

Mixed Emotions: Facial Emotion Recognition for People with Autistic Spectrum Condition

Author: Mark Channer

MSc Computer Science project report

Department of Computer Science and Information Systems
Birkbeck College, University of London

September 2016

This report is substantially the result of my own work, expressed in my own words, except where explicitly indicated in the text. I give permission for it to be submitted to the JISC Plagiarism Detection Service. The report may be freely copied and distributed provided the source is explicitly acknowledged

Contents

1. Introduction.....	1
2. Specification and Design	3
2.1 Development Methods	3
2.2 Application Architecture.....	3
2.3 Version 1.0 Class Design (Java)	4
2.3.1 GameBoard	6
2.3.2 Selections	7
2.3.3 GamePiece	7
2.3.4 BoardPopulator and MatchFinder	8
2.3.5 GameModel.....	8
2.3.6 GameController and GameBoardView	9
2.3.7 Complete Design for Version 1.0	10
2.4 Version 2.0 Class Design (Android)	10
2.4.1 Game Loop Design Pattern	10
2.4.2 Refactoring GamePiece	11
2.5 Version 3.0 Class Design (Android)	12
2.5.1 Refactoring BoardPopulator	12
2.5.2 Complete Design for Version 3.0	13
2.6 Algorithm Design.....	13
2.6.1 Potential Scenarios.....	13
2.6.2 Locating Matches	16
2.6.3 Removing Matches	17
2.6.4 Updating the Board	18
3. Software Implementation.....	19
3.1 The Android Activity Lifecycle.....	19
3.2 Game View Implementation	20

3.2.1 Screen Dimensions and Scaling.....	20
3.2.2 Rendering to the Screen	21
3.2.3 Announcing the Matched Emotion with Audio	22
3.2.4 Handling Screen Touch Selections	23
3.2.5 Management of Rendering with a Secondary Thread.....	24
3.3 Game Logic Implementation	26
3.3.1 Level-Specific Emoticon Creation.....	26
3.3.2 Populating the GameBoard	27
3.3.3 Finding Matches.....	28
3.3.4 Updating the GameBoard 2d Array	28
3.3.5 Putting it all Together	29
3.4 User Interface.....	30
4. Testing, Analysis and Evaluation	31
4.1 Testing Framework	31
4.2 Testing Strategy	32
4.3 Unit Testing	32
4.3.1 Local Testing	32
4.3.2 Instrumental Testing	34
4.4 User Testing	34
4.4.1 Round 1	34
4.4.2 Round 2.....	35
5. Project Evaluation.....	36
5.1 Limitations and Contributions	36
5.2 Lessons Learned.....	36
6. Conclusions and Future Work	38
6.1 Summary	38
6.2 Future Work.....	38

Acknowledgements

I would like to thank my supervisor Sergio Gutierrez-Santos for guidance, Esha Massand for initial research recommendations, Nicole Channer for help with the coordination of participants for user testing, and the parents of the children who kindly agreed to volunteer.

Abstract

This report documents the software development process for an Android application that is designed to assist children with Autism Spectrum Condition to improve their facial emotion recognition skills. It is not intended that the application be used as a replacement for software currently in use by specialist early intervention centres, but to provide an enjoyable accompaniment that, through its association with fun, may increase interest in human facial expressions and social interaction.

Project Supervisor: Sergio Gutierrez-Santos

1. Introduction

People with Autism Spectrum Condition (ASC) are known to have difficulties with facial emotion recognition. It is thought that an improvement in facial emotion recognition skills could encourage people with ASC to increase their level of social interaction and thereby improve their quality of life.

A variety of software has been developed to help people with ASC to practice recognizing emotions in a predictable environment that is free of social demands. However, due to relatively high drop-out rates during user testing of some such software, it is thought that the currently available software may not be sufficiently enjoyable for the target user. In addition, there is an abundance of applications that simply involve labelling a displayed emotion without sufficient motivation for continued engagement.

The goal of this project is to deliver a working application (henceforth referred to as *Mixed Emotions*) that helps individuals with ASC to improve their facial emotion recognition skills. *Mixed Emotions* is of the *Matching Tile game* sub-genre. For each level of the game, the user swaps face images on a grid that features one of five different expressions. If a swap leads to three or more consecutive faces that express the same emotion, that particular emotion is announced, points are given, and the grid is manipulated accordingly after the removal of the matches.

It must be emphasized that it is not intended that *Mixed Emotions* be used as a substitute for software that is currently available in specialist early intervention centres, but rather to be an enjoyable accompaniment; by encountering emotions in a fun yet predictable environment, it is hoped that the application can encourage a more positive association with human facial expressions and social interaction in general.

Some additional aims of the project are to improve object oriented design skills and learn new technologies: Both Android and JavaScript were unfamiliar prior to the commencement of the project.

Although the majority of the project proceeded in accordance with the plan given in the project proposal, there are two significant changes to note: As the amount of data to be saved in *Mixed Emotions* is relatively small, the Android `SharedPreferences` framework was used in place of an SQLite database. For an application such as *Mixed Emotions* that does not require a large amount of data to be saved, the `SharedPreferences` key/value store simplifies the storing and reading of data.

Regarding user testing, the temporary closure of the Puzzle Centre over the summer months meant that it was not possible to carry out user testing there. However, due to the fact that a family member who is on the autistic spectrum attended the Puzzle centre up until last year, it was still possible to assemble a sufficient number of volunteers for user testing through personal connections.

2. Specification and Design

This section describes how object oriented principles were applied to the design of *Mixed Emotions* to create an application that is maintainable, flexible and reusable. As frequent iterations of the game were delivered throughout the project, the design of two working prototypes and the final version are presented individually. The overall aim is to show how the design of the application was refined and improved upon with each iteration. The design of the algorithms employed to perform operations on the game board are also outlined.

2.1 Development Methods

There is no single formal development methodology that was strictly adhered to, but it was decided that for a project with a relatively short deadline, and technology that was initially unfamiliar, frequent iterations together with user testing would provide the least risk and enable requirements changes to become apparent at an earlier stage of the development process. Iterative planning is something that can be applied to many different processes and is more of a best practice than a methodology. However, many Extreme Programming (XP) principles were adhered to, such as design simplicity, refactoring and a consistent coding style that is as self-documenting as possible.

2.2 Application Architecture

In order to decouple the game logic from its rendering in the Graphical User Interface (GUI), *Mixed Emotions* uses the Model-View-Controller (MVC) pattern. The logic is stored in a *model* package, while the GUI output is handled by objects within a *view* package. Communication between the model and view is orchestrated by a *controller* which acts as a bridge between the two. There are several advantages to this design: because the model is unaware of how it will be displayed in the GUI, the view's rendering of it can be changed without the need to update

any of the code in the model classes. Furthermore, because the responsibilities of the view are restricted, there is a greater opportunity for it to be reused in other applications.

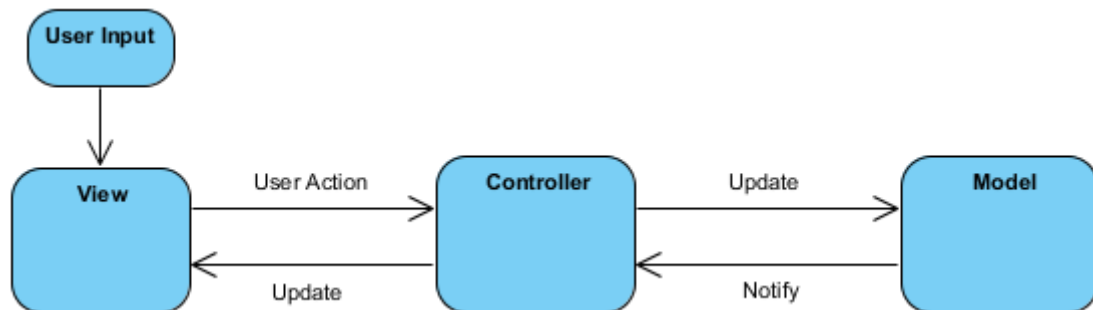


Fig 2.1: Flow of control between the MVC packages [1, p. 36]

2.3 Version 1.0 Class Design (Java)

An effort was made to produce a basic working prototype at an early stage in the development process. The first iteration of *Mixed Emotions* was therefore designed in Java to grasp the requirements for the game logic without the distraction of animation, sound, or the initially unfamiliar Android framework. Version 1.0 of *Mixed Emotions* is played from a simple command line interface with the game pieces (henceforth referred to as *emoticons*) represented as String values which are specified as follows:

AN = Angry face

EM = Embarrassed face

HA = Happy face

SA = Sad face

SU = Surprised face

Figure 2.2 below shows the command line interface of Version 1.0 when handling a swap of two emoticons that lead to a match. The game logic in this first iteration would later form the basis of the model layer in subsequent Android prototypes.

```

      0      1      2      3      4      5      6      7
0 | SU | HA | SU | HA | EM | SA | SU | SU |
1 | SU | SU | SA | EM | SA | AN | SA | SU |
2 | AN | HA | SA | EM | EM | AN | EM | AN |
3 | HA | SA | EM | SA | EM | SA | AN | SA |
4 | AN | SA | EM | SU | SA | SU | EM | HA |
5 | AN | HA | HA | AN | HA | HA | SA | SU |
6 | HA | HA | EM | AN | SA | EM | EM | SU |

Enter coordinates by column and row in the form 3,4:
Coordinates: 2,5
Selection 1: (HA)
Coordinates: 2,6
Selection 2: (EM)
Selections swapped. Board display before removal of matches:

      0      1      2      3      4      5      6      7
0 | SU | HA | SU | HA | EM | SA | SU | SU |
1 | SU | SU | SA | EM | SA | AN | SA | SU |
2 | AN | HA | SA | EM | EM | AN | EM | AN |
3 | HA | SA | EM | SA | EM | SA | AN | SA |
4 | AN | SA | EM | SU | SA | SU | EM | HA |
5 | AN | HA | EM | AN | HA | HA | SA | SU |
6 | HA | HA | HA | AN | SA | EM | EM | SU |

Vertical matches: EMBARRASSED (2,5) (2,4) (2,3)
Horizontal matches: HAPPY (0,6) (1,6) (2,6)
Matches removed. Board display after removal of matches:

      0      1      2      3      4      5      6      7
0 | HA | AN | SA | HA | EM | SA | SU | SU |
1 | SU | HA | EM | EM | SA | AN | SA | SU |
2 | SU | SU | SA | EM | EM | AN | EM | AN |
3 | AN | HA | SA | SA | EM | SA | AN | SA |
4 | HA | SA | SU | SU | SA | SU | EM | HA |
5 | AN | SA | SA | AN | HA | HA | SA | SU |
6 | AN | HA | SA | AN | SA | EM | EM | SU |

```

Fig 2.2: Version 1.0 command line output showing a swap that yields a vertical and a horizontal match

The major components and functionality for *Mixed Emotions* were first determined by writing a description of the game in natural language based on the following flow-of-events:

1. User selects two adjacent game pieces (emoticons) on the game board to swap:
 - 1.1. If swap results in three or more consecutive matching horizontal/vertical emoticons:
 - 1.1.1. Consecutive matching emoticons are highlighted
 - 1.1.2. Emoticon is announced via audio output
 - 1.1.3. Reward given
 - 1.1.4. Matching emoticons removed from the board, briefly leaving blank tiles
 - 1.1.5. Any emoticons above the blank tiles are shifted down to fill them
 - 1.1.6. Randomly generated emoticons fill the subsequent blank tiles at top of board
 - 1.1.6.1. If new board layout contains more matches, return to step 1.1.1
 - 1.1.7. Check a swap yielding consecutive matches is possible for next turn:

- 1.1.7.1. If not, populate the board again, but any earned points are kept
- 1.2. If swap does not yield a line of consecutive emoticons:
 - 1.2.1. Swapped emoticons are returned to previous position
2. End of swap

Nouns and verbs were highlighted, with nouns corresponding to potential objects and verbs to methods. With this approach, the following classes were identified and documented as UML class diagrams: **GameBoard**, **Selections**, and **GamePiece**. A **GameModel** was also added to coordinate the game.

2.3.1 GameBoard

The **GameBoard** class contains the general rules for its structure and interaction with the emoticons which it holds in a 2d array of abstract **GamePiece** elements (henceforth referred to as **GameBoard 2d array**). The array is encapsulated within the **GameBoard** class so that it can control access to the array elements and handle any illegal index values that may be entered. To enable reuse of the **GameBoard** class in other games, it was given a constructor that takes two integer arguments for specifying the size of its 2d array, in addition to a default constructor. If no arguments are passed to the **GameBoard** constructor, the array will be initialized to a size of 8 x 7. In order to minimize side effects, the only mutator provided is a *setGamePiece* method for adding emoticons to the array.

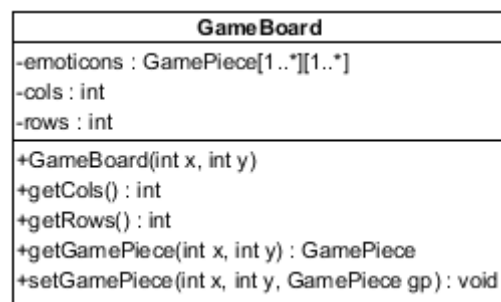


Fig 2.3 Version 1.0 GameBoard, which holds a 2d array of GamePiece elements

2.3.2 Selections

The **Selections** class stores the user selections and also contains methods that check whether the selections are legal according to the rules of the game. As a single swap requires the selection of two emoticons, the **Selections** class holds two arrays which each store the column and row value of the selected emoticon's location within the **GameBoard** 2d array.

Selections
-selection01 : int[2] -selection02 : int[2]
+getSelection01() : int [2] +setSelection01(int x, int y) : void +getSelection02() : int [2] +setSelection02(int x, int y) : void +selection01Made() : boolean +sameSelectionMadeTwice() : boolean +selectionsNotAdjacent() : boolean +secondSelectionToFirstSelection() : void +resetSelections() : void

Fig 2.4: Version 1.0 Selections class

2.3.3 GamePiece

In Version 1.0, an emoticon needs to know the type of emotion that it represents and the coordinates of its position within the **GameBoard** 2d array. The coordinate values are used to output the location of matches in the command line, but in subsequent game versions they also specify where they should be rendered in the GUI. In order to avoid code repetition, as embodied by the DRY Principle, state and behaviour that is common to all emoticons is stored in an abstract **GamePiece** parent class, which is implemented by five emoticon subclasses that each represent a different emotion (Fig 2.5). In hindsight, the decision to create a separate subclass for each emoticon was a somewhat naïve design choice as the differences between each of the subclasses is minimal. This design was improved upon in subsequent versions, an outline of which is given in the description of the Version 2.0 prototype.

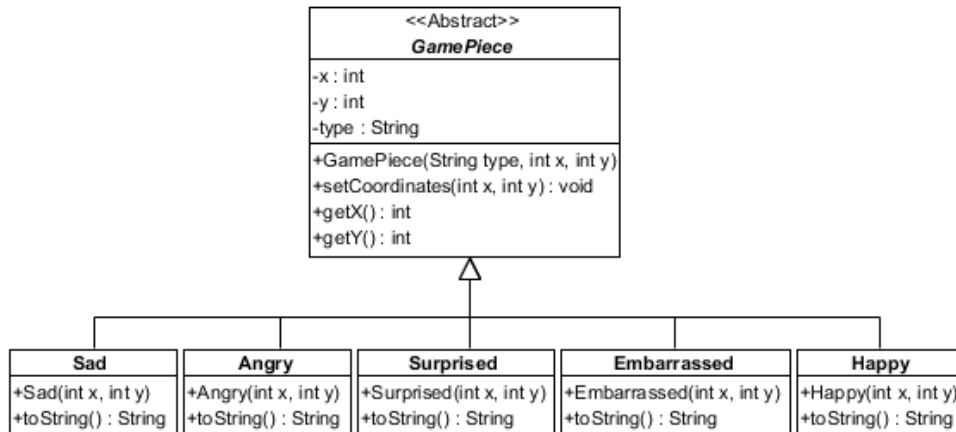


Fig 2.5: Version 1.0 abstract GamePiece class with five subclasses

2.3.4 BoardPopulator and MatchFinder

Returning briefly to the written description of the game, an examination of highlighted verbs yielded the following method names: `select`, `swap`, `giveReward`, `populateBoard`, `getRandomGamePiece`, `findVerticalMatches` and `findHorizontalMatches`. To break the game down further into self-contained components that have a single, well-defined responsibility, two additional classes were incorporated into the design: `BoardPopulator` and `MatchFinder`. For Version 1.0, the `MatchFinder` class contains methods `findVerticalMatches` and `findHorizontalMatches`, and the `BoardPopulator` class has two methods: `populateBoard` and `getRandomGamePiece`. Note here that the `BoardPopulator` class is violating the Single Responsibility Principle as it is both populating the `GameBoard` 2d array and generating the `Emoticon` objects. This design was improved upon in Version 2.0.

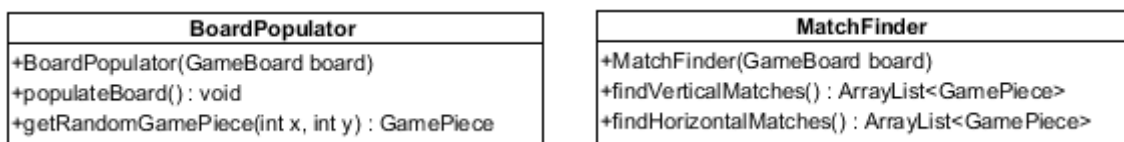


Fig 2.6: Version 1.0 BoardPopulator and MatchFinder classes

2.3.5 GameModel

The business logic of *Mixed Emotions* is coordinated by a `GameModel` class. `GameModel` contains references to a *controller* and all of the classes outlined in this section so far. It calls the relevant methods on these classes to coordinate the game. In Version 1.0, `GameModel` also

houses the `select`, `swap` and `giveReward` methods. However, these methods are abstracted out into other classes in subsequent versions of the game to promote a high degree of cohesion and enable such functionality to be reused elsewhere if necessary.

GameModel
-controller : GameController
-board : GameBoard
-matchFinder : MatchFinder
-populator : BoardPopulator
-selections : Selections
+handleSelection(int x, int y) : void
+swap(Selections selections) : void
+remove(List matches) : void
+lowerGamePieces() : void

Fig 2.7: Version 1.0 GameModel class

Rather than giving the `GameModel` responsibility for instantiation of the above classes, they are passed in as a constructor injection which reduces the degree of coupling. To further conform to good object oriented design principles, all of the described classes are coded to an interface rather than an implementation. This design allows different implementations to be injected into the `GameModel` without the need for any modification to it.

2.3.6 GameController and GameBoardView

The `GameController` and `GameBoardView` classes complete the design of the MVC architecture for Version 1.0. The `GameBoardView` is part of the view package and is coordinated by the `GameController` in the controller package. As Version 1.0 does not feature a GUI, the responsibilities of `GameBoardView` are to simply output a primitive representation of the `GameBoard` 2d array in the command line.

GameController	GameBoardView
-gameModel : GameModel	-board : GameBoard
-gameBoardView : GameBoardView	+GameBoardView(GameBoard gameBoard)
+displayBoard() : void	+drawBoard() : void
+displayText(String str) : void	+printText(String str) : void

Fig 2.8: Version 1.0 GameController and GameBoardView

2.3.7 Complete Design for Version 1.0

This concludes the design for Version 1.0. A UML class diagram of the complete design for the model is given below in Figure 2.9.

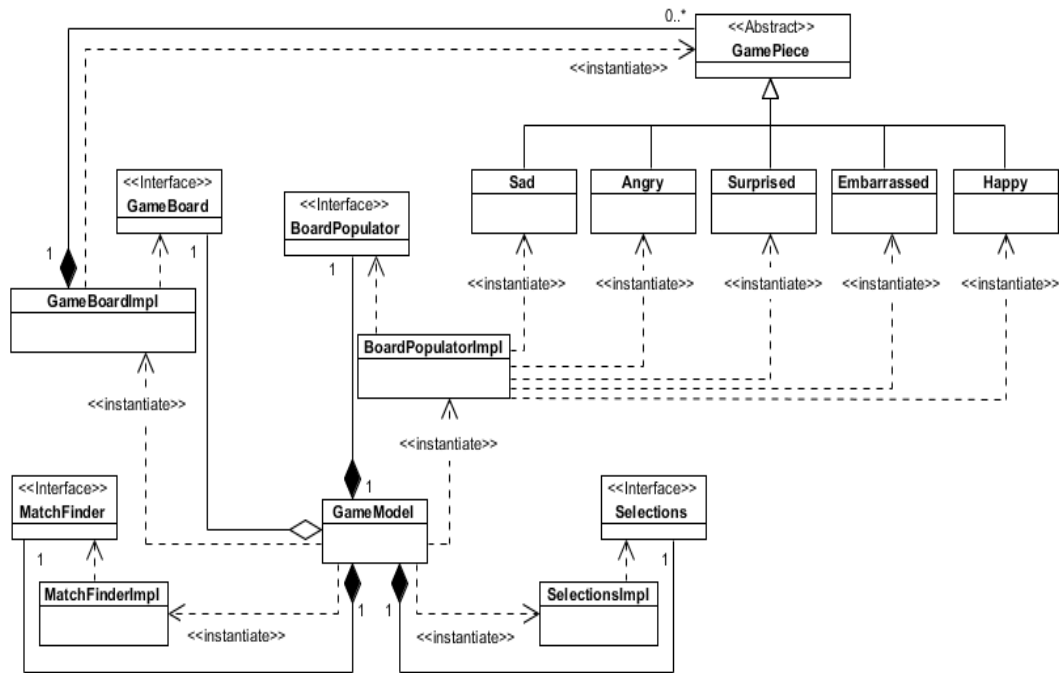


Fig 2.9: Complete Design of Model package for Version 1.0

2.4 Version 2.0 Class Design (Android)

Version 2.0 was the first Android prototype of *Mixed Emotions*. Much of the design focuses on providing a GUI representation of the game that animates states changes to the business logic. Because an MVC architecture was used, much of the code from Version 1.0 could be reused without major changes being necessary.

2.4.1 Game Loop Design Pattern

The business logic of *Mixed Emotions* is represented in the GUI by a `GameBoardView` class. A common game design pattern is used to achieve this, which is known simply as the game

loop pattern [2]. With each iteration through the loop, the game state is updated and then rendered to the GUI (Fig 2.10).

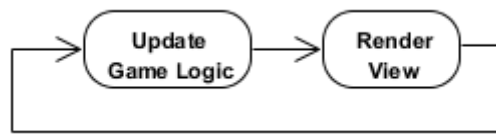


Fig 2.10: The Game Loop Pattern

2.4.2 Refactoring GamePiece

In order for the emoticons to have a graphical representation of themselves in the GUI, each was given a reference to a **Bitmap** object of a face expressing the emotion being represented by the emoticon. In Version 1.0, each emoticon was given x and y fields to represent its location in the **GameBoard** 2d array. However, in Version 2.0, these field values are used in combination with the bitmap dimensions to calculate where in the GUI each emoticon bitmap should be rendered.

An improvement was made to the design of the **GamePiece** and its five emoticon subclasses: Rather than having a separate emoticon subclass to represent a potentially ever-expanding set of emotions, the parts of the emoticon that vary are encapsulated and passed into it via constructor injection. This modification enables one **Emoticon** class to be used for the creation of all emoticons. An additional **GamePiece** subclass was also added to the design: **BlankTile**. This class takes a translucent **Bitmap** that enables tiles to appear briefly empty until an emoticon is housed within it.

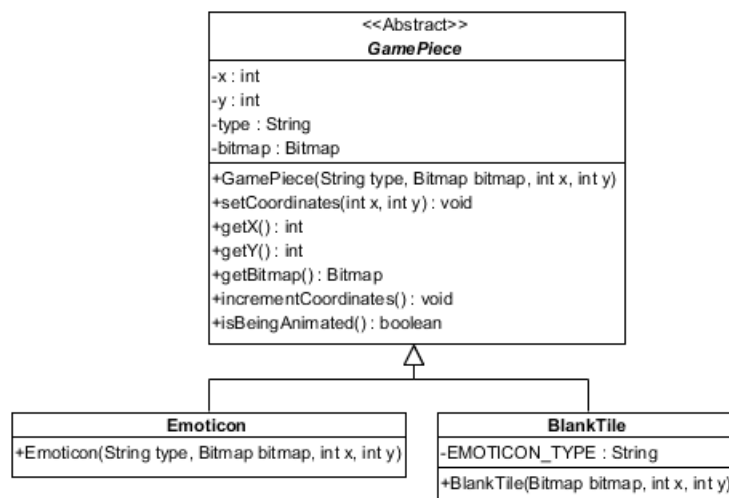


Fig 2.11: Version 2.0 **GamePiece** parent class with **Emoticon** and **BlankTile** child classes

2.5 Version 3.0 Class Design (Android)

The first round of user testing highlighted the fact that there was a paucity of emotions available in Version 2.0 of the game so additional levels were subsequently introduced for Version 3.0. To accommodate the extra levels, several additional classes were added to the design: `LevelManager`, `GamePieceFactory`, and `ScoreBoard`. A `BoardManipulator` class was also introduced to perform manipulation operations on the `GameBoard` 2d array. A `BitmapCreator` was also added for optimization of `Bitmap` object creation.

2.5.1 Refactoring BoardPopulator

It has already been noted that, in the design of Version 1.0, the `BoardPopulator` class violates the Single Responsibility Principle by both generating `Emoticon` objects and positioning them in the `GameBoard` 2d array. To improve the design in Version 3.0, the `getRandomGamePiece` method was moved out into a new abstract `GamePieceFactory` class. For each level of *Mixed Emotions*, a different subclass implements `getRandomGamePiece` to handle the creation and return of emoticons featuring in that particular level of the game. In order for `BoardPopulator` to continue being able to populate the `GameBoard` 2d array, its `populateBoard` method now has a `GamePieceFactory` parameter that subclasses can be passed into for polymorphic generation of emoticons for the required game level. Instantiation of the correct `GamePieceFactory` subclass is handled by a `LevelManager` class.

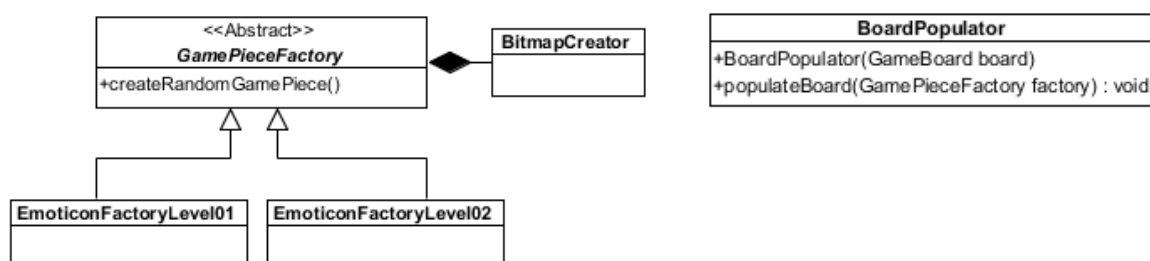


Fig 2.12: `GamePieceFactory` and subclasses (left) which are passed to the `populateBoard` method (right)

2.5.2 Complete Design for Version 3.0

Figure 2.13 below shows the final design of *Mixed Emotions* with the previously stated design modifications included.

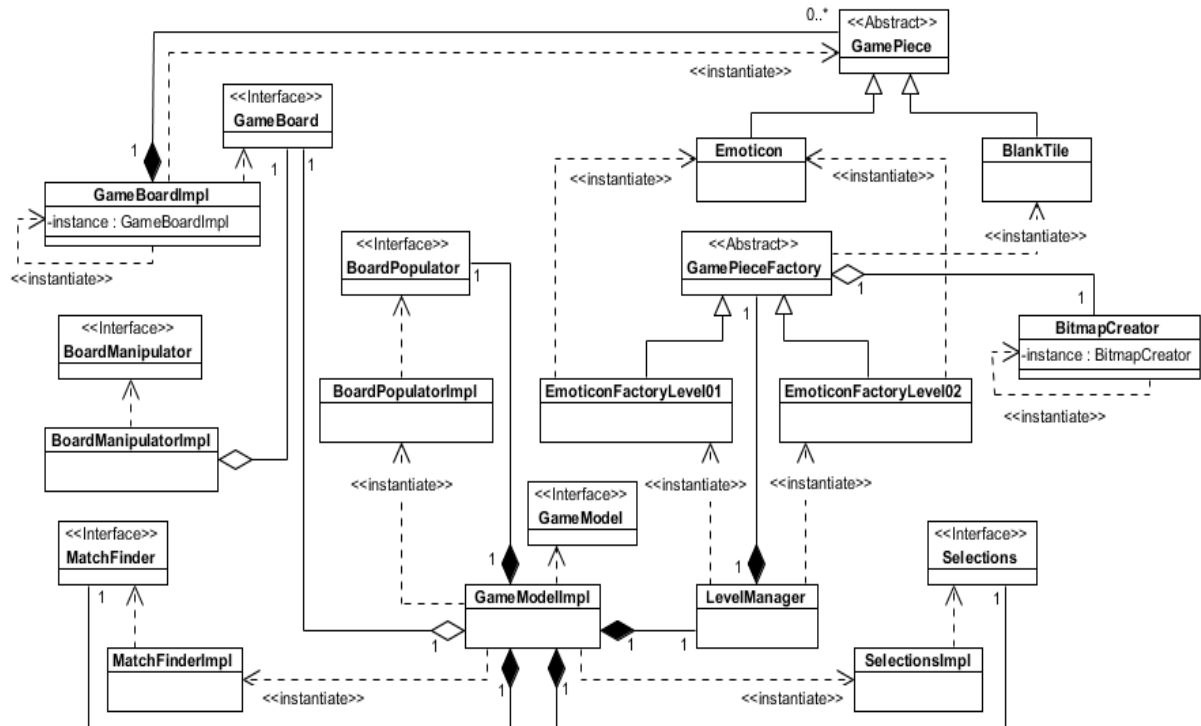


Fig 2.13: Complete Design of Model Package for Version 3.0

2.6 Algorithm Design

In order to provide a clear explanation of the algorithm requirements for *Mixed Emotions*, three examples are firstly given of less-common emoticon match formation possibilities that were considered.

2.6.1 Potential Scenarios

Once a legal swap has been made by the user, the application must be able to execute the following operations on the `GameBoard` 2d array:

1. Locate vertically or horizontally matching emoticon types of three or more in length
2. Remove matches
3. Update its state


The below key gives the initials of four emoticon types used in the following examples. The colours will highlight matches and indicate the state of particular emoticons involved in the manipulation of the GameBoard 2d array.

A = Angry emoticon


E = Embarrassed emoticon

H = Happy emoticon

S = Sad emoticon

 = Indicates matching emoticons







 = Indicates randomly generated, non-matching emoticons **R**







 = Indicates emoticons to be shifted down the board

Example 1 below gives a scenario where the user swaps emoticons at 7, 2 and 7, 3 which yields two vertical matches within the same column. This shows the necessity for the algorithm to be able to locate and return more than one match within a particular column. An additional point is that, as the two matches fill the column to the top of the board, there are no emoticons above the blank tiles to shift down and the removed matches must therefore be replaced entirely by randomly generated emoticons.

Example 1

	0	1	2	3	4	5	6	7
0	E	H	S	E	S	S	H	E
1	S	A	A	S	A	A	E	E
2	A	E	A	E	S	H	E	H
3	S	A	H	A	A	E	S	E
4	A	E	E	A	E	S	H	H
5	H	A	E	H	A	E	S	H
6	H	H	S	H	S	S	E	E

	0	1	2	3	4	5	6	7
0	E	H	S	E	S	S	H	 E
1	S	A	A	S	A	A	E	 E
2	A	E	A	E	S	H	E	 E
3	S	A	H	A	A	E	S	 H
4	A	E	E	A	E	S	H	 H
5	H	A	E	H	A	E	S	 H
6	H	H	S	H	S	S	E	E

	0	1	2	3	4	5	6	7
0	E	H	S	E	S	S	H	 R
1	S	A	A	S	A	A	E	 R
2	A	E	A	E	S	H	E	 R
3	S	A	H	A	A	E	S	 R
4	A	E	E	A	E	S	H	 R
5	H	A	E	H	A	E	S	 R
6	H	H	S	H	S	S	E	E

Example 2 shows the user's next swap which leads to a match of intersecting vertical and horizontal emoticons. Note that the emoticon located on tile 2, 3 forms part of both matches. In this scenario, if one of the matches is removed prior to the other being found, the absence of the emoticon at 2, 3 would lead to a loss of the other match. Additionally, it cannot be assumed that all emoticons above a match will always move down the board the same number of tiles; the emoticon at 2, 0 would drop down by three tiles, whereas the other two emoticons at 3, 2

and 4, 2 only have to drop down to the tile immediately below. Once the location of the emoticons that were present on the board have been updated, the vacant tiles must then be filled by randomly generated emoticons.

Example 2

	0	1	2	3	4	5	6	7
0	E	H	S	E	S	S	H	R
1	S	A	A	S	A	A	E	R
2	A	E	A	E	S	H	E	R
3	S	A	H	A	A	E	S	R
4	A	E	E	A	E	S	H	R
5	H	A	E	H	A	E	S	R
6	H	H	S	H	S	S	E	E

	0	1	2	3	4	5	6	7
0	E	H	S	E	S	S	H	R
1	S	A	A	S	A	A	E	R
2	A	E	A	E	S	H	E	R
3	S	H	A	A	A	E	S	R
4	A	E	E	A	E	S	H	R
5	H	A	E	H	A	E	S	R
6	H	H	S	H	S	S	E	E

	0	1	2	3	4	5	6	7
0	E	H	R	R	R	S	H	R
1	S	A	R	E	S	A	E	R
2	A	E	R	S	A	H	E	R
3	S	H	S	E	S	E	S	R
4	A	E	E	A	E	S	H	R
5	H	A	E	H	A	E	S	R
6	H	H	S	H	S	S	E	E

Example 3 gives a swap that yields two horizontal matches within the same row which, upon removal, leads to a subsequent match of *embarrassed* emoticons. This shows how an arbitrary number of matches can continue to occur after a single swap.

Example 3

	0	1	2	3	4	5	6	7
0	E	H	R	R	R	S	H	R
1	S	A	R	E	S	A	E	R
2	A	E	R	S	A	H	E	R
3	S	H	S	E	S	E	S	R
4	A	E	E	A	E	S	H	R
5	H	A	E	H	A	E	S	R
6	H	H	S	H	S	S	E	E

	0	1	2	3	4	5	6	7
0	E	H	R	R	R	S	H	R
1	S	A	R	E	S	A	E	R
2	A	E	R	S	A	H	E	R
3	S	H	S	E	S	E	S	R
4	A	E	E	A	E	S	H	R
5	H	A	E	H	A	E	S	R
6	H	H	H	S	S	S	E	E

	0	1	2	3	4	5	6	7
0	R	R	R	R	R	R	H	R
1	E	H	R	R	R	S	E	R
2	S	A	R	E	S	A	E	R
3	A	E	R	S	A	H	S	R
4	S	H	S	E	S	E	H	R
5	A	E	E	A	E	S	S	R
6	H	A	E	H	A	E	E	E

	0	1	2	3	4	5	6	7
0	R	R	R	R	R	R	R	R
1	E	H	R	R	R	R	H	R
2	S	A	R	E	S	S	E	R
3	A	E	R	S	A	A	E	R
4	S	H	S	E	S	H	S	R
5	A	E	E	A	E	E	H	R
6	H	A	E	H	A	S	S	R

2.6.2 Locating Matches

Having considered the less-common scenarios that must be handled after a swap, the design of the algorithm used in the `findVerticalMatches` method will now be described.

Fig 1 below shows a post-swap match present on the board at 0, 1 to 0, 3. The algorithm begins at 0, 6 and adds the *sad* emoticon to the List (Fig 2). It then moves up one tile to 0, 5 (Fig 3) and compares the *sad* emoticon on the current tile with the emoticon most recently added to the list. They are both of the same type so the emoticon at 0, 5 is added to the list.

	0	1	2	3	4	5	6	7
0	E	H	S	A	E	E	H	A
1	H	A	E	H	A	E	E	H
2	H	E	A	S	H	S	H	A
3	H	H	S	E	A	E	S	H
4	E	E	A	S	E	H	E	S
5	S	E	E	A	S	E	S	A
6	S	H	A	H	A	S	A	S

Fig 1

	0	1	2	3	4	5	6	7
0	E	H	S	A	E	E	H	A
1	H	A	E	H	A	E	E	H
2	H	E	A	S	H	S	H	A
3	H	H	S	E	A	E	S	H
4	E	E	A	S	E	H	E	S
5	S	E	E	A	S	E	S	A
6	S	H	A	H	A	S	A	S

Fig 2

	0	1	2	3	4	5	6	7
0	E	H	S	A	E	E	H	A
1	H	A	E	H	A	E	E	H
2	H	E	A	S	H	S	H	A
3	H	H	S	E	A	E	S	H
4	E	E	A	S	E	H	E	S
5	S	E	E	A	S	E	S	A
6	S	H	A	H	A	S	A	S

Fig 3

We then move one tile up to 0, 4 and compare the *embarrassed* emoticon on the current tile with the *sad* emoticon most recently added to list (Fig 4). The emoticons are not of the same type which indicates this is the end of any match that may be present. The contents of the list are therefore checked (without adding the *embarrassed* emoticon). If the size of the list is 3 or greater, it is a match and the list is added to a list of matching lists. In this case, the size of the list is only 2, so the list is not added. Regardless of whether the size of the list is 3 or more, a new list is created and the *embarrassed* emoticon that was found at 0, 4 is now added to the list. We proceed one tile up to 0, 3 and compare the *happy* emoticon on the current tile with the *embarrassed* emoticon most recently added to list (Fig 5). As the emoticon types differ, the same process as before is repeated and we now have a list containing one *happy* emoticon. Proceeding upwards to 0, 2. We compare the *happy* emoticon on the current tile with the emoticon most recently added to list (Fig 6). They are both of the *happy* variety, so the emoticon located at 0, 2 is added to the list.

	0	1	2	3	4	5	6	7
0	E	H	S	A	E	E	H	A
1	H	A	E	H	A	E	E	H
2	H	E	A	S	H	S	H	A
3	H	H	S	E	A	E	S	H
4	E	E	A	S	E	H	E	S
5	S	E	E	A	S	E	S	A
6	S	H	A	H	A	S	A	S

Fig 4

	0	1	2	3	4	5	6	7
0	E	H	S	A	E	E	H	A
1	H	A	E	H	A	E	E	H
2	H	E	A	S	H	S	H	A
3	H	H	S	E	A	E	S	H
4	E	E	A	S	E	H	E	S
5	S	E	E	A	S	E	S	A
6	S	H	A	H	A	S	A	S

Fig 5

	0	1	2	3	4	5	6	7
0	E	H	S	A	E	E	H	A
1	H	A	E	H	A	E	E	H
2	H	E	A	S	H	S	H	A
3	H	H	S	E	A	E	S	H
4	E	E	A	S	E	H	E	S
5	S	E	E	A	S	E	S	A
6	S	H	A	H	A	S	A	S

Fig 6

Moving up to 0, 1 we compare the *happy* emoticon on the current tile with the emoticon most recently added to list (Fig 7). They are both of the *happy* variety, so that emoticon is also added to the list. Moving one tile up the column to 0, 0. We again compare the *embarrassed* emoticon on the current tile with the *happy* emoticon most recently added to list (Fig 8). As they differ, the list is checked without adding the *embarrassed* emoticon to it. The lists contains three *happy* emoticons and therefore passes the criteria for a match and is added to the list of matching lists. The benefit of this algorithm design is that it ensures matching emoticons of any number equal to or above three are correctly handled and is not limited to matches of exactly three emoticons.

	0	1	2	3	4	5	6	7
0	E	H	S	A	E	E	H	A
1	H	A	E	H	A	E	E	H
2	H	E	A	S	H	S	H	A
3	H	H	S	E	A	E	S	H
4	E	E	A	S	E	H	E	S
5	S	E	E	A	S	E	S	A
6	S	H	A	H	A	S	A	S

Fig 7

	0	1	2	3	4	5	6	7
0	E	H	S	A	E	E	H	A
1	H	A	E	H	A	E	E	H
2	H	E	A	S	H	S	H	A
3	H	H	S	E	A	E	S	H
4	E	E	A	S	E	H	E	S
5	S	E	E	A	S	E	S	A
6	S	H	A	H	A	S	A	S

Fig 8

	0	1	2	3	4	5	6	7
0	E	H	S	A	E	E	H	A
1	H	A	E	H	A	E	E	H
2	H	E	A	S	H	S	H	A
3	H	H	S	E	A	E	S	H
4	E	E	A	S	E	H	E	S
5	S	E	E	A	S	E	S	A
6	S	H	A	H	A	S	A	S

Fig 9

2.6.3 Removing Matches

This algorithm is comparatively trivial and simply involves taking the coordinates of the emoticons contained in the list of matches and replacing those coordinates on the GameBoard 2d array with a BlankTile object.

2.6.4 Updating the Board

We start at 0, 6 and proceed up the column, checking for blank tiles, the first of which is found at 0, 3 (Fig 2). It marks the location of the blank tile and proceeds up the column until either finding a tile that contains an emoticon or until the top of the column is reached. An *embarrassed* emoticon is located at 0, 0 (Fig 3).

	0	1	2	3	4	5	6	7
0	E	H	S	A	E	E	H	A
1		A	E	H	A	E	E	H
2		E	A	S	H	S	H	A
3		H	S	E	A	E	S	H
4	E	E	A	S	E	H	E	S
5	S	E	E	A	S	E	S	A
6	S	H	A	H	A	S	A	S

Fig 1

	0	1	2	3	4	5	6	7
0	E	H	S	A	E	E	H	A
1		A	E	H	A	E	E	H
2		E	A	S	H	S	H	A
3	?	H	S	E	A	E	S	H
4	E	E	A	S	E	H	E	S
5	S	E	E	A	S	E	S	A
6	S	H	A	H	A	S	A	S

Fig 2

	0	1	2	3	4	5	6	7
0	E	H	S	A	E	E	H	A
1		A	E	H	A	E	E	H
2		E	A	S	H	S	H	A
3	?	H	S	E	A	E	S	H
4	E	E	A	S	E	H	E	S
5	S	E	E	A	S	E	S	A
6	S	H	A	H	A	S	A	S

Fig 3

The blank tile on 0, 3 is replaced by the *embarrassed* emoticon that was located at 0, 0, and the tile at 0, 0 is set as a blank tile (Fig 4). We then continue up the board checking for blank tiles, one of which is found on the next tile at 0, 2 (Fig 5). As before, the location of the blank tile is marked and we continue up the column. The top of the column is reached without locating any emoticons. This indicates that there are no more emoticons in the present column to shift down so a random emoticon is created to replace the blank tile at 0, 2 (Fig 6). This algorithm continues until the remainder of the column is populated. This algorithm is repeated until the entire GameBoard 2d array has been updated.

	0	1	2	3	4	5	6	7
0		H	S	A	E	E	H	A
1		A	E	H	A	E	E	H
2		E	A	S	H	S	H	A
3	E	H	S	E	A	E	S	H
4	E	E	A	S	E	H	E	S
5	S	E	E	A	S	E	S	A
6	S	H	A	H	A	S	A	S

Fig 4

	0	1	2	3	4	5	6	7
0		H	S	A	E	E	H	A
1		A	E	H	A	E	E	H
2	?	E	A	S	H	S	H	A
3	E	H	S	E	A	E	S	H
4	E	E	A	S	E	H	E	S
5	S	E	E	A	S	E	S	A
6	S	H	A	H	A	S	A	S

Fig 5

	0	1	2	3	4	5	6	7
0		H	S	A	E	E	H	A
1		A	E	H	A	E	E	H
2	R	E	A	S	H	S	H	A
3	E	H	S	E	A	E	S	H
4	E	E	A	S	E	H	E	S
5	S	E	E	A	S	E	S	A
6	S	H	A	H	A	S	A	S

Fig 6

3. Software Implementation

This section describes the implementation of *Mixed Emotions*. A brief introduction to the Android Activity Lifecycle is given, followed by discussion of the chosen approach to animating the game. The use of multithreading to coordinate the business logic with a secondary thread is also described, followed by an outline of significant interfaces and classes that were implemented.

3.1 The Android Activity Lifecycle

The user interface of an Android application is represented by an **Activity**. An **Activity** can be in different states which are collectively called the Activity Lifecycle. The three states of relevance to this explanation are *running*, *paused*, and *stopped*. Implementation of an **Activity** is achieved by extending the Android API's **Activity** class. This class has several methods that are called by the Android system when the **Activity** changes state, and can be overridden to specify the operations that should be carried out upon a change in state. These include `onCreate`, `onResume` and `onPause`, which can each be found in Figure 3.1 [3].

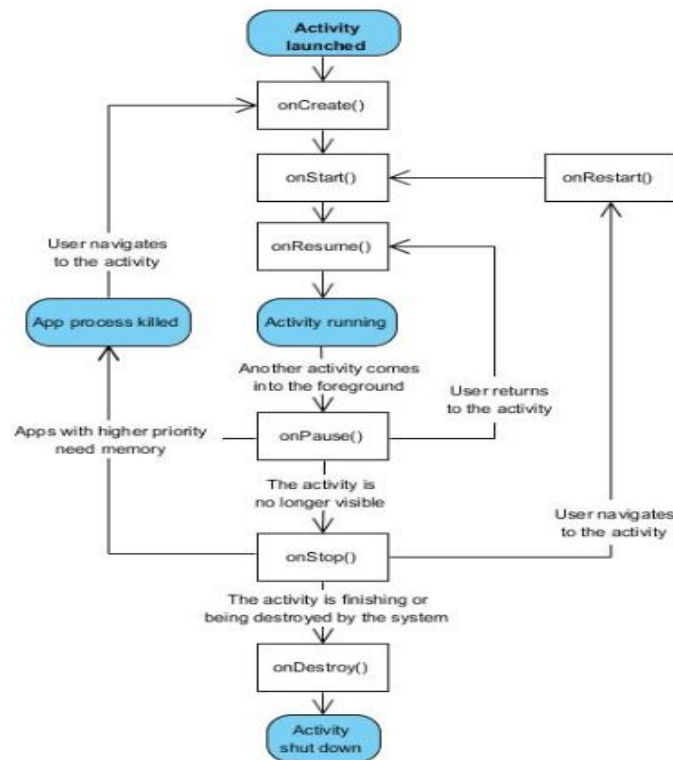


Fig 3.1: The Android Activity Lifecycle [4]

The model and view modules of *Mixed Emotions* are instantiated within the `onCreate` method as this is where an `Activity` sets up its initial state and leaves the most time for instantiation. The main game loop is called from the `onResume` method. This ensures that the game resumes when a user returns to it from another `Activity`. The game is paused when the `onPause` method is called.

3.2 Game View Implementation

3.2.1 Screen Dimensions and Scaling

The `GameBoard` 2d array is represented in the GUI as a grid of tiles that each contains an emoticon, as can be seen below in Figure 3.2.

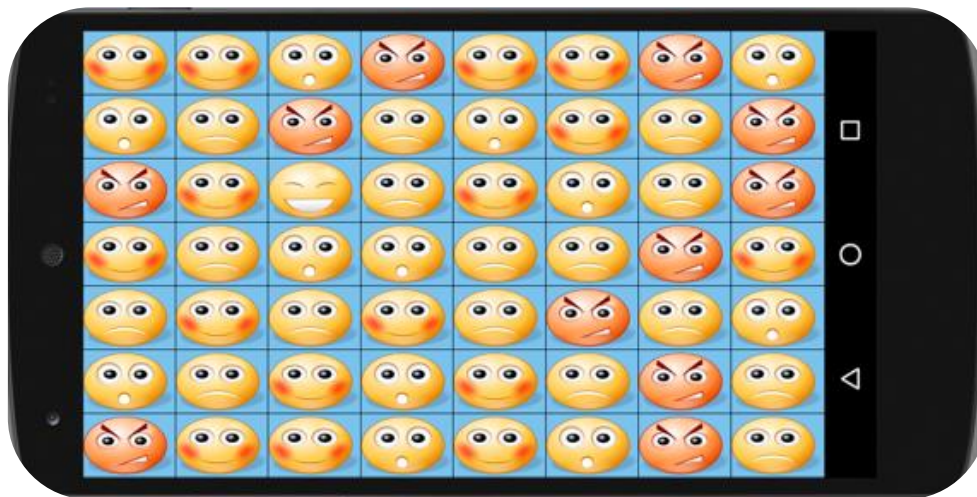


Fig 3.2: Version 2.0 Implementation of Mixed Emotions

Ensuring the even distribution of tiles and emoticons over the entire screen was initially problematic. Due to the differences in screen size and density across various Android devices, the use of hard-coded pixel values to position the graphics can cause unpredictable changes to the layouts and had to be avoided [5]. Instead, a layout that can adapt to different devices was implemented with the below calculation:

$$\text{tile width} = \text{screen width} / \text{array columns}$$

$$\text{tile height} = \text{screen height} / \text{array rows}$$

The code below shows how the tile dimensions were then passed into the `GameBoardView` used to set the GUI layout:

```
Display display = getWindowManager().getDefaultDisplay();
Point screenSize = new Point();
display.getSize(screenSize);

tileWidth = screenSize.x / MAX_COLUMNS;
tileHeight = screenSize.y / MAX_ROWS;
boardView = new GameBoardView(this, screenSize.x, screenSize.y, tileWidth, tileHeight);
setContentView(boardView);
```

The tile dimensions are also passed into a `BitmapCreator` object, which was implemented for the creation and scaling of the emoticon bitmaps. The `BitmapCreator` utilizes the Android `BitmapFactory` to scale the bitmaps, then it stores them as attributes for returning to the `GamePieceFactory` subclasses upon request via *getters*. This ensures that the expensive procedure of creating and scaling emoticon bitmaps is not repeated throughout the game. As an extra measure, the `BitmapCreator` is implemented with the singleton design pattern to ensure that there is never more than one instantiation of a `BitmapCreator` object at a time.

3.2.2 Rendering to the Screen

The `GameBoardView` of *Mixed Emotions* extends the Android system's `View` class. A `View` is an area on the screen that has responsibility for drawing and the handling of events. Bitmaps can either be rendered into a `View` object or to a `Canvas`. The `GameBoardView` renders to a `Canvas` because the Android Developer guide recommends this for applications that must continuously redraw themselves [6]. The x and y coordinate values of a `Canvas` begin in the top-left corner at 0, 0 and increase across the screen and downwards as shown below.

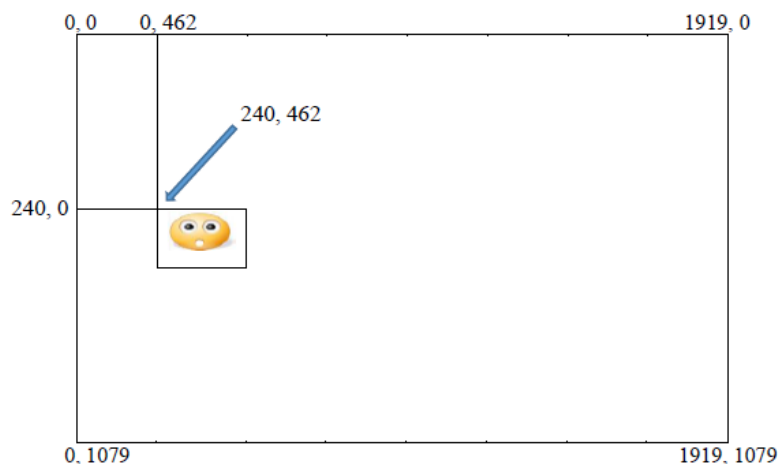


Fig 3.3: Coordinate system for a screen resolution of 1920 by 1080 pixels

The `GameBoardView` calls a *getter* method on each emoticon to obtain the required bitmap and x, y field values for rendering. The bitmap is then drawn to the `Canvas` at a position that represents the emoticon's location within the `GameBoard` 2d array. When animating an emoticon's movement between tiles, such as for a swap, care must be taken to ensure that its final destination is always within the boundaries of a tile. The calculation to achieve this is given below:

Right horizontal movement = *x value* + (*bitmap width* × *number of tiles to move*)

Left horizontal movement = *x value* – (*bitmap width* × *number of tiles to move*)

Downwards vertical movement = *y value* + (*bitmap height* × *number of tiles to move*)

Upwards vertical movement = *y value* – (*bitmap height* × *number of tiles to move*)

In order to achieve the effect of animation, the emoticon x or y values are incremented gradually until they reach the newly calculated value. The emoticon `Bitmap` is rendered to the `Canvas` after each increment, achieving the effect of animation.

3.2.3 Announcing the Matched Emotion with Audio

The Android `SoundPool` class was used to extract sound files of the declared emotions. The sound files were recorded in the OGG format as recommended. One consideration was that more than one matching emotion is occasionally present. Announcing every emoticon would be tedious for the player, so if more than one matching type is found, the phrase “Mixed Emotions” is announced. The code to find the appropriate OGG file for a match is below.

```
public void announceMatchedEmoticons(MatchContainer matchContainer) {
    ArrayList<LinkedList<GamePiece>> matchingX = matchContainer.getMatchingX();
    ArrayList<LinkedList<GamePiece>> matchingY = matchContainer.getMatchingY();
    if (mixedEmotionsSameDirection(matchingX)
        || mixedEmotionsSameDirection(matchingY)
        || mixedEmotionsCrossDirection(matchingX, matchingY)) {
        playSound(MIXED_EMOTIONS);
    } else if (!matchingX.isEmpty()) {
        playSound(matchingX.get(0).getFirst().toString());
    } else if (!matchingY.isEmpty()) {
        playSound(matchingY.get(0).getFirst().toString());
    }
}
```

3.2.4 Handling Screen Touch Selections

A common frustration for young tablet users is when a finger or thumb inadvertently rests on the corner of the screen. As the digit is registered by the running application as part of a touch event it can cause the screen to become unresponsive [7]. To minimize such a problem, the decision was made to handle touch events in the `GameBoardView`, rather than in the `GameController`, so touch responses are restricted to the game board itself, rather than the entire screen of the device. The implementation of the `onTouchEvent` method can be seen below. The Android `MotionEvent` parameter is an Android class that contains the coordinates for the specific part of the screen that has been touched. It is therefore passed on as an argument when the `handleTouch` method in `GameController` is called.

```
@Override
public boolean onTouchEvent(MotionEvent event) {
    controller.handleTouch(event);
    return true;
}
```

The `handleTouch` method below retrieves the touched screen coordinates from the `MotionEvent` object and divides them by the tile dimensions. The result of this calculation gives the location of the selected emoticon in the `GameBoard` 2d array. The location is then passed to the `handleSelection` method in the `GameModel` for handling the business logic concerning the selection.

```
@Override
public void handleTouch(MotionEvent event) {
    int screenX = (int) event.getX();
    int screenY = (int) event.getY();
    if (event.getAction() == MotionEvent.ACTION_DOWN) {
        gameModel.handleSelection(screenX / tileWidth, screenY / tileHeight);
    }
}
```

Care was taken in the naming of the methods and parameters to make the code as readable as possible. It is hoped that the `handleSelection` method reads somewhat like a story.

```

@Override
public void handleSelection(int x, int y) {
    if (!selections.selection01Made()) {
        selections.setSelection01(x, y);
        gameBoard.highlightTile(x, y);
    } else {
        handleSecondSelection(x, y);
    }
}

private void handleSecondSelection(int x, int y) {
    selections.setSelection02(x, y);
    gameBoard.clearHighlights();
    if (selections.sameSelectionMadeTwice()) {
        selections.resetSelections();
    } else if (selections.notAdjacent()) {
        selections.secondSelectionToFirstSelection();
        gameBoard.highlightTile(x, y);
    } else {
        boardManipulator.swap(selections);
        checkForMatches(selections);
        selections.resetSelections();
    }
}

```

3.2.5 Management of Rendering with a Secondary Thread

In order to keep the `GameBoardView` free to handle user events, it launches a secondary thread to manage the main game loop which continuously updates the model code and renders its state to the Canvas. The `SurfaceView` class that is extended by `GameBoardView` provides a secondary thread with its own Canvas. `SurfaceView` also enables access to a `SurfaceHolder` interface which controls the drawing surface size and format. The implementation is shown below (Javadoc comments omitted for brevity).

```

public void resume() {
    running = true;
    gameViewThread = new Thread(this);
    gameViewThread.start();
}

@Override
public void run() {
    Canvas canvas;
    while (running) {
        if (surfaceHolder.getSurface().isValid()) {
            canvas = surfaceHolder.lockCanvas();
            controller.updateModel();
            drawGameBoard(canvas);
            surfaceHolder.unlockCanvasAndPost(canvas);
        }
    }
}

```

The business logic in the main thread and the animation managed by the secondary thread are synchronized with a lock object to ensure that the business logic waits for the animation to complete before proceeding. For example, when the user selects two emoticons, the `BoardManipulator` swaps their location within the `GameBoard` 2d array. With every call to `updateModel` from the secondary thread, the x, y pixel coordinates of the emoticons are incremented. The business logic must be interrupted until the animating emoticons have reached their destination on the grid. Once the emoticons have been animated to their new position in the GUI, the business logic is resumed. The `waitForSwapAnimationToComplete` method below is called by the `BoardManipulator` after it has swapped the 2d array elements. This is to prevent the board from being checked and pieces subsequently removed before the animation of the swap has finished.

```
private void waitForSwapAnimationToComplete() {
    synchronized (lock) {
        animatingSwap = true;
        while (animatingSwap) {
            try {
                lock.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

The `incrementSwapCoordinates` method is called via the controller with each iteration around the secondary thread's game loop. When the animation of all emoticons is complete, the business logic is resumed by releasing the lock on the object and calling `notifyAll`.

```
private void incrementSwapCoordinates() {
    boolean stillAnimating = false;
    for (int y = COLUMN_BOTTOM; y >= COLUMN_TOP; y--) {
        for (int x = ROW_START; x < COLUMNS; x++) {
            if (board.getGamePiece(x, y).isBeingAnimated()) {
                stillAnimating = true;
                board.getGamePiece(x, y).incrementCoordinates();
            }
        }
    }
    if (!stillAnimating) {
        synchronized (lock) {
            if (animatingSwap) {
                animatingSwap = false;
                lock.notifyAll();
            }
        }
    }
}
```

3.3 Game Logic Implementation

3.3.1 Level-Specific Emoticon Creation

The `createRandomGamePiece` method must only return `Emoticon` objects for the current level of the game being played. The factory method design pattern was used to ensure this by deferring the instantiation of each emoticon to the relevant `GamePieceFactory` subclass, of which there is one for each level. The advantage of this design is that new levels can easily be added to the game in the future just by creating a new subclass of `GamePieceFactory`, and no changes to the `GamePieceFactory` itself are necessary. In this way, the design satisfies the Open/Closed Principle by keeping the `GamePieceFactory` ‘open for extension but closed for modification’ [8].

```
public GamePiece createRandomGamePiece(int x, int y) {  
    return getRandomGamePiece(x, y);  
}  
  
protected abstract GamePiece getRandomGamePiece(int x, int y);
```

The code below shows the implementation of the `getRandomGamePiece` method for the `GamePieceFactory` subclass that is responsible for level 1 of the game.

```
protected GamePiece getRandomGamePiece(int x, int y) {  
    Random random = new Random();  
    String type = null;  
    Bitmap bitmap = null;  
    switch (random.nextInt(5)) {  
        case 0:  
            type = "ANGRY";  
            bitmap = bitmapCreator.getAngryBitmap();  
            break;  
        case 1:  
            type = "HAPPY";  
            bitmap = bitmapCreator.getHappyBitmap();  
            break;  
        case 2:  
            type = "EMBARRASSED";  
            bitmap = bitmapCreator.getEmbarrassedBitmap();  
            break;  
        case 3:  
            type = "SURPRISED";  
            bitmap = bitmapCreator.getSurprisedBitmap();  
            break;  
        case 4:  
            type = "SAD";  
            bitmap = bitmapCreator.getSadBitmap();  
            break;  
        default:  
            break;  
    }  
    return new Emoticon(type, bitmap, x, y);  
}
```


The `LevelManager` class is responsible for instantiating the relevant subclass of `GamePieceFactory`. This is indicated by the `level` parameter. The strategy design pattern is used here to enable `GamePieceFactory` subclasses to be created dynamically. This was implemented in the `LevelManager` `setGameLevel` method, which can be seen below.

```
@Override
public void setGameLevel(int level) {
    if (level == LEVEL_ONE) {
        factory = new EmoticonFactoryLevel01(tileWidth, tileHeight);
    } else if (level == LEVEL_TWO) {
        factory = new EmoticonFactoryLevel02(tileWidth, tileHeight);
    } else {
        throw new IndexOutOfBoundsException("Level out of bounds");
    }
}
```

3.3.2 Populating the GameBoard

As outlined in Section 2, the `BoardPopulator` class has a `populate` method that takes a `GamePieceFactory` argument which it uses to populate the `GameBoard` 2d array with emoticons of the required game level. The algorithm traverses the array, adding an `Emoticon` object to each tile. An additional consideration is avoiding the presence of matches when the 2d array is first populated. This is achieved with a call to a private method that checks the type of emoticon located on the neighbouring two tiles in either direction of the current tile to ensure that no matches would be caused if the just-created emoticon were to be placed within the 2d array. If it would cause a match, the emoticon is discarded and another emoticon is randomly created, then the tiles are checked for matches again.

```
@Override
public void populate(GamePieceFactory factory) {
    GamePiece emoticon;
    for (int x = ROW_START; x < COLUMNS; x++) {
        for (int y = COLUMN_TOP; y < ROWS; y++) {

            do {
                emoticon = factory.getRandomGamePiece(x, y);
            } while (gamePieceTypeCausesMatch(emoticon));

            gameBoard.setGamePiece(x, y, emoticon);
        }
    }
}
```


3.3.3 Finding Matches

There are two algorithms that search the board for matches. The `findVerticalMatches` implementation is given below. A `LinkedList` was used for the `matchingEmoticons` list, simply because it has a `getLast` method that `ArrayList` does not have.

```
private ArrayList<LinkedList<GamePiece>> findVerticalMatches() {
    LinkedList<GamePiece> matchingEmoticons = new LinkedList<>();
    ArrayList<LinkedList<GamePiece>> allVerticalMatches = new ArrayList<>();
    GamePiece emo;
    for (int x = ROW_START; x < COLUMNS; x++) {
        matchingEmoticons.add(gameBoard.getGamePiece(x, COLUMN_BOTTOM));

        for (int y = (COLUMN_BOTTOM - 1); y >= COLUMN_TOP; y--) {
            emo = gameBoard.getGamePiece(x, y);
            if (!emo.toString().equals(matchingEmoticons.getLast().toString())) {
                examineList(matchingEmoticons, allVerticalMatches);
                matchingEmoticons = new LinkedList<>();
            }
            matchingEmoticons.add(emo);
            if (y == COLUMN_TOP) {
                examineList(matchingEmoticons, allVerticalMatches);
                matchingEmoticons = new LinkedList<>();
            }
        }
    }
    return allVerticalMatches;
}
```

3.3.4 Updating the GameBoard 2d Array

Below is the implementation of the algorithm to update the `GameBoard` 2d array, as outlined in Section 2. The `waitForDropAnimationToComplete` method interrupts the business logic until the downwards animation of the emoticons is complete.

```

@Override
public void updateBoard(GamePieceFactory factory) {
    int runnerY;
    for (int x = ROW_START; x < COLUMNS; x++) {
        for (int y = COLUMN_BOTTOM; y >= COLUMN_TOP; y--) {
            if (gameBoard.getGamePiece(x, y).toString().equals(BLANK)) {
                runnerY = y;
                while ((runnerY >= COLUMN_TOP) &&
                    gameBoard.getGamePiece(x, runnerY).toString().equals(BLANK)) {
                    runnerY--;
                }
                if (runnerY >= COLUMN_TOP) {
                    int tempY = gameBoard.getGamePiece(x, y).getArrayY();
                    gameBoard.setGamePiece(x, y, gameBoard.getGamePiece(x, runnerY));
                    gameBoard.getGamePiece(x, y).setArrayY(tempY);
                    gameBoard.setGamePiece(x, runnerY, factory.createBlankTile(x, runnerY));
                } else {
                    gameBoard.setGamePiece(x, y, factory.getRandomGamePiece(x, y));
                }
            }
        }
    }
    waitForDropAnimationToComplete();
}

```

3.3.5 Putting it all Together

The implementation of the `GameModel` class can be seen below. It brings together the functionality of the various components discussed to handle a match that has been discovered on the board. Again, care was taken in the naming of methods and attributes to make the code as clear as possible for the reader.

```

private void checkForMatches(Selections selections) {
    MatchContainer matchContainer = matchFinder.findMatches();
    if (matchContainer.hasMatches()) {
        handleMatches(matchContainer);
    } else {
        controller.playSound(INVALID_MOVE);
        boardManipulator.swapBack(selections);
    }
}

private void handleMatches(MatchContainer matchContainer) {
    GamePieceFactory gamePieceFactory = levelManager.getGamePieceFactory();
    do {
        currentLevelScore += matchContainer.getMatchPoints();
        controller.updateScoreBoardView(matchContainer.getMatchPoints());
        gameBoard.highlightTile(matchContainer);
        controller.playSound(matchContainer);
        controller.controlGameBoardView(ONE_SECOND);
        boardManipulator.removeFromBoard(matchContainer, gamePieceFactory);
        boardManipulator.lowerGamePieces(gamePieceFactory);
        matchContainer = matchFinder.findMatches();
    } while (matchContainer.hasMatches());
    checkForLevelUp();
}

```

3.4 User Interface

Individuals with ASC can sometimes find excessive stimuli distressing so the implementation of the user interface naturally required some thought. As previously mentioned, the `onTouchEvent` method was implemented in the `GameBoardView` class rather than the `GameController` to ensure that only the grid area itself is responsive to touch. A border was implemented around the grid area of the GUI to separate it from the edges of the screen that might inadvertently be touched by little thumbs or fingers holding the device. Colours in the blue hue sectors were used as children with ASC have shown a preference for them and evidence suggests that they find such colours have a calming influence [7]. Finally, any other interactive elements other than the grid were avoided as this has been shown to cause distraction to the target user.

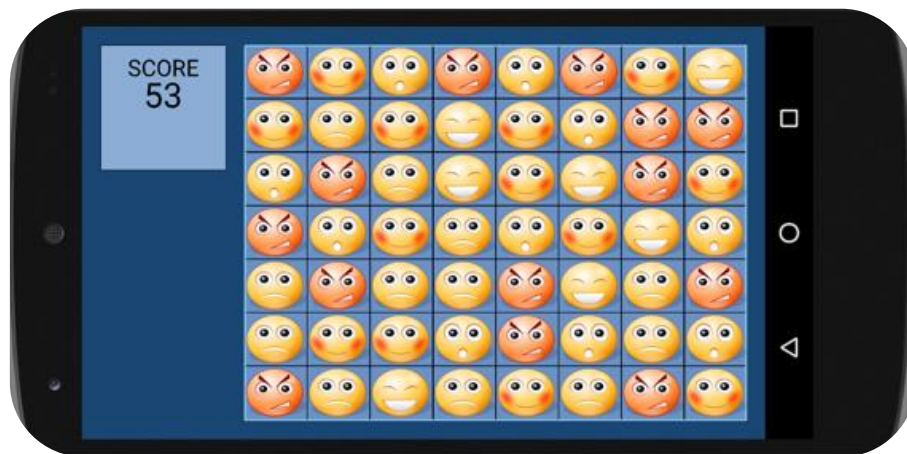


Fig 3.4: Version 3.0 GUI for Nexus 5

The JavaScript prototype that was subsequently discarded also featured the same design.

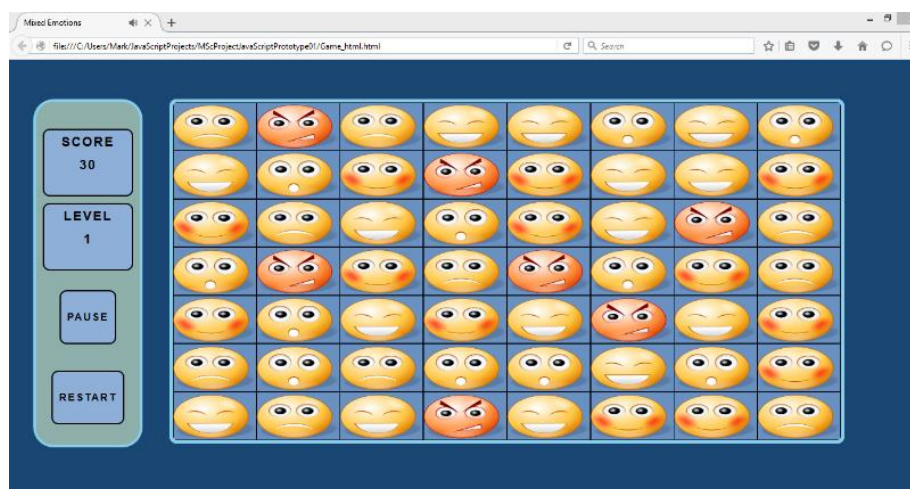


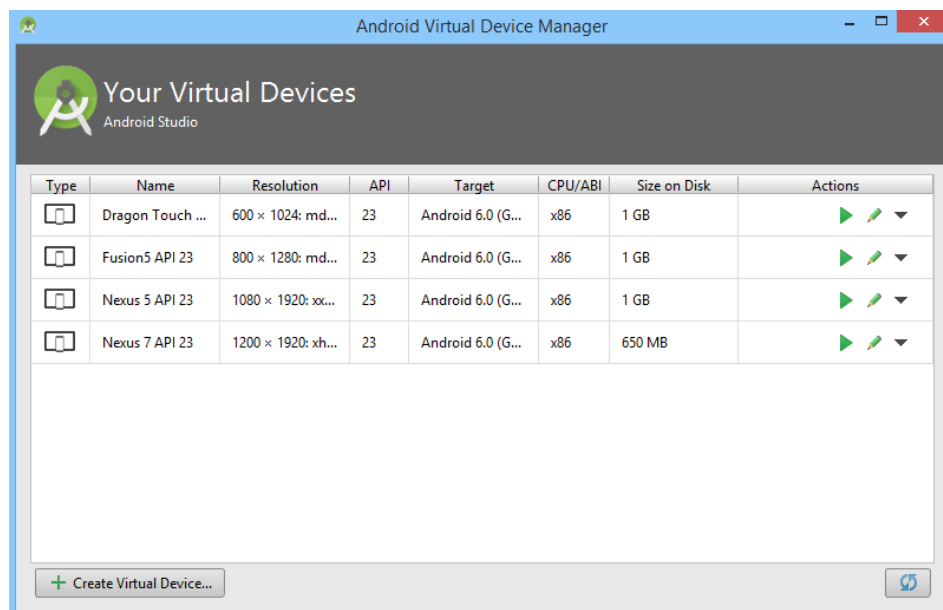
Fig 3.5 Graphical User Interface for JavaScript Prototype

4. Testing, Analysis and Evaluation

This section describes the testing process for the *Mixed Emotions* implementation. The choice of testing strategy is described and the two rounds of user testing are outlined.

4.1 Testing Framework

Android has a Testing Support Library that provides a framework for testing Android apps, which includes APIs for functional user interface tests and JUnit 4. All tests can be run from the Android Studio IDE. Android Studio includes a variety of Android Virtual Devices (AVDs) that can model the screen sizes and densities of targeted devices. These were used to confirm that the game scales correctly in the GUI and runs well on a variety of devices. A disadvantage to using AVDs is that they do not give a very reliable indicator of performance and also take up a large amount of memory on the computer that the Android IDE is being run on [9].



Virtual Devices for testing Mixed Emotions

4.2 Testing Strategy

The original plan was to use a TDD methodology, but maintaining the discipline to continue writing a series of test cases without yet having produced anything tangible proved difficult. However, the value in having a solid battery of tests was not underestimated and, where possible, care was taken to ensure that each test covered only the smallest unit of code, with particular attention being paid to border cases and null values. Exception cases were also tested. Whenever a test failed, work on fixing the bug would only begin once it had been established that the bug could be reproduced consistently. The smallest possible unit of code that could be tested was usually the individual methods of a class. These were tested in separate test methods so that it would be clear which particular method was broken if any test were to fail at a later date. In order to avoid time-consuming back-tracking over previously working code, existing unit tests were run whenever the code underwent significant refactoring or modification.

4.3 Unit Testing

Android Studio enables two categories of unit testing: *local unit tests* and *instrumented unit tests*. Local unit tests run on the Java Virtual Machine (JVM) while instrumented unit tests can be run on either an AVD or a real device. Local unit tests produce results faster than instrumented tests because the unit test code does not need to be loaded into an AVD or device. Local tests were therefore preferred when the class to be tested had no dependencies on the Android framework that could not be mocked. Instrumental tests were employed when access to information about the application environment was required.

4.3.1 Local Testing

An initial problem that was encountered in the testing of *Mixed Emotions* concerned the population of the `GameBoard` 2d array. Because it is populated with emoticons at random, the results of any operations to it would not produce consistent, testable results. To overcome this, a mock `BoardPopulator` class was created that populates the 2d array in a predictable way. As can be seen in the code below, the 2d array is populated with emoticons that each have a type value that is dependent on the value of the counter variable. As the counter is incremented with each iteration, the value for each type of emoticon is unique. This ensures matches will not be inadvertently formed when the 2d array is initially populated. Test emoticons can then be positioned on the board in locations that satisfy the requirements of each particular test. Note the use of the `@Mock` annotation to enable the mocking framework, Mockito, to mock

the `Bitmap` object that needs to be passed in to the constructor of each emoticon. This removes the need for wasteful bitmap creation.

```
@Mock
Bitmap bitmap;

public MockBoardPopulator(GameBoard gameBoard) {
    this.gameBoard = gameBoard;
}

@Override
public void populate() {
    int counter = 0;
    GamePiece emoticon;
    for (int x = ROW_START; x < COLUMNS; x++) {
        for (int y = COLUMN_TOP; y < ROWS; y++) {
            String type = counter <= 9 ? "0" + counter : "" + counter;
            emoticon = new Emoticon(x, y, TILE_WIDTH, TILE_HEIGHT, bitmap, type);
            gameBoard.setGamePiece(x, y, emoticon);
            counter++;
        }
    }
}
```

In the below `testBaseCaseVerticalMatches` test method, emoticons that have real type values are added to the 2d array at locations which will form matches. The below method concentrates on testing the corner cases of the array for matches.

```
@Test
public void testBaseCaseVerticalMatches() throws Exception {
    // Sets 4 vertical matches
    gameBoard.setGamePiece(0, 0, new Emoticon(0, 0, TILE_WIDTH, TILE_HEIGHT, bitmap, ANGRY));
    gameBoard.setGamePiece(0, 1, new Emoticon(0, 1, TILE_WIDTH, TILE_HEIGHT, bitmap, ANGRY));
    gameBoard.setGamePiece(0, 2, new Emoticon(0, 2, TILE_WIDTH, TILE_HEIGHT, bitmap, ANGRY));

    gameBoard.setGamePiece(0, 4, new Emoticon(0, 4, TILE_WIDTH, TILE_HEIGHT, bitmap, HAPPY));
    gameBoard.setGamePiece(0, 5, new Emoticon(0, 5, TILE_WIDTH, TILE_HEIGHT, bitmap, HAPPY));
    gameBoard.setGamePiece(0, 6, new Emoticon(0, 6, TILE_WIDTH, TILE_HEIGHT, bitmap, HAPPY));

    gameBoard.setGamePiece(7, 0, new Emoticon(7, 0, TILE_WIDTH, TILE_HEIGHT, bitmap, EMBARRASSED));
    gameBoard.setGamePiece(7, 1, new Emoticon(7, 1, TILE_WIDTH, TILE_HEIGHT, bitmap, EMBARRASSED));
    gameBoard.setGamePiece(7, 2, new Emoticon(7, 2, TILE_WIDTH, TILE_HEIGHT, bitmap, EMBARRASSED));

    gameBoard.setGamePiece(7, 4, new Emoticon(7, 4, TILE_WIDTH, TILE_HEIGHT, bitmap, SAD));
    gameBoard.setGamePiece(7, 5, new Emoticon(7, 5, TILE_WIDTH, TILE_HEIGHT, bitmap, SAD));
    gameBoard.setGamePiece(7, 6, new Emoticon(7, 6, TILE_WIDTH, TILE_HEIGHT, bitmap, SAD));

    matchContainer = matchFinder.findMatches();
    matchingX = matchContainer.getMatchingX();

    // check vertical matches found
    assertEquals(4, matchingX.size());
    assertEquals(HAPPY, matchingX.get(0).get(0).toString());
    assertEquals(ANGRY, matchingX.get(1).get(0).toString());
    assertEquals(SAD, matchingX.get(2).get(0).toString());
    assertEquals(EMBARRASSED, matchingX.get(3).get(0).toString());
}
```

4.3.2 Instrumental Testing

Although many of the problems that were encountered when testing could be overcome with Mockito and other home-made mock objects, the classes that require access to resources such as bitmaps, or those that have dependencies on the Android system `Context` object, could not be mocked in the same way. Fortunately, application context and instrumentation can be obtained with the Android `InstrumentationRegistry` class. One example of its use is given below in a `BitmapCreator` test. The `testGetAngryBitmap` method calls a `getTargetContext` method which returns a `Context` object containing information regarding the application environment. This allows access to an emoticon bitmap for testing.

```
@Test
public void testGetAngryBitmap() throws Exception {
    Context context = InstrumentationRegistry.getTargetContext();
    bitmapCreator.prepareScaledBitmaps(context, TILE_WIDTH, TILE_HEIGHT);
    Bitmap angryBitmap1 = bitmapCreator.getAngryBitmap();

    Bitmap unscaledBitmap = BitmapFactory.decodeResource(context.getResources(), R.drawable.angry);
    Bitmap angryBitmap2 = Bitmap.createScaledBitmap(unscaledBitmap, TILE_WIDTH, TILE_HEIGHT, false);
    assertTrue(angryBitmap1.sameAs(angryBitmap2));
}
```

4.4 User Testing

4.4.1 Round 1

The first round of testing was conducted on Version 2 of *Mixed Emotions*. Eight participants who had all received an official diagnosis of high functioning autism (five girls and three boys aged between 5 – 11 years old) used the application at home under the watch of a parent for at least one hour. Results were generally mixed but useful feedback was received.

The first criticism was regarding the music. Two of the participants found the music to be too distracting. When questioned on whether this concerned the volume of the music or the music itself, the feedback was that it was too fast or too lively. Music of a more relaxing nature was used in the subsequent version of the game.

In addition, two of the younger children found the game difficult to play. One participant was reported to have simply tapped various emoticons at random on the screen in the hope that something would happen and lost interest after five minutes.

On the other hand, the children above 8 years old were reported to have enjoyed the game and were able to recall the emotions that featured in the game and correctly match them with physical Picture Exchange Communication System (PECS) cards displaying the same emotion. The only other criticism was regarding the fact that only one level of the game was available at the time, which raised a good point that the extra levels would be a valuable opportunity to increase the breadth of emotions that the game could accommodate.

4.4.2 Round 2

Approximately five weeks after the feedback from Version 2 of *Mixed Emotions*, a subsequent round of testing was carried out on Version 3, with changes recommended from the first round of testing incorporated.

The music had been changed to something of a slower pace, which feedback indicated caused less agitation amongst participants. The inclusion of an additional level received favourable feedback and appeared to increase the length of time that the application was played for.

On the whole, the levels of agitation seemed to be minimal and the target user's interaction with the game appeared to be largely an enjoyable one that brought about positive reinforcement of the concept of emotions. No other problems with the game were reported.

5. Project Evaluation

This section describes particular weaknesses and strengths of *Mixed Emotions* and details some of the lessons that were learned during the development of the application.

5.1 Limitations and Contributions

The two rounds of user testing highlighted the fact that *Mixed Emotions* does have some limitations. However, it should be noted that any software that is designed for people with ASC to help improve emotion recognition skills will be limited to some degree because, no matter how realistic the simulation, it will never be as valuable as real life experience.

As the game focusses on the matching of emotions in quick succession, it is possible that the searching for matches could take priority over consideration of the emotions themselves. Although naming the matching emotions can be valuable practice, the user is not required to interact with them in any meaningful way that represents a true social interaction with another human being.

With that said, *Mixed Emotions* does appear to make some contribution to the area of facial emotion recognition. Firstly, many of the volunteers did appear to genuinely enjoy the game and the majority stated that their experience of playing the game was positive. This could be partly due to the care taken in minimizing distraction in the GUI and using a simple design with colours that are suggested to be calming. Furthermore, many of the user test participants were able to recall at least four of the emotions present in the first round of the game, and many were able to correctly match these with expressions displayed on the PECS cards.

5.2 Lessons Learned

Without doubt, the benefit of producing frequent iterations of an application was strongly reinforced. Although the design and implementation of *Mixed Emotions* started from a

primitive command line interface written in Java, it proved to be most beneficial for the implementation of the business logic. It was also reassuring to have something working at an early stage that could be slowly improved upon.

The benefit of building and running frequent tests was also reinforced. On several occasions, the running of tests highlighted problems in the code that had not existed a short while before, enabling bugs to be quickly located. Had the tests not highlighted the problem at such an early stage, it would have wreaked havoc and cost much valuable time.

6. Conclusions and Future Work

This final section provides a reflection on the project and considers opportunities for future work on the application.

6.1 Summary

On reflection, the project has been a success for a variety of reasons. First and foremost, a working Android application has been implemented that target users have tested and enjoyed using. Although the extent to which *Mixed Emotions* is able to help improve the emotion recognition skills of people with ASC is still not entirely certain, the enjoyment experienced by the target users, and the favourable association with emotions that it generated are positive.

On a personal note, the skills and experience gained from working on this project have been of immense benefit. It is often said that the best way to learn a programming language is to actually try to build something with it. As someone who had the tendency to spend more time reading about coding than actually doing it, actually having a goal to work towards whilst learning Android accelerated the learning curve significantly. In addition, applying design patterns to the design of the application, discovering SOLID principle violations, and learning to refactor the design has been immensely valuable.

6.2 Future Work

Work on *Mixed Emotions* will continue from here. As children with ASC sometimes find the transition to different activities to be quite a difficult task, it may be beneficial to introduce an egg timer animation to the game and give parents the option of setting a message to be displayed or played when the egg timer has finished to introduce the new activity, such as “Now it’s bath time!”. In this way it is hoped that *Mixed Emotions* can become more than just a game for the children, but also a tool that makes a small contribution to the family as a whole.

References

- [1] B. Hardy, in *Android Programming: The Big Nerd Ranch Guide*, 2013, p. 36.
- [2] R. Nystrom, “Game Programming Patterns,” [Online]. Available: <http://gameprogrammingpatterns.com/game-loop.html>. [Accessed 18 09 2016].
- [3] [Online]. Available: <https://developer.android.com/training/basics/activity-lifecycle/starting.html>. [Accessed 26 8 2016].
- [4] [Online]. Available: <https://developer.android.com/reference/android/app/Activity.html>. [Accessed 18 9 2016].
- [5] [Online]. Available: <https://developer.android.com/training/basics/supporting-devices/index.html>. [Accessed 18 09 2016].
- [6] [Online]. Available: <https://developer.android.com/guide/topics/graphics/2d-graphics.html>. [Accessed 18 09 2016].
- [7] “Designing Environments,” [Online]. Available: <http://www.autism-architects.com/wp-content/uploads/downloads/2012/06/CB-Presentation-Catania-2010.pdf>. [Accessed 18 09 2016].
- [8] “OODesign,” [Online]. Available: <http://www.oodesign.com/open-close-principle.html>. [Accessed 18 09 2016].
- [9] [Online]. Available: <https://developer.android.com/topic/libraries/testing-support-library/index.html>. [Accessed 18 09 2016].