

Application Development Assessment 1 Ashley Ausitn

by Ashley Austin

FILE	11673_ASHLEY_AUSTIN_APPLICATION_DEVELOPMENT_ASSESSMENT_1_ASHLEY_AUSITN_164031_915246386.DOCX		
TIME SUBMITTED	20-DEC-2018 01:58AM (UTC+0000)	WORD COUNT	5338
SUBMISSION ID	97796529	CHARACTER COUNT	31731

LEVEL 5 ASSESSMENT SPECIFICATION

Student name:	Ashley Austin	Student ID number:	179591
Programme:	Applied Computing		
Module:	Application Development		
Module code:	DUM5002	Contribution to Overall Module Assessment (%):	50%
Lecturer:	Daniel Davies	Internal Verifier:	Denise Hudson
Assignment Title:	Design patterns	Word count (or equivalent):	2500 Words
Submission deadline:	December 21 st 2018	Return date of provisional marks & written feedback:	January 11 th 2019
Submission method:	All written assessments, where practical and possible, must be submitted via Turnitin unless otherwise instructed by the Lecturer. (Please DO NOT put this assessment specification into Turnitin or it will match many similarities with other students' submissions.) Alternative submission method (if applicable): Late submission of the assessment will result in a late penalty mark. Penalties for late submission: Up to one week late, maximum mark of 40%. Over one week late, 0%. Only the Extenuating Circumstances Panel may approve a change to submission dates.		
Academic honesty / referencing:	Academic honesty is required. In the main body of your submission you must give credit to authors on whose research and ideas your work is based. Append to your submission a reference list that indicates the books, articles, etc. that you have used, cited or quoted in order to complete this assessment.		

Module Learning Outcomes (from module syllabus)

1 - Evaluate the concepts, principles and technologies and techniques used in the development and creation of interactive experiences

2 - Select and deploy appropriate scripting and programming languages and techniques in the implementation of applications for real-world problem scenarios.

Level 5

Applied Computing

Application Development

Assessment 1

Design Patterns

By: Ashley Austin

Date: December 21st, 2018

Table of Contents

Heading	Page Number
Introduction	4
Part A	5 – 12
Design Patterns	6 – 12
Façade Design Pattern	7 – 9
Observer Design Pattern	10 – 11
MVVM Pattern	12
Part B	13 – 18
Exercise 1	13
Exercise 2	14
Exercise 3	15
Testing	16 – 18
Conclusion	19
References	20
Appendix	21 - 45

Introduction

This assessment will be split into two sections, Part A and Part B. Within Part A I am going to investigate software/design patterns, creating examples to demonstrate their use. I will also be investigating MVVM and the purpose of this pattern, also discussing the drawbacks and benefits of this approach. Within Part B, I have been tasked to create three different applications within Android Studio, I will be creating and discussing these applications. Finally, I will include a section on testing these applications to ensure that they function as intended.

Part A

Within this section I will be looking into the use of software patterns for application development. Additionally, I will be looking into the use of the façade and observer design patterns. I will also be discussing UX design patterns, architectural design patterns and software design patterns.

Finally, I will discuss the MVVM pattern. Discussing its purpose and considering the benefits or drawback of using this approach during android development.

Design Patterns

Design patterns can be split into three main sections. These include;

UX design patterns. During application development, you will also need to consider what the user will see. Therefore, developers can use UX design patterns to their advantage. These are essentially standardised practices for layouts of applications. Making sure to efficiently structure your application for it to be convenient and visually pleasing, this can include visual design choices such as a keypad layout for a mobile application. A developer would use a UX design pattern to ensure that the layout for this is in correlation with other applications.

Architectural design patterns. These are essentially software design patterns but with a much broader scope.

Finally, you have a software design pattern. This can be defined as a general repeatable solution to many common occurrences with software design and development. Vladan D. [1] suggest that software patterns are an attempt to describe successful solutions to common software problems. It is essentially a template that a developer can utilise to further understand how to formulate an application.

Furthermore, design patterns can be utilised to improve development speed and efficiency. As stated by Christopher G. Lasater [2] design patterns allow the code you write to be easier to implement, build and maintain. This is achieved by providing thoroughly tested paradigms. Reusing these design patterns can help the developer to potentially avoid unforeseen problems that may arise in the future. They can also help to make the code easier to read for developers who are familiar with that pattern, this in turn can allow the developer to easily check for issues within the code.

Dinesh Rajput [3] suggests that design patterns can be split into three main categories. These include;

- Creational Design Patterns

The programs based on these patterns are more flexible in deciding object creation according to your demands and your use cases for the application.

- Behavioural Design Patterns

Patterns under this category, characterize the ways in which classes or object distribute responsibility. An example of this pattern would be the observer, which I will be discussing further.

- Structural Design Patterns

Finally, structural design patterns cover the object composition of the classes. An example of this that I will be covering further is the façade design pattern.

Façade design pattern

The façade pattern is heavily based on how your code should be structured and presented to a potential client. This is achieved by providing an interface for a set of interfaces within a subsystem. This source states that [4] the façade pattern can take a complex subsystem and make it easier, achieved by implementing a façade class that provides one reasonable interface. Although within a façade there is capability to have multiple facades for any subsystem. This can help with the overall presentation of the program, as the client will only see the façade as a simplified version of the application. Whereas it has control over a large subsystem. This can be particularly useful when dealing with subsystems consisting of many classes and methods.

As stated by Source Making [5] a façade pattern has a five-step checklist. Firstly, the developer needs to identify a simpler, unified interface for the subsystem or component. Then a “wrapper” class needs to be instantiated that will encapsulate the subsystem. Then the façade/wrapper will need to capture the complexity and collaborations of the component and delegate it to the appropriate methods. Then it needs to be ensured that client use is to the façade only and finally consideration if multiple facades would add value to the application or aid the client.

Some of the advantages of the façade pattern include;

- Simplification of a complex library of potential classes and methods.
- Potential for multiple entry points for the subsystem, simplifying dependencies.
- Capability for high maintenance and readability, due to its well-designed API (Application Programming Interface).
- The dependency on external code can be reduced, allowing for the application to be more flexible.

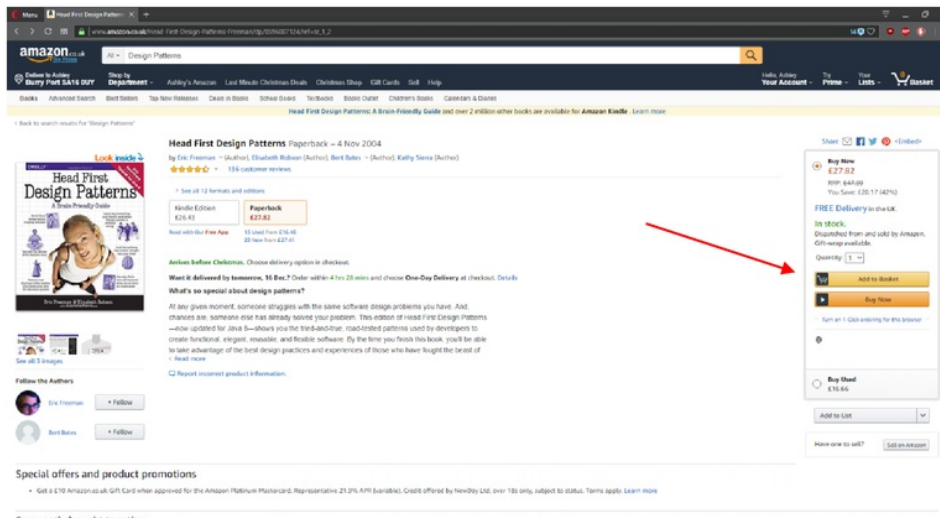
Although this does not mean that this pattern does not have its disadvantages. Such as;

- It may add a layer of indirection, making the program seem more complicated than it may be. Which in turn could affect performance.
- Developers not familiar will need to learn to use the API. Although this is true for most design patterns.

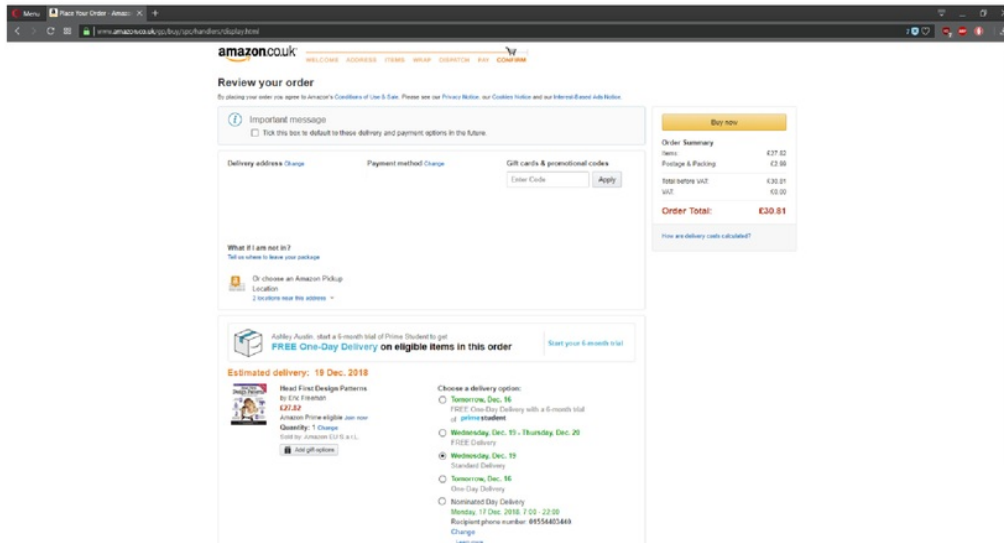
Considering that façade is heavily directed towards potential clients, I will be going over two examples of this pattern. Firstly, looking into how Amazon utilises the façade pattern to improve frequent customer experience.

Amazon One-Click Ordering

The main goal of the façade pattern as explained is to provide one main interface to a large subsystem. Therefore, Amazon’s one click ordering system demonstrates this perfectly. The process for ordering an item on Amazon can be quite frustrating for frequent users.

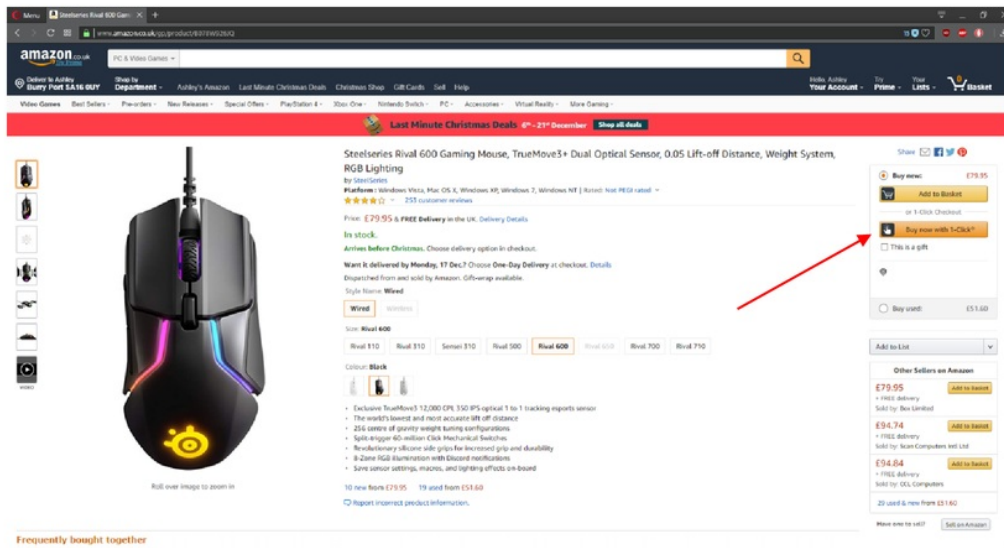


You need to first add the item to your cart.



Then after going through the process of confirming your address, item delivery and credit card information you can then purchase the item.

One click ordering aims to simplify this process, by providing a single option to the user while a main process is going on in the background. In terms of the façade pattern you can take the one click order button as the main façade, while the processes such as delivery location, credit card details and item delivery are the subsystem.



Here is an item that has one click ordering available. Below this is the default address of your account. Once this button has been pressed an order will be placed with the default credit card information and address assigned to your account, without having to go through the entire process of ordering manually.

Observer Design Pattern

The observer is a behavioural design pattern. This design pattern can be used when there is a one-to-many relationship between objects within an application. When this object is changed, all the following subscribed observers or dependant objects will be notified. As Aaron Saray states that [6] the observer pattern is based around state changes, A state change can simply be an object being transformed from one thing to another. When a state change like this occurs, the observer pattern can be used to then notify a subscribed observer of the system to receive push notifications on the changes.

As this source suggests [7] there are two main roles within this pattern. The observer and the observed. Although this is a broader summary of the main components. Therefore, Andrew Powell-Morse states that [8] in the observer can be split into three main components. These include: The Provider; this will send notifications to the observers. The Observer; this will receive the notifications of data from the Provider, whilst also keeping track of the data received to handle and potentially ignore data notifications that may have already been sent. Then the Data; this is the section that the provider will be sending to the observers.

The advantages of the observer design pattern include;

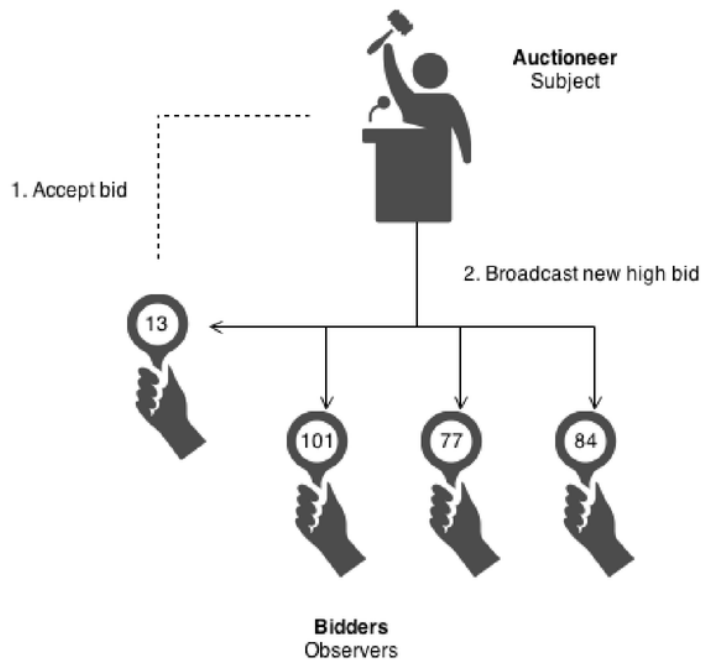
- A unique observer can be added or removed at any point in time.
- Runs off the principle of loose coupling, meaning that the system will function with little to no knowledge of the definitions within other components of the program. This means that the observer does not understand if the process is being acted upon, it will continue to perform its process from start to finish. Some of the advantages of loose coupling include;
 1. Reduced risk that changes made within other classes or objects may negatively affect other classes or objects.
 2. Simplifies the testing, troubleshooting and maintenance of the application.
 3. Allows the application to be dissected into easily definable classes and objects.
- Sending data to other objects can be done without requiring changing the observer or subject classes.

Some of the disadvantages of this pattern include;

- If the observer class is not implemented correctly, this can cause the program to become complicated and potential incur performance issues as a result.
- You may find some inconsistencies with the push notifications when using the observer pattern, although these will be rare.
- The observer pattern relies heavily on inheritance as the observer interface needs to be implemented.

I will be providing an example of this pattern and its use. Looking into a realistic situation where the observer pattern may be applicable.

The Observer Design Pattern and Auctions



This example as provided by Source Making [9] demonstrates firstly the one-to-many relationship that the pattern relies on. The bidders all utilise a paddle to notify the auctioneer when they want to place a bid. Therefore “observing” when a bidder’s paddle is used to then confirm and accept this bid. This in turn will notify the bidders when a paddle is raised increasing the bid price and then allowing them to form a new bid.

MVVM Pattern

The MVVM pattern consists of three main sections. These include Model, View and ViewModel. Each of these sections has a unique function within the model. I will be discussing each of them, in turn discussing the model's purpose.

Model

This will represent the actual data that the application is dealing with. An example of the kinds of data an android application could use includes log in information, or address details. This section of the model will ensure that the data entered the application is held. Although this section does not have the capability to manipulate the information provided, it is not responsible for the presentation of data.

View

This is what the end user will see. It covers the presentation of the data that the Model provides. The aim of this section is to simplify the data in a way that the end user will understand. For example, the model may contain the current date and time stored as seconds. But the view will aim to formulate this data into something that the end user can read.

ViewModel

The ViewModel is used as a bridge between the Model and View sections. It acts as the main controller of the application. It provides a separation between the two previous sections. Therefore, the model will only hold the data that it understands, and the view will provide this data so that the end user can understand.

This model aim is to provide a separation between the different classes within an application, by creating small views and viewmodels with specific responsibilities.

The advantages of the MVVM model include;

- The separation of code makes it much easier to maintain the application, allowing the developer to make changes to a section of the application without it indirectly affecting another class or object.
- Repeated code within android development such as "OnClickListener" will not be needed due to the View class.
- Ability to freely add or replace pieces of code that do similar functions to the correct places within the architecture.

Some disadvantages include;

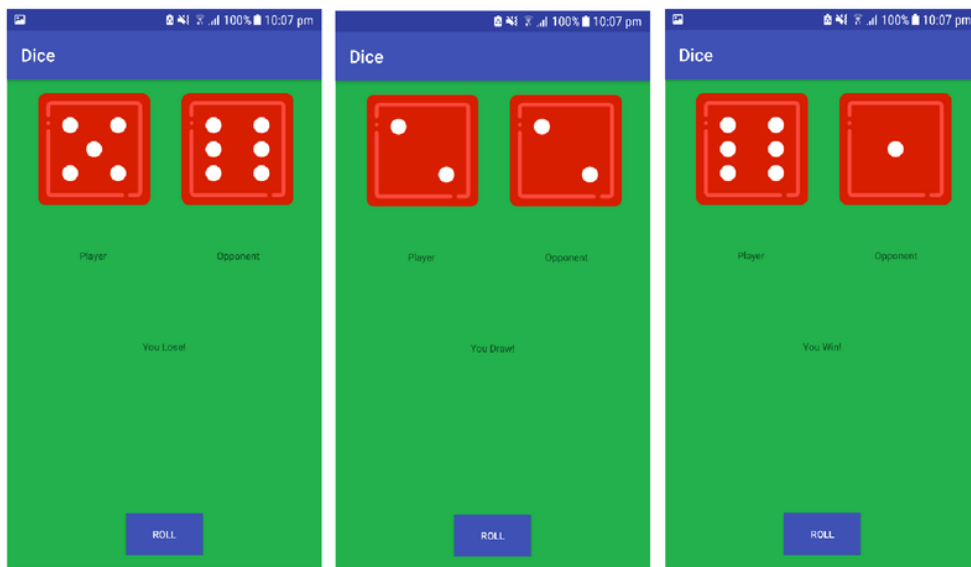
- For simple programs and user interfaces, MVVM may be a little too much. Therefore, the scope of the application and requirement for MVVM will need to be considered.
- Debugging can sometimes be quite problematic for applications that use MVVM due to data bindings.

Part B

I have been tasked to create three applications, within this section I will be presenting and discussing the applications that I have created. I will also be thoroughly testing these applications to ensure they function as intended.

Exercise 1 – Dice Game

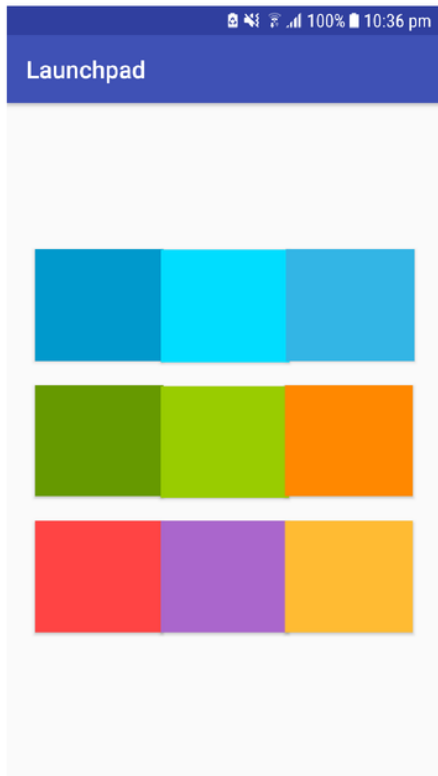
The objective of this game is to receive a higher number than your opponent, based on a six-sided dice. An array of images was created for this application, each representing a side of the dice. Once the button is pressed, the randomNumberGenerator function will select a number from one to six. This is applied for both the player and the opponent. The win, loss and draw conditions are all contained within if statements. The result is displayed to the user once the button has been pressed. Here are examples of each of the potential results.



The code for this application can be found in the appendix (B.1)

Exercise 2 – Launchpad

This application provides a nine by nine grid of squares that allows the user to play a song, each of the squares corresponds to a sound file that will be played when the user selects it. This is a preview of the application.

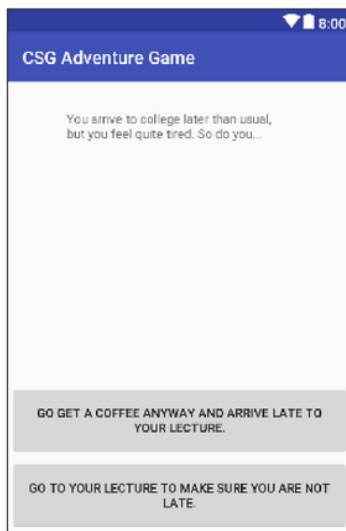


This is the user interface for the application. To provide an example of its function, the top three buttons from left to right will play "note1" through to "note3".

The code for this application can be found in the appendix (B.2)

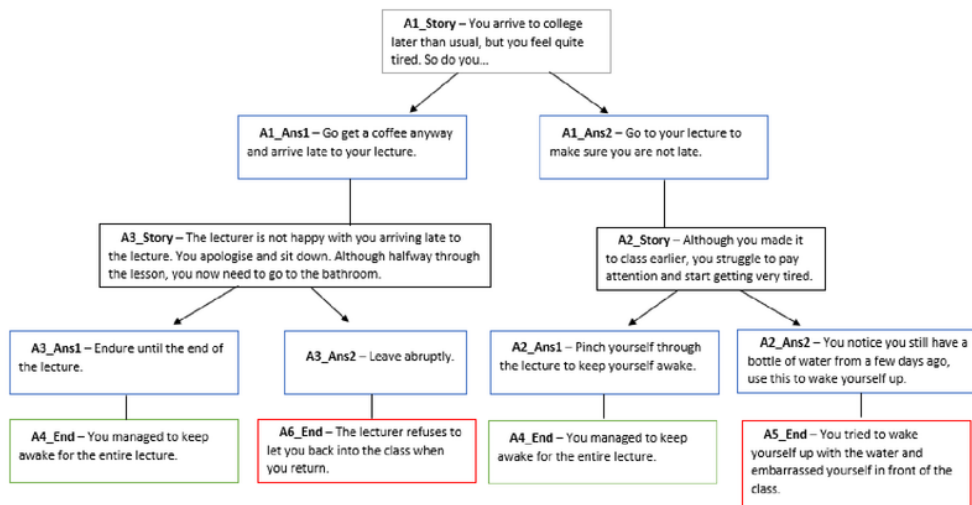
Exercise 3 – CSG Text Adventure

For this exercise I have created a short text-based adventure, providing multiple choices to the user. Once they have reached either a good or bad ending, they are then prompted to close the application to try again. The app functions by applying each option and current event to an index number. It then uses if statements to determine the players current progress.



This is an example of the applications initial question and the two answers. Following is a storyboard of the application.

(Top Button = Left) (Bottom Button = Right)



The code for this application can be found in the appendix (B.3)

Testing

Dice Game

Test Num.	Test Data	Test Purpose	Expected Result	Actual Result
1.0	Win, Loss and Draw conditions.	Thoroughly test and ensure that each of the potential outcomes are functioning as intended. I will be pressing the "Roll" button a total of 5 times.	Application will generate a random outcome each time, providing an appropriate result.	Generated a random result 5 times. Working as intended. Debugging for each of these conditions can be found in the appendix.
1.1	Random number functioning.	Test to see if the random number is working as intended, providing a random number from 1 to 6. I will be debugging the application to see what numbers the function provides.	New random number is generated from 1 to 6 each time the roll button is pressed	New random number generated each time the function is ran.
See Appendix for test images and results.				

Launchpad

Test Num.	Test Data	Test Purpose	Expected Result	Actual Result
2.0	Sound files assigned to each button.	Test to see if the sound files assigned to each button function as intended.	Top left button will play "note1", the following buttons to the right will play "note2" and "note3" respectively. Buttons below will follow the same pattern.	Each button played the correct sound file. Since I cannot use images to show the sound being played. A screenshot of debugging within android studio has been provided.
See Appendix for test images and results.				

CSG Text Adventure

Test Num.	Test Data	Test Purpose	Expected Result	Actual Result
3.0	When the application is reopened, the current state is saved.	Test to see that if the application is closed while the user is currently playing. Their previous state should be saved.	When closed and reopened, the application should remember the previous state.	Application state saved once the program is reopened.
3.1	Testing each storyline.	Test that each branch of story is functioning as intended.	Each option will display without any errors.	Each branch of storyline functioned as intended.
3.2	Testing the win/loss conditions.	Test that each of the four win/loss conditions function, then closing the application once the notification is received.	Each of the conditions will function as intended.	Each win/loss condition was displayed and returned the user to their home screen, closing the application.
See Appendix for test images and results.				

Conclusion

Within this assignment I have covered design patterns, looking particularly into their intended use. I have also discussed two design patterns, façade and observer, looking into their functions, main purpose and the advantages and disadvantages of each pattern. Some discussion on the MVVM pattern was also included, looking into the advantages and disadvantages of said pattern.

I also discussed the applications that I have created, presenting them and discussing their functions. Following I included some testing of the applications, looking into if their main functions are working as intended. I also ensured that the processes within the code were functioning as intended.

References

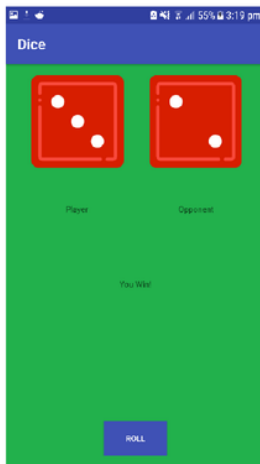
- [1] V. Devedzic, Handbook of Software Engineering and Knowledge Engineering, N/A: School of Business Administration, Univeristy of Belgrade., 2002.
- [2] C. G. Lasater, "Design Patterns," in *Design Patterns*, N/A, Wordware Publishing, Inc., 2010, p. 1.
- [3] D. Rajput, "Spring 5 Design Patterns," in *Master efficient applicaiton development with patterns such as proxy, singleton, the template method and more*, N/A, Packt Publishing Ltd., 2017, p. 41.
- [4] E. F. E. R. K. S. B. B. Eric Freeman, "Head First Design Patterns," in N/A, N/A, O'Reilly Media, Inc., 2004, pp. 258 - 264.
- [5] G. F. M. P. Alexander Shvets, "Facade Design Pattern," Source Making, N/A N/A 2007-2018. [Online]. Available: https://sourcemaking.com/design_patterns/facade. [Accessed 14 December 2018].
- [6] A. Saray, "Problem and Solution," in *Professional PHP Design Patterns*, N/A, John Wiley & Sons, 2009, 2009, p. 13.1..
- [7] R. H. Dustin DiAx, "The Observer Pattern," in *Pro JavaScript Design Patterns*, N/A, Apress, 2008, p. 215.
- [8] A. Powell-Morse, "Behavioral Design Patterns: Observer," AirBrake.io, 29 August 2017. [Online]. Available: <https://airbrake.io/blog/design-patterns/observer-design-pattern>. [Accessed 16 November 2018].
- [9] Source Making, "Observer Design Pattern," Source Making, 2007 - 2018. [Online]. Available: https://sourcemaking.com/design_patterns/observer. [Accessed 16 December 2018].

Appendix

Testing

1.0 ~ Win, Loss and Draw conditions.

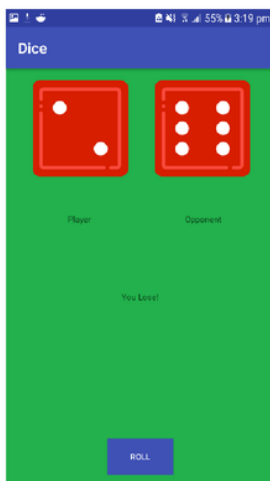
Roll 1;



Player: 3, Opponent: 2

Win.

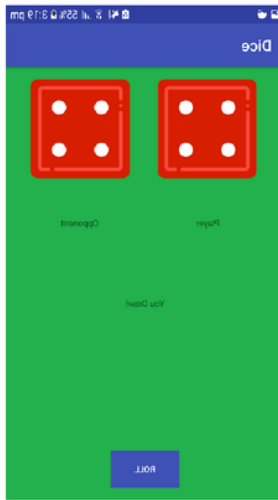
Roll 2;



Player: 2, Opponent: 3

Loss.

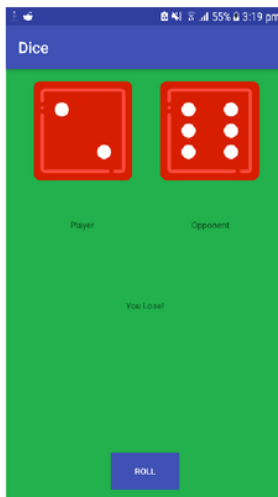
Roll 3;



Player: 4, Opponent: 4

Draw.

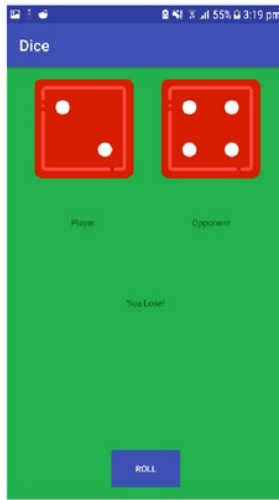
Roll 4;



Player: 2, Opponent: 6

Loss.

Roll 5;

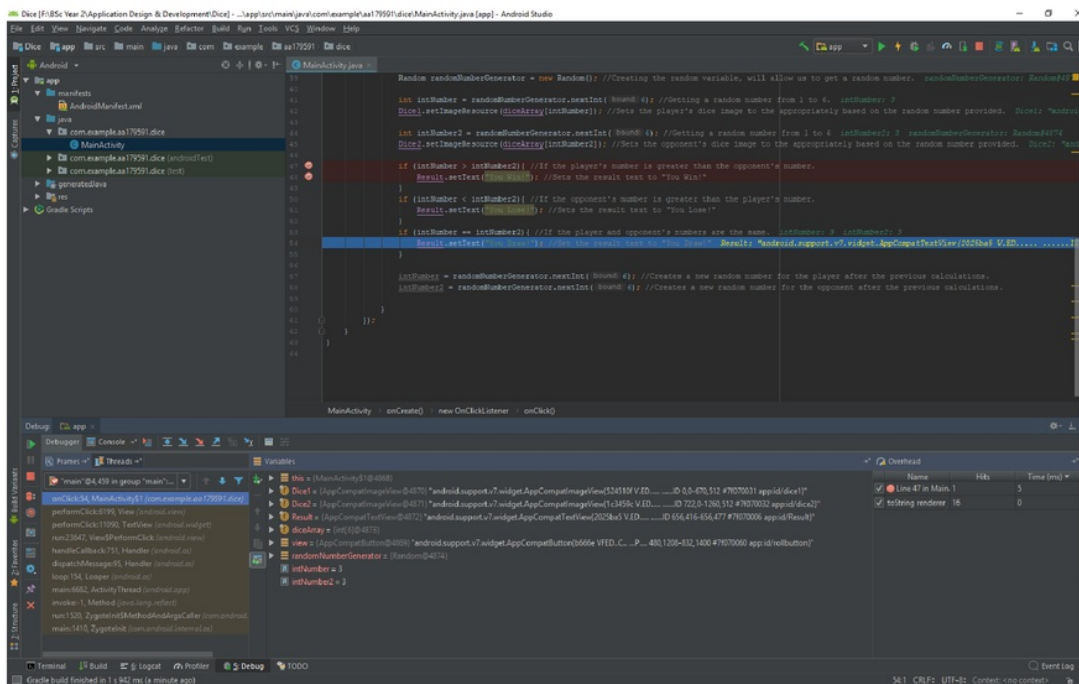


Player: 2, Opponent: 4

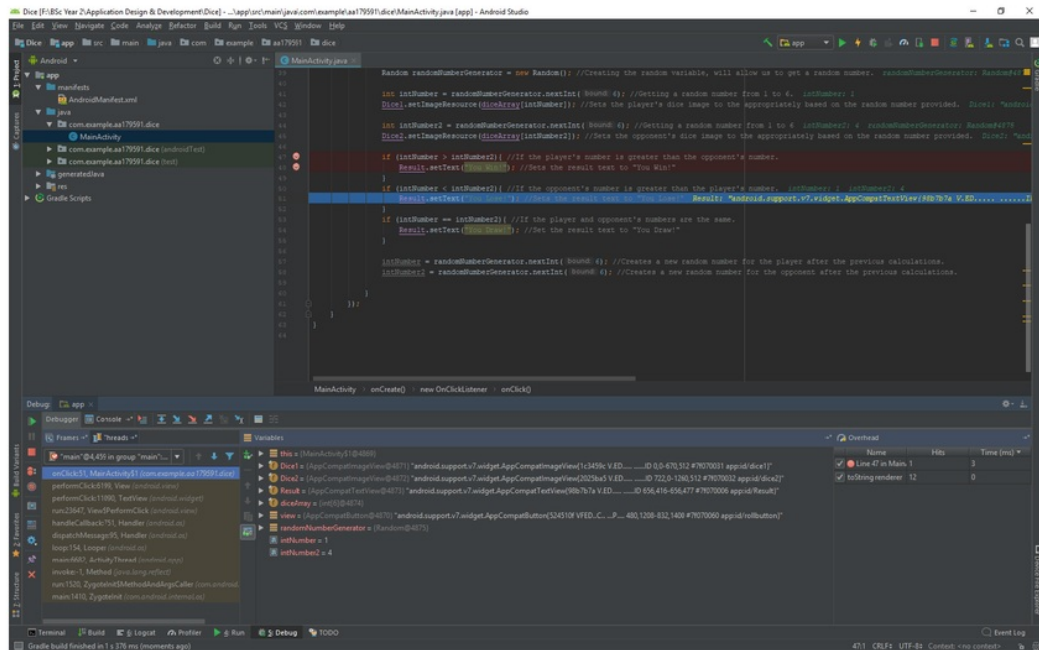
Loss.

Debugging

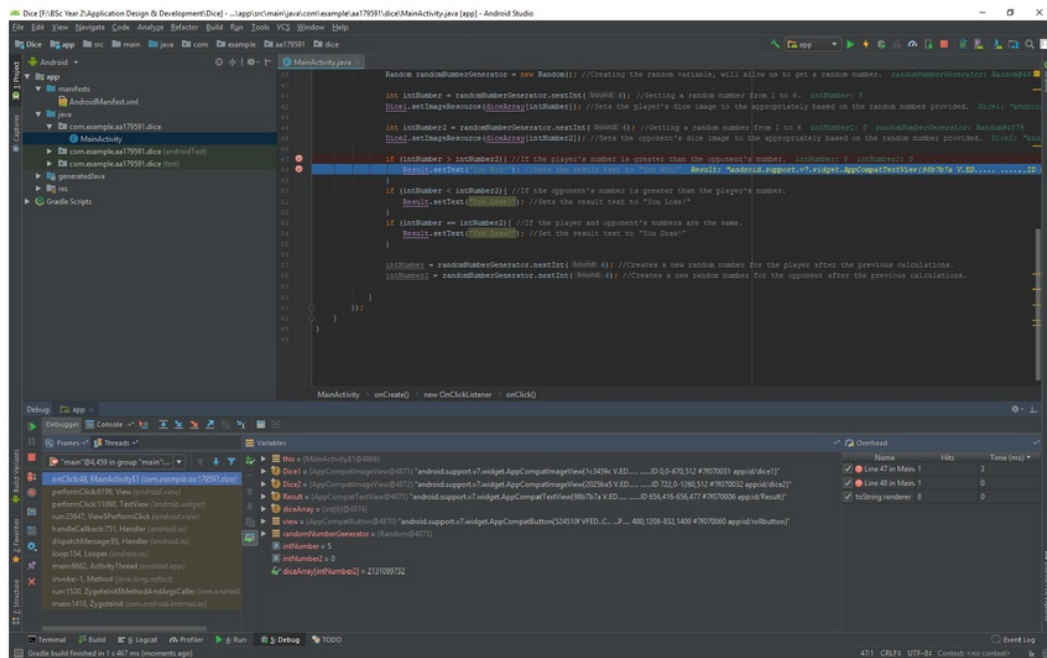
Draw



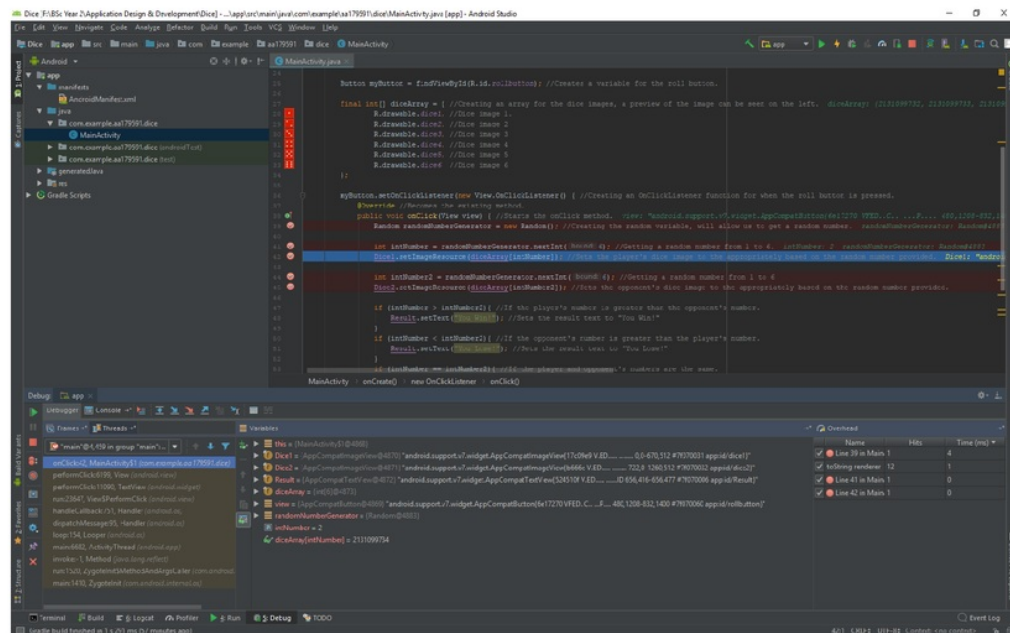
Loss



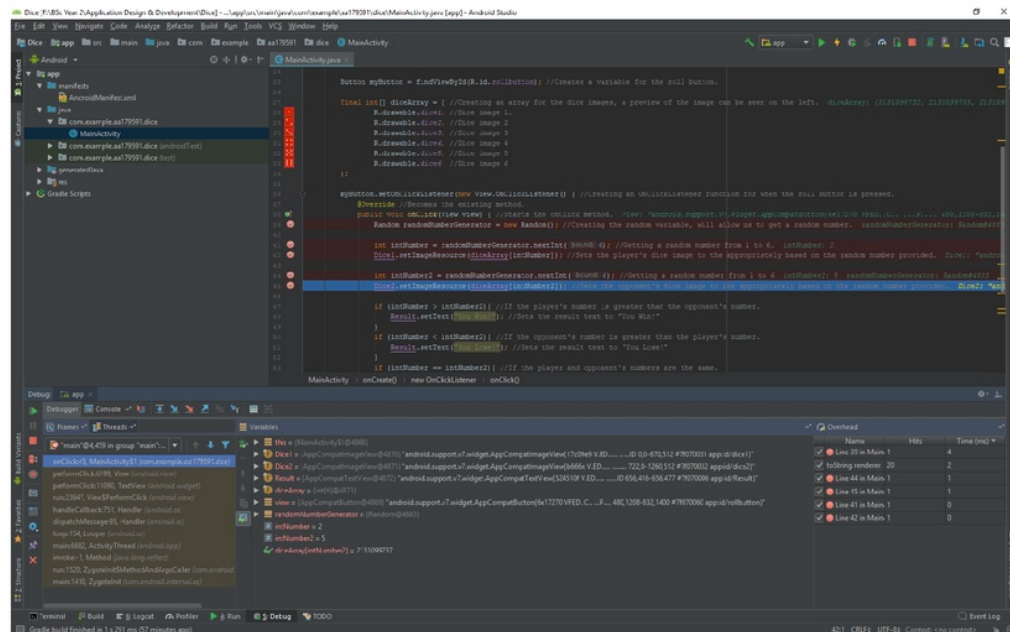
Win



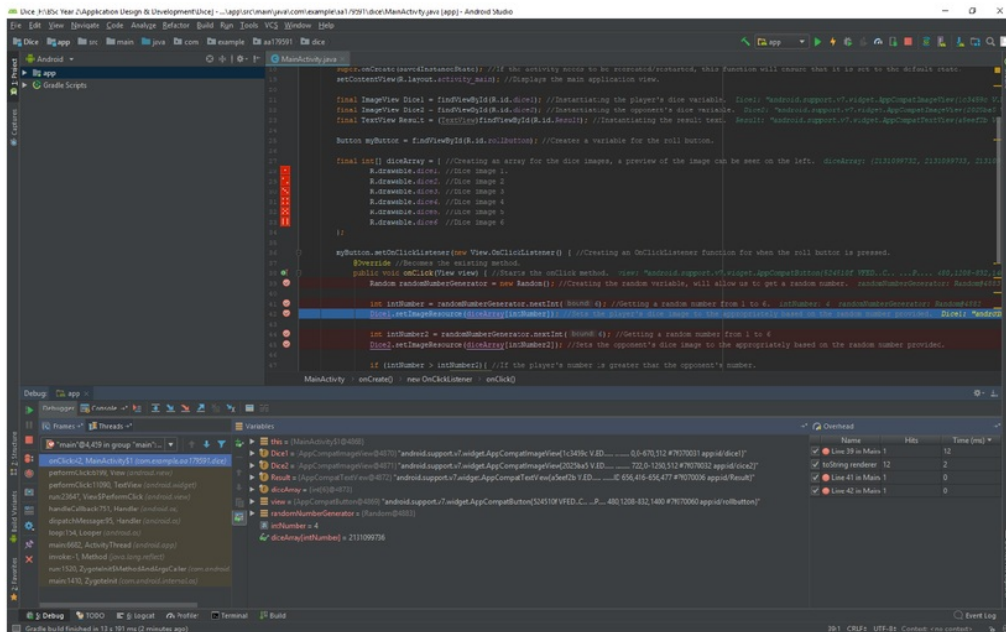
1.1 ~ Random Function



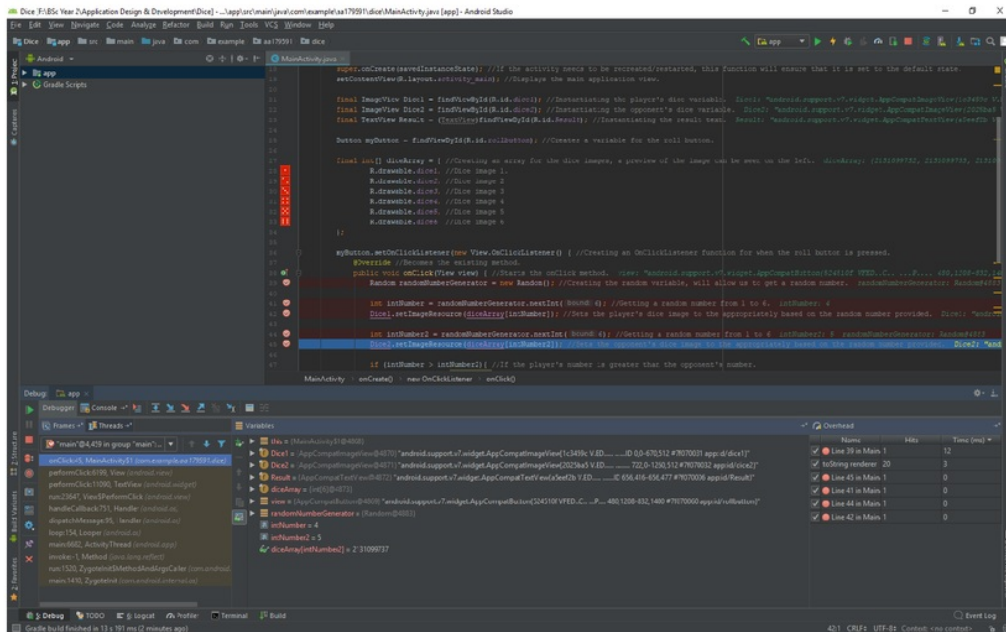
Generated the number 2



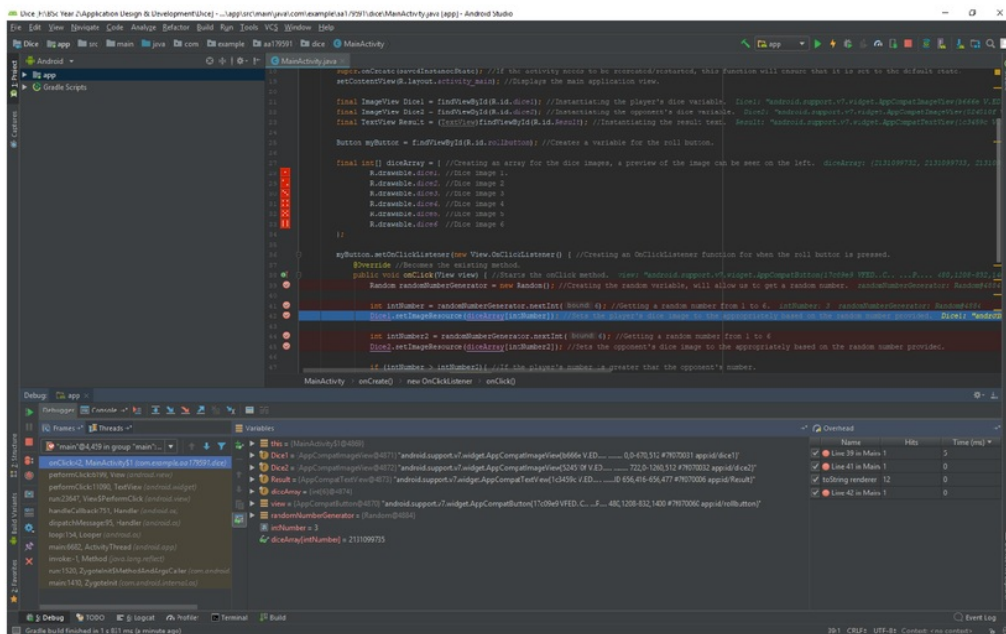
Generated the number 5



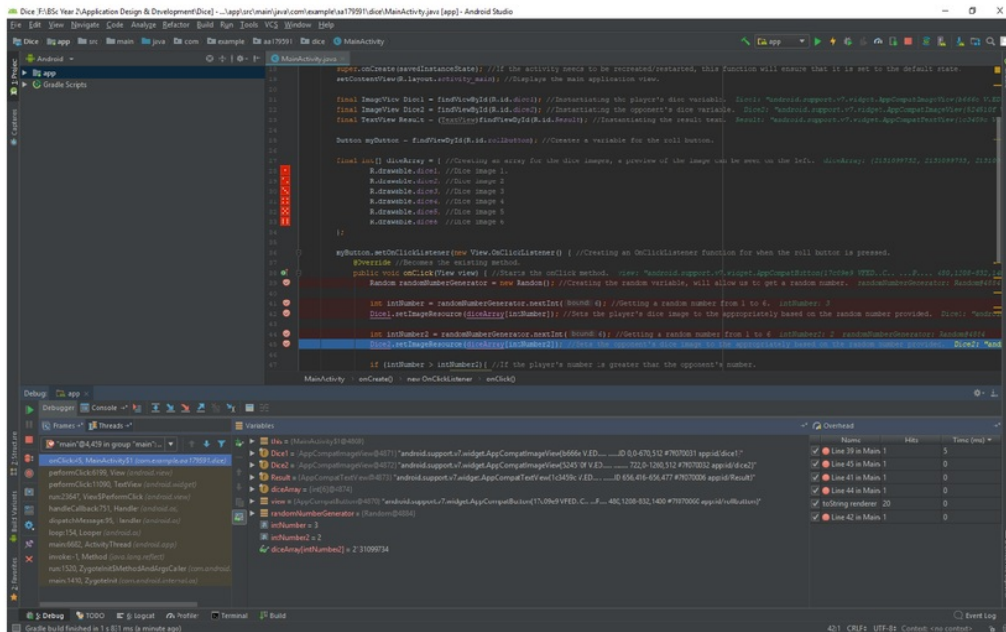
Generated the number 4



Generated the number 5

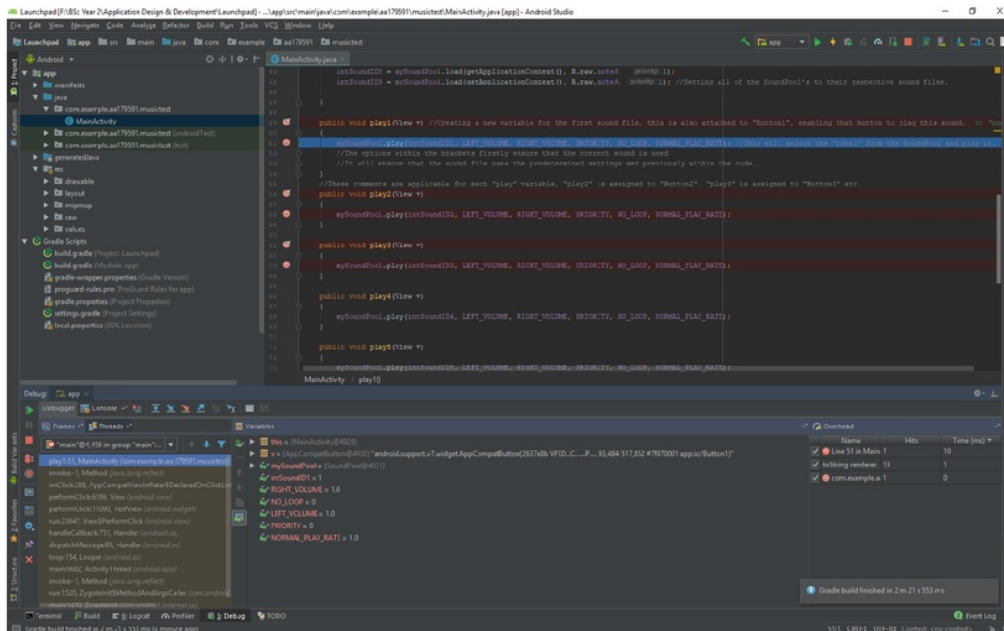


Generated the number 3

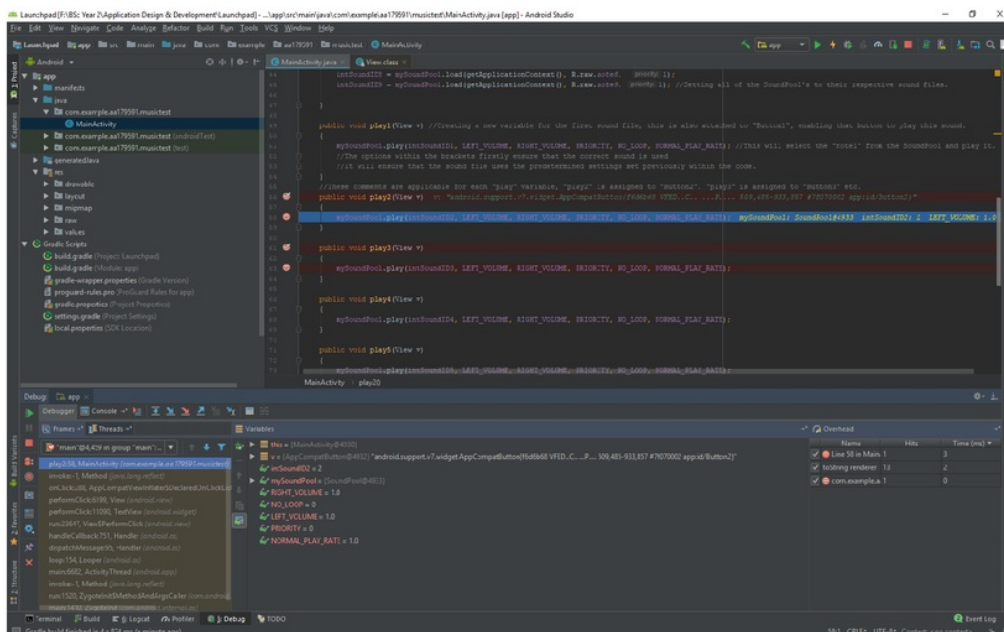


Generated the number 2

2.0 ~ Sound Files Functioning



After pressing the top left button, the function loaded “intSoundID1” which corresponds to the respective note.



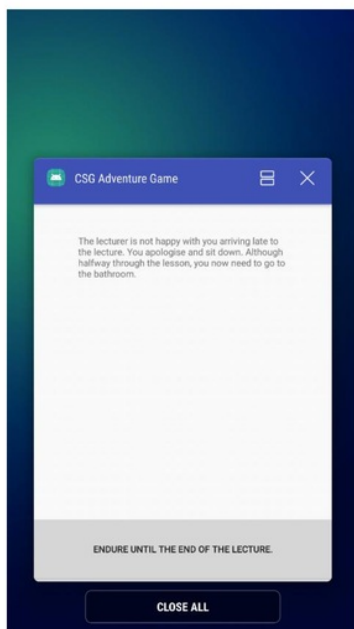
Here it has selected “intSoundID2” which corresponds to the correct note.



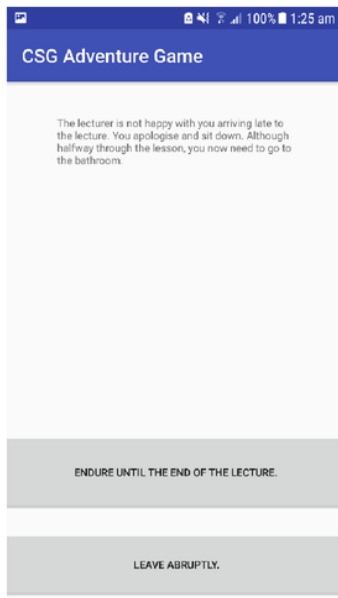
3.0 ~ Saving Current State



Application was left at this section.

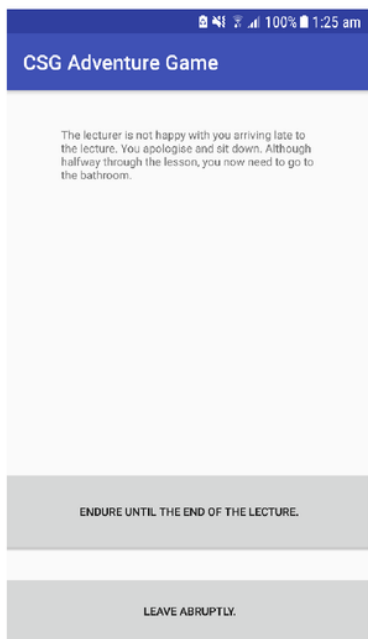
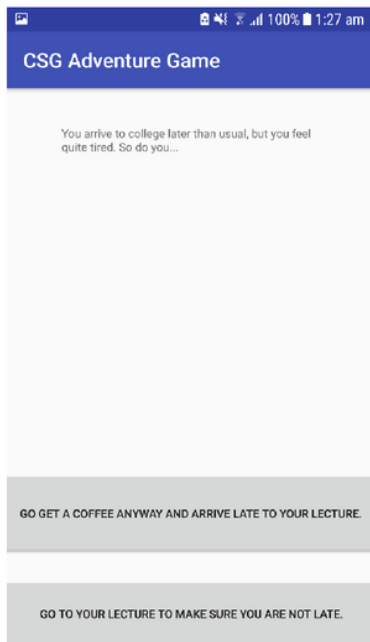


Closed and reopened the application.

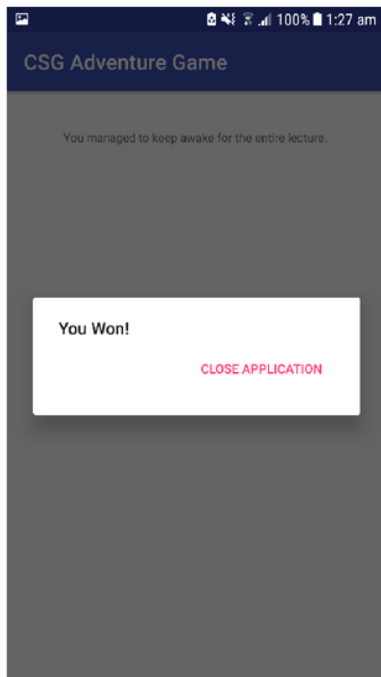


Reopened in the correct location.

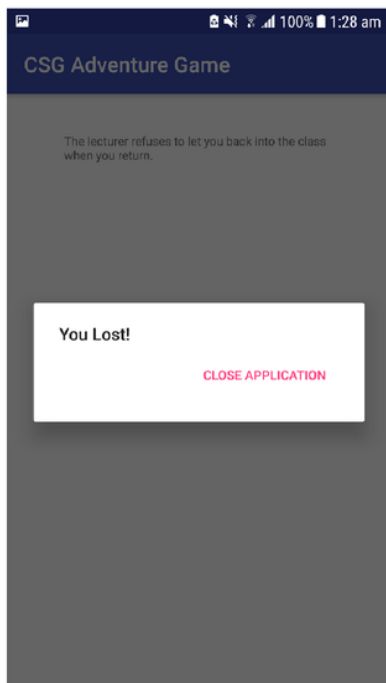
3.1 ~ Testing Each Storyline



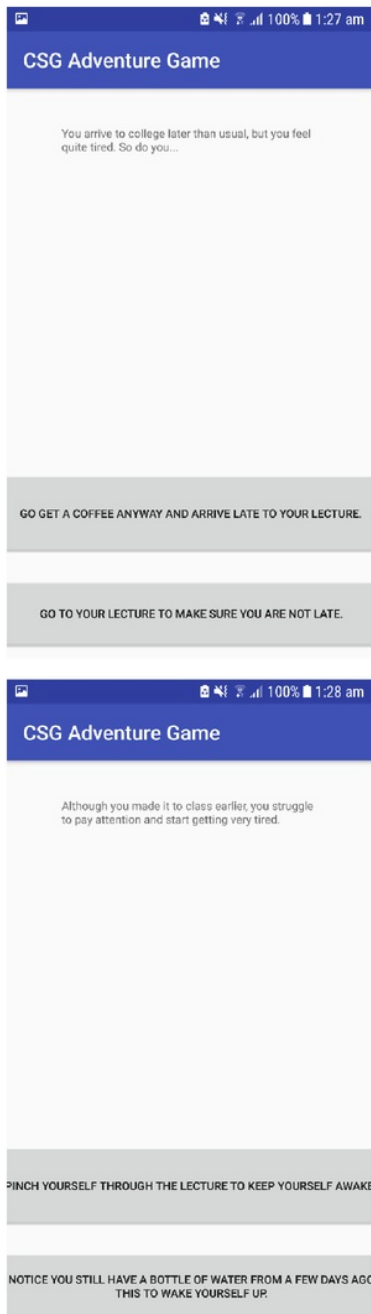
Result after pressing the top button.



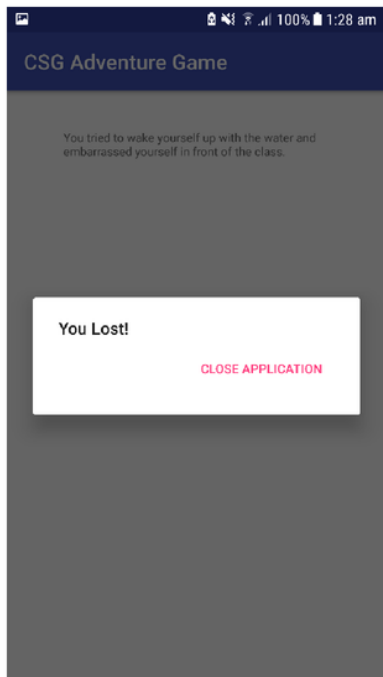
Result after selecting the top option.



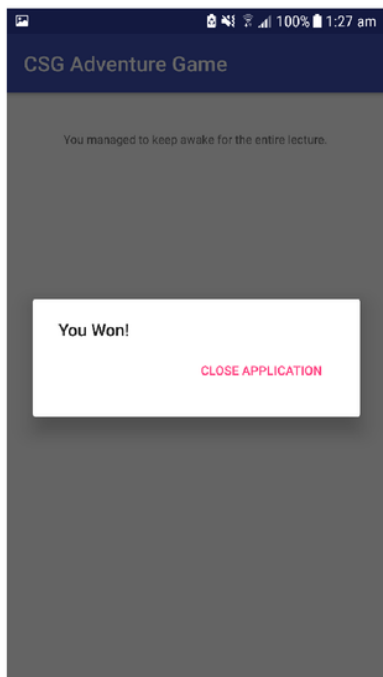
Result after selecting the bottom button.



Result after selecting the bottom button

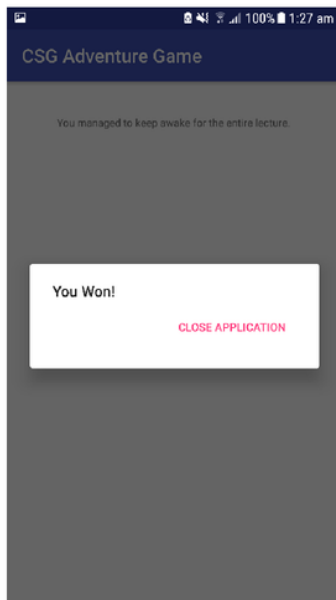


Result after selecting the bottom button.

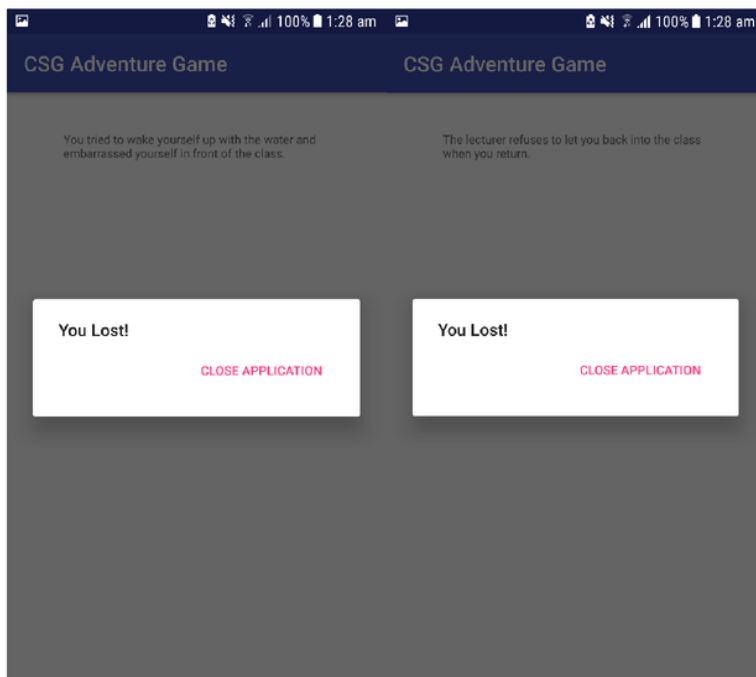


Result after selecting the top button.

3.2 ~ Testing win/loss conditions



This win condition is used twice in the application, for the button combination top-top and bottom-top respectively.



Here are the two loss conditions, each for selecting bottom-bottom and top-bottom. Once the "Close application" button is selected the application then closes.

Application Code

(B.1) Dice Game Code

MainActivity.java

```
package com.example.aal79591.dice;

import android.media.Image;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.ImageView;
import android.widget.Button;
import android.widget.TextView;
import org.w3c.dom.Text;

import java.util.Random;

public class MainActivity extends AppCompatActivity { //Instantiates the main class
for the application.

    @Override //Becomes the existing method.
    protected void onCreate(Bundle savedInstanceState) { //This starts the dice
game.
        super.onCreate(savedInstanceState); //If the activity needs to be
recreated/restarted, this function will ensure that it is set to the default state.
        setContentView(R.layout.activity_main); //Displays the main application
view.

        final ImageView Dice1 = findViewById(R.id.dice1); //Instantiating the
player's dice variable.
        final ImageView Dice2 = findViewById(R.id.dice2); //Instantiating the
opponent's dice variable.
        final TextView Result = (TextView)findViewById(R.id.Result);
//Instantiating the result text.

        Button myButton = findViewById(R.id.rollbutton); //Creates a variable for
the roll button.

        final int[] diceArray = { //Creating an array for the dice images, a
preview of the image can be seen on the left.
            R.drawable.dice1, //Dice image 1.
            R.drawable.dice2, //Dice image 2
            R.drawable.dice3, //Dice image 3
            R.drawable.dice4, //Dice image 4
            R.drawable.dice5, //Dice image 5
            R.drawable.dice6 //Dice image 6
        };

        myButton.setOnClickListener(new View.OnClickListener() { //Creating an
OnClickListener function for when the roll button is pressed.
            @Override //Becomes the existing method.
            public void onClick(View view) { //Starts the onClick method.
                Random randomNumberGenerator = new Random(); //Creating the random
variable, will allow us to get a random number.

                int intNumber = randomNumberGenerator.nextInt(6); //Getting a
random number from 1 to 6.
                Dice1.setImageResource(diceArray[intNumber]); //Sets the player's
dice image to the appropriately based on the random number provided.

                int intNumber2 = randomNumberGenerator.nextInt(6); //Getting a
random number from 1 to 6
                Dice2.setImageResource(diceArray[intNumber2]); //Sets the
opponent's dice image to the appropriately based on the random number provided.
```

```

        if (intNumber > intNumber2){ //If the player's number is greater
than the opponent's number.
            Result.setText("You Win!"); //Sets the result text to "You
Win!"
        }
        if (intNumber < intNumber2){ //If the opponent's number is greater
than the player's number.
            Result.setText("You Lose!"); //Sets the result text to "You
Lose!"
        }
        if (intNumber == intNumber2){ //If the player and opponent's
numbers are the same.
            Result.setText("You Draw!"); //Set the result text to "You
Draw!"
        }

        intNumber = randomNumberGenerator.nextInt(6); //Creates a new
random number for the player after the previous calculations.
        intNumber2 = randomNumberGenerator.nextInt(6); //Creates a new
random number for the opponent after the previous calculations.

    }
}
}

```

(B.2) Launchpad Code

MainActivity.java

```
package com.example.aal79591.musictest;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.media.AudioManager;
import android.media.SoundPool;
import android.view.View;

public class MainActivity extends AppCompatActivity { //Starts the main class.

    SoundPool mySoundPool; //Creates a new SoundPool that the application will use
    to hold the applications sound files.

    private int intSoundID1;
    private int intSoundID2;
    private int intSoundID3;
    private int intSoundID4;
    private int intSoundID5;
    private int intSoundID6;
    private int intSoundID7;
    private int intSoundID8;
    private int intSoundID9; //Instantiating the ID's for the sound files.

    private final int NR_OF_SIMULTANEOUS_SOUNDS = 1; //Limiting the amount of
    active sounds played to one.
    private final float LEFT_VOLUME = 1.0f; //Setting the left sound volume.
    private final float RIGHT_VOLUME = 1.0f; //Setting the right sound volume. Both
    volumes set at the same volume to not confuse the user.
    private final int NO_LOOP = 0; //Ensures that the sound files do not loop.
    private final int PRIORITY = 0; //Defaults the priority to 0, when a button is
    pressed the priority is changed to 1.
    private final float NORMAL_PLAY_RATE = 1.0f; //Ensures the sound files are not
    played at a faster or slower rate than normal.

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState); //Ensures when the application is
        reset, the default home screen is displayed.
        setContentView(R.layout.activity_main); //Sets the current view to the
        application.

        mySoundPool = new SoundPool(NR_OF_SIMULTANEOUS_SOUNDS,
        AudioManager.STREAM_MUSIC, 0); //Creating a new SoundPool.

        intSoundID1 = mySoundPool.load(getApplicationContext(), R.raw.note1, 1);
        intSoundID2 = mySoundPool.load(getApplicationContext(), R.raw.note2, 1);
        intSoundID3 = mySoundPool.load(getApplicationContext(), R.raw.note3, 1);
        intSoundID4 = mySoundPool.load(getApplicationContext(), R.raw.note4, 1);
        intSoundID5 = mySoundPool.load(getApplicationContext(), R.raw.note5, 1);
        intSoundID6 = mySoundPool.load(getApplicationContext(), R.raw.note6, 1);
        intSoundID7 = mySoundPool.load(getApplicationContext(), R.raw.note7, 1);
        intSoundID8 = mySoundPool.load(getApplicationContext(), R.raw.note8, 1);
        intSoundID9 = mySoundPool.load(getApplicationContext(), R.raw.note9, 1);
        //Setting all of the SoundPool's to their respective sound files.

    }

    public void play1(View v) //Creating a new variable for the first sound file,
    this is also attached to "Button1", enabling that button to play this sound.
    {
        mySoundPool.play(intSoundID1, LEFT_VOLUME, RIGHT_VOLUME, PRIORITY, NO_LOOP,
        NORMAL_PLAY_RATE); //This will select the "note1" from the SoundPool and play it.
        //The options within the brackets firstly ensure that the correct sound is
```

```

used
    //It will ensure that the sound file uses the predetermined settings set
    previously within the code.
}
//These comments are applicable for each "play" variable, "play2" is assigned
to "Button2", "play3" is assigned to "Button3" etc.
public void play2(View v)
{
    mySoundPool.play(intSoundID2, LEFT_VOLUME, RIGHT_VOLUME, PRIORITY, NO_LOOP,
NORMAL_PLAY_RATE);
}

public void play3(View v)
{
    mySoundPool.play(intSoundID3, LEFT_VOLUME, RIGHT_VOLUME, PRIORITY, NO_LOOP,
NORMAL_PLAY_RATE);
}

public void play4(View v)
{
    mySoundPool.play(intSoundID4, LEFT_VOLUME, RIGHT_VOLUME, PRIORITY, NO_LOOP,
NORMAL_PLAY_RATE);
}

public void play5(View v)
{
    mySoundPool.play(intSoundID5, LEFT_VOLUME, RIGHT_VOLUME, PRIORITY, NO_LOOP,
NORMAL_PLAY_RATE);
}

public void play6(View v)
{
    mySoundPool.play(intSoundID6, LEFT_VOLUME, RIGHT_VOLUME, PRIORITY, NO_LOOP,
NORMAL_PLAY_RATE);
}

public void play7(View v)
{
    mySoundPool.play(intSoundID7, LEFT_VOLUME, RIGHT_VOLUME, PRIORITY, NO_LOOP,
NORMAL_PLAY_RATE);
}

public void play8(View v)
{
    mySoundPool.play(intSoundID8, LEFT_VOLUME, RIGHT_VOLUME, PRIORITY, NO_LOOP,
NORMAL_PLAY_RATE);
}

public void play9(View v)
{
    mySoundPool.play(intSoundID9, LEFT_VOLUME, RIGHT_VOLUME, PRIORITY, NO_LOOP,
NORMAL_PLAY_RATE);
}
}

```

(B.3) CSG Adventure Game Code

MainActivity.java

```
package com.example.ashleya.textgame;

import android.app.AlertDialog;
import android.content.DialogInterface;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity { //Sets the main class for the
application, called MainActivity.

    private Button mTopButton; //These are used to create a private button,
allowing me to enter text into the buttons.
    private Button mBottomButton; //The same case is used for the bottom button.
    private TextView mTextView; //Will allow me to set custom text within the
TextView box.

    private int mStoryIndex; //I will be using this to relate to the index numbers
within the "clsStoryProgress" array, allowing me to progress the story with the use
of if statements.
    private int mStory; //Private integer value for the text required for the
story.
    private int mTopButtonIndex; //Integer value that will allow me to set the
current top and bottom button index.
    private int mBottomButtonIndex; //Allowing me to set the text within a button
based on an array value.

    private clsStoryProgress[] mStoryBank = new clsStoryProgress[] { //Array of
story parts and answers.
        new clsStoryProgress(R.string.A1_Story, 1), //Story Title 1
        new clsStoryProgress(R.string.A1_Ans1, 2), //Answer 1 for Story 1
        new clsStoryProgress(R.string.A1_Ans2, 3), //Answer 2 for Story 1
        new clsStoryProgress(R.string.A2_Story, 4), //Story Title 2
        new clsStoryProgress(R.string.A2_Ans1, 5), //Answer 1 for Story 2
        new clsStoryProgress(R.string.A2_Ans2, 6), //Answer 2 for Story 2
        new clsStoryProgress(R.string.A3_Story, 7), //Story 3 Title
        new clsStoryProgress(R.string.A3_Ans1, 8), //Answer 1 for Story 3
        new clsStoryProgress(R.string.A3_Ans2, 9), //Answer 2 for Story 3
        new clsStoryProgress(R.string.A4_End, 10), //Ending 1
        new clsStoryProgress(R.string.A5_End, 11), //Ending 2
        new clsStoryProgress(R.string.A6_End, 12) //Ending 3
    }; //This array will allow me to set the buttons and textview's current text
with the use of the index numbers.

    @Override //Sets this as the main view
    protected void onCreate(Bundle savedInstanceState) { //Creating a new
savedInstanceState variable
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        if (savedInstanceState != null) { //If the user is currently progressing
through the application
            mStoryIndex = savedInstanceState.getInt("StoryIndex"); //This will save
the users location, when they exit the application and come back they will be
redirected to their last story section.
        }
        else //If no instance of the user is saved
        {
            mStoryIndex = 0;
            mTopButtonIndex = 1;
            mBottomButtonIndex = 2;
        }
    }
}
```

```

    } //Then this will return them to the default values, providing the
    introduction and two options.

    mTopButton = (Button) findViewById(R.id.button); //Linking the top button
    to the corresponding button on the "activity_main.xml".
    mBottomButton = (Button) findViewById(R.id.button2); //Same case for this
    button as the top one.
    mTextView = (TextView) findViewById(R.id.textView); //Also linking the text
    view from the same xml.

    mStory = mStoryBank[mStoryIndex].getStoryID(); //mStory variable becomes
    the story ID.
    mTextView.setText(mStory); //Sets the text view to the current mStory text
    based on the ID it currently has.

    mTopButton.setOnClickListener(new View.OnClickListener() { //Setting a
    click option for the top button.
        @Override
        public void onClick(View view) {
            checkStoryPath(1); //Defining the button to user selection 1.
            UpdateStory(); //This variable is defined later in the code,
            increments the current index by 1.
        }
    });

    mBottomButton.setOnClickListener(new View.OnClickListener() { //Same case
    for the top button, defining the same settings for the bottom button.
        @Override
        public void onClick(View view) {
            checkStoryPath(2); //Sets this button to user selection 2.
            UpdateStory(); //Increments index by 1
        }
    });

}

private void checkStoryPath(int userSelection) //Creating a new variable for
the story progression.
{
    int ChosenAnswer = mStoryBank[mStoryIndex].isAnswer();

    if (userSelection == 1) //If the user has pressed the top button and..
    {
        if (mStoryIndex == 0) //Checks if story index is 0, which will start
        the A1_Story. Placing it into the TextView.
        {
            mStoryIndex = 5; //Places the A3_Story text into the text view.
            mTopButtonIndex = 7; //Sets the top button as A3 Ans1.
            mBottomButtonIndex = 8; //Sets the bottom button to A3_Ans2.
        }
        else if (mStoryIndex == 6) //If A3_Ans1 is pressed this function will
        start.
        {
            mStoryIndex = 8; //Sets the TextView to A4 End.
            End_Of_story_win(); //Runs the corresponding function, which will
            then end the application.
        }
        else if (mStoryIndex == 3) //If the user selects A2_Ans1
        {
            mStoryIndex = 8; //Sets the TextView to A4_End
            End_Of_story_win(); //Runs the end function.
        }
    }
    if (userSelection == 2) //If the user has pressed the bottom button, it
    will run the following commands.
    {

```

```

        if (mStoryIndex == 0) //Checks if the TextView is currently displaying
A1_Story.
        {
            mStoryIndex = 2; //Sets the TextView to A2_Story.
            mTopButtonIndex = 4; //Sets the top button to A2_Ans1.
            mBottomButtonIndex = 5; //Sets the bottom button to A2_Ans2.
        }
        else if (mStoryIndex == 3) //If the user has selected the button
A2_Ans2.
        {
            mStoryIndex = 9; //Sets the TextView to A5_End.
            End_Of_story_loss(); //Ends the story with the loss function.
        }
        else if (mStoryIndex == 6) //If the user has selected the A3_Ans2
option.
        {
            mStoryIndex = 10; //Set the TextView to display A6_End.
            End_Of_story_loss(); //End the application with the use of the loss
function.
        }
    }
}

@Override
public void onSaveInstanceState(Bundle outState){
    outState.putInt("Story Index", mStoryIndex);
    super.onSaveInstanceState(outState);
}

private void End_Of_story_win(){ //Setting when the user chooses an answer that
is considered a "win"
    mTopButton.setVisibility(View.GONE); //Ensures both the buttons are removed
    mBottomButton.setVisibility(View.GONE);

    AlertDialog.Builder alert = new AlertDialog.Builder(this); //Creating a
dialogue box.
    alert.setTitle("You Won!"); //Sets the text of the dialogue box that
appears.
    alert.setCancelable(false); //Removes the ability to cancel/close the
dialogue box.
    alert.setPositiveButton("Close Application", new
DialogInterface.OnClickListener() { //Setting a button within the box to close the
application.
        @Override
        public void onClick(DialogInterface dialogInterface, int i) {
            finish(); //This function will close the application
        } //This onClick function will close the application once the option
has been pressed
    });
    alert.show(); //Displays the alert.
}

private void End_Of_story_loss(){ //Setting for when the user chooses an option
that is considered a "loss"
    mTopButton.setVisibility(View.GONE);
    mBottomButton.setVisibility(View.GONE);

    AlertDialog.Builder alert = new AlertDialog.Builder(this);
    alert.setTitle("You Lost!");
    alert.setCancelable(false);
    alert.setPositiveButton("Close Application", new
DialogInterface.OnClickListener() {
        @Override
        public void onClick(DialogInterface dialogInterface, int i) {
            finish();
        }
    });
}

```

```

        alert.show();
    }

    private void End_Of_story(){ //Setting for when the Story index is equal to 0
        mTopButton.setVisibility(View.GONE);
        mBottomButton.setVisibility(View.GONE);

        AlertDialog.Builder alert = new AlertDialog.Builder(this);
        alert.setTitle("Game Over");
        alert.setCancelable(false);
        alert.setPositiveButton("Close Application", new
        DialogInterface.OnClickListener() {
            @Override
            public void onClick(DialogInterface dialogInterface, int i) {
                finish();
            }
        });
        alert.show();}

    private void UpdateStory() //Creating a new variable to update the story.
    {
        mStoryIndex = (mStoryIndex + 1) % mStoryBank.length; //Increments the story
        index by 1 and then takes the remainder($).

        if (mStoryIndex == 0){ //This ensures that if the result of the calculation
        above is 0
            End_Of_story(); //The application will close.
        }

        mStory = mStoryBank[mStoryIndex].getStoryID(); //mStory becomes equal to
        the current Storyindex.
        mTextView.setText(mStory); //Sets the text of the TextView to the current
        story array position.
        mStory = mStoryBank[mTopButtonIndex].getStoryID(); //mStory becomes equal
        to the top button index.
        mTopButton.setText(mStory); //Sets the text of the top button to the
        current top button index.
        mStory = mStoryBank[mBottomButtonIndex].getStoryID(); //mStory becomes
        equal to the bottom button index.
        mBottomButton.setText(mStory); //Sets the text of the bottom button index.

    }
}

```

clsStoryProgress.java

```
package com.example.ashleya.textgame;

/**
 * Created by AshleyA on 23/10/2018.
 */

public class clsStoryProgress {

    private int mStoryID; //Instantiating the StoryID integer.
    private int mAnswer; //Instantiating the Answer integer.

    public clsStoryProgress(int StoryResourceID, int intAnswer){ //Publicly
accessible function for each of the integer variables.
        mStoryID = StoryResourceID; //StoryID becomes StoryResourceID.
        mAnswer = intAnswer; //Answer variable becomes equal to intAnswer.
    }

    public int getStoryID() {return mStoryID;}

    public void setmStoryID(int StoryID) {mStoryID = StoryID;}

    public int isAnswer() {return mAnswer;} //Used to assign index numbers to each
"story" and "answer"

    public void setmAnswer(int answer) {mAnswer = answer;}

}
```