PAPL Ltd.

How to ask your boss for a raise

Up941424 - Technical Report

Introduction

This project aims to build upon PAPL ltd's product based on the George Perec novel "The Art of Asking Your Boss for a Raise", which is a traditional Command Line Interface that allows a user to navigate an underlying Decision Map.

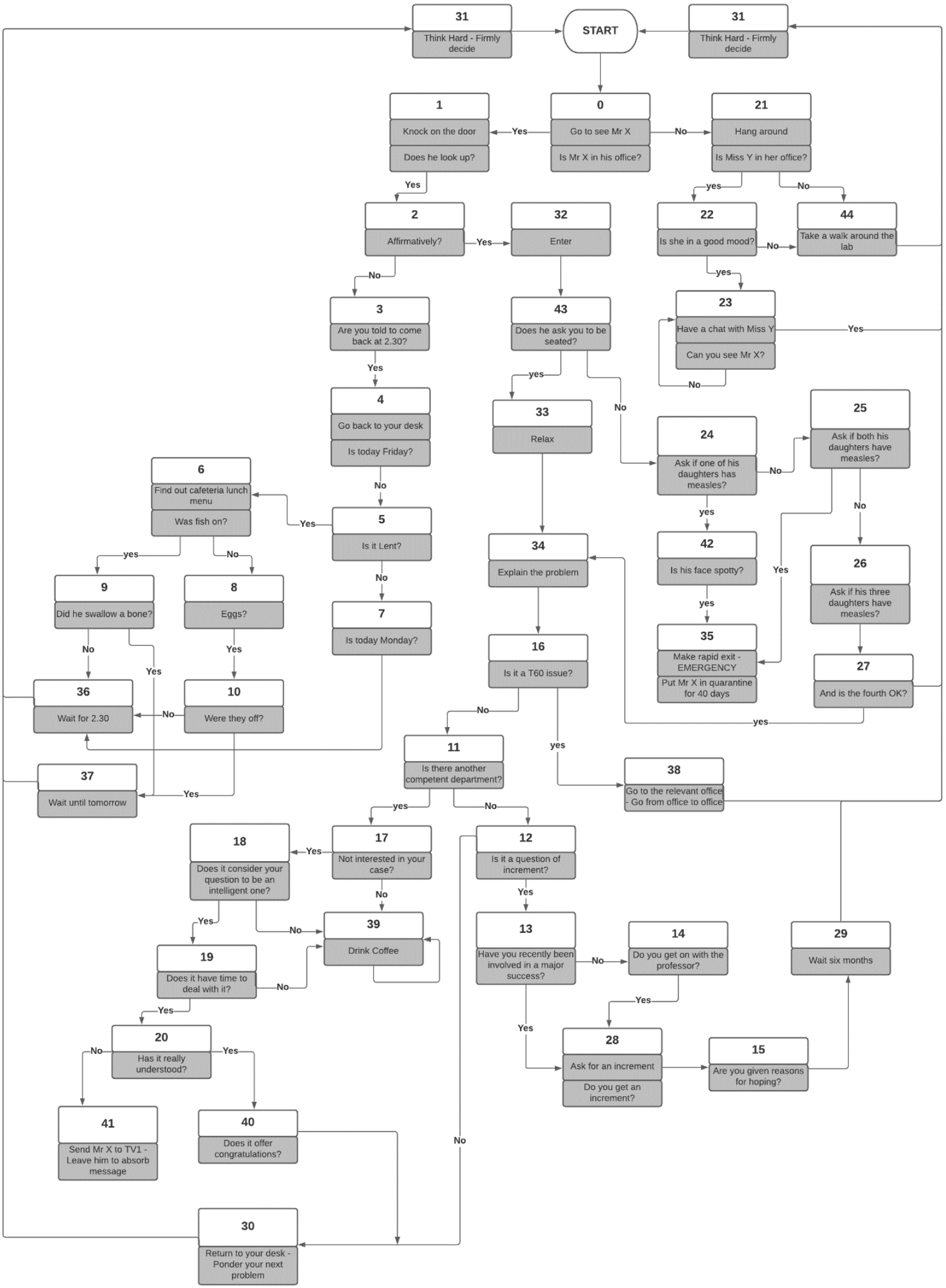
The project aim is to design and implement an interactive front-end for this application that is usable on android devices. This report documents the progression of the project from inception to completion.

First Implementation Proposal

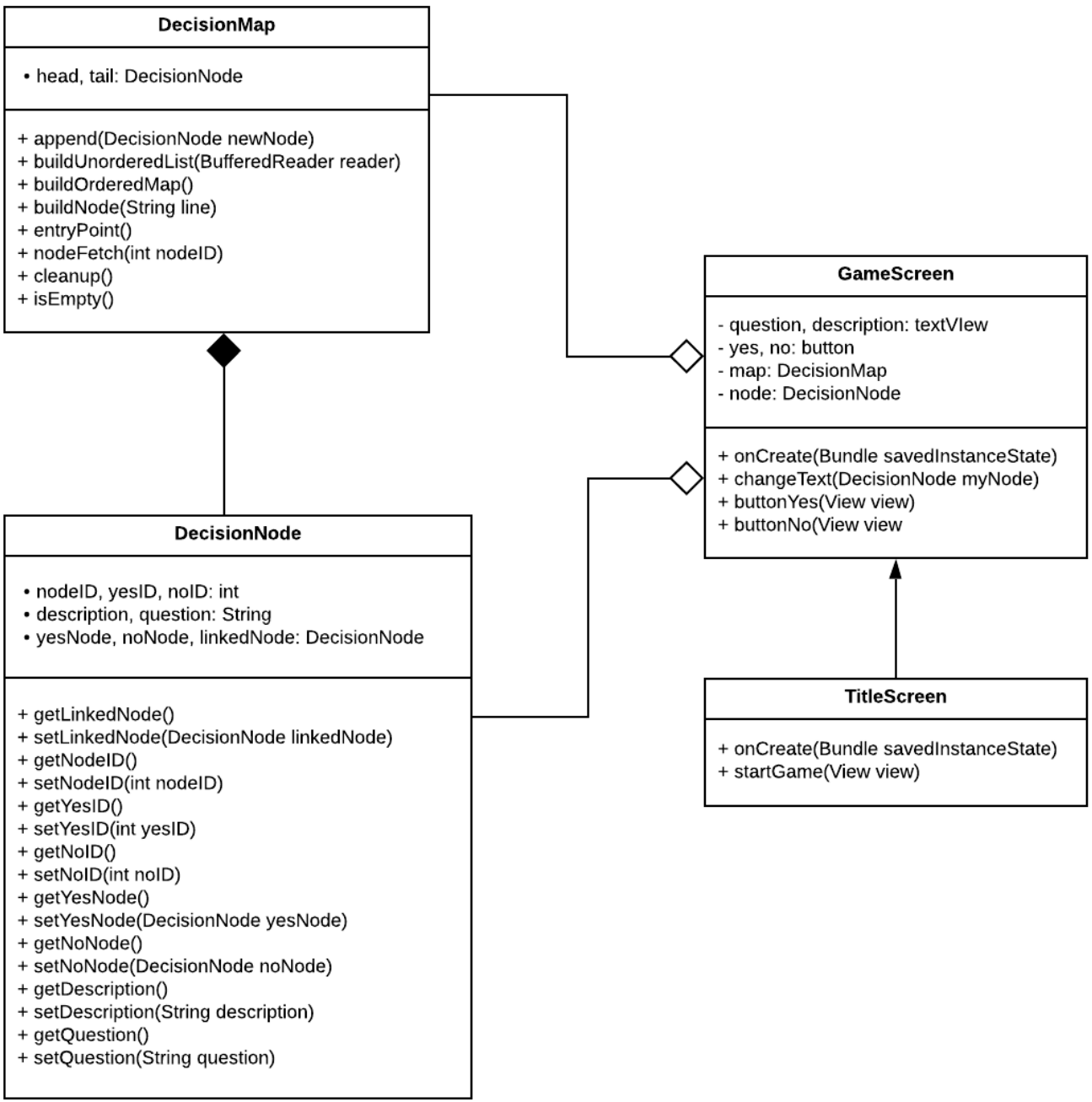
- Aim is to achieve a basic functioning and interactive Android interface for the application.
- This implementation of the app will feature no changes to the decision map design.
- While It is expected that changes will need to be made to the existing codebase for it to function as an android application, these changes should be kept to a minimal to maximise time efficiency.

Decision Map

The only representation of the decision map received was the comma separated file (CSV) file itself, so my first course of action was to draw a diagram representation of the map in Lucidchart. This will help me to visualise how the map functions as well as visualising any changes that might be made to the design in future implementations, helping to avoid making permanent changes directly to the CSV file.


START
0
Go to see Mr X
Is Mr X in his office?
Yes
1
Knock on the door
Does he look up?
No
21
Hang around
Is Miss Y in her office?
31
Think Hard - Firmly decide
31
Think Hard - Firmly decide
Yes
2
Affirmatively?
Yes
32
Enter
No
3
Are you told to come back at 2.30?
Yes
4
Go back to your desk
Is today Friday?
No
5
Is it Lent?
Yes
6
Find out cafeteria lunch menu
Was fish on?
yes
9
Did he swallow a bone?
No
8
Eggs?
No
36
Wait for 2.30
Yes
10
Were they off?
No
Yes
37
Wait until tomorrow
Yes
No
7
Is today Monday?
yes
22
Is she in a good mood?
No
44
Take a walk around the lab
yes
23
Have a chat with Miss Y
Can you see Mr X?
Yes
No
43
Does he ask you to be seated?
yes
33
Relax
No
24
Ask if one of his daughters has measles?
No
25
Ask if both his daughters have measles?
yes
42
Is his face spotty?
Yes
No
26
Ask if his three daughters have measles?
yes
35
Make rapid exit - EMERGENCY
Put Mr X in quarantine for 40 days
27
And is the fourth OK?
yes
34
Explain the problem
16
Is it a T60 issue?
No
yes
11
Is there another competent department?
yes
No
38
Go to the relevant office - Go from office to office
18
Does it consider your question to be an intelligent one?
Yes
17
Not interested in your case?
Yes
No
12
Is it a question of increment?
Yes
39
Drink Coffee
No
Yes
19
Does it have time to deal with it?
No
Yes
20
Has it really understood?
No
Yes
41
Send Mr X to TV1 - Leave him to absorb message
40
Does it offer congratulations?
13
Have you recently been involved in a major success?
No
14
Do you get on with the professor?
Yes
Yes
28
Ask for an increment
Do you get an increment?
15
Are you given reasons for hoping?
29
Wait six months
No
30
Return to your desk - Ponder your next problem

Custom Class Design (UML Class Diagram)



Custom Class Design

Updates to Existing DecisionMap & DecisionNode Classes

The *DecisionMap* and *DecisionNode* classes were transferred along from the original Command Line implementation of the app. However, the following changes to the existing *DecisionMap* class were necessary to allow it to function as an android application.

- The *constructor* was updated, allowing the front end to read the CSV file from the resources folder.

```
// constructor loads up data file
public DecisionMap(Context context)  {
    InputStream inFile = context.getResources().openRawResource(R.raw.dataraise);
    BufferedReader reader = new BufferedReader(new InputStreamReader(inFile, Charset.forName("UTF-8")));
    buildUnorderedList(reader);
    buildOrderedMap();
}
```

- The *buildOrderedMap()* was updated corresponding changes to the constructor.

```
// converts each line from csv file into node and adds it to list
public void buildUnorderedList(BufferedReader reader)  {


    DecisionNode node;
    String line = "";
    try  {
        while( (line = reader.readLine()) != null)  {
            // split data by comma
            line.split( regex: ",");

            // read data
            node = buildNode(line);
            append(node);
        }
    }  catch (IOException e)  {
        Log.wtf( tag: "DecisionMap", msg: "Error reading data file on line " + line);
    }
}
```

TitleScreen Class

I wanted the first thing that the user sees when initiating the app to be attractive and engaging and decided the best way to achieve this was through a title screen. The title screen presents the user with the title of the game in large font and a button that starts the game.

GameScreen Class

Rather than have separate screens for each node traversed in the map, my approach was to update the text on a single screen according to the current node being traversed.

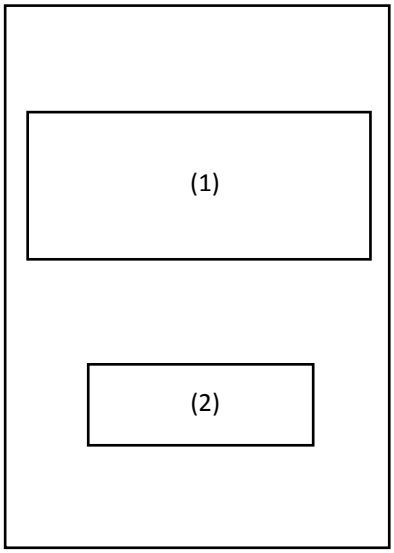
```
public void changeText(DecisionNode myNode)  {
    node = myNode;
    description.setText(myNode.getDescription());
    question.setText(myNode.getQuestion());
}
public void buttonYes(View view)  {
    changeText(node.getYesNode());
}
public void buttonNo(View view)  {
    changeText(node.getNoNode());
}
```

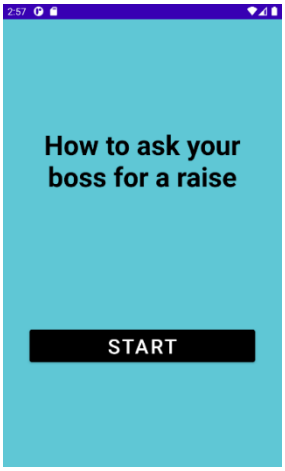
Graphical User Interface Design

Title Screen

(1) Main title display

(2) Start button to enter game screen





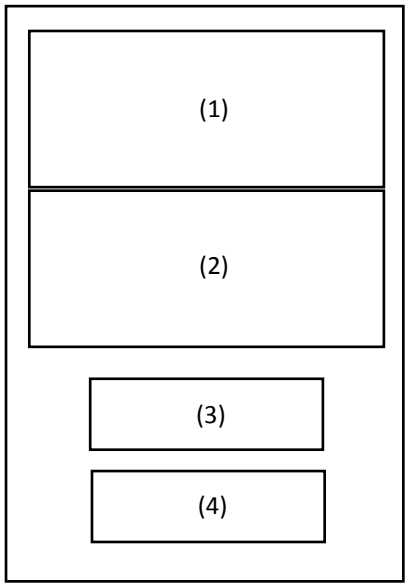
Game Screen

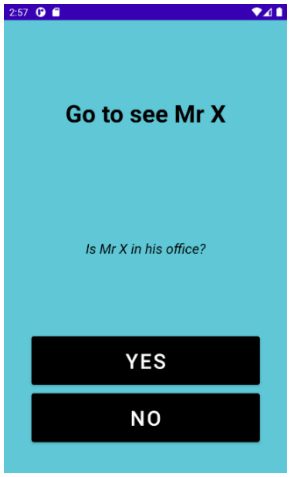
(1) Displays node description

(2) Displays node question

(3) Button for yes option

(4) Button for no option





Link to video report update

Video report reflecting on the first implementation of the application and future plans.

https://www.youtube.com/watch?v=U3yE_l2b5QM&feature=youtu.be

Proposed Updates

Add further exception handling

Custom exception handlers and catching of errors must be implemented so that future developers will have an easier time catching bugs when making updates to this application.

Add more decisions

The original application only gives the user the same two options for each question. I plan to make the game feel more engaging by adding more decision options.

Add winning and losing

The original application loops infinitely. By providing ways to complete or fail the game, the user will have a tangible goal to work towards, adding depth to gameplay.

Update decision map

In order to pave the way for the above two proposals, I propose making changes to the underlying decision map. It may be necessary to completely rewrite the decision map from scratch.

Make improvements to graphical user interface (GUI)

Application currently has a very basic looking GUI, improving upon this and making it more attractive will increase customer engagement and retention.

Add inventory system

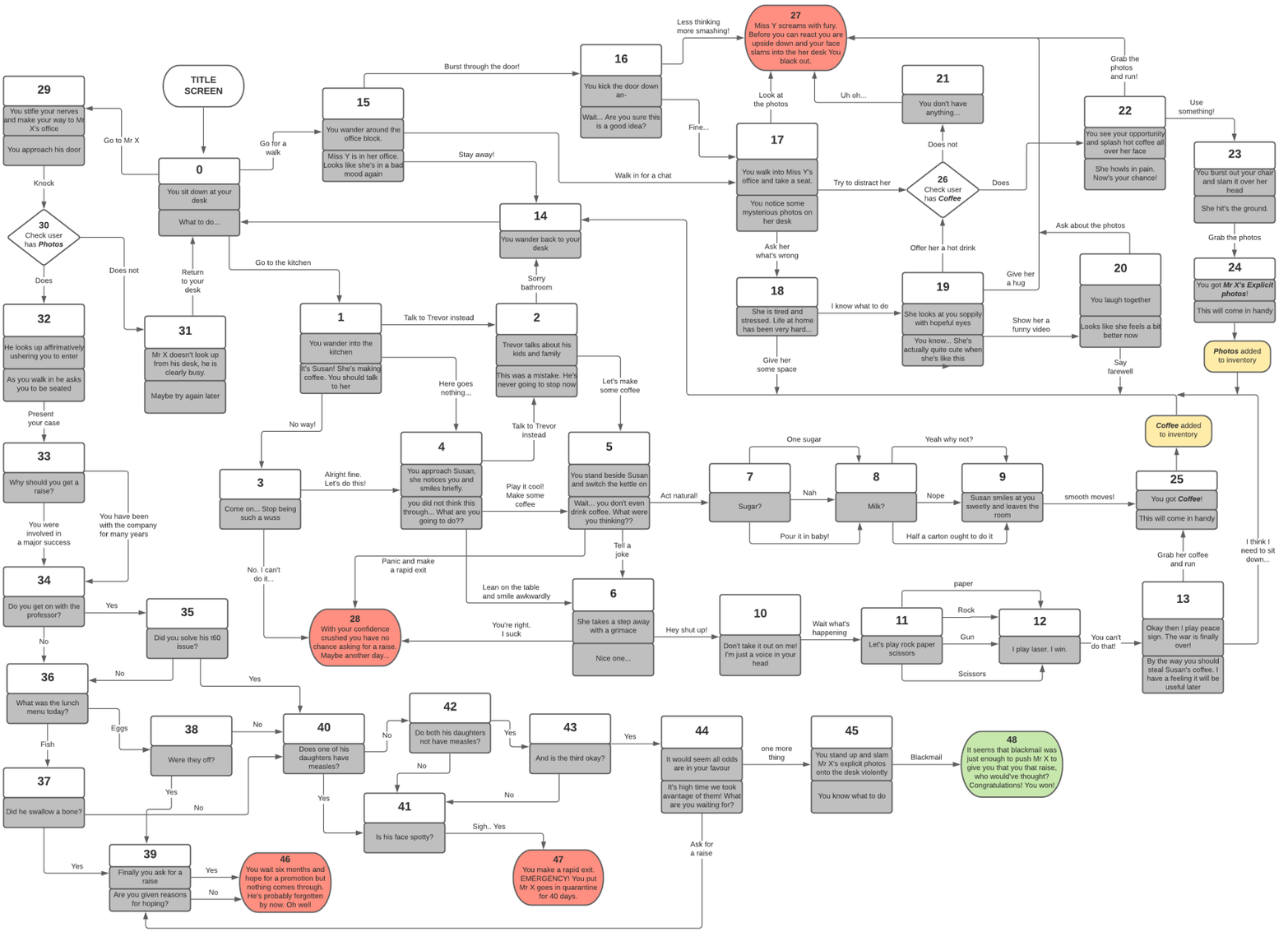
In order to add further depth and replayability to the game I propose adding an inventory system. The inventory system will include items that must be picked up in order to progress and complete the game.

Updated Design & Implementation

Updated Decision Map

The decision map was redesigned so that it featured the following:

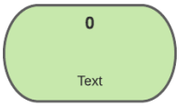
- A max of three decisions instead of two.
- *"Winning"* and *"losing"* nodes in which the player can fail or complete the game.
- "*Item*" and "*decision*" nodes in which the player can pick up or use items to progress through the game.



Title screen: the user can click the start button to begin the game from here.

Action node: displays the node description/question and presents the user with options that will determine how the map is traversed.

Death node: if a user reaches this node they will fail the game and be prompted to try again



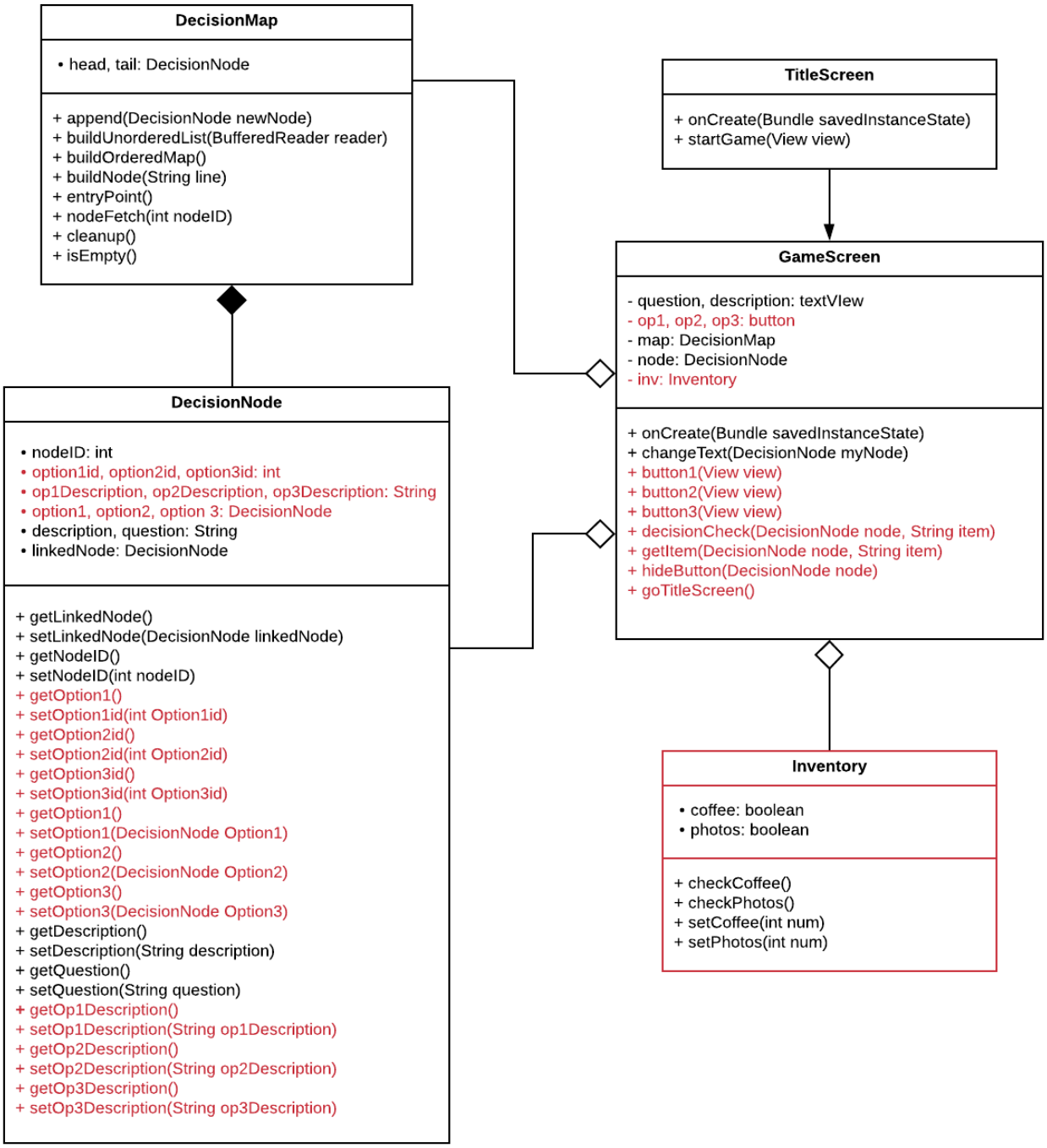
Victory node: if a user reaches this node they will win the game and be prompted to try again

Item node: If a user reaches this node a specific item will be added to their inventory

Decision node: Checks whether a user has a specific item and moves the user to relevant node

Updated UML Class Diagram

(Updates highlighted in red)



Custom Class Design Updates & New Additions

DecisionMap and DecisionNode Class updates

- In order to include three decision options, four new items needed to be added to the CSV file. One new item to determine the third option of map traversal, and three items to represent the unique choices for each traversal option. This is demonstrated in the CSV file line below:

	A	B	C	D	E	F	G	H	I
1	0	15	1	29	You sit at your desk with your	You release a sigh of boredom.	Go for a walk	Go to the kitchen	Visit Mr X

- The DecisionMap and DecisionNode classes also needed to be updated so that the new CSV file format could be read.

```
// converts CSV line into a readable node
private DecisionNode buildNode(String line)   {
    String[] stringArray = line.split( regex: ",");
    DecisionNode n = new DecisionNode();

    n.setNodeID(parseInt(stringArray[0]));
    n.setOption1id(parseInt(stringArray[1]));
    n.setOption2id(parseInt(stringArray[2]));
    n.setOption3id(parseInt(stringArray[3]));

    n.setDescription(stringArray[4]);
    n.setQuestion(stringArray[5]);

    n.setOp1Description(stringArray[6]);
    n.setOp2Description(stringArray[7]);
    n.setOp3Description(stringArray[8]);

    return n;
}
```

```
// grab IDs and link paths for yes and no decisions
public void buildOrderedMap()  {

    if(head == null)  {  return; }

    DecisionNode nodeLinker = head;

    while(nodeLinker != null)  {

        int option1id = nodeLinker.getOption1id();
        int option2id = nodeLinker.getOption2id();
        int option3id = nodeLinker.getOption3id();

        DecisionNode option1 = nodeFetch(option1id);
        DecisionNode option2 = nodeFetch(option2id);
        DecisionNode option3 = nodeFetch(option3id);

        nodeLinker.setOption1(option1);
        nodeLinker.setOption2(option2);
        nodeLinker.setOption3(option3);

        nodeLinker = nodeLinker.getLinkedNode();
    }
    cleanup();

}
```

Inventory Class

- This custom class was necessary in order to implement the new item and decision nodes included in the updated decision map design.
- The class enables users to pick up items when traversing through item nodes, as well as enabling them to use the items they acquire when traversing through decision nodes.
- I chose to implement this as a separate class rather than directly into the GameScreen class, making it easier for future developers to add further updates to the inventory system, such as new items.

```
public class Inventory {

    boolean coffee = false;
    boolean photos = false;


    public boolean checkCoffee()  { return coffee; }
    public boolean checkPhotos()  { return photos;  }
```

```
public void setCoffee(int num)  {
    switch(num)  {
        case 1:
            coffee = true;
            break;
        case 0:
            coffee = false;
            break;
    }
}
```

GameScreen Class Updates

Accounting for new decision map design

- The gamescreen logic needed to be updated to account for three decision buttons, rather than two.
- Since the new map design features "special" nodes such as decision and item nodes that cause something to happen, the following methods were necessary to add to the class to determine these "special" nodes during map traversal.

```
// checks if current node adds an item to player inventory
if (qstn.equals("This will come in handy"))  {
    getItem(desc);
}

// checks if current node is decision node
if (qstn.equals("DECISION"))  {
    decisionCheck(node, desc);
    return;
}
```

Cleaning up Graphical User Interface

In both the previous and updated map design there is a varying amount of decisions, sometimes users will be able to pick from three decisions and sometimes only two. This posed a problem because the user interface would still present the user with three decision buttons even if there were only two options to pick from. This may come across as confusing or lazy to the user, therefore, a method to hide unneeded buttons was necessary to add to the class.

```
// cleans up GUI by hiding needless buttons
public void hideButton(DecisionNode node)  {
    if (node.getOp2Description().equals("-")) {
        op2.setVisibility(View.INVISIBLE);
    } else {
        op2.setVisibility(View.VISIBLE);
    }
    if (node.getOp3Description().equals("-")) {
        op3.setVisibility(View.INVISIBLE);
    } else {
        op3.setVisibility(View.VISIBLE);
    }
```

Returning to Title Screen

The new map design features victory and failure nodes in which the user should be prompted to play again. In order to achieve this, I simply allowed users to return to the title screen upon winning or losing, where they can press the start button to play again.

Design & use of Custom Exception Handling

Custom Exceptions

CSVFormatError is thrown for any formatting issues in the CSV file, such as too little or too many items per line.

```
package com.example.bossraise;

public class CSVFormatError extends NullPointerException {

    public CSVFormatError(String message) { super(message); }
}
```

invalidCSVitem is thrown if the CSV file contains invalid arguments, such as characters for node IDs.

```
package com.example.bossraise;

public class invalidCSVitem extends IllegalArgumentException {

    public invalidCSVitem(String message)  { super(message);
    }
```

Error Throwing

Invalid amount of CSV files on a line:

```
// Check CSV file has correct amount of items per line
if(stringArray.length != 9)  {
    throw new CSVFormatError("wrong ammount of items on CSV line for node " + stringArray[0]);
}
```

```
E/CSV format error: Issue with CSV file format. wrong ammount of items on CSV line for node 0
```

Empty or unreachable CSV file:

```
// throw error if CSV file is empty
if(!reader.ready())  {
    throw new CSVFormatError("CSV file appears to be empty");
}
```

```
E/CSV format error: Issue with CSV file format. CSV file appears to be empty
```

Node IDs are not integer values:

```
private int getID(String[] arr, int index)  {
    for (char c : arr[index].toCharArray()) {
        if (!Character.isDigit(c)) {
            throw new invalidCSVitem("All node IDs must be integer, error found at node " + arr[0]);
```

```
E/CSV item error: Invalid CSV argument found. All node IDs must be integer, error found at node 0
```

Catching Exceptions

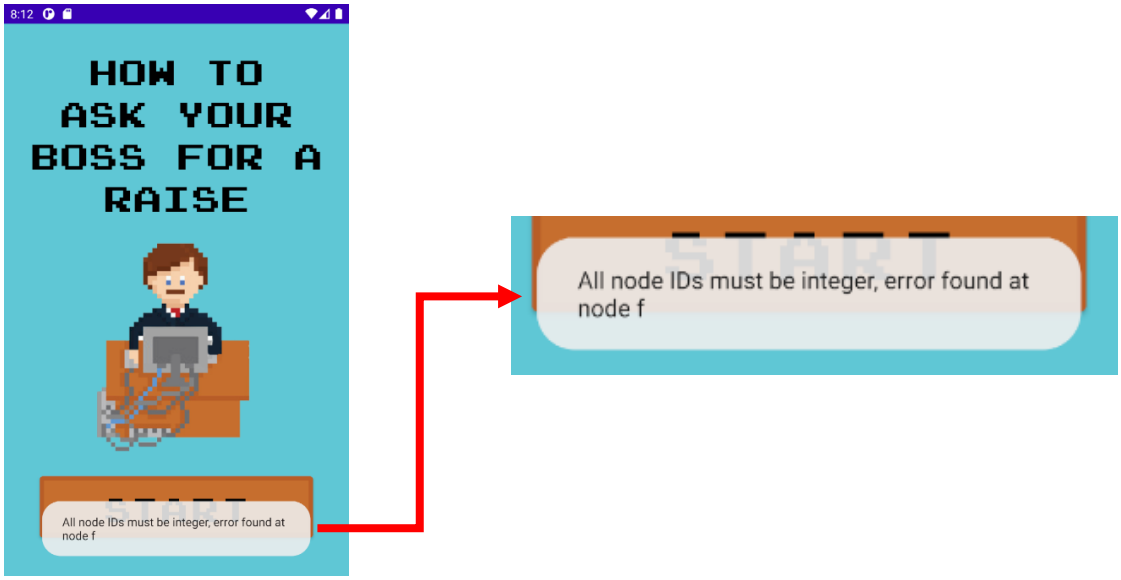
Errors are caught upon initiation of the main activity screen. This will halt the application and log valuable error information to the console as well as displaying the error on the application's user interface.

```
try {
    map = new DecisionMap(getApplicationContext());

    inv = new Inventory();
    inv.setPhotos(0);
    inv.setCoffee(0);

    node = map.entryPoint();
    changeText(node);

} catch(Resources.NotFoundException e)  {
    Log.e( tag: "CSV not found",  msg: "CSV file was not found. " + e.getMessage());
    exceptionToast(getApplicationContext(), e.getMessage());
    finish();
}  catch(CSVFormatError | invalidCSVitem e)  {
    Log.e( tag: "CSV item error",  msg: "Invalid CSV argument found. " + e.getMessage());
    exceptionToast(getApplicationContext(), e.getMessage());
    finish();
} catch(IOException e)  {
    Log.e( tag: "Logic error",  msg: "Issue with CSV file data. " + e.getMessage());
    exceptionToast(getApplicationContext(), e.getMessage());
    finish();
} catch(Exception e)  {
    Log.e( tag: "Unhandled Exception", e.getMessage());
    exceptionToast(getApplicationContext(), e.getMessage());
    finish();
}
```



Link to Final Demonstration Video

https://www.youtube.com/watch?v=bc-MCErU3oc&feature=youtu.be

User Interface Design Updates

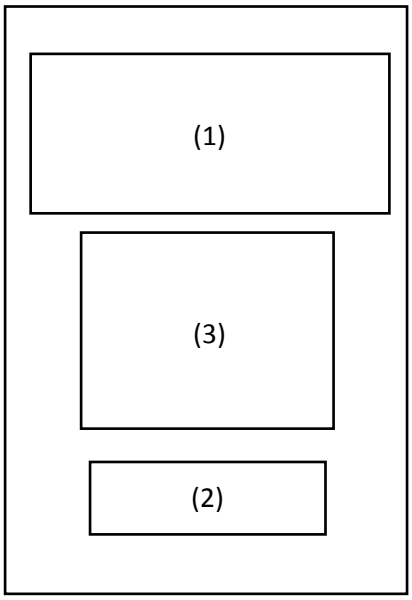
Title Screen

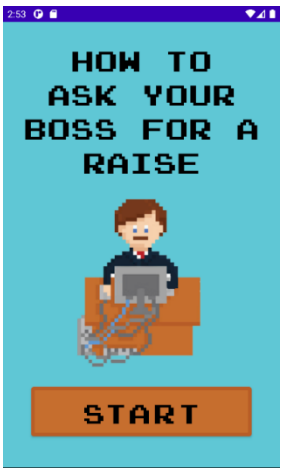
Image was added

(1) *Main title display*

(2) *Start button to enter game screen*

(3) *Image display*





Game Screen

Third button was added

(1) *Displays node description*

(2) *Displays node question*

(3) *Button for option1*

(4) *Button for option2*

(5) *Button for option3*