. It utilizes the GameMap and DataModel

classes to manage the game state and provides methods for initializing the game, updating the game state, handling user input, and rendering the game.

Class: GameMap

The GameMap class represents the game map in Sokobon. It is responsible for storing and managing the game objects on the map. The map is represented as a two-dimensional grid of cells, where each cell can contain a game object such as a wall, floor, box, goal, or player. The class provides methods for initializing the map, retrieving and setting game objects at specific positions, checking for wall collisions, checking if a level is completed, and rendering the map.

Class: GameObject

The GameObject class is the base class for all game objects in Sokobon. It defines common properties and behaviors for game objects, such as position and icon. Each game object has a unique ID that identifies its type. The class provides methods for retrieving and setting the position of a game object, getting its ID, and rendering the game object on the game map.

Class: Wall

The Wall class represents a wall object in Sokobon. Walls are immovable objects that block the player and other game objects from moving through them. The class extends the GameObject class and sets the ID and icon for wall objects.

Class: Floor

The Floor class represents a floor tile in Sokobon. Floors are empty spaces on the game map where other game objects can be placed. The class extends the GameObject class and sets the ID and icon for floor tiles.

Class: Goal

The Goal class represents a goal tile in Sokobon. Goals are special tiles where boxes need to be pushed to in order to complete the level. The class extends the GameObject class and sets the ID and icon for goal tiles.

Class: Player

The Player class represents the player object in Sokobon. The player is controlled by the user and can move around the game map, pushing boxes to complete the level. The class extends the GameObject class and sets the ID and icon for the player object. It provides methods for moving the player in different directions and handling interactions with other game objects, such as Movingboxes and goal tiles.

Class: MovingBox

The MovingBox class represents a box object that can be moved in Sokobon. It extends GameObject class and adds additional functionality to handle box movement. It provides methods for moving the box in different directions, checking for collisions with walls and other boxes, and checking if the box is on a goal tile.

Class: DataModel

The DataModel class represents the data model of the Sokobon game. It stores and manages the game state, including the current level and the number of completed levels. The class provides methods for initializing the data model, updating the game state, and retrieving the game state information.

Class: SoundsEffects

The SoundsEffects class is responsible for playing sound effects in Sokobon. It uses the Swing library to play sound files. The class provides static methods for playing different sound effects, such as level-up sounds, horn sounds, box movement sounds, and new level sounds. It also implements the ChangeListener interface to listen for changes in the game state and play the corresponding sound effects.

2.2 Long descriptions of the classes

2.2.1 DataModel

The given code represents a Java class called **DataModel** in the **sokobon.models** package. It is used to store and manage the data of a game, as well as update observers when the data changes. Here is a breakdown of what the code does:

The code defines the **DataModel** class and imports necessary dependencies. The class declares instance variables, including:

- **listeners** : An **ArrayList** to store change listeners for the observer pattern.
- **map** : A 2D array of **GameObject** representing the current state of the game.
- **cols** and **rows** : Integers representing the size of the game.

- **levelCount** : The number of levels in the game.
- **currentLevel** : The index of the current game level.
- **levels** : An instance of the **Level** class to access the game levels.
- **gameMap** : An instance of the **GameMap** class for interaction with the game objects.
- **soundsToPlay** : An integer indicating the sound to play.

The constructor initializes the **DataModel** object by converting the initial level map (provided by **Level**) into a matrix of **GameObject** instances. There are several methods to manipulate and access the game data, such as:

- **addGameMap** : Associates a **GameMap** object with the **DataModel** .
- **addLevels** : Sets the game levels and initializes the current level.
- **attachGameMapToGameObjects** : Assigns the **GameMap** to each **GameObject** in the map.
- **getLevelCount** : Returns the number of levels in the game.
- **getCurrentLevel** : Returns the index of the current level.
- **convertCharMatrixToGameObjectMatrix** : Converts a char matrix map to a **GameObject** matrix.
- **getData** : Returns the current game data as a **GameObject** matrix.
- **attach** : Attaches a change listener to the **DataModel** .
- **update** : Updates the game data with a new map and notifies observers.
- **checkWin** : Checks if the player has won the game by examining the game state.
- **notifyObservers** : Notifies the registered observers that the data has changed.
- **updateLevel** : Updates the current level when the player wins the game.
- **loadState** : Loads a game state from a file and updates the game data accordingly.
- **getSoundsToPlay** and **setSoundsToPlay** : Getter and setter for the sound to play.

The provided code primarily focuses on managing the game data and notifying observers.

2.2.2 GameState

The given code represents a Java class called **GameState** in the **sokobon.models** package. This class is responsible for saving and loading the current state of the game.

Here's a breakdown of the class:

The class contains two static methods: **savecurrent**stateItem and loadastateItem . The **savecurrentstateItem** method is used to save the current state of the game to a file. It takes a **DataModel** object as a parameter, which represents the game data model. Inside the **savecurrentstateItem** method, the current state of the game is retrieved from the **DataModel** object. The method generates a file name based on the current level and the timestamp. It checks if the directory **/gameStates** exists and creates it if it doesn't. The method then writes the game state to a file in the **/gameStates** directory. It writes the data as a character matrix (char[][]) row by row. The **loadastateItem** method is used to load a game state from a file. It takes a menuBarView object and a **DataModel** object as parameters. Inside the **loadastateItem** method, a JFileChooser dialog is used to select a file from the **/gameStates** directory. The selected file is then passed to the **loadState** method of the **DataModel** object to update the game state.

2.2.3 menuBarView

Overall, the GameState class provides functionality for saving and loading game states using file I/O operations.

2.2.4 GameMap

The given code represents a Java class called **GameMap** in the **sokobon.views** package. This class acts as an interface between the game objects and the data model. It is responsible for updating the data model based on the actions performed by the game objects.

The class extends **JComponent** and implements the **ChangeListener** interface. It has instance variables such as **map** , **player** , **cols** , **rows** , **gameMap** , and **dataModel** . The **map** variable stores the current state of the game as a 2D array of **GameObject** objects. The **player** variable represents the player object that the user controls. The **cols** and **rows** variables store the size of the current game. The **gameMap** variable is a **JLabel** used to display the current state of the game. The **dataModel** variable represents the data model of the game.

The class provides various methods to interact with the game objects and update the data model:

- **setDataModel** : Sets the data model of the game.
- **getGameObject** : Gets the game object at the specified row and column.
- **setGameObject** : Sets a new game object at the specified row and column and updates the data model.
- **isEmpty** : Checks if the specified row and column is empty (Floor object).
- **isWall** : Checks if the specified row and column is a wall (Wall object).
- **isInRange** : Checks if the specified row and column is within the range of the current map.
- **isMovingBox** : Checks if the specified row and column contains a moving box (MovingBox object).
- **isPlayer** : Checks if the specified row and column contains the player object.
- **addPlayer** : Updates the player position and notifies the observers.
- **addMovingBox** : Adds a moving box to the specified row and column and updates the data model.
- **addWall** : Adds a wall to the specified row and column and updates the data model.
- **isMarked** : Checks if the specified row and column is marked (goal object).
- **getCols** : Returns the width of the map.
- **getRows** : Returns the height of the map.
- **toString** : Returns a string representation of the map.
- **setMap** : Copies the given map to the current map.
- **stateChanged** : Updates the map and repaints the game map label (Jlabel)when the state changes.
- **update** (private): Updates the data model with the given map.
- **update** (private): Updates the data model with the specified row, column, and game object.
- **toMultiLineHTML** (private): Converts the given map to a multi-line HTML string for display in a JLabel.

Overall, the **GameMap** class acts as a bridge between the game objects and the data model, allowing them to interact and update each other as the game progresses.

2.2.5 GraphPresenter

Sokobon includes two classes: **GameMap** and **GraphPresenter** . These classes are responsible for displaying the game map and updating it based on the game state.

The **GameMap** class extends **JComponent** and implements the **ChangeListener** interface. It serves as an interface between the game objects and the data model. It stores the current state of the game in a 2D array called map and provides methods to manipulate and retrieve information about the game objects. It also includes methods to add player, moving boxes, walls, and update the data model. The **GameMap** class is responsible for rendering the game map in a graphical way using a **JLabel** component.

The **GraphPresenter** class extends **JPanel** and implements the **ChangeListener** interface. It is used to display the game map graphically. It also stores the current state of the game in a 2D array called **gameObjects** . The **GraphPresenter** class overrides the **paintComponent** method to render the game map using Graphics2D and Image objects.

Both classes receive a **DataModel** object in their constructors, which is the underlying data model for the game.

Overall, these classes work together to provide a graphical representation of the game map and update it based on the game state stored in the data model.

2.2.6 GameObject

The code provided defines an abstract class called **GameObject** , which serves as a superclass for various game objects in the Sokobon game.

Here are the key features of the **GameObject** class:

- It implements the **Serializable** interface, indicating that instances of **GameObject** can be serialized.
- It defines instance variables to store the position of the game object (**posRow** and **posCol**) as row and column coordinates.
- It has a reference to the **GameMap** class, which represents the game map and allows the game object to interact with its neighbors.
- It contains a **BufferedImage** variable named **bufferedImage** , which is used by the **GraphPresenter** class to paint the game object.
- The **GameObject** class has a character identifier (**id**) that is used to distinguish between different types of game objects.
- It provides methods to get and set the position, get the character identifier, set the game map, set the icon image for the game object, and define movement methods for up, down, left, and right directions.

The **GameObject** class is intended to be subclassed by specific game objects, such as players, boxes, and walls. Each subclass is expected to implement the movement methods (**moveUp()** , **moveDown()** , **moveLeft()** , and **moveRight()**) based on its specific behavior in the game.

Overall, the **GameObject** class provides a common structure and functionality for game objects in the Sokobon game.

2.2.7 Level

The code provided defines a **Level** class that represents the different levels in the Sokobon game. Here's an overview of the class:

- The class has constants **CountLevel** , **height** , and **width** to define the number of levels, height of the map, and width of the map, respectively.
- The **mapLevel** instance variable is a three-dimensional array (**char[][][]**) used to store the map layouts for each level.
- The constructor Level() initializes the **mapLevel** array and calls the **setLevelMaps()** method to define the different levels and their layouts.
- The **setLevelMaps()** method assigns specific map layouts to each level using nested arrays of characters. Each character represents a different element in the game (e.g., wall 'hashtag', player 'p', box 'o', goal 'g', empty space ' ').

- The **getMapLevel(int level)** method returns the map layout for a specific level. It checks if the level number is within the valid range and returns the corresponding map layout.
- The **countLevel()** method returns the number of defined levels.
- The **main()** method demonstrates how to use the **Level** class by creating an instance and printing the map layouts for each level using the **printLevelMap()** method.
- The **printLevelMap(char[][] mapLevel)** method is a helper method used to print the map layout to the console.

Overall, the **Level** class provides a convenient way to define and access different map layouts for the Sokobon game levels.

2.2.8 Sokobon

The code provided represents the starting point of the Sokobon game in the **Sokobon** class. Here's an overview of the class:

- The **main** method is the entry point of the game.
- An instance of the Level class is created to access the different game levels.
- An instance of the **menuBarView** class is created to represent the game window.
- A **DataModel** object is created, passing the **Level** instance to it.
- Various view components, such as **GameMap** , **SoundsEffects** , **ControllerWindow** , **LevelDisplay** , and **GraphPresenter** , are created and associated with the **DataModel** .
- The view components are added to the game window (**menuBarView**).
- The layout and positions of the view components are set using **setBounds()** and **setLocation()** methods.

The **Sokobon** class also overrides the **toString()** method to provide a string representation of the game session, including the dimensions of the game window and the number of levels.

Overall, the **Sokobon** class sets up the game window, creates the necessary components for the game, and establishes the connections between the data model and the view components. It serves as the starting point for running the Sokobon game.

2.2.9 MovingBox

The code provided defines the **MovingBox** class in the **sokobon.GameObjects** package. Here's an overview of the class:

- **MovingBox** is a subclass of the **GameObject** class and represents a movable box in the Sokobon game.
- The class has two constructors: the default constructor and a parameterized constructor that takes the row and column position of the box on the game map.
- In the constructors, the id of the **MovingBox** is set to 'o' , and the icon for the box is set to "**crate.png**" using the **setIconGameObject()** method inherited from **GameObject** .
- The class provides methods to move the box in different directions: **moveUp()** , **moveDown()** , **moveLeft()** , and **moveRight()** . These methods internally call the **move()** method, passing the corresponding direction as an argument.
- The **move(int direction)** method is responsible for moving the box in the chosen direction. It updates the box's position and checks the new position for validity and collision with other game objects.
- If the new position is outside the game map's range or collides with another object, the box remains in its current position.
- If the new position is valid and empty, the box moves to that position. If the old position was marked as a goal, it is updated accordingly.

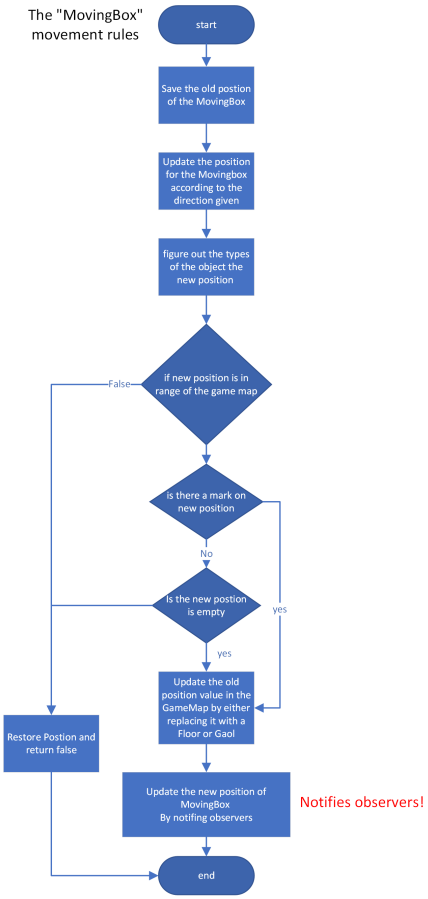

Figure 2: A Flowchart showing the MovingBox Logic

- If the new position is marked as a goal, the box converts into a marked box by updating its icon and id. If the old position was marked, it is updated as a goal. Otherwise, it is updated as an empty floor tile.
- The class also provides private helper methods **convertToMarked()** and **convertToUnMarked()** to handle the conversion of the box's type when it reaches or steps out of a goal destination.

Overall, the **MovingBox** class encapsulates the behavior and attributes of a movable box in the Sokobon game. It provides methods to move the box, update its position and appearance, and handles collision with other game objects.

2.2.10 Player

The code provided defines the **Player** class in the **sokobon.GameObjects** package. Here's an overview of the class:

- **Player** is a subclass of the GameObject class and represents the player-controlled character in the Sokobon game.
- The class has two constructors: the default constructor and a parameterized constructor that takes the row and column position of the player on the game map.
- In the constructors, the **id** of the **Player** is set to **'p'** , and the icon for the player is set to "player.png" using the **setIconGameObject()** method inherited from GameObject .
- The class provides private constants for the four directions: **UP** , **DOWN** , **RIGHT** , and **LEFT** .
- The class provides private helper methods **move(int direction)** and **pull(int direction)** to handle the movement of the player.

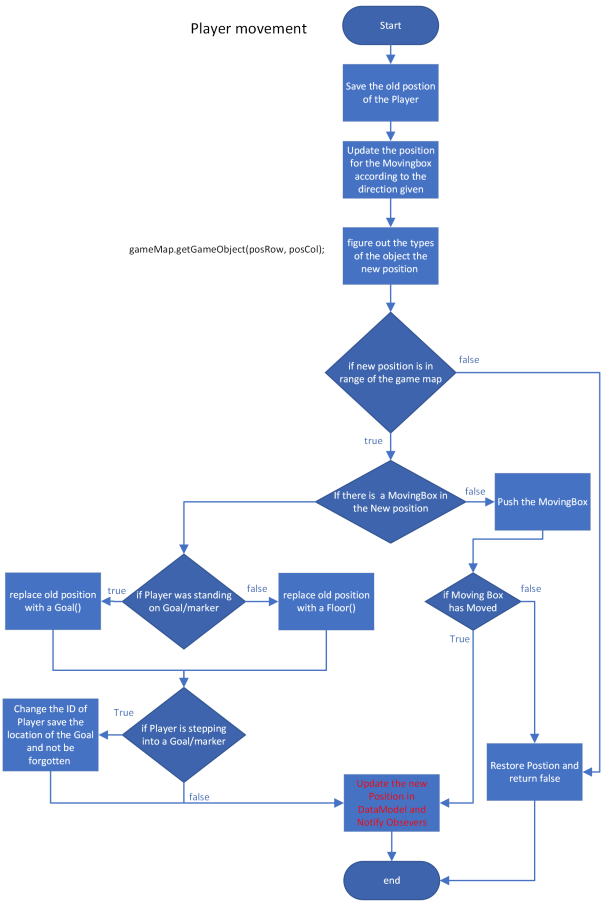

Figure 3: A Flowchart showing the Player Logic

- The **move(int direction)** method is responsible for moving the player in the chosen direction. It checks the new position for validity and collision with walls and crates.
- If the new position is outside the game map's range or collides with a wall, the player remains in its current position.
- If the new position contains a crate, the **move(int direction)** method calls the corresponding movement method (**moveUp()** , **moveDown()** , **moveLeft()** , **moveRight()**) of the crate object to attempt to move it.
- If the crate is successfully moved, the player moves to the new position. If the old position was on a goal, it is updated accordingly.
- If the new position is a goal tile, the player's id is set to 'x' to indicate that it is standing on a goal.
- The class provides public movement methods **moveUp()** , **moveDown()** , **moveLeft()** , and **moveRight()** to allow the player to move in the corresponding directions. These methods call the move(int direction) method internally.
- The class also provides additional methods **moveLeftAndPull()** , **moveRightAndPull()** , moveUpAndPull() , and **moveDownAndPull()** that combine player movement with pulling a crate in the specified direction.

Overall, the **Player** class encapsulates the behavior and attributes of the player character in the Sokobon game. It provides methods to move the player, handle collision with walls and crates, and update the player's position and appearance.

2.2.11 SoundsEffects

The code provided defines the **SoundsEffects** class in the **sokobon.views** package. Here's an overview of the class:

- The **SoundsEffects** class is used to play sound effects in the Sokobon game. It contains static methods for playing different sound effects.
- The class implements the **ChangeListener** interface, which allows it to listen for changes in the associated **DataModel** object.
- The class has static constants **SOUND2** , **SOUND3** , **SOUND4** , and **SOUND5** , which represent the file paths of different sound effects.
- The constructor of the **SoundsEffects** class takes a **DataModel** object as a parameter and attaches itself as a listener to the **DataModel** using the **attach()** method.
- The **playSound()** method is a private helper method that plays a sound effect given the name of the sound file.
- The method uses the Swing library to play the sound effect in a separate thread to avoid blocking the main thread.
- The class provides static methods **playSound2()** , **playSound3()** , **playSound4()** , and **playSound5()** that can be called to play specific sound effects.
- The **main()** method provides an example usage of the sound effects by playing each sound effect with a delay in between.
- The **delay()** method is a private helper method that introduces a delay in milliseconds.
- The **stateChanged()** method is called when the **DataModel** changes and determines which sound effect to play based on the **soundsToPlay** value in the **DataModel** .
- Depending on the value of **soundsToPlay** , the corresponding sound effect is played by calling the respective static method.

Overall, the **SoundsEffects** class provides methods to play sound effects in the Sokobon game. It uses the Swing library to play the sound effects in separate threads and listens for changes in the DataModel to play specific sound effects based on the game state.

2.3 Application Requirements

For Grade 3:
The Model-View-Controller (MVC) or Subject-Observer patterns are emphasized in the design specifications for the Sokobon game to ensure a flexible and modular architecture. Two independent views should be available in the game: a Swing-based graphical output and a separate console-based or game capture view for debugging and analysis. It is recommended that input devices, such keyboards, mice, and mobile phone apps, be repluggable using techniques that make switching between control systems simple. The Sokobon game design achieves a dynamic and adaptable framework, enabling different viewpoints and a variety of input possibilities, by satisfying these characteristics.
For Grade 4:
The code included a system for sound notifications using the observer pattern, A facility to save and load the current game state, and a JUnit4 test to test the functions.
For Grade 5:
The programmer must implement the suggested methods of music and animation to create a fully functional logical puzzle application. The application code must contain all the logic and details of the game, such as which pictures or animations the framework should render, what should happen when a button is pressed, and other details. Although the framework creates the painting, the application programmer must instruct the framework in the application code on which picture to paint at which coordinates. The audio is no different. The application programmer must instruct the framework on which song to play. In the application code, all the data structures that store things like object coordinates and object values are built and initialized.

2.4 Design of the Application

The MVC pattern makes the program, which, despite its simplicity, can be challenging to apply. An MVC pattern was used in the application's design to boost code efficiency, but this is only true for more complex applications like games. Three classes comprise the MVC pattern's Model, View, and Controller portion.

3 Testing section

This section discusses how the code and game were tested and debugged to ensure they worked perfectly.

Level.java tested with a main function.

```
Level 1:
# # # # # # # # # #
#                 #
#                 #
#   p o g g       #
#                 #
#                 #
o                 #
# # # # # # # # # #
```

Listing 1: The output of testing Level.java

3.1 Code debugging

Various tests were run on the code. Data was sent around without a GUI to test the MVC pattern. The console, over time, provided evidence of this. OutOfException, wrong implementation, and common issues were checked and fixed using Eclipse's debugging tool and the console. Taking things one step at a time and moving cautiously is key in this situation. The code below might represent a typical debugging scenario during the project, as it showed in Listing[**??**]. If This method did not solve the problem, using Eclipse debugging solved the problem, as shown in Figure[4].

As a result, this debugging method was implemented across the entire Code to prevent issues that are difficult to find or identify. It is shown in Figure [2]. Using non-static methods and appropriate code structure makes the code simple to test using JUnit4, as it is shown in figure[5].

```
1
2     public static void main(string args[]){
3         System.out.prinln(gMap);
4         //gMap.addPlayer(2,2);
5         gMap.player.moveDown();
6        // gMap.player.moveUP();
7        // gMap.player.moveLeft();
8        // gMap.player.moveRight();
9     }
```

Listing 2: Print out the result

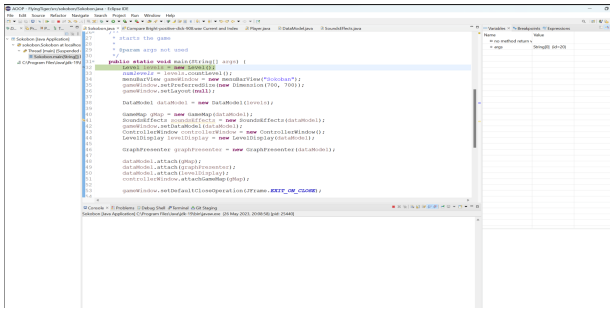

Figure 4: Step into the code

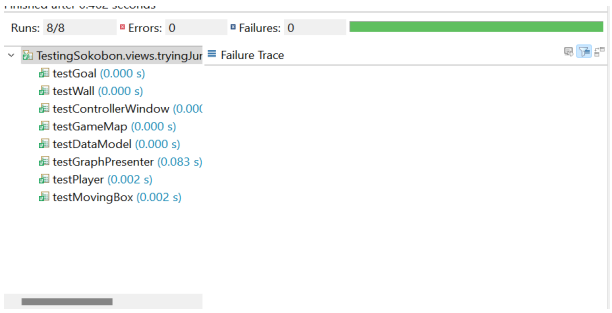

Figure 5: JUnit4 test

3.2 Game debugging

To secure the game's operation and a positive user experience, several difficulties and issues were faced throughout the development of the Sokobon game.

Controlling player mobility posed a serious difficulty. When the player object passed through boxes or walls, it caused issues that confused players and interfered with gameplay. The leading causes of these problems were discovered using console debugging. Discrepancies and mismatches might be found and corrected by comparing the wall and player positions, resulting in legitimate player movements.

Placement of the crates and the justification for designating spots presented another difficulty. When placed properly, crates weren't always recognized as labeled. It will print off your victory to address this, and the containers will be dark when it arrives at the proper location. Monitoring the behavior of crate items and the effects of these techniques during gameplay sessions was made possible via real-time debugging. Any anomalies in crate placement and marking logic were found and fixed by careful study of console outputs and logic tracing as it is shown in figure[6].

Adding audio and visual components together presented further difficulties. The smooth loading, rendering, and positioning of pictures within the game environment needed debugging. Sound effects were included to improve the player experience, and debugging was required to ensure appropriate playback and triggering. These visual and aural components were effectively incorporated into the Sokobon game through iterative testing and debugging, boosting its appeal and immersion.

Overall, the process of debugging games was crucial in locating and fixing a variety of problems that cropped up during the creation phase. Problems with player movement, crate placement logic, and visual/aural integration were successfully fixed through console debugging, step-by-step analysis, and real-time monitoring, leading to a playable and entertaining Sokobon game.

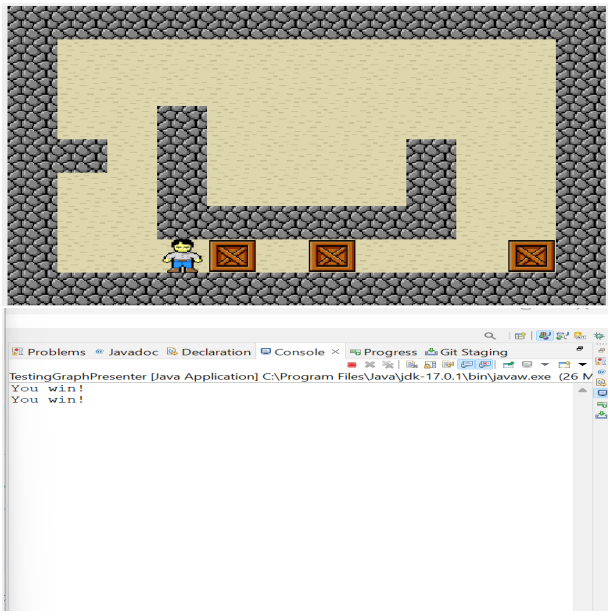

Figure 6: Checking Win

4 Interesting parts

Creating levels and maps was an interesting part, and trying to add some sound if the player won or another sound if the player lost is a funny sound. Add sound when the player is moving or trying to pull or push something to show that the player is exhausted, as it is in Listing[3].

```
    private static void playSound(final String soundName) {
        // play sound
        new Thread(new Runnable() {
            public void run() {
                try {
                    AudioInputStream audioInputStream = AudioSystem
                            .getAudioInputStream(new File(soundName).
    getAbsoluteFile());
                    Clip clip = AudioSystem.getClip();
                    clip.open(audioInputStream);
                    clip.start();
                } catch (Exception e) {
                    System.out.println("Error playing sound");
                    e.printStackTrace();
                }
            }
        }).start();
    }
```

Listing 3: Sound Code

5 Results and discussions

The project's outcomes met the original goals. To enhance the entire experience, several things may be done differently. One proposal for enhancement is to replace manually building many maps and levels with a text file-based approach. For upcoming additions, this would offer greater customization options and flexibility.

On the other hand, the separately developed Grid-Layout was developed using a design thinking technique that was successful and produced adequate graphical results. The game's overall experience was improved by the successful implementation of reading audio and graphics data. The addition of music, animation, and all the intended features proved that the project was successfully finished.

Despite the fact that there is always space for improvement. The project results were considered

satisfactory in light of the team of students' combined talents and knowledge. A more polished and refined game would result from implementing the suggested enhancements described above.

5.1 Requirements met

We were aiming on Grade 5, but I think we met Grade 4.

- A mixture of MVC and Observer pattern: Done! See section2.2.1
- more than two distinct "views": ViewPresenter + GameMap + Level
- Notifing observers, check for win, and keep track of level and save game state: in DataModel
- repluggable methods: the methods are in Player, implemented by windowcontrollerclass
- system for sound notifications using the observer pattern: See SoundEffects.java
- A facility to save and load the current game state: GameState.java
- for javaDoc; see folder doc1
- framework: almost there!

6 version control / GIT

- What's version control?

  Game version control, also known as version control or revision control, refers to managing and tracking changes made to a game's source code, assets, and other related files during and after development. It is a system that allows game developers and teams to manage multiple versions or iterations of their game and track changes made to each version.

  Version control is important in game development as it enables team members to collaborate on different aspects of the game. It helps developers track and manage changes, merge changes made by multiple team members, and roll back to previous versions if necessary. In addition, it provides a reliable backup mechanism, making it easier to identify and resolve any issues or errors encountered during development.

  Game version control systems typically use a repository to store all the game's files and assets. Developers can check out, modify, and check in files, and the system keeps track of changes made by each team member. This allows for effective collaboration while keeping your game's source code and assets organized and accessible.

  Common version control systems used in game development are Git, Mercurial, and Perforce. These systems provide features such as branching (creating separate lines of development), merging (combining changes from different branches), tagging (marking specific versions), and history tracking, making them efficient for game development projects. It provides good version control.

- what are some popular tools?

  Git: Git is a distributed version control system widely used in game development. It allows developers to track changes, create branches, merge code, and collaborate effectively. GitHub and GitLab are popular hosting platforms for Git repositories.

  Perforce: Perforce is a centralized version control system widely used in the games industry. It offers features such as file locking, asset management, and collaboration tools designed specifically for game development workflows.

  Subversion (SVN): SVN is a centralized version control system that offers a simpler and more familiar workflow compared to Git. Although its popularity has waned in recent years, it is still used by some game development studios.

- Are they paid?

  Git: Git is a free and open source version control system widely used in game development. It's free and can be used with hosting platforms such as GitHub, GitLab, and Bitbucket that offer free repositories for public projects.

Perforce: Perforce, a centralized version control system commonly used in the games industry, typically requires a paid license. Cost may vary based on factors such as number of users, features, support options, etc.

Subversion (SVN): SVN is a free, open source, centralized version control system. No license fee.

- what are the difficulties with starting to use VC?

  Learning curve: Adopting a new version control system often requires learning new concepts, commands, and workflows. For developers new to VC, it can be difficult at first to understand how to use the system effectively. However, with practice and guidance, these difficulties can be overcome.

  Setup and Configuration: Setting up a version control system and configuring it to work with a particular project or development environment can be complicated, especially for beginners. This includes installing software, configuring repositories, setting user permissions, etc. Incorrect configuration can cause problems and hinder collaboration.

  Merge Conflicts: Conflicts can occur when multiple developers are working on the same file at the same time and trying to merge their changes. Merge conflicts occur when the system cannot automatically decide which changes to keep. Resolving conflicts may require manual intervention and careful consideration to ensure that all changes are applied correctly.

  Collaboration and Communication: Effective collaboration and communication are essential when using version control. Developers need to understand how to communicate changes, review code, and manage conflicts and disagreements with other team members. Establishing clear procedures and rules can help mitigate these challenges.

  Project Size and Performance: As your project grows, your version control system's performance can degrade. Large repositories with many files, frequent commits, and huge histories can slow down operations like clone, branch, and merge. Efficient organization and optimization strategies are required to mitigate these performance issues.

  Maintenance and Management: Maintaining and managing a version control system requires constant attention. This includes managing user access, backups, repository maintenance, and ensuring systems are up to date. Effective management tasks require adequate knowledge and resources.

- How easy is it to get used to the workflow? was it helpful in our case? give a picture.
- what is ssh?

  SSH stands for Secure Shell. It is a network protocol that allows secure remote access and communication between two computers over an insecure network. SSH provides a secure channel for executing commands, transferring files, and managing network services.

7 JavaDoc

During the course of the project, we have commented the majority of the code. This means methods, variables, classes and usage. We did this through the javaDoc tool, where we can generate complete documentation for the project as shown in the figure[8].

8 Discussion

9 Refrences

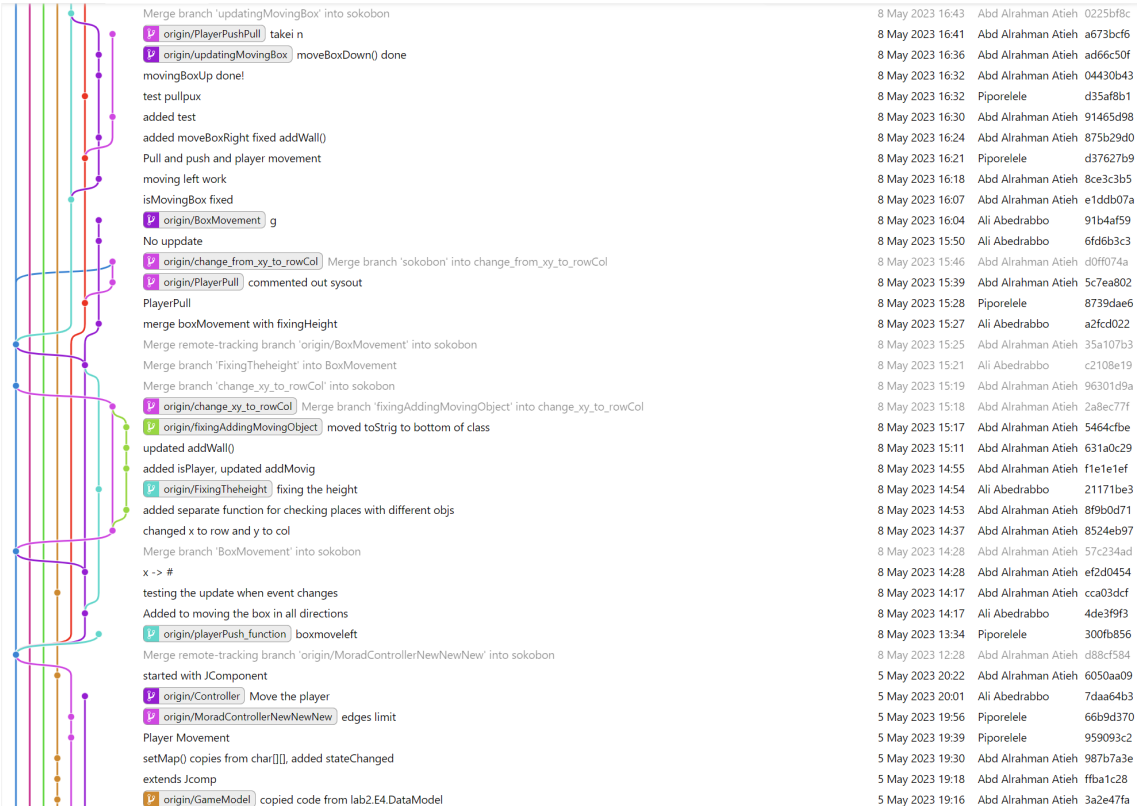

Figure 7: A snippet from git history

Figure 8: Screenshots from javadoc