Advanced Object-Oriented Programming

Sokoban Project

Khaled Mhjazi

Hind Dakkeh

Group 22

Contents

1. Introduction

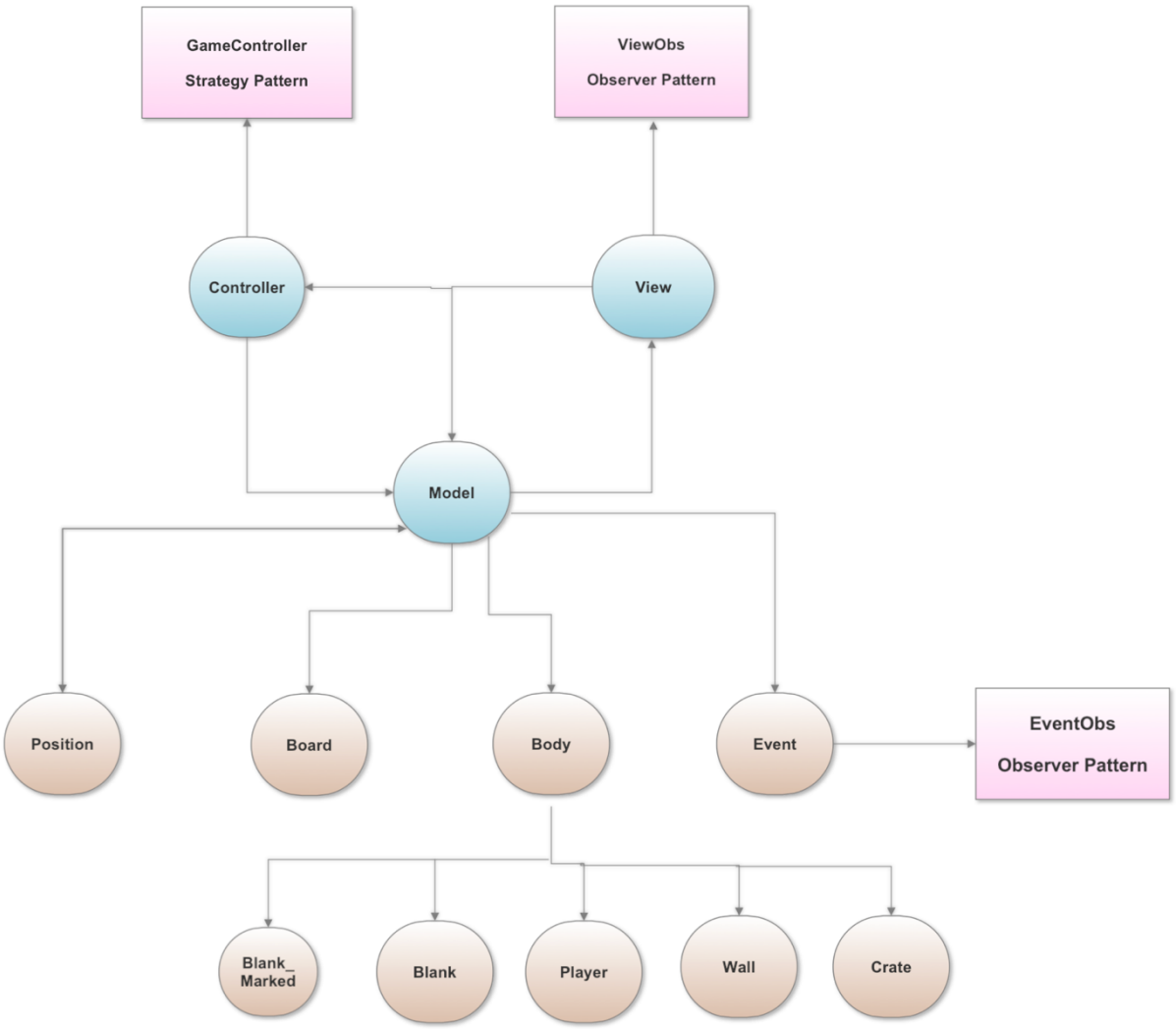
Java is a programming language that belongs to the high-level category. It follows a class-based and object-oriented approach to programming. One of the core design principles of Java is to minimize the number of implementation dependencies as much as possible.
In this project, our aim was to develop a reusable 2D framework in Java for implementing Sokoban, a popular Japanese puzzle game. The requirements following grade 3 were imposed where the main design was chosen around the MVC pattern.
Sokoban is known for its challenging gameplay mechanics and requires players to strategically move objects within a confined space. To ensure a well-structured and reusable codebase, we adopted the Model-View-Controller (MVC) architectural pattern as the main design approach.

This report provides an overview of our implementation of the Sokoban game. This includes an explanation of the process and application of the framework, as well as delve into the game's logic. By following the MVC pattern, we aimed to achieve a modular.

2. Design

2.1. Diagram

By following the Diagram observe that MVC pattern is very helpful to structured and organized the game. The Model component separates the game's logic and data from the user interface, ensuring modularity and ease of modification. The View component handles the visual representation of the game. The Controller acts as the intermediary, facilitating communication between the user input, the Model, and the View.



2.2. Framework

A framework can be likened to a toolbox containing a collection of libraries and pre-existing code designed to facilitate application development. It serves as a valuable resource that allows developers to streamline their work and avoid the repetition of common tasks. In the world of application and program design, frameworks play a crucial role as they significantly enhance productivity and save programmer's valuable time.

Within the context of our project, we have implemented an efficient and reusable framework specifically tailored for building 2D puzzle games. The framework we have created serves as a solid foundation for developing similar games with ease. In our case, we utilized this framework to develop the Sokoban game.

2.3. Model

The Model section of the project encompasses the fundamental components that form the core of the Sokoban game, essentially acting as its rulebook. Within the Model, key functionalities are implemented, including reading the game map, placing the player character on the map, establishing connections between various game elements, and adding markers to indicate the positions of objects. It is responsible for setting the current level of the game and calculating the next level when appropriate. Additionally, the Model handles player directions and events, ensuring they are appropriately transmitted to observers. It serves as the central hub, facilitating the flow of information and computations between different components. The Model maintains a reference to the View component and listens for controller input from the Controller class [3].

2.4. View

The View component takes charge of the visual presentation of the Sokoban game, managing all aspects related to output. It encompasses the View, Controller, Observer, and Model, working together to update the game board and interact with the graphical user interface (GUI). The View utilizes the ViewObs interface to communicate with the GUI (GameGraphical) and employs the observer pattern to access the Control (ViewControl) functionality.

2.5. Controller

The Controller section of the project is responsible for managing and facilitating user input and controls. It serves as the framework for incorporating additional control functionalities into the game and acts as a foundation for other classes such as Keyboard and GameController. One of the primary controllers utilized in this implementation is the Keyboard controller. It enables the player to control the game using specific keyboard inputs.

The controls that have been implemented are as follows:

- Arrow keys: These keys allow the player to control the movement of their character within the game.

- R key: Pressing the R key triggers a game reset, allowing the player to start the game from the beginning.

2.6. Patterns

Patterns play a crucial role in software development, providing proven solutions to common design problems. In our project, we have utilized two important patterns: the Observer pattern and the Strategy pattern.

2.6.1. Observer pattern

The Observer pattern is employed to establish a communication mechanism between different components of the project [2]. This pattern ensures loose coupling between objects, allowing for flexibility and extensibility. In our implementation, the Model acts as the observable, while the View and other relevant components act as observers. By using the Observer pattern, updates and events occurring within the Model can be efficiently propagated to the interested observers, such as the View, ensuring that all necessary information is appropriately communicated. Examples of this in code are ViewControl and GameGraphical being observers of the subject of it which is ViewObs.

```
package view;

import model.Board;

public interface ViewObs {

    /**
     * Update board for the observers
     * */

        void updateBoard(Board board);
```

2.6.2. Strategy pattern

The Strategy pattern implementation in our project involves the use of interfaces to define a common set of methods [2]. One such interface is GameController, which is implemented in the Keyboard class. By implementing this interface, the Keyboard class gains access to essential methods for controlling the game.

The interfaces act as references to the Controller class, which contains important methods responsible for game control. This implementation approach using the Strategy pattern and interfaces provides flexibility and extensibility for integrating different input devices and control mechanisms. It allows seamless integration of peripherals like keyboards and ensures effective communication between user input and the game's internal processes. This pattern enhances interactivity and responsiveness within the game, this approach facilitates code reusability, as different validation algorithms can be easily plugged into the system without affecting the overall structure of the game.

```java
package controller;

public interface GameController {

    /**
     * Adds a reference to the Controller class
     * @param controller the controller object
     */
    void addController(Controller controller);

}

```

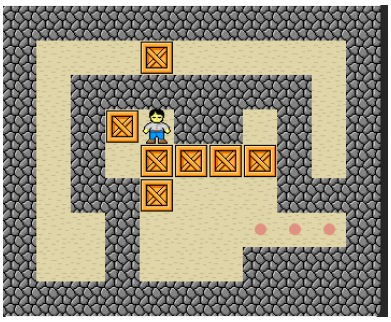
3. Testing

3.1. Move two objects:

In Sokoban, the player typically pushes boxes to specific targets. Moving two boxes down: it's usually not possible to move two boxes down consecutively in a single step.

Moving two walls right: walls are typically immovable objects that block the player's path the player character cannot push or move walls.

Moving a box and a wall up or left: the player character cannot move a box and a wall together in the upward or leftward direction.



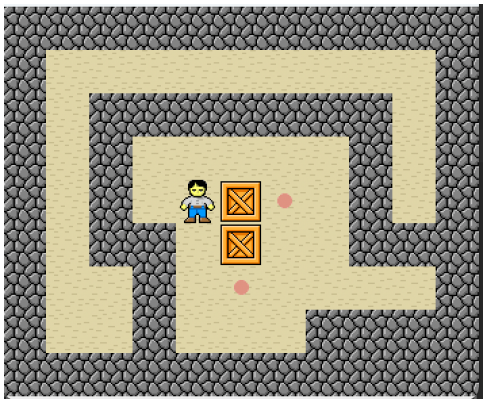
After the sequence of movements, an assertion (assertTrue()) is made to check if the player's position is a specific row and column on the game board. If the condition evaluates to true, it means the player cannot moved and stayed in row 3 and column4.

```
12
13•void moveTwo()  {
14
15      Model model = new Model("./Sokoban/src/TestPlan/t0");
16      model.movePlayer(Controller.Direction.UP);
17      model.movePlayer(Controller.Direction.DOWN);
18      model.movePlayer(Controller.Direction.LEFT);
19      model.movePlayer(Controller.Direction.RIGHT);
20
21      assertTrue(model.getPlayer().getP().getRow() == 3 && model.getPlayer().getP().getCol() == 4);
22  }
23
```

Sokoban has specific rules and limitations for moving objects, including boxes and walls. The player can only push one box at a time, and walls are immovable obstacles. Understanding and adhering to these rules is essential for solving Sokoban puzzles effectively.

3.2. Win the game

This test case focuses on a simple scenario where the player moves right and then down, making sure that the game detects the winning condition when all blank Marked positions are filled with boxes.



After the movements, the code performs an assertion using (assertTrue()) to check if the game board is in a winning state. It does this by calling the (win()) method on the (model.getBoard()) object. The (win()) method evaluates the current state of the game board and returns `true` if the win condition is met, indicating that all boxes are correctly placed on target positions.

```
26
27•void WinGame() {
28
29      Model model = new Model("./Sokoban/src/TestPlan/t1");
30
31        model.movePlayer(Controller.Direction.RIGHT);
32         model.movePlayer(Controller.Direction.DOWN);
33
34         assertTrue(model.getBoard().win());
35
36 }
37
```

4. **Interesting parts**

This method serves to propagate the game board's current state to all registered observers. It facilitates the communication between the model and the view components, allowing the views to update their display based on the new board information.

```
 * Method to show the board to all the view observers
 * */
public void display() {

    for(ViewObs obs: observers)
        obs.updateBoard(model.getBoard());
}
```

Method handles key presses and performs different actions based on the specific key pressed. It updates the direction of the controller object for player movements and allows the player to restart the level by pressing the R key. It is very interesting that we know how the keyboard works in the Java language.

```
public void keyPressed(KeyEvent e) {

    switch (e.getKeyCode()) {

    //  move left
    case 37 :  con.setDirection(Controller.Direction.LEFT);
    break;

    // move up
    case 38 :  con.setDirection(Controller.Direction.UP);
    break;

     // move Right
    case 39 :  con.setDirection(Controller.Direction.RIGHT);
    break;

      // move down
    case 40 :  con.setDirection(Controller.Direction.DOWN);
    break;

    // By pressing R to reset the game

    case 82 : con.restartLevel();
```

The code calculates the difference between the row and column numbers of newP and pos. It then doubles these differences by multiplying them by 2. The purpose of this calculation is to determine the target position where the box should be moved. The target position is based on the player's intended movement direction, as indicated by newP. This code is used to calculate the target position for the box when the player attempts to move it. It ensures that the box moves two steps in the desired direction from its current position.

```
// moving two steps in the column and row direction.
int row1 = 2*(newP.getRow()- p.getRow());
int col1 = 2*(newP.getCol()- p.getCol());

// check if there is the wall in the new position
switch(board.getPlan(newP.getRow(),newP.getCol()).getBlock()) {
    case WALL: //Wall
        break;

        // check if there are two crates or crate and wall (behind each other).

    case CRATE_MARKED ,CRATE:

        switch (board.getPlan(p.getRow() + row1, p.getCol() +col1).getBlock()){

            case WALL,CRATE_MARKED,CRATE:

                break;
```

5. Results

The successful implementation of all functionalities in a Sokoban game depends on how closely the requirements were followed during development. Suppose all the required features, such as player movement, box pushing, wall collision detection, and win condition checks, have been implemented and are working correctly. In that case, it can be considered a successful implementation.

To determine if the game works as intended, one must test various scenarios, including moving the player, pushing boxes, handling collisions with walls, and checking win conditions. The game behaves correctly in these scenarios and meets the desired requirements, which can work as intended. While following the requirements closely is essential, it's also important to have some flexibility during implementation. Sometimes, there may be multiple valid approaches to solving a problem or implementing a feature. As long as the core functionality is present and the game behaves as expected, minor deviations from the requirements might not be critical.

The ease of modification will depend on the code's design and structure to extend the Sokoban game with new functionality. Good software design principles, such as modularity, encapsulation, and separation of concerns, can make the program more flexible and easier to extend. The program is easy to modify in case a better implementation of a part is possible because the code has been well-organized and follows good design practices. The successful implementation of a Sokoban game depends on meeting the requirements, thorough testing, and adhering to good software design principles.

6. version control / GIT

Version control systems, such as Git, offer a structured and efficient approach to managing and tracking changes in software development projects. They provide numerous benefits, including enhanced collaboration, improved code quality, and increased productivity for teams. By using Git, developers can easily collaborate with team members, even if they are geographically dispersed. Version control through Eclipse and Github opened many possibilities and flexibility for this project. Git's version control capabilities provide a detailed history of changes, including information about who made each change and when, which helps us to work with the project more intelligently. It was an exciting experience that taught us how to work on the same project from different places and save every edit.

References

[1] Wojciech Mostowski. Project requirements
https://bb.hh.se/ultra/courses/_15161_1/outline/edit/document/_503751_1?courseId=_15161_1&view=content
Last accessed 2023-05-16

[2] Wojciech Mostowski. Lecture 5
https://bb.hh.se/ultra/courses/_15161_1/outline/edit/document/_502760_1?courseId=_15161_1&view=content Last accessed 2023-05-16

[3] Wojciech Mostowski. Lecture 7
https://bb.hh.se/ultra/courses/_15161_1/outline/edit/document/_504710_1?courseId=_15161_1&view=content Last accessed 2023-05-16

[4] Wikipedia. Sokoban https://en.wikipedia.org/wiki/Sokoban Last accessed 2023-05-16

[5] Wikipedia. MVC https://en.wikipedia.org/wiki/Model–view–controller Last accessed 2023-05-16