# Implementing a tile based game using advanced object oriented design patterns

Jahnstedt, Per
perjah20@student.hh.se

Eklund, Rasmus
rasekl19@student.hh.se

## Contents

# 1 Introduction

We were tasked to implement the game Sokoban, see https://en.wikipedia.org/wiki/Sokoban for detailed description of the game. We received a project description that specified the outline of the project and how the implementation of Sokoban had to be done. We interpreted the requirements as follows:

1. The game had to be represented by a model.
2. The game had to have a graphical representation of the game model.
3. We had to have debug window.
4. The game had support various method of inputs.
5. The game had to be implemented using various design patterns such as:
   - Model-View-Controller
   - Observer
   - Strategy

Besides the minimum requirements stated above, we added the following functionalities to our list of requirements:

1. Generating sound for different type events occuring in the game.
2. The ability to save and load the game using java serialization.
3. Creating a framework to build any tile based game upon.

The end product was supposed to be a playable Sokoban game, which supported playing with the help of the arrow keys on the keyboard or pressing the buttons in the graphical user interface for the game. The majority of our decisions on how to implement the project were impacted by the course format and what we learned during the course.

To test the application, we used JUnit to develop some automated testing methods that focused on game model functionality like movement, collision, and buttons. We also did some manual testing in which we ran the application and purposefully attempted to do things that shouldn't be possible in the game.

We believe there are some key distinction between our game and other game implementations in general like the concurrency of executable code. Our program does not handle the execution and update of observers at the same time. According to us, this is a drawback in our design because it forces things to happen in a specific order. The game model is changed first, and then each observer is notified and updated one by one. In a large program with numerous observers, each of whom must execute complex algorithms, this would result in a significant performance reduction, potentially ruining the gaming experience for the players.

# 2 Design

We began by analyzing the characteristics of tile-based games. We concluded that all tile-based games are a $Y * X$ grid, where $Y$ and $X$ are positive integers ($X, Y \in Z^+$). Based on this we started to create a base class for creating a 2D-array consisting of integers. In retrospect we should have made this base class generic, allowing the 2D-array to include any datatype. This base class is called TileGameModel, and contains all functions for accessing and manipulating the game grid, see fig. 1 for analysis of 2D tile based game.

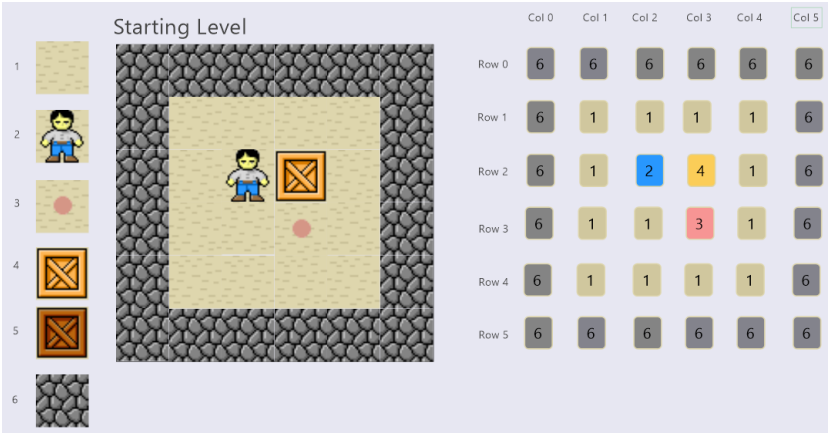


Figure 1: Tile Game Analysis

We proceeded with designing a base graphical user interface (GUI) that could act as a base for representing a game model visually, see fig.2. This base class is called TileGameGUI and is an abstract class with abstract methods that the programmer has to implement.

The last base class is the controller and is used to manipulate the game model, it only contains one method and the algorithm of this method can be changed at run time to allow different kinds of manipulation of the game model using the strategy pattern, this class is called TileGameController and the strategy interface ButtonStrategy, see sample code below.

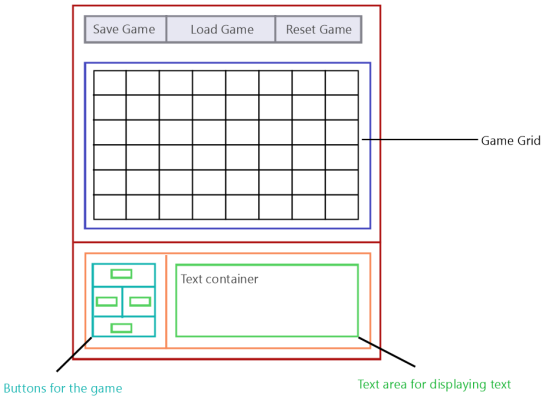


Figure 2: Tile Game GUI concept

```
public interface ButtonStrategy<T extends TileGameModel> {
    void executeMethod(T gameModel);
}
public abstract class TileGameController<T extends TileGameModel> {
    public void addGameModel(T aGameModel) {
        gameModel = aGameModel;
    }
    public void handleButtonPress(ButtonStrategy<T> buttonStrategy){
        buttonStrategy.executeMethod(gameModel);
    }
    private T gameModel;
}
public class MoveButton implements ButtonStrategy<SokobanGameModel> {
    private final Directions direction;
    public MoveButton(Directions aDirection) {
        this.direction = aDirection;
    }
    @Override
    public void executeMethod(SokobanGameModel gameModel) {
        gameModel.moveCharacter(direction);
    }
```

Combining these three base classes together we get a TileGame class which implements the MVC pattern. At this milestone we were at a crossroad to decide whether to continue with our requirements list of implementing observer pattern or implement the mediator pattern. The mediator pattern would allow us to further decouple our classes but in the end we decides to continue with the observer pattern since it is important to not deviate to far from the project plan, this resulted in the interface GameObserver. Combining all of the classes into a main class called TileGame resulted in the finished framework for creating tile based game, see fig.3. By separating modules of the program into classes and making sure that each class is responsible for a specific task in the program enables modification of particular modules easy. The main concept of the framework is that the person should be able to focus on the model and game logic instead of having to think about how they want to structure the graphic layout. However this comes at the expense of the developer not being able to change the layout. This framework can be thought of as a quick way to test game concepts for rapid development. An example diagram of how we used the framework can be seen in fig.4 and fig.5.

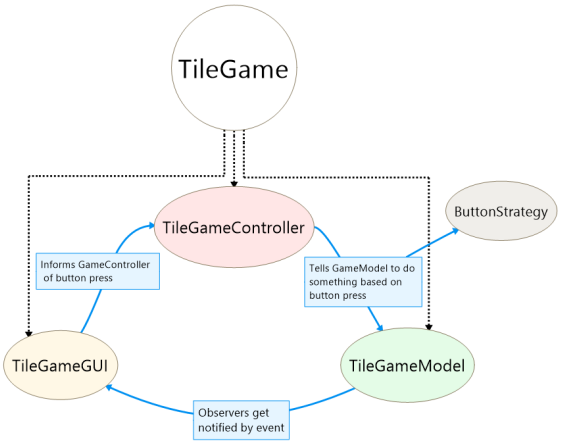


Figure 3: Tile Game Framework

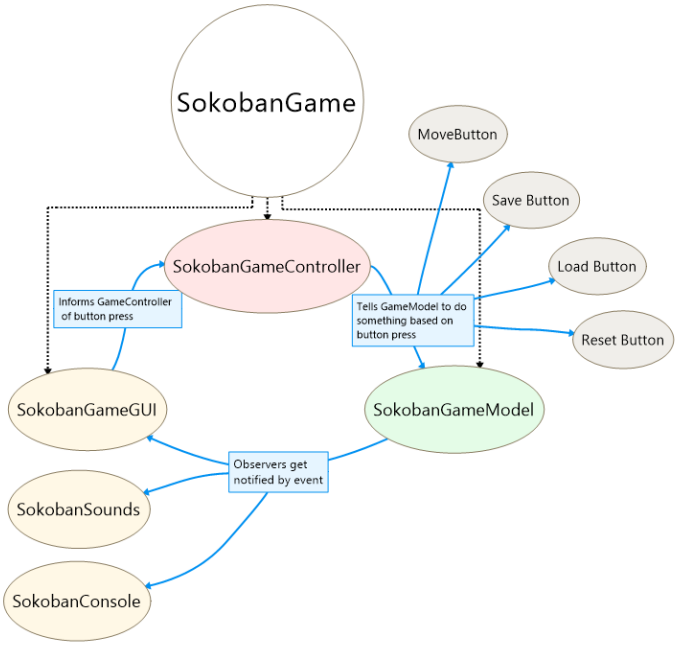


Figure 4: Sokoban extending Game Framework

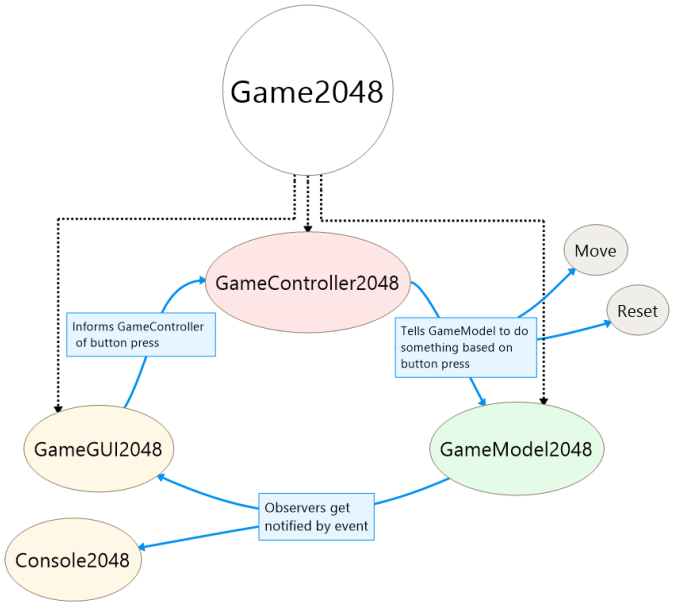


Figure 5: 2048 extending Game Framework

# 3 Testing

As a part of the project we where tasked to test certain aspects of the code.
The parts that we tested where:

- Player movement
- Crate movement
- Collision
- Reset button

We started off by creating 3 test maps with different structures. One map is created to test different possible collisions, one map is made to test the pushing mechanics and in the last map we test the movement without collision or pushing. To test the collision in the game we created a test method where we retrieve the test map for collision and then use our move methods in every direction, each time checking if the current map is the same as the original test map and if the event that should occur when you can't move is activated. Shown below is an example of the testing we did with the collision.
When testing the pushing and moving functions we created methods for moving in every direction and set the current map to be either the pushing or moving test map accordingly. When one of these test methods are being run we check if the tiles in that direction is updated as they should and also check if the right event occurs.

```
@Test
void playerMovingIntoCollision() {
    // Testing Collision
    sokobanGameModel = new SokobanGameModel(new int[][][]{testLevelCollision});
    sokobanGameModel.startGame();
    sokobanGameModel.moveCharacter(NORTH);
    compareGrids(testLevelCollision);
    assertEquals(TRIED_TO_MOVE, sokobanGameModel.getLastEvent());
    sokobanGameModel.moveCharacter(EAST);
    compareGrids(testLevelCollision);
    assertEquals(TRIED_TO_MOVE, sokobanGameModel.getLastEvent());
    sokobanGameModel.moveCharacter(WEST);
    compareGrids(testLevelCollision);
```

```
    assertEquals(TRIED_TO_MOVE, sokobanGameModel.getLastEvent());
    sokobanGameModel.moveCharacter(SOUTH);
    compareGrids(testLevelCollision);
    assertEquals(TRIED_TO_MOVE, sokobanGameModel.getLastEvent());
}
```

# 4 Interesting parts

In our opinion one of the interesting parts is the usage of strategy pattern. To be able to change a methods algorithm at runtime to perform different but related tasks on the game model is impressive. This essentially removed the dependency for switch statements in the controller.

While implementing the functions for moving/updating tiles in the game grid, we ran into an interesting problem: how were we supposed to keep track of the tiles underneath the player when he moved to a new tile, and also what tile we should replace the player with once he was moved? Our solution was to create a stack that we started by pushing a sand tile onto at the start of a new map, and when the player moves, the stack pops the tile to the previous location of the player and pushes the next tile onto the stack. For example if the player is standing on a sand tile and wants to move a filled crate, the stack pops the sand tile to its prior location and then pushes a dot tile to the stack to keep track of what's below the player as he moves. See sample code below.

```
private void checkNextTile(int nextLocation, Directions direction){
    if (nextLocation == CRATE) {
        tileStack.push(SAND);
        moveCrate(direction);
    } else if (nextLocation == FILLEDBOX) {
        tileStack.push(DOT);
        moveCrate(direction);
    } else tileStack.push(nextLocation);
}
```

Another issue worth mentioning was a difficulty with 2D array referencing while attempting to reset the game state. We discovered that when we were testing our reset button, we weren't getting the old 2D array, but rather an updated array with the same values as the current game state. The issue was that every time we moved in the game, we were also changing the initial 2D array. To be able to return to the old game state when the reset button was pressed, we decided that we needed to make copies of the 2D arrays each time we loaded in a new array. See sample code below.

```
public int[][] makeCopyOf2DArray(int [][] arrayToMakeACopyOf) {
    int[][] newArray = new int[arrayToMakeACopyOf.length][];
    for(int i = 0; i < arrayToMakeACopyOf.length; i++)
        newArray[i] = arrayToMakeACopyOf[i].clone();
    return newArray;
}

public int[][] getGameState() {
    return makeCopyOf2DArray(gameGrid);
}
```

# 5 Results

Looking back on the project and our implementation, we'd like to claim that we successfully implemented a working version of Sokoban as well as all of the requirements we set out to meet. The game features sounds that are generated based on what event has occured, a debugging terminal that prints out information about the game after each interaction with it, and a GUI that the user can use to play the game, all of which are updated using the observer pattern. It's easy to add extra functionality to the program thanks to the modular design, and it's simple to add new ways to control or manipulate the game thanks to the controller's ability to handle algorithms and button strategies.

When we started this project, the end goal was to implement the game Sokoban. But the end result of our project is a framework that, with the use of object oriented techniques, has reusable modules that can be used for building larger applications in the aspect of tile based games.


Figure 6: Finished game