Halmstad University

Advanced Object Oriented Programming

Java Environment

Final Project

x

Author:
Simon Thelin
Oskar Svensson

Supervisor:
Wojciech Mostowski
Mahsa Varshosaz

May 27, 2016

Abstract

The purpose of the project was to produce and develop a framework with an application connected to it. The students of this report implemented the game Sokoban and an easy version of TicTacToe. The framework was designed to be patternless and the application to the MVC(Model View Controller) pattern. The framework will adapt its own pattern to the applications given pattern. The framework handles image reading, animation reading and audio files. The framework can also handle keyboard events and painting objects to an JFrame. During the implementation there were several problems encountered such as how to handle the player movements. It was resolved with matrices containing data telling where the object can move and not move. Therefore there was a lot of debugging resolving this part the result came out to be solid. The framework together with the application works properly and it is simple to continue developing the framework.

Figure 1: Sokoban implementation with the framework

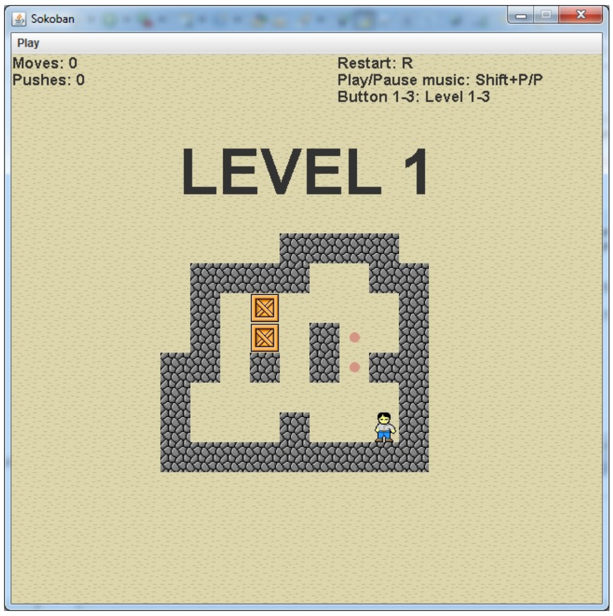
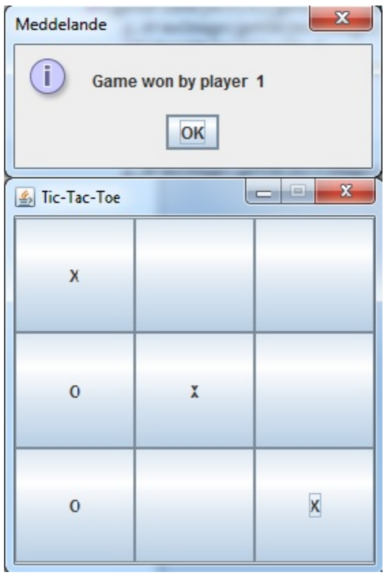


Figure 2: TicTacToe implementation with the framework



Contents

1 Introduction

The specification given to the students was to create a framework. This was solved and a solid solution came out from it. The framework was connected to an application and the result was an implementation of the 2D puzzle game of Sokoban on the given framework. The framework requirements were clear. The framework should help the programmer to to write an application to verify that operations such as reading images, audio files and such was done inside the framework and not in the application. The developers were given a great deal of freedom to solve these issues. The framework design was decided to be patternless for the best possible dynamic functionality. The application requirements were clear. The application should put the necessary data in the different data structures to verify that the framework had something to work with. The application design was set to a MVC pattern[9]. When the framework is implemented with the application, the framework will adapt to the given application pattern. The programmer does not need to use all the parts of the framework. The report has a section that discusses the testing and debugging that was conducted during the implementation of the framework and application with a deeper look into the debugging of the code and the game-play. There is also a section called interesting parts. This section is interesting due to the implementation of the framework and especially how the framework interacts with the application and the design result. This covers the good and bad ideas.

2 Framework

2.1 Framework Requirements

Unlike the application requirement all the non logical and not game specified parts of the code should be in the framework, e.g. the painting of objects, animations inside a frame and the playback of sounds. The framework should be made for the application programmer to start coding for a 2D logical puzzle game without any modifications of the framework.

2.2 Framework Design

The design of the framework is patternless. A framework is code ready for use usually packaged in a way that makes creating an application much easier[4]. When the framework is patternless it allows the application programmer to choose what pattern design to have on the application. This will then be heritaged to the framework so the framework can be a part of many design patterns for a given application. If the framework would have had a pattern the application would have been forced to follow the same pattern. Therefore the framework will now follow the same pattern as the application. The framework consists of three abstract classes, Engine, PaintMap and KeyActions.

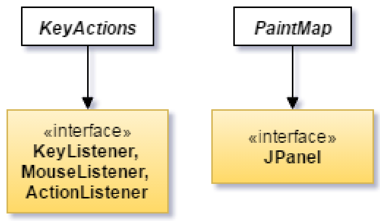
The abstract framework class Engine is the heart of the framework. The Engine class is supposed to help the application programmer to develop the core of the game. Engine provide several abstract methods that should be implemented in the application that is used within the framework. Such as reading images, animations and audio files. Therefore the code becomes more facilitating during implementation. This design confirm the efficiency when testing the code and testing because of the freedom of choice.

The abstract framework class PaintMap extends JPanel and enables the drawing and painting of images. The only code that the application programmer need to write is the coordinates for the object to be painted. The thought of this part of the framework is that the developer does not have to use this part but it is there and ready to use. Especially for a game like Sokoban or another game where an object moves around inside a JFrame and there should be a representation of this object as an image.

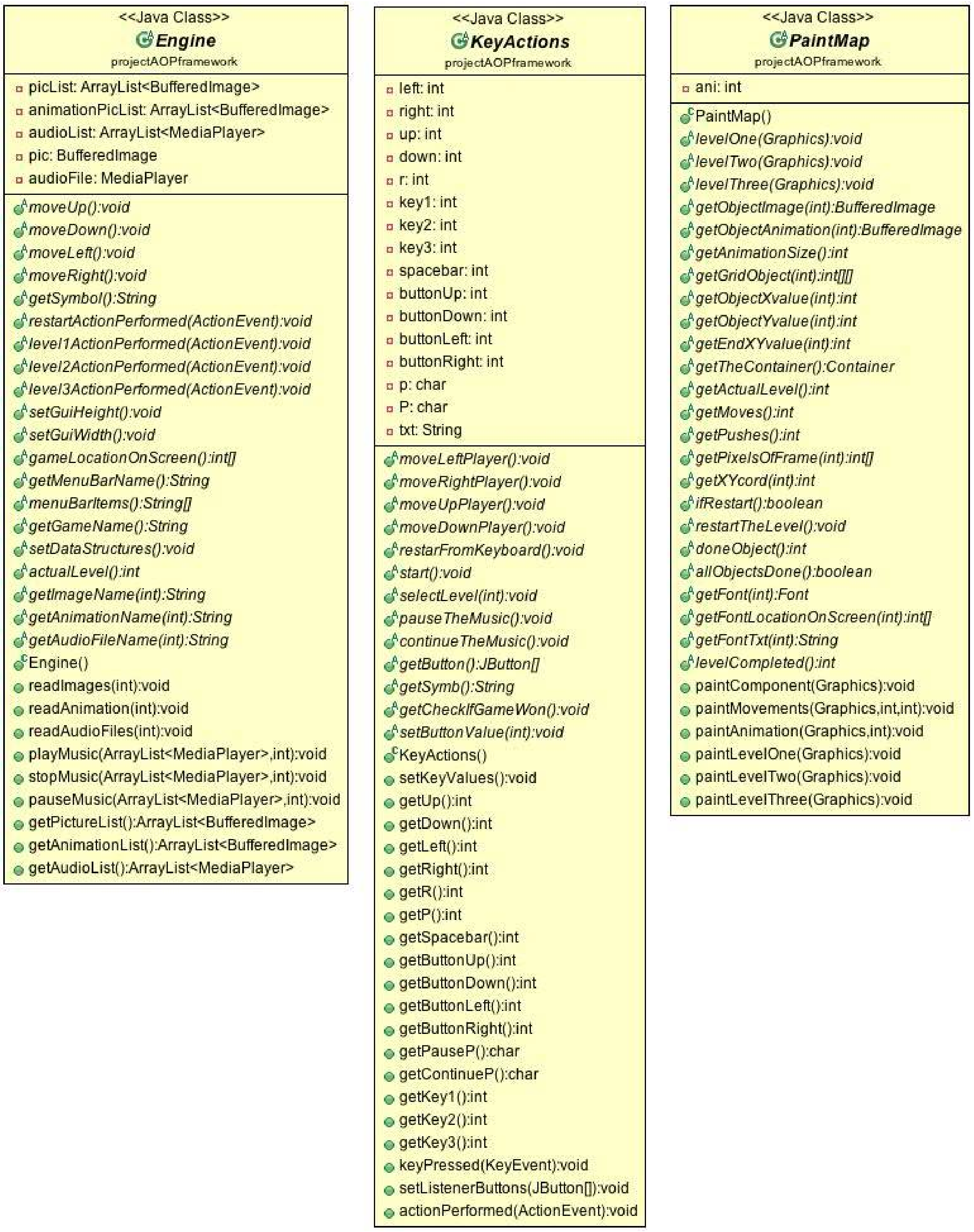
The abstract framework class KeyActions implements the interfaces KeyListener, MouseListener and ActionListener. This part of the framework tells the application programmer the end user actions without having to implement a listener for the events. It is up to the application programmer to choose which interface that suits the application best. This part of the framework helps the programmer not having to spend time on the actual reading from the keyboard input, it is easy to use and is an efficient part of the framework.

The framework is also designed not to hold any form of static methods. This is to confirm that the code is not hard wired to any particular method. It will also provide a better chance of testing and debugging the code[1].

Figure 3: Abstract Framework Classes Implementing Interfaces



2.2.1 UML Framework Design



```
<<Java Class>>
Engine
projectAOPframework

picList: ArrayList<BufferedImage>
animationPicList: ArrayList<BufferedImage>
audioList: ArrayList<MediaPlayer>
pic: BufferedImage
audioFile: MediaPlayer

moveUp():void
moveDown():void
moveLeft():void
moveRight():void
getSymbol():String
restartActionPerformed(ActionEvent):void
level1ActionPerformed(ActionEvent):void
level2ActionPerformed(ActionEvent):void
level3ActionPerformed(ActionEvent):void
setGuiHeight():void
setGuiWidth():void
gameLocationOnScreen():int[]
getMenuBarName():String
menuBarItems():String[]
getGameName():String
setDataStructures():void
actualLevel():int
getImageName(int):String
getAnimationName(int):String
getAudioFileName(int):String
Engine()
readImages(int):void
readAnimation(int):void
readAudioFiles(int):void
playMusic(ArrayList<MediaPlayer>,int):void
stopMusic(ArrayList<MediaPlayer>,int):void
pauseMusic(ArrayList<MediaPlayer>,int):void
getPictureList():ArrayList<BufferedImage>
getAnimationList():ArrayList<BufferedImage>
getAudioList():ArrayList<MediaPlayer>
```

```
<<Java Class>>
KeyActions
projectAOPframework

left: int
right: int
up: int
down: int
r: int
key1: int
key2: int
key3: int
spacebar: int
buttonUp: int
buttonDown: int
buttonLeft: int
buttonRight: int
p: char
P: char
txt: String

moveLeftPlayer():void
moveRightPlayer():void
moveUpPlayer():void
moveDownPlayer():void
restarFromKeyboard():void
start():void
selectLevel(int):void
pauseTheMusic():void
continueTheMusic():void
getButton():JButton[]
getSymb():String
getCheckIfGameWon():void
setButtonValue(int):void
KeyActions()
setKeyValues():void
getUp():int
getDown():int
getLeft():int
getRight():int
getR():int
getP():int
getSpacebar():int
getButtonUp():int
getButtonDown():int
getButtonLeft():int
getButtonRight():int
getPauseP():char
getContinueP():char
getKey1():int
getKey2():int
getKey3():int
keyPressed(KeyEvent):void
setListenerButtons(JButton[]):void
actionPerformed(ActionEvent):void
```

```
<<Java Class>>
PaintMap
projectAOPframework

ani: int

PaintMap()
levelOne(Graphics):void
levelTwo(Graphics):void
levelThree(Graphics):void
getObjectImage(int):BufferedImage
getObjectAnimation(int):BufferedImage
getAnimationSize():int
getGridObject(int):int[][]
getObjectXvalue(int):int
getObjectYvalue(int):int
getEndXYvalue(int):int
getTheContainer():Container
getActualLevel():int
getMoves():int
getPushes():int
getPixelsOfFrame(int):int[]
getXYcord(int):int
ifRestart():boolean
restartTheLevel():void
doneObject():int
allObjectsDone():boolean
getFont(int):Font
getFontLocationOnScreen(int):int[]
getFontTxt(int):String
levelCompleted():int
paintComponent(Graphics):void
paintMovements(Graphics,int,int):void
paintAnimation(Graphics,int):void
paintLevelOne(Graphics):void
paintLevelTwo(Graphics):void
paintLevelThree(Graphics):void
```

Tools used: [12]

3 Application

3.1 Application Requirements

In order to have a fully functioning logical puzzle application the programmer need to implement the given methods inside the framework class Engine. All the logic and the specific things about the game need to be in the application code, e.g. which images or animations the framework should paint, what action should be performed after a button is pressed and so on. The painting itself is made in the framework but the application programmer need to tell the framework what image to paint at what coordinates in the application code. The same goes for the audio. The application programmer need to tell the framework what song to play. All the data structures that hold e.g. object coordinates and object values are created and initialized in the application code.

3.2 Application Design

The application is designed according to the MVC pattern[9] which works in a pretty simple way but can be hard to implement. By designing the application in a MVC pattern[9] we increase the efficiency of the code but this only applies for larger applications which games usually are. The MVC pattern[9] consists of three classes, Model, View and Controller.

The Controller extends View and contains all methods associated with end user actions. It manipulates the values inside Model through View. The Controller have an inner class called MyKeyListener which verifies that the real time reading of the end user input is read all the time. It extends the abstract framework class KeyActions which provides the functionality of reading events.

The View extends Model which enables View to read all new coordinates and data correctly. Thanks to View extending Model all three classes are able to interact with each other. The View contains the GUI[6] that the end user sees. Inside View there is an inner class called Map which extends the abstract framework class PaintMap. Map initiates the call to PaintMap to confirm the repainting of objects for every move done.

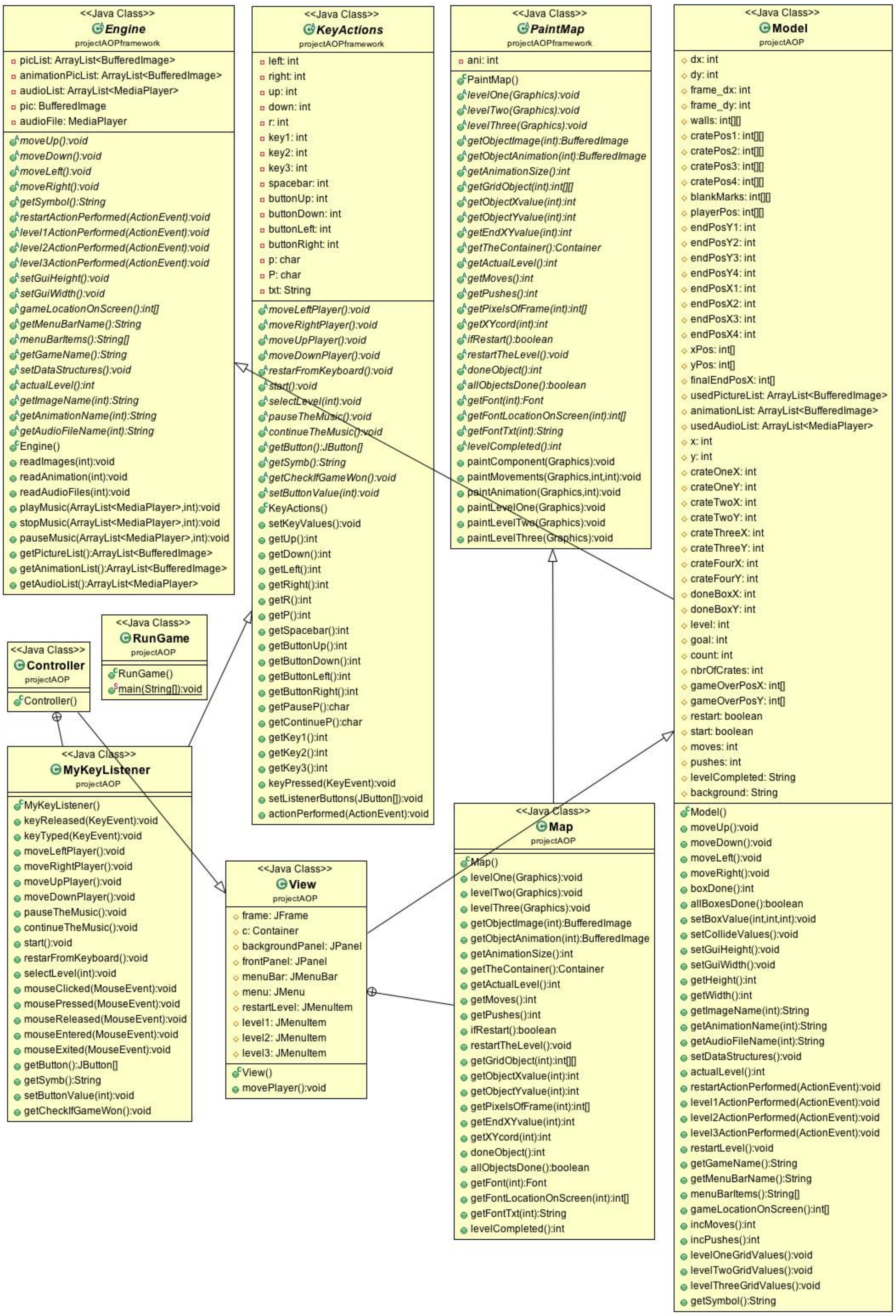
The Model extends the abstract framework class Engine. It contains all the data structures needed for the given application. The class verify that the input from the user read from Controller is updated within the View class. Model is the main point of the MVC pattern[9]. All of this verify the data within the project is updated with the real time information that the application needs.

The application is designed not to hold any form of static methods. This is to confirm that the code is not hard wired to any particular method. It will provide a better chance of testing and debugging the code[1]. The only method that is static is within a main method inside a RunGame class, that initiates the Controller class.

```
package projectAOP;
public class RunGame{
        public static void main(String[] args) {
                @SuppressWarnings("unused")
                Controller c = new Controller();
        }
}
```

The maps that are used in this project are designed by others and the information about this is found here[10].

3.2.1 Application With Framework UML Design



Tools used: [12]

4 Test

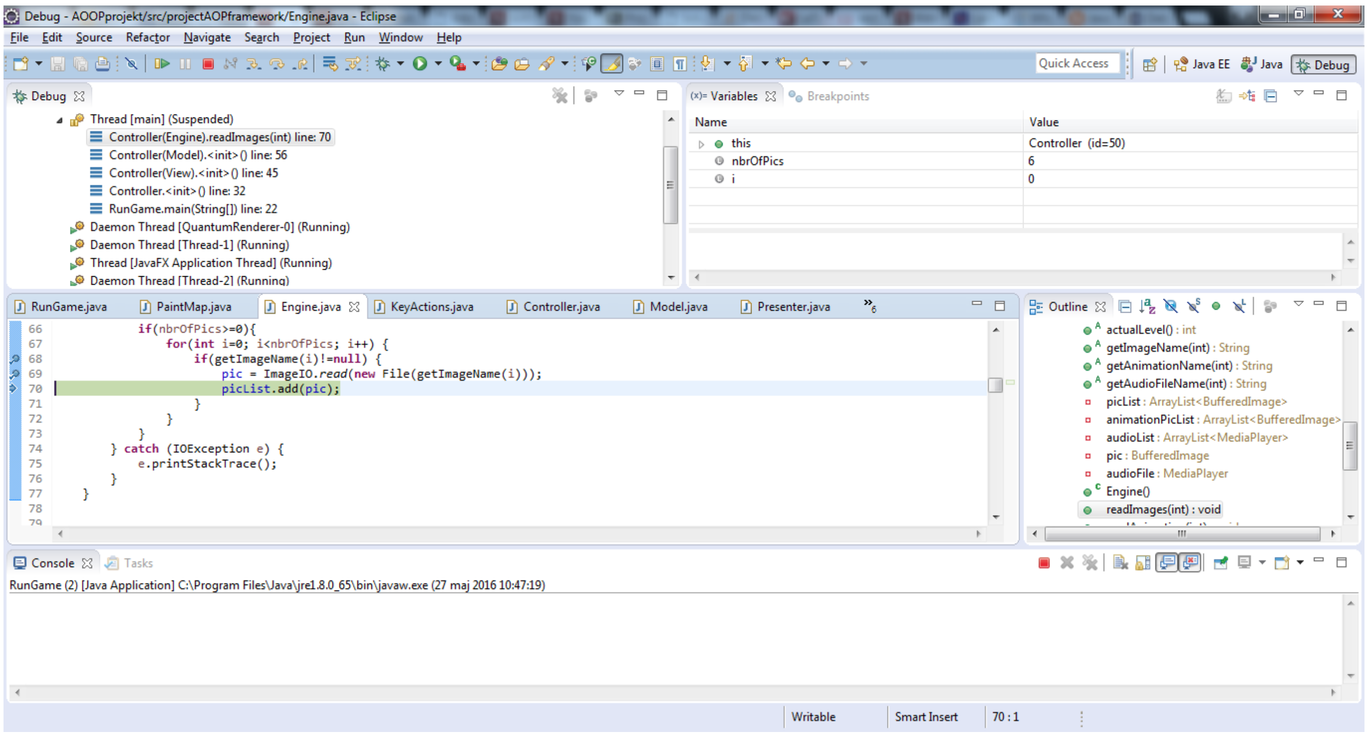
4.1 Code debugging

The code was tested in several steps. First the MVC pattern[9] was tested by passing around data without involving GUI[6]. This was confirmed by the console[2] over a period of time. Typical problems such as NullPointerException occurred and this was checked and resolved by using both the console and the debugging tool inside Eclipse[3]. The important part here is to take one problem at the time and going step by step. A typical debugging scenario during the project could look like the code shown below.

```
/*
* Check why the calls are NullPointerExceptions, starting out with
* checking the problem from the top down with printing out in console
*/
System.out.println(getFontLocationOnScreen(1)[0]);
System.out.println(getFontLocationOnScreen(2)[1]);
System.out.println(getFontTxt(1));

g.drawString(getFontTxt(1),getFontLocationOnScreen(1)[0],
getFontLocationOnScreen(1)[1]);
g.drawString(getFontTxt(2),getFontLocationOnScreen(2)[0],
getFontLocationOnScreen(2)[1]);
```

If the problem was still not resolved by doing this in different steps, the debugging part of Eclipse also worked fine[3].

Figure 4: Eclipse Debug



```
Thread [main] (Suspended)
    Controller(Engine).readImages(int) line: 70
    Controller(Model).<init>() line: 56
    Controller(View).<init>() line: 45
    Controller.<init>() line: 32
    RunGame.main(String[]) line: 22
Daemon Thread [QuantumRenderer-0] (Running)
Daemon Thread [Thread-1] (Running)
Thread [JavaFX Application Thread] (Running)
Daemon Thread [Thread-2] (Running)
```

Name	Value
this	Controller (id=50)
nbrOfPics	6
i	0

```
            if(nbrOfPics>=0){
                for(int i=0; i<nbrOfPics; i++) {
                    if(getImageName(i)!=null) {
                        pic = ImageIO.read(new File(getImageName(i)));
                        picList.add(pic);
                    }
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

```
actualLevel() : int
getImageName(int) : String
getAnimationName(int) : String
getAudioFileName(int) : String
picList : ArrayList<BufferedImage>
animationPicList : ArrayList<BufferedImage>
audioList : ArrayList<MediaPlayer>
pic : BufferedImage
audioFile : MediaPlayer
Engine()
readImages(int) : void
```

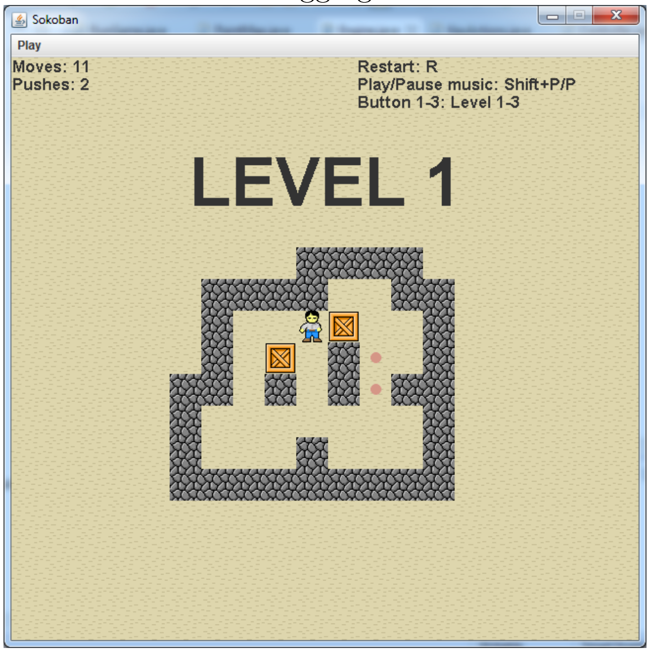
```
RunGame (2) [Java Application] C:\Program Files\Java\jre1.8.0_65\bin\javaw.exe (27 maj 2016 10:47:19)
```

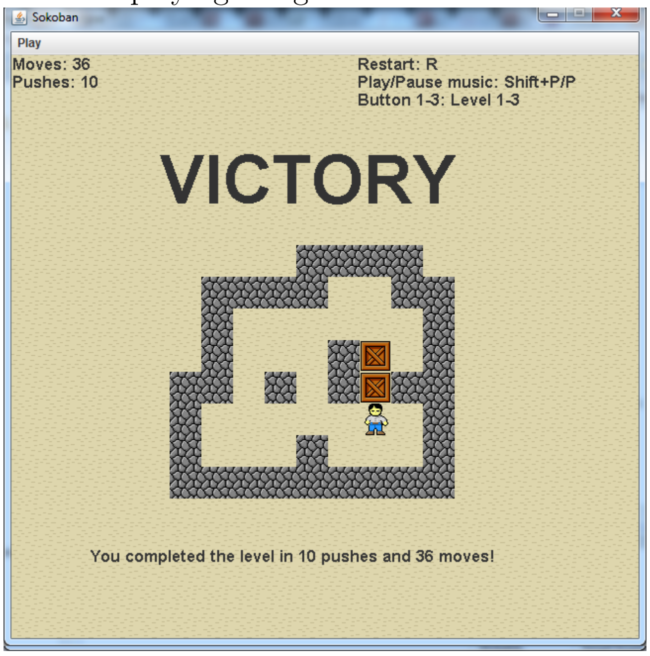
Therefore this way of debugging was made over all parts of the framework and application to verify the prevention of bugs that are hard to discover or detect. JUnit[8] was not used due to the time pressure. The code itself is easy to test with JUnit[8] due to the use of non static methods and proper code structure.

4.2 Game debugging

The game-play testing was made in the same fashion as the code debugging. In these scenarios the actual GUI[6] part was involved. One example of where the console debugging[2] was useful was in the player movement methods. The player movements are depending on values in 2D arrays connected to objects. A crate is connected to an 2D array called cratePos. The 2D array is 20x20, or 640x640 in pixels in this project. To check if a player movement is valid, first check if there is a wall in the position the player wants to move to, in other words, check the 2D array called walls, which hold values for each position where there is a wall. The movements were tested one method at a time. For moveUp the 2D array walls value were printed in the console[2]. The players next position([x][y-1]) value was interesting combined with the value of the walls array. If there was a mismatch this would not move the object to the new wanted position. Problems occurred such as the player object moved through the wall and through the crate object which created confusion. After debugging one step at the time the problem was solved. The same way of thinking was applied for moving crate objects in the games map, to confirm that the crate moved in the right position and also to avoid the crate to go beyond walls. The console[2] debugging made it possible to see the values in the 2D object arrays in an easy way which made it easier to code the logic for the movements.

Another problem that occurred during the development was when a crate object was placed over a marked spot. The crates would not be indicated as marked and the logic with this was confusing. After some debugging it resulted in two methods called doneObject() and allObjectsDone(). The doneObject() method returns the actual crate value and when all the crates are in the correct position the allObjectsDone() will provide a flag that is equal to true. While coding these methods, there were some debugging with them in real time while playing the game.




Sokoban
Play
Moves: 36
Pushes: 10
Restart: R
Play/Pause music: Shift+P/P
Button 1-3: Level 1-3
VICTORY
You completed the level in 10 pushes and 36 moves!

5 Interesting Parts

The player animation is an interesting, yet simple, implementation. The animation is created by taking the player image and shortening the arms and legs into several images. These images are then put in an arraylist and printed out in a circular way. Below the method paintAnimation is shown.

```java
/**
 * The method paints the player as an animation,
 * changing its picture for each step.
 *
 * @param Graphics g, int val
 */
 public void paintAnimation(Graphics g, int val){
    g.drawImage(getObjectAnimation(val),
    getPixelsOfFrame(1)[getXYcord(1)],
    getPixelsOfFrame(2)[getXYcord(2)],
    getTheContainer());
    ani++;
 }
```

And this method gets called from the method paintMovements() inside the abstract framework class PaintMap.

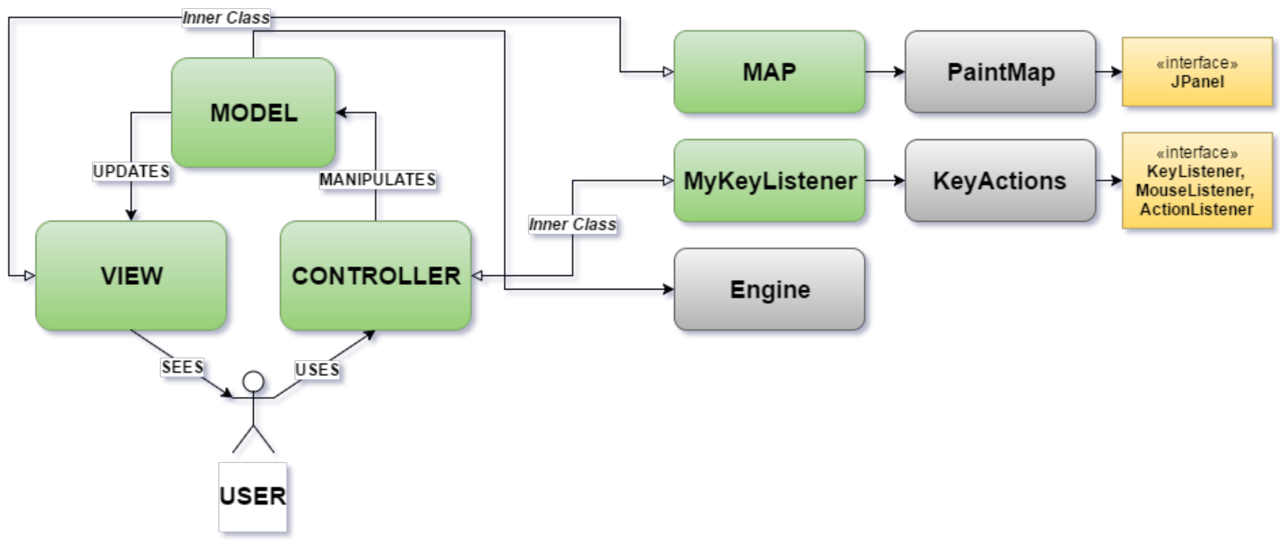
```java
/**
 * The method paints all the moving objects of the map.
 * If the position of the object is changed, it will
 * paint the object at its new position.
 *
 * @param Graphics g, int height, int width
 */
 public void paintMovements(Graphics g, int height, int width){
    ...some code later

    if(ani>getAnimationSize()-1){
        ani=0;
    }
    paintAnimation(g, ani);
 }
```

The reading of audio and image files happens in an efficient way. The method readImages reads a given number of images. The image names are stored in the arraylist getImageName. They are then read as images into another arraylist called picList, where they can be used. The audio files is read in the same way.

```java
/**
 * The method reads a given number of pictures, where zero is the
 * lowest value and indicates one file, and puts them inside
 * an ArrayList called picList of the type BufferedImage.
 *
 */
 public void readImages(int nbrOfPics) {
    try {
        if(nbrOfPics>=0){
            for(int i=0; i<nbrOfPics; i++) {
                if(getImageName(i)!=null) {
                    pic = ImageIO.read(new File(getImageName(i)));
                    picList.add(pic);
                    }
                }
            }
        } catch (IOException e) {
        e.printStackTrace();
    }
 }
```

The framework pattern will adapt dynamically corresponding to the application pattern. Below is figure five and it shows the connection. The framework has now adapted the MVC pattern[9] through the application.

Figure 5: [5]



To be able to work with this framework and application a lot of information has been read and analyzed from the book Object-oriented Design and Patterns[13].

6 Results

The result of the framework together with the application came out as expected. Some parts could have been improved, such as having one big matrix instead of having several matrices containing the data for the different objects. On the other hand the design thinking with the own developed Grid-Layout worked out satisfying and gave a proper graphical outcome. The implementation of reading images and audio files came out solid and good working.

The functionality that was supposed to be implemented was implemented, such as sound, animation, the game of Sokoban and a side implementation of the game TicTacToe.

Sprites[11] could have been used instead of repainting every object for every move. This could be implemented in a new class that would be a part of the framework.

The framework is easy to modify, add new classes and functionality because of the approach of having it patternless.

There is always room for improvement but the outcome with the knowledge that the students possessed the outcome was good.

The Swedish book "Java direkt med Swing" was a great source of information when working with the Swing library[14]. Jan Skansholm[7] is a Swedish professor at Chalmers University.

7 References

[1] Avoid static methods. https://dzone.com/articles/why-static-bad-and-how-avoid. Accessed: 2016-05-25.

[2] Console Eclipse. http://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fviews%2Fconsole%2Fref-console_view.htm. Accessed: 2016-05-25.

[3] Debugging with Eclipse. https://wiki.eclipse.org/Debug. Accessed: 2016-05-25.

[4] Design Patterns. http://stackoverflow.com/questions/320142/design-patterns-vs-frameworks. Accessed: 2016-05-25.

[5] Drawing Help. https://www.draw.io/. Accessed: 2016-05-25.

[6] GUI. https://en.wikipedia.org/wiki/Graphical_user_interface. Accessed: 2016-05-25.

[7] Jan Skansholm. http://www.cse.chalmers.se/~skanshol/. Accessed: 2016-05-25.

[8] JUnit Eclipse. https://en.wikipedia.org/wiki/JUnit. Accessed: 2016-05-25.

[9] MVC Pattern. https://sv.wikipedia.org/wiki/Model-View-Controller. Accessed: 2016-05-25.

[10] Sokoban Maps. http://sokoban-jd.blogspot.se/2011/10/sokoban-lessons-two-boxes.html. Accessed: 2016-05-25.

[11] Sprites. https://en.wikipedia.org/wiki/Sprite_(computer_graphics). Accessed: 2016-05-25.

[12] UML generator for Eclipse. http://www.objectaid.com/. Accessed: 2016-05-25.

[13] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *Object-Oriented Design Patterns by Cay Horstmann*. Cay S Horstmann, 2005.

[14] Jan Skansholm. *Java direkt med Swing*. Studentlitteratur, 2014.