Aberystwyth University

CS12420 - Software Development

Individual Assignment

'Blockmation' Documentation

Author:
James Euesden - jee22

May 2, 2013

Abstract

Documentation for creating an Animation Suite using Java Swing and AWT to make cartoon-like animations that may be saved as a .txt file in a particular format, with the ability to open, edit and run '*scotty.txt*' as a minimum requirement.

Included in this document is the analysis, design, testing, testing evidence and results, conclusion and self-evaluation.

Contents

1 Introduction

The problem set for this assignment was to create an Animation Suite, 'Blockmation', so that a user could build simple cartoon type animations, using small blocks of colour to paint a picture onto a grid. Using multiple of these grids as 'frames' comprises the animation. The user should be able to view their animation in play back. This was required to be completed with Java, using Swing and AWT without the use of a GUI builder.

The animations must be able to be saved to file, as a .txt format, and loaded in for later editing or play back. The .txt files start with the amount of frames in the animation, the amount of rows/columns per animation and then the animation itself, with colours represented by text characters.

This document serves to demonstrate how I tackled this problem and came up with a solution for it, then improved upon the requested design.

2 Analysis

2.1 My approach

I decided that the best way to tackle this problem would be to have two different panes, one for the Director(creation) and one for the Display(animation play back). I felt that having these two panes in one window was one of the most efficient and clean ways to tackle this and so I chose to use a tabbed window. I also felt that visuals in this application were extremely important, as it must be easy and intuitive for a person to use, and so conform to many norms and standards for current popular image manipulation programs.

My first step was to get a very basic version of the program working, focusing on the drawing aspect in the Director panel. I knew that should I be able to get a functional Director panel, many of the different methods that were used would be transferable to the Display and it's functionality. To implement this, I attempted to use a Model-View-Controller approach. This can be seen from my UML Class Diagram (fig 2). To help aid me in starting this application, I observed the code from Lynda Thomas' 'Life' example. While I gained ideas and found this code to be a helpful aid to start, much of the code and the way it was written for a different use meant it was not applicable to my own application, and so while there are some similarities in how the grid is drawn, the resulting classes and product is very different.

SuiteDirector is my Controller, housing the panels, buttons and menus, SuiteCanvas is my View, that holds the methods for drawing onto a 'canvas' and how to display it when added to the SuiteDirector. Finally was my BoardModel, the Model and data storage and handling class. This is where the characters for the colour representations are stored, moved, changed and new 'grids' are made. Later on in development, this is where the handling of frame switching was added.

Once the simplest version of the Director was created, I added some extra functionality, such as file choosing, 36 custom colours, frame previews, 3 different types of tools and more. All of this can be seen in my Use Case Diagram(fig 1). A number of methods were transferable to the Display side of the application and so I began to work on creating the base functionality there. As before, I added extra features, such as a speed slider and the ability to pause and click though frames previous and next to the current frame.

2.2 Use Case Diagram

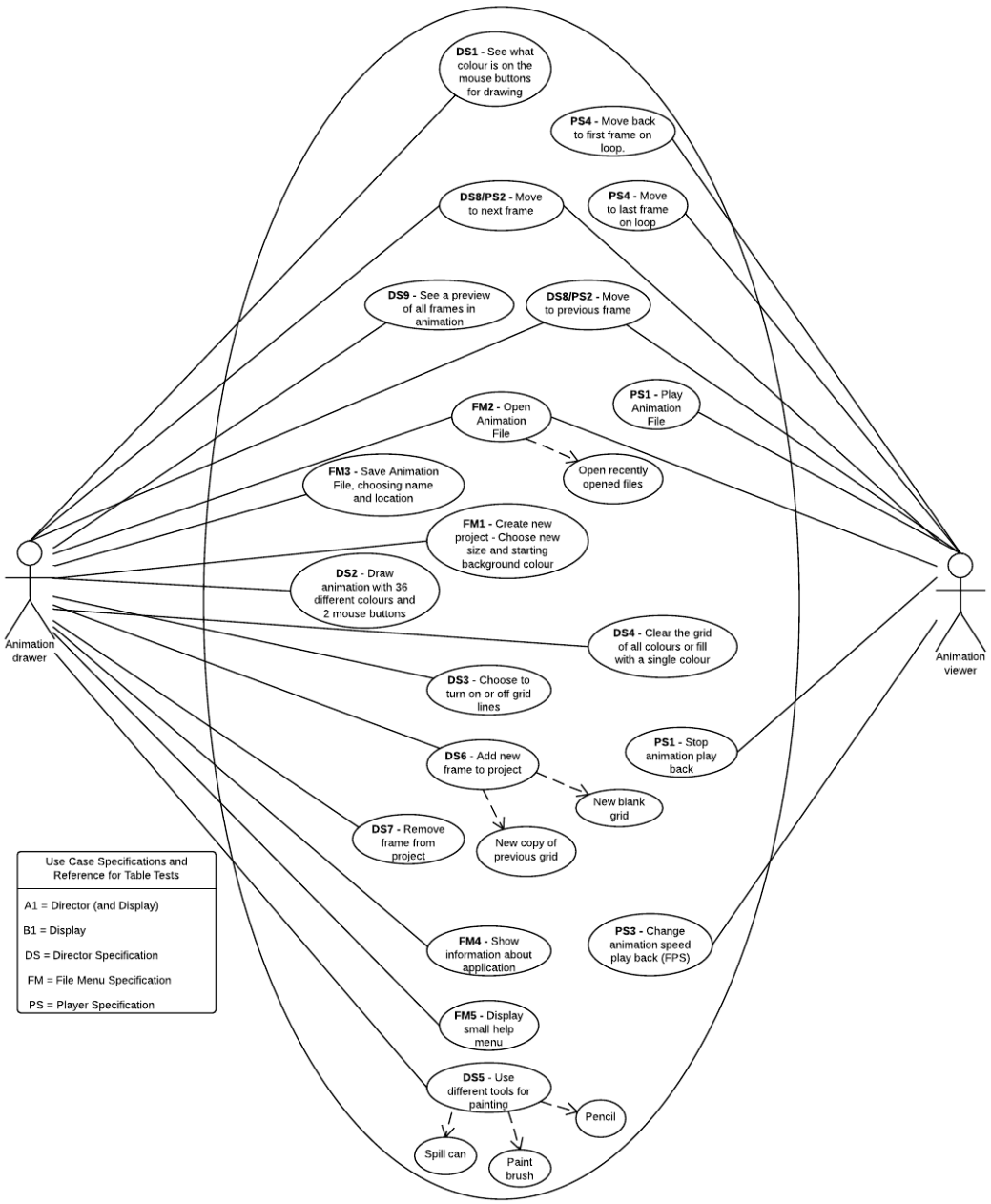


Figure 1: Use Case Diagram for Blockmation, showing all actions the user should be able to accomplish with the application.

As shown in the Use Case Diagram, a number of additional features beyond those in the project specification were added to the list of features to be included with the application. I felt this made the application a much better end-user experience and was better for me to learn from while programming. The final program would act more like an something that would be available for use outside of being just an assignment project.

3 Design

3.1 UML Class Diagram

The consequence of wanting to implement so many additional features required a number of extra classes to function. The final count is 17 classes, although I feel that should I have been more efficient with setting up JPanels and perhaps an Interface for my View classes, there would have been more. The classes and their associations are shown below in my UML Class Diagram which helped me plan out the structure of my application as it was developed. Note: I have not shown where classes inherit from Swing/AWT classes, such as JPanel and ActionListener. This was to save space on the diagram as so many classes would so this.

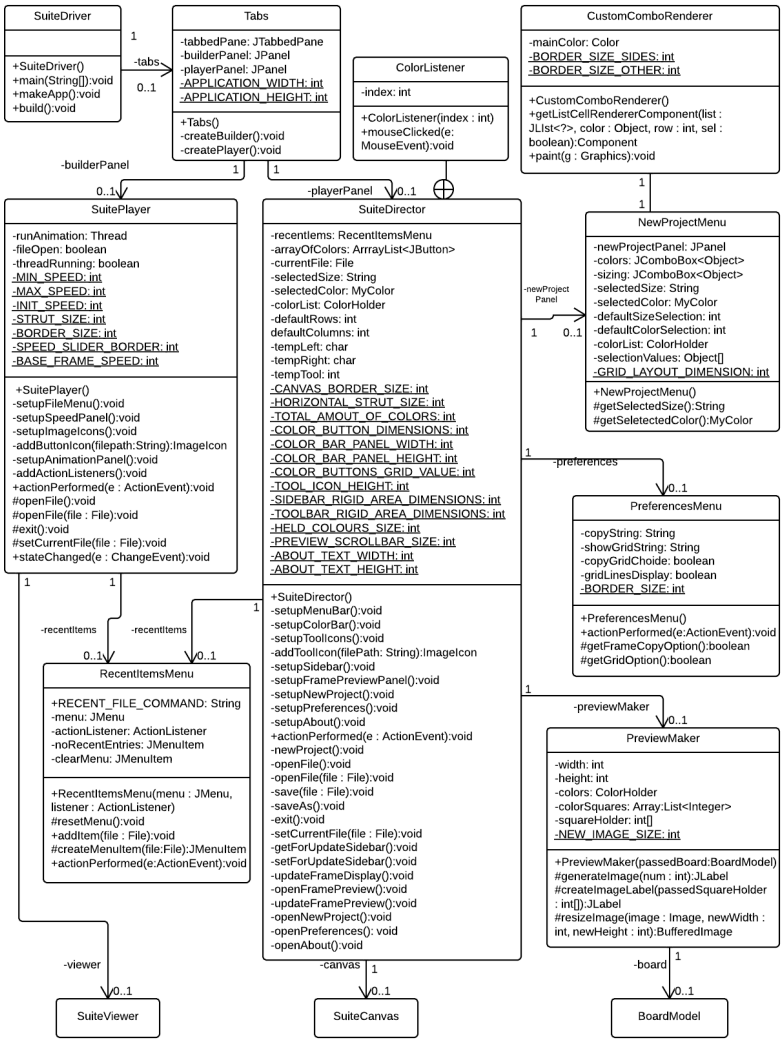


Figure 2: UML Class Diagram - Controller and various Menu classes with associations.

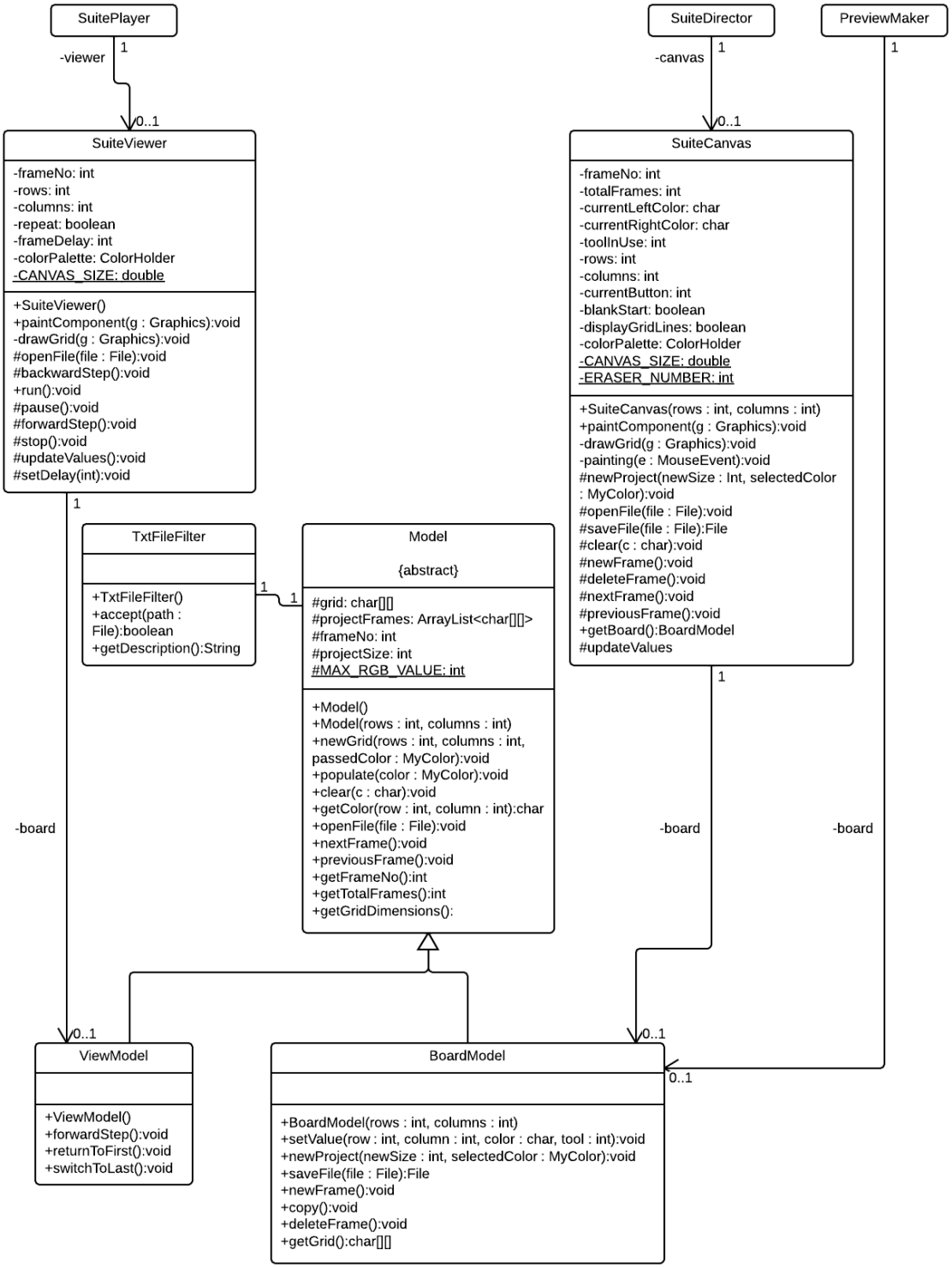


Figure 3: UML Class Diagram - View and Model classes.

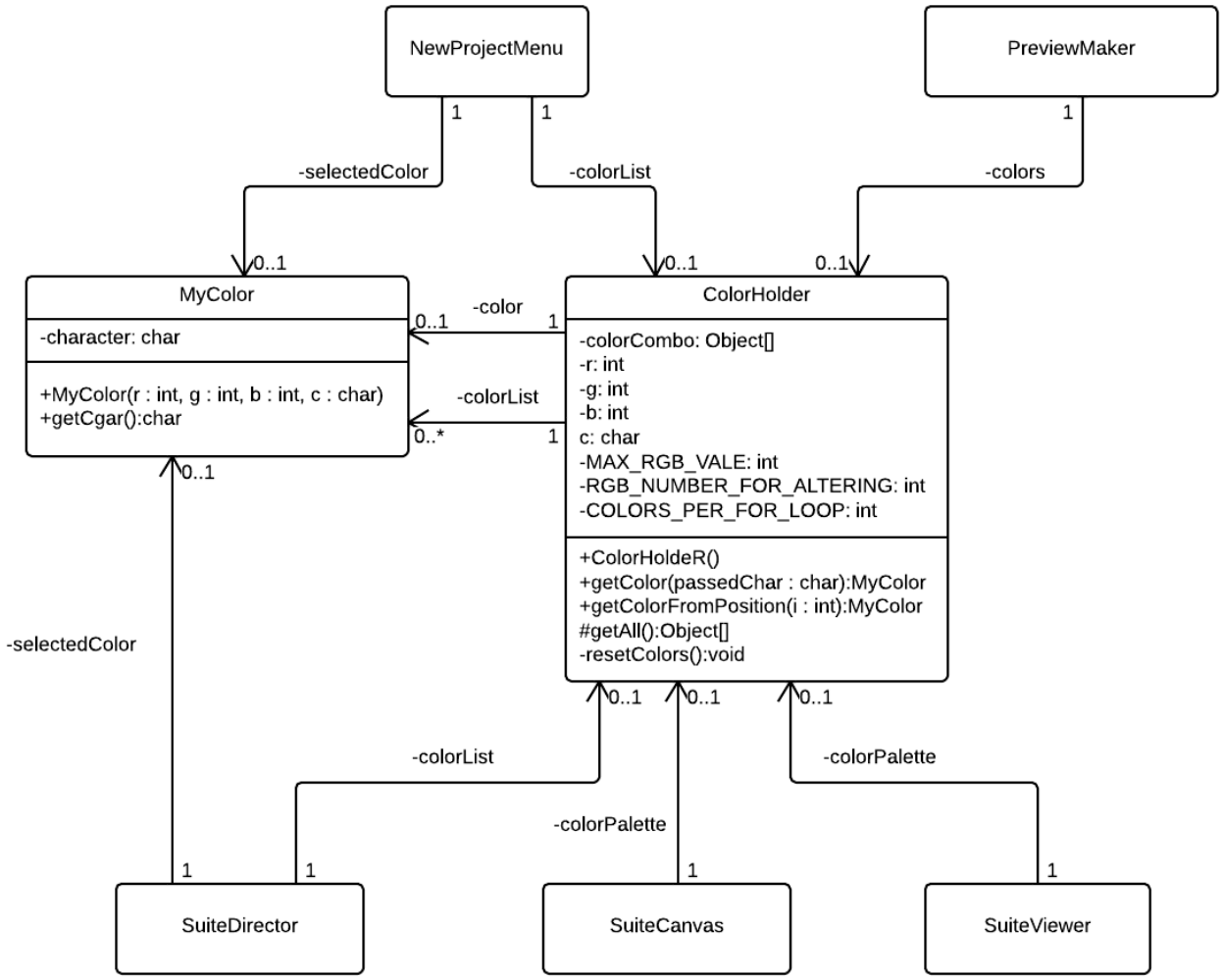


Figure 4: UML Class Diagram - MyColor, ColorHolder and their associations.

In order to preserve space in the Class Diagram, the following classes have some variables and methods that I felt were not necessary to include to see implementation removed. These are:

- SuitePlayer: The majority of the Swing field components, such as JMenu menuBar and JLabel sliderLabel.
- SuiteDirector: The majority of the Swing field components, such as JMenuItem newMenuItem and JPanel colorHolderPanel.
- SuiteCanvas: Many 'Getter' and 'Setter' methods, such as getLeftColor() and setBlankStart(boolean).

To see a full list of fields and classes, consult the JavaDoc provided with this documentation.

3.2 Design implementation

To show how the functions from the Use Case Diagram are implemented a brief description of each class and the most important methods and algorithms are required. For a full detailed description of each class, method and function, consult the JavaDoc provided with this documentation.

3.3 Class Description

SuiteDriver The main class that sets the look and feel and runs the application by creating a new instance of Tabs.

Tabs The overall frame of the application, providing the tabs for Director and Display to be attached to. A restriction on the dimensions of the application window are set here. This was due to how the grid is drawn. While the amount of rows/columns can be changed, the actual grid size is static. Therefore making the window bigger or smaller would be a pointless operation.

SuiteDirector The main panel and controller for the Director side of the Animation Suite. It is from here that buttons for tools, colours and menu operations are set, along with what should be done when a user presses one of these buttons.

The panel has a BorderLayout.

- North: The Menu bar, to call operations for making a new project, saving, opening an existing project, creating new frames, setting different preferences and more. Most notable from this are the New Project that opens up a new window with a Renderer for a ComboBox (covered elsewhere), the Preferences menu and the utilization of a JFileChooser in order to let users select their file to open and name a file to save. I took this a step further even by using a custom FileFilter to only display directories and files ending with .txt. More information can be found on this when we come to discuss this class.

- East: The Side bar. This panel contains the tool selection, colour selection from 36 different colours and shows what colour is currently on each button. To have so many buttons for colours without having many different JButtons (e.g. redButton, darkRedButton, etc), I created a custom MouseListener (ColorListener) that when added to one of the colours, created from the same JButton, takes an index of where that colour button is stored in an ArrayList of ColourButtons, so when they are clicked by a user, it recognizes which button has been clicked and on what colour.

  There are a number of JLabels on this panel giving useful information to the user. In order to get all of these JLabels and JPanels to sit correctly together to be visually pleasing, panels made to simply be 'containers' are used. These may hold multiple items inside of them, but are added as a single object to the main panel in order to line up all of the different elements. The colours on the mouse buttons display is a good example of this.

  Each of the tools have a small Icon next to them. These were created by myself specifically for this project. I felt this was the best way to show to a user what tools they could use. The images are quite conventional to many other popular Image Manipulation applications. They are added to the system using a ClassLoader, as the image files are stored in a Resource folder in the same directory as the source folder that holds the .java files.

  The images are loaded in through an InputStream and made into Images with ImageIO to read the InputStream. From Images, they are converted into ImageIcons and are then ready to be added to JRadioButtons in place of the radio buttons.

- South: Frame previews are displayed to the user, staying updated every time a new frame is created, a frame is deleted or the user moves through the frames using a short cut key, the buttons on this panel or the Project menu items for moving through frames.

The implementation of this involves getting all of the frames currently stored within the BoardModel and returning them, one by one as an Image on a JLabel. The conversion from 2D char array to Image is covered in PreviewMaker. Borders are set around each of the preview images (Gray), with the current active frame bordered in Black to make it visible to the user which frame they were currently editing.

- Center: The SuiteCanvas class that extends JPanel is added here and comprises the draw space. The algorithm for the drawing is discussed in SuiteCanvas.

These components together comprise the Controller for the application that the user can control any of the actions from. Most action results are simply passing or getting values from the SuiteCanvas class, or BoardModel class through SuiteCanvas. As an example, getting the colours on the mouse buttons is done by saving whatever character represents the particular colour the user chose for that mouse click and painting that colour onto a JPanel, while as in order to move frames, all information is handled in BoardModel through SuiteCanvas. SuiteDirector only makes the call to the method and updates any values after the call has been completed.

SuitePlayer The 'Display' (or 'Player') side of the Animation Suite Controller. This class is similar to SuiteDirector, but controls the Animation play back side. It allows the user to load in and watch a previously saved animation. With this they have the capabilities to Play, Pause, Skip a frame forward/backwards, Stop and increase the frame speed. There are custom icons made by myself for the animation control buttons.

The JSlider to change the frame speed works by having settings of 1 through to 8. When a user changes the slider, a field is updated in the class and records what speed is set. This number is then used to divide the base frame speed and uses this new number for the delay between frame changes, which is just an infinite loop of nextFrame() until the last frame, when returnToFirst() from Model is called.

SuiteViewer This class is the Player version of the Director's 'SuiteCanvas'. It is the panel that the Animation is displayed on in SuitePlayer. Many methods are similar to those in SuiteCanvas, with slight differences for things such as whitespace between squares.

NewProjectMenu A class that extends JPanel. In an effort to keep larger code separate, this menu item is in it's own class. When File - New Project is clicked by a user from SuiteDirector, a window pane of this class appears, allowing the user to choose what new size and background colour they would like for their new project using JComboBoxes. The choice the user last made is stored in memory for use again.

To choose a colour, the actual colours themselves are displayed, as opposed to a description of them. This was achieved through the use of a custom ListCellRenderer (CustomComboRenderer).

CustomComboRenderer Converts the display of the MyColors in the NewProjectMenu JComboBox from a toString description into the colours themselves to be displayed.

TxtFileFilter A class that is sent the file path that the user sees in JFileChooser and only returns items to be shown if they are a Directory or end with .txt. This encourages a user to save their file as a .txt or open only .txt files.

PreferencesMenu Much like NewProjectMenu, PreferencesMenu keeps the code in SuiteDirector by constructing it's window in a separate class. It is a class that allows a user to alter options about the way the application works. One of the options they are given is whether to display the grid lines or not, by changing a boolean in SuiteCanvas that alters the drawing method to paint 'blank' squares as filled white rectangles rather than gray boxes and removes whitespace between squares.

The other option is to start each new frame with a blank (white) frame or as a copy of the previous frame. Again, this is just changing a boolean in SuiteCanvas. Based on the state of this boolean, when a

new frame is made, either a new grid is made and populated with the character for clear or a copy of the previous grid is made.

PreviewMaker In order to convert the 2D char arrays of frames in BoardModel to Images for use in the preview panel at the bottom of SuiteDirector, this class is created and passed the current BoardModel every time a frame is added, deleted or the user moves to the next/previous frame.

Once it has the board, SuiteDirector calls the generateImage method, with the parameter of what frame to create an Image of. The method creates a new ArrayList and a new 2D char array, which is a copy of the frame in the board's ArrayList at the location passed as a parameter. With this copy of the grid, each of the colours is extracted as char and compared to the colours held in ColorHolder until a match is found and returned. This matching colour then has its red, green and blue values (RGB) added into the new ArrayList.

The process of getting the characters from the grid and storing the RGB values is repeated for every character in the grid. Once the for loops are finished, the ArrayList is converted into a regular array and passed as a parameter to create the Image and JLabel. Here, a new BufferedImage a made at the size of the grid and a new WritableRaster is made based on this BufferedImage, with it's pixels set from the array of RGB int values. The result is then passed to another method to resize the Image and then become an ImageIcon.

To resize the Image, Graphics2D is made from a new BufferedImage, passed the Image made in the previous method and paints the image at a new size provided in the parameters. The result is returned, made into an ImageIcon and added to a JLabel, which is then itself returned to SuiteDirector to be added to the preview bar at the bottom of the application window.

RecentItemsMenu As the name implies, this class holds a record of the most recently used items saved and/or opened with SuiteDirector. This works in conjunction with JFileChooser. Every time a new file is opened, it's file path is added to the list of files it stores and an ActionListener is assigned to it (the class itself is the ActionListener). This new entry can be used as a button to re-open the file while the application has a memory of it. There is no limit to the amount of recent files that can be stored and the list can be cleared any time with the click of 'Clear List'. This class came from and was written Neil Taylor. The implementation and usage of it in SuiteDirector and SuitePlayer are of my own work. As this was an extra feature, I felt it would be acceptable to re-use his code.

SuiteCanvas The canvas where the user can draw is controlled and painted by this class. In order for a user to draw, a MouseListener and MouseMotionListener are used together. In this way, it allows the use of mouse dragged with both left and right mouse buttons. When the user clicks a square to draw, mouseListener stores what mouse button they used to memory and calls painting(). This works for single clicks. To make the full 'painting' effect with dragging, mouseDragged calls painting() to take the colour and placement of the mouse, while mousePressed() gets and stores the mouse button used to memory. Working together, these two methods allow the user to smoothly draw upon the canvas.

The painting() method sends parameters to the BoardModel class. Namely, where the mouse is (x,y), what colour they are using (char representation) and what tool they are using (int representation). The other side of this, drawing the grid to be displayed to the user, is drawgrid.

drawgrid() iterates through every element in the 2D char array of the current frame in BoardModel. For every 'Z' returned, the method uses draw3DRect to draw a gray square onto the grid space at the location (this depends on what size the grid is and how many rows/columns. It may also be a white square painted if grid lines are turned off). For every other character returned, the colour it represents is found using ColorHolder and this is painted on using g.fill3DRect and the correct position on the grid.

Other methods in this class deal with passing on parameters and method calls to BoardModel, updating values and getters and setters for tool values, rows and columns.

Model Most of the data manipulation is handled here, where multiple 2D char arrays are kept as representations of the grids held in the current animation. This is an abstract class that BoardModel and ViewModel

inherit from to get most of their methods. The most important methods in this class are newGrid and the nextFrame and previousFrame methods. Others in the class are getters/setters or basic data manipulation, such as writing a single char passed as a parameter to every space in the array of the current grid. The openFile() method is also handled here after being passed the File path of where to load.

The newGrid() method works by creating a new 2D char array based on the requested size, passed as parameters, plus 2 for buffers around the edge of the grid (used in searching algorithms to stop NullPointerExceptions from occurring). The new grid is then filled with the colour from the parameters. If the current amount of grids stored in the ArrayList to represent the list of frames is less then two, this grid is added to the ArrayList twice, otherwise it is just added once. The reason for this is to stop Exception occurring from a user trying to remove what the class considers the last frame. The extra frame is a buffer.

The nextFrame() and previousFrame() methods are very similar methods that get the frame that is either one higher or one lower in the ArrayList of frames, respectively. The important difference comes in with previousFrame. When the user has been drawing on the current last-in-sequence frame, the frame is not actually saved to the ArrayList. For this reason, it is necessary that previousFrame() checks to see if the frame the user is working on is the current last frame. If it is, it is added to the ArrayList and then the previous frame is displayed.

For the openFile() method, the current frames are removed from the ArrayList. The first two lines of the .txt file are read in, and these contain the amount of frames and the amount of rows in the project file. Using these, a series of for loops are ran that creates a new grid for each frame and then takes each line (row) of text from the file and stores at as a string. Then for each character in the row/columnn, each character from the String is individually extracted and saved to the corresponding space in the new grid. As a precaution, should the amount of characters be less than those expected, 'blank' characters will be added to the end of that row until it matches. This avoids any NullPointerExceptions.

BoardModel Inheriting from Model, only a few additional methods are in this class. Most notably are setValue, which handles setting the character to the correct space in the 2D arrays, and how to tackle different tool types, saveFile for saving a project to disk and newFrame, which will add a new frame at the end of a project, or if the current active frame is not the last frame, will add the frame after the current frame. These two scenarios must be declared separately due to the 'buffer' frame at the end of the ArrayList.

Method setValue(), at it's most basic form, adds a single character to the location row and column passed in the parameters. The process becomes a little more complex should the user be using a spill can or paint brush tool. For the paint brush, the same colour value is set to the rows one more and one less than the square clicked and one column more and one less, making a cross pattern. Using this at the edge of the grid, with a block or two that would be painted outside of the grid makes no difference, as they are caught by the buffers in place. The buffers are an extra element added to the start and end of every array in the 2D char array that help avoid ArrayOutOfBounds and NullPointer Exceptions.

Spill can works on a principle of checking to see what the squares above, below and to the sides are. If they are the same as the current colour then they are painted the new colour too. The method to spill paint is called again for each colour that needed to be painted in this way, with the old colour being passed as a parameter to check if it is applicable. Through this, every colour directly touching a square that is of the same colour as the original colour that the user clicked is repainted the new colour.

The method to save a file is practically the same as loading a file, except rather than store the characters in the ArrayList in BoardModel, they are written out with a PrintWriter to a .txt file.

ViewModel Also inheriting from Model, this class only adds three new methods, as this class is for use in the Display side of the Animation Suite. forwardStep() checks that the current active frame is not the last. If it isn't, then the current grid is set as the next grid in the ArrayList. returnToFirst() sets the current active grid to the first. This is used for when the animation is to loop after reaching the end of the frames. switchToLast() is practically the same as return to first except it moves the animation to the very last frame by updating the grid to be the second to last frame.

ColorHolder To make 36 colours, the class ColorHolder creates a new instance of MyColor, sets it's RGB values and character and then adds it to an ArrayList. In order to set the RGB values, a series of for loops are run, each lasting for 5 iterations where a value or red, green or blue is altered, or two or three at a time. On each run through, the newly created colour is stored. The character selection is done by simply incrementing a character each time a new colour is added and saving it to that colour. This class also has methods to return the full list as an array (for the JComboBox) and to search for particular colours, based on their position in the ArrayList or their respective characters.

MyColor A small extension on the Color class from the Java libraries. MyColor inherits Color and adds the ability for a character to be stored with the Color and later return this character for searching and saving methods.

4 Testing

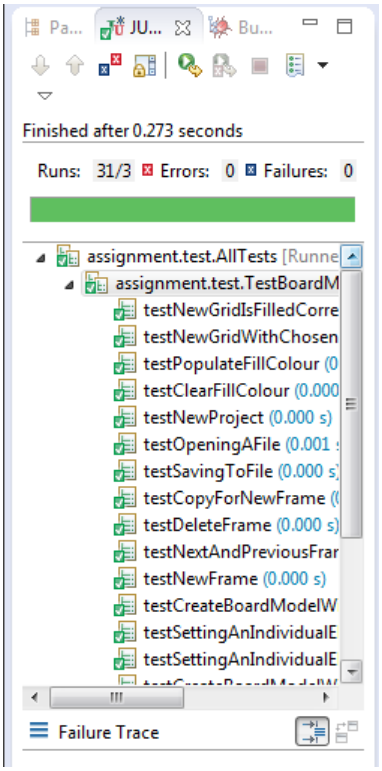
4.1 My Approach

As the application was developed, I repeatedly carried out various forms of testing and bug checking, either using the white box method of checking single blocks of code to check their functionality, or using black box and running the application as a whole to see how each method functioned within the class, whether it worked correctly and did what it was set to do based upon the Use Case Diagram planning.

In order to see what was going on with each method, I would try System.out.println statements that printed out when the application reached a particular point or printed out the state and values of fields and variables. While these were particularily useful, using break points and Eclipse's in-built debugger were also helpful to see a step by step run through of my application and how it acted in real time to find any issues that were occurring.

Once I felt the application was complete, I built JUnit tests to test the Model and data classes for consistency and robustness. In particular I was testing that values stay consistent when entered, no data would be unexpectedly changed and my save and load methods of the data functioned correctly. An example of the tests is shown below.

4.2 JUnit Examples



(a) JUnit tests succeeding.

```
@Test
public void testSavingToFile() {
    BoardModel board = new BoardModel(10, 10);
    board.clear('r');
    board.setValue(2, 2, 'f', 1);
    board.saveFile(saveFile);

    board.openFile(saveFile);
    char[][] testGridArray = new char[board.getGridDimensions() + 2][board
            .getGridDimensions() + 2];
    for (int i = 1; i <= board.getGridDimensions(); i++) {
        for (int j = 1; j <= board.getGridDimensions(); j++) {
            testGridArray[j][i] = board.getColor(j, i);
        }
    }
    for (int i = 1; i <= board.getGridDimensions(); i++) {
        for (int j = 1; j <= board.getGridDimensions(); j++) {
            assertEquals("Grids do match up - Loading unsuccessful",
                    testGridArray[j][i], board.getColor(j, i));
        }
    }
    assertEquals("Frame amount is not that expected from load", 1,
            board.getTotalFrames());
}

@Test
public void testNewFrame() {
    board.newFrame();
    char[][] testGrid = board.getGrid(2);
    assertEquals("Not as many Frames as expected", 2,
            board.getTotalFrames());
    assertEquals("New Frame is not as expected", 'Z', testGrid[4][4]);
}
```

(b) An example of my JUnit tests, here for BoardModel.

```
@Test
public void testBackwardStep() {
    view.forwardStep();
    view.forwardStep();
    view.previousFrame();
    assertEquals("Backwards step failed", 2, view.getFrameNo());
}

@Test
public void testForwardStep() {
    view.forwardStep();
    view.forwardStep();
    view.forwardStep();
    assertEquals("Forward step failed", 4, view.getFrameNo());
}

@Test
public void testNewGrid() {
    view.newGrid(50, 50, new MyColor(255,255,255, '}'));
    assertEquals("Size of new grid does match up!", 50,
            view.getGridDimensions());
    assertEquals("Colors of new grid not set correct", '}',
            view.getColor(5, 10));
}
```

(c) An example of my JUnit tests

Figure 5: Screen shots from Eclipse, showing my JUnit testing of the Model.

4.3 Test Tables

Another useful technique for testing the application in ways that could not necessarily be done with JUnit testing was by using Test Tables. I have provided my Test tables below. I used them primarily for testing aspects of the application that involved the GUI and how a user would use it themselves,as these could not be tested by JUnit or in the code itself.

For each Use Case in the Use Case Diagram, I constructed and executed a test table as best I could. Not all cases were applicable and some were not able to take abnormal input. The Requirement numbers are references to my Use Case Diagram.

ID	Requirement	Description	Inputs	Expected output	Result of Test	Pass/ Fail	Comments
A1. 1	DS1	Choose new colours for mouse buttons and see them updated.	Left click Blue – See Blue	See blue on the left button	Screen shot SS1 is displayed	P	
			Right click Purple – See Purple	See purple on the right button	Screen shot SS1 is displayed	P	
			Middle Click Blue	See no result	Screen shot SS1 is displayed	P	
A1. 2	DS2	Draw with 36 different colours and 2 mouse buttons.	Use all 36 colours and both mouse buttons.	See each colour used on the grid, from both buttons	Screen shot SS2 is displayed	P	
A1. 3	DS3	Choose to turn on or off grid lines	Turn grid lines on	See grid lines	Screen shot SS3 is displayed	P	
			Turn grid lines off	Do not see grid lines	Screen shot SS4 is displayed	P	
A1. 4	DS4	Clear the grid of all colours or fill with a single colour	Clear grid of all colour	See no colours on the grid	Screen shot SS5 & SS6 are displayed	P	
			Fill grid with Left colour	See the grid all as colour on left mouse button	Screen shot SS7 is displayed	P	
			Fill grid with Right colour	See the grid all as colour on right mouse button	Screen shot SS8 is displayed	P	
A1. 5	DS5	Use different tools for painting	Pencil	See a single square filled	Screen shot SS9 is displayed	P	
			Paint brush	See a cross shape filled	Screen shot SS10 is displayed	P	
			Spill can	See a whole connecting are filled	Screen shot SS11 is displayed	P	

(a)

A1. 6	DS6	Add new frame to project	New blank grid	A new blank grid is made	Screen shot SS12 is displayed	P	
			New copy of previous grid	A new grid with the same colours as the previous grid	Screen shot SS13 is displayed (Shows Preferences menu and blank frame)	P	
A1. 7	DS7	Remove frame from project	Remove last frame	Last frame deleted	Screen shot SS14 and SS15 are displayed	P	
			Remove frame mid-project	Active frame deleted	Screen shot SS15 and SS16 are displayed	P	
A1. 8	DS8	Move to Next/Previous Frame	Move through frames	Move through frames	Screen shot SS17 is displayed	P	
A1. 9	FM1	Create new project with new background colour and size	New project	New project created	Screen shot SS18 is displayed	P	
			New project with new size and colour	New project with different size and background color	Screen shot SS19 is displayed	P	
A1. 10	FM2	Open Animation File	Open a real animation text file	File is opened and displayed	Screen shot SS20 and SS21 are displayed	P	
			Open a wrong text file	Alert message is displayed and nothing else happens	Screen shot SS22 is displayed	P	Test passed – Alert message given
A1. 11	FM3	Save Animation File	Save an animation to disk (.txt)	Saves the file to disk	Screen shot SS23 is displayed	P	
A1. 12	FM4	Show information about application	Click 'About' on menu	About menu pops up	Screen shot SS24 is displayed	P	
B1. 1	PS1	Play animation	Click play button	Animation begins to play	Screen shot SS25 is displayed	P	

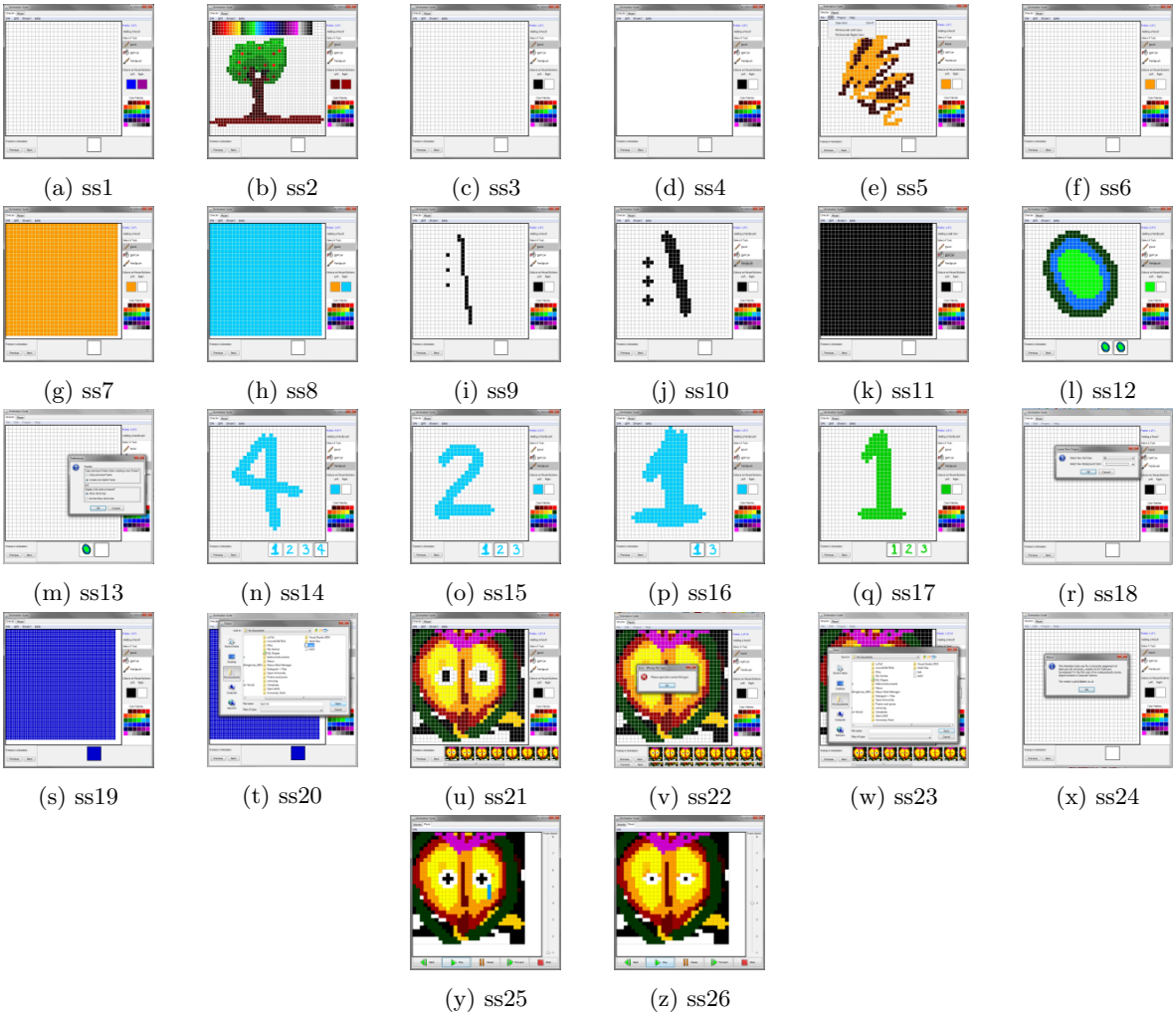
(b)

B1. 2	PS2	Pause and move to next and previous frame	Use appropriate buttons	Pause animation and change frames	No screen shot as proof due to nature of test.	P	
B1. 3	PS3	Change speed of animation play back	Change speed setting to 4	Speed is changed to represent 4x speed.	Screen shot SS26 is displayed	P	

(c)

Figure 6: Test Tables used for testing the majority of the Use Case from a full application perspective.

4.4 Test Table Screen Shot Evidence



(a) ss1 (b) ss2 (c) ss3 (d) ss4 (e) ss5 (f) ss6

(g) ss7 (h) ss8 (i) ss9 (j) ss10 (k) ss11 (l) ss12

(m) ss13 (n) ss14 (o) ss15 (p) ss16 (q) ss17 (r) ss18

(s) ss19 (t) ss20 (u) ss21 (v) ss22 (w) ss23 (x) ss24

(y) ss25 (z) ss26

Figure 7: Screen shots for Test Table Evidence.

On top of all of the technical tests, I found one of the best ways of finding out how to improve the application was through the use of human testing. By asking others to use the program as they would use it naturally. Specifically I wanted to see what they would do with the program and if they could 'break' it in some way. Through this I found new and better ways of improving features, such as the opening of files and checking for error correction, displaying frames and using the mouse on colour buttons.

5 Conclusion

5.1 Application result

The end solution to the original problem is an application with one window and two panes, for Director and Display. The way a user sees and interacts with the application was designed with the old style of MS paint in mind, albeit a very stripped down version without the many tools or colours. It can save and load files in the correct format, saved as a .txt file, from sizes of 5 up to 100 rows and columns.

I am pleased with the result and feel happy that it both matches up to the basic specification set out by the original design brief and adds enough extra features to be an application that could potentially be used for more than just an assignment piece. In particular, I am proud of my approach of using 36 colours and how I implemented it and the algorithms for tool use.

I found the majority of the assignment to build this application both enjoyable and reasonably challenging. The most difficult parts were to think of how to include 36 colours, working out the best way of using the custom ListCellRenderer, making ImagePreviews and thinking of the algorithms for the best way to keep all of the values updated. Most everything else I found relatively easy or at least not too difficult.

My most useful tools were lecture notes, Java books I own, various google searches for small solutions and, of course, experience from working on the mini-assignments and last semester's assignment.

5.2 Potential improvements

As happy as I am with the result, there are a number of improvements that I would like to do had I realized before the implementation would take time re-writing code later. First, I would use an interface for the SuiteCanvas and SuiteViewer, as the two classes have similar methods and yet are not identical. I would potentially do the same for SuiteViewer and SuiteDirector too.

If I had more time, there are also a number of extra features I would like to add. The first would be the ability to import JPG images as a base for a frame. I would also want to be able to export GIF animations. Adding these two would make the application more practical in the real world. On top of this, more tools, perhaps a line, square and circle tool and being able to copy and paste sections. Two features I tried to implement were an Undo button and JColorChooser.

I found the Undo feature to be out of the scope of this assignment. In order to have it completely functional for every action taken by a user, I would have to be saving a copy of the whole application after every action. A dot, a squiggle, a new frame, frame delete, etc. As for JColorChooser, as much as I wished to implement it, I found it to be difficult to implement in this assignment as we were locked into using character representations for colours in order to save out to .txt. Without the requirement of having to save in a particular format, there would not be a restriction on how many hundreds to thousands of different colours that could be used with JColorChooser.

Finally, one feature I planned on implementing but never finished was the Help menu. This was down to time constraints as I felt that as nice as it would be to have the Help menu, it did not add to the current functionality of the basic specification and would be very time consuming to write a whole manual on how to use the application. This is something I would have done at the very end.

6 Self-Evaluation

There is a complete self-evaluation PDF file included in this folder. The following is an exert from this file. The full PDF document should be consulted for my Self-Evaluation Form.

6.1 Expected grade and why I think this?

Expected Letter Grade: B+ (82/100)
How many hours (approx.) did you spend on this assignment? 120 hours

I am expecting higher marks for the complete working code and the additional features. My UML, JUnit, documentation and JavaDoc may not receive higher marks though.

I feel that a lot of the more complex algorithms were explained the most in my JavaDoc and code comments. However, when it came to writing the documentation and discussing the algorithms/classes, because I had already talked about them in JavaDoc I found myself at a loss to explain them further and so feel that my documentation may not be awarded as many marks as it could be.

The structure of my documentation may not be the easiest to read and for the class descriptions I am unsure if I have actually written too much or too little. These things that I feel I may be penalized for are areas I wish to improve on in the next year for future assessments.

6.2 What did I learn?

I feel I have learned a great deal about how to build GUIs that are not only functional but also how to make them appealing to a user. I learned how to make an application be intuitive to a user, based on what seems 'natural' to do and by using standards and conforming to how other popular applications and image manipulation programs function.

I know far more now about class inheritance and implementation than previously, and have learned about attempting to use the Model-View-Controller for design. Code aside, I have gained a better understanding of thinking of algorithms and how to solve problems for bits of functionality that I wanted the application to have both in methods and as a whole application.

My understanding of testing has also been improved, and I have a much better appreciation for the different types of testing, when to use them and how to implement them. I also feel that I have learned how to better manage time. While I feel that I have taken this assignment right up to the deadline, I have managed to get the core application and the majority of the extra features I wished to include added in time, along with the analysis, design and testing. I have come to appreciate that continuous work on the project and getting tasks done sooner is by far the best way to tackle an issue.

Overall, after completing this assignment I feel I am a much improved Java programmer, better logical thinker, wiser planner and smarter designer!