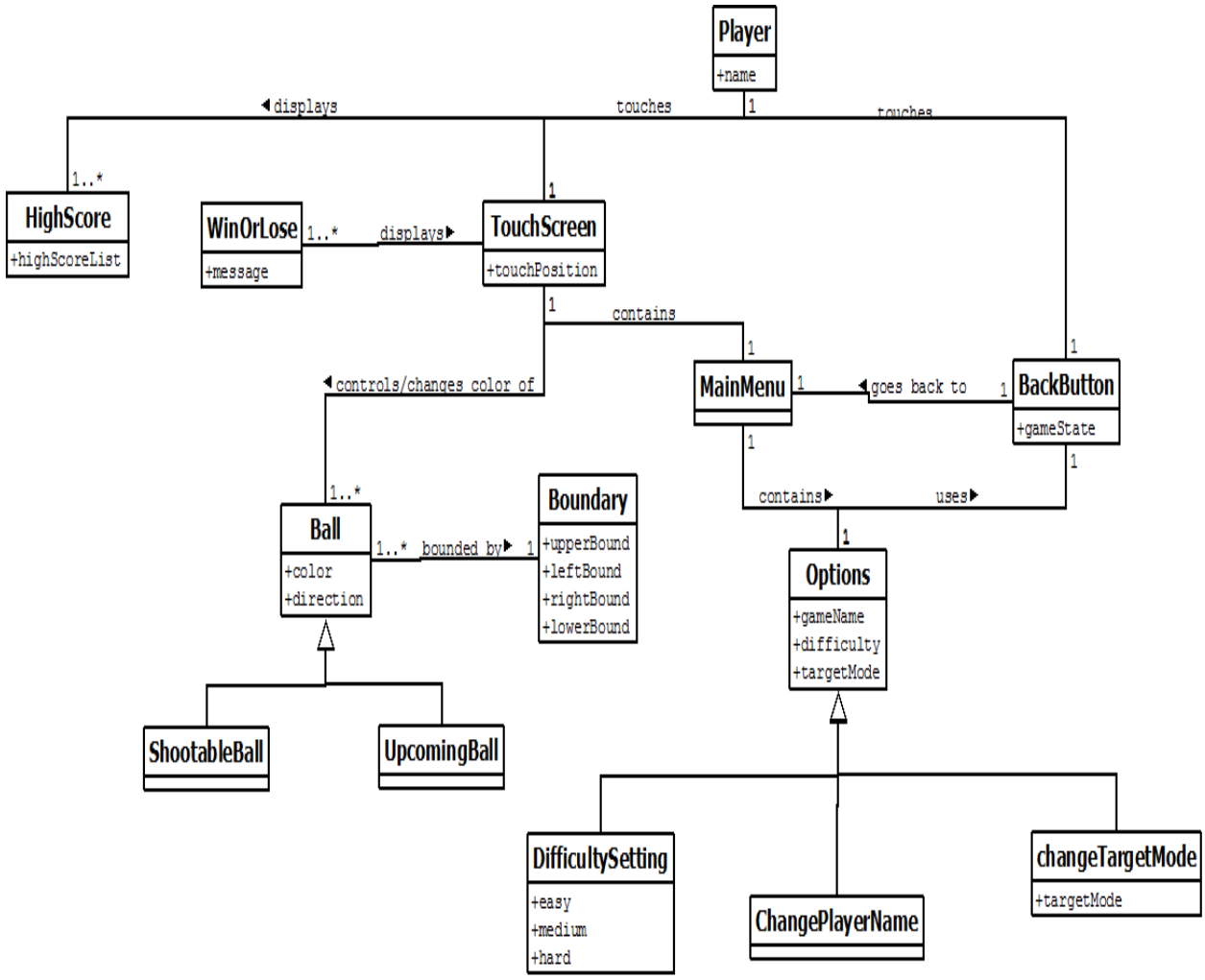
Frozen Bubble (previously Sliding Menu)

Massimo Catanzariti 1696645
Nicolas Chausseau 6431526
Jonathan Bobak 1945947
Astou Sene 9557679
Walter Alexander Chacon 9238662

Summary of Project

For our team project, we have chosen a project called Frozen Bubble. The original java program was coded by Guillaume Cottenceau. The android port has been coded by Pawel Aleksander Fedorynski and Eric Fortin. This is a small Android video game coded in Java using the Android SDK. In this video game, the player aims to shoot bubbles to attach them to other bubbles of same color. Points are awarded when 3 or more bubbles of the same color are attached. We have chosen this project mainly because we want to either enhance or develop skills as Android developers and touch the game development domain since game applications are a popular type of program to develop. This project allows us to analyze and work with touch listeners, graphics and animations, and the Android SDK in general.

Conceptual Diagram (From Milestone 2)



UML diagram of System

(classes of interest → see file Class Diagram ver3.0.gif in same repository)

The system diagram (class diagram of the system) represents the most important parts of the android app. It includes the classes needed for the game to work properly. Since we are mainly concerned about the mechanics of the game, some classes that involved Bit Map wrapping and

images have been excluded since they are meant more for the visual aesthetics of the game rather than how the gameplay and interface is implemented.

The main class that creates the other objects in the application is called the FrozenBubble. This class initializes the menu, the game view, the options, sprites in the game, the orientation of the screen according to the mobile device, and everything else that is associated with the setup of the game. The FrozenBubble class has associations with the GameView and GameThread class. It also implements the interfaces GameListener and AccelerometerListener. The GameView class is essentially the touch screen of the application. It pertains to all the events applicable (such as swiping or single touch) when touching the mobile application screen. The GameView has associations with the GameListener interface and the GameThread class. The GameThread is the class that decides what to do with the touch events. This leads to the association relationship with the actual game itself, the FrozenGame class. The FrozenGame class is a generalization of the GameScreen class; a class that updates the screen in which the user interacts with. The FrozenGame class is associated with BubbleSprite class, BubbleManager class, LaunchBubbleSprite class, MalusBar class, HighScoreManager class, and the LevelManager class. These classes deal with the various objects and data in the game such as the bubbles and their behavior, the high score, and the current or future levels of the game. The BubbleSprite class is also associated with the BubbleManager class and he HighScoreManager class is associated with the HighScoreDB class. There is also a ComputerAI class that is associated with the FrozenGame class and the Freil class. The ComputerAI class determines how the computer will play against the user. The class also implements the OpponentListener interface. The Freile class is also associated with the OpponentListener.

Other classes that do not have relationships with the FrozenGame class but are still essential to the application are the CollisionHelper, SplashScreen, PreferenceActivity, AccelerometerManager, and the HighScoreDO classes. The CollisionHelper is an essential helper class for how the game will play. The SplashScreen class manages the splash screen that appears when the game is launched. This class allows the player to start the game with a button. PreferenceActivity allows the user to change the settings of the game (difficulty, full screen, etc.). The AccelerometerManager allows the game view to be rotated according to the orientation of the mobile device. Finally, the HighScoreDO class collects data for determining a high score in the game.

More or less, the class diagram of the system maps to most of the architecture of the

conceptual diagram. For instance, the TouchScreen can be interpreted as the GameView, GameThread, GameScreen, and FrozenGame classes. The HighScore and WinOrLose classes can be mapped to the HighScoreManager, HighScoreDA, and HighScoreDO classes. The Ball class maps to the BubbleSprite, BubbleManager, and the Freile classes. The Boundary class can be interpreted as the CollisionHelper class as well as the BubbleManager and LaunchBubbleSprite classes. The MainMenu class is the SplashScreen class, and the Options class is the PreferenceActivity class.

Some of the classes that are not similar are the Player, Options, and BackButton classes. The Player class is essentially the user of the application, and a user cannot be a class. The Options class, which is mapped as PreferenceActivity, is not associated with the MainMenu; it is a class on its own. Also not associated with the MainMenu is the BackButton. The BackButton class was more of a reference to a physical button on a mobile device and therefore will not interfere with the architecture of the project.

Even with these slight difference, we believe the architecture (functionality) of the system will not change and it will be very close to what our conceptual diagram indicates it will be. The core implementations of the classes match with the conceptual classes. There may be more than one actual class that make up a conceptual class but at least the general idea and structure of the project is implemented the way we originally believed it to be.

Tool used: ObjectAid UML Explorer - This tool allowed us to extract the classes used in the project into a UML class diagrams. This tool made it much easier to extract all attributes and functions and place them in a class diagram rather than going through each class and finding each and every attribute and function. Furthermore, the tool provided all the types of relationships between the classes such as associations, generalizations, and interface implementations.

Relationship between BubbleSprite and BubbleManager class.

```
public class BubbleSprite extends Sprite {
        …
        private BubbleManager bubbleManager;
        …
          // Class constructor used when restoring the game state from a bundle.
```

```
        public BubbleSprite(…, BubbleManager bubbleManager, …) {
                …
                this.bubbleManager = bubbleManager;
                …
        }
         //Class constructor used when creating a launched bubble.
        public BubbleSprite(…, BubbleManager bubbleManager, SoundManager …) {
                …
                this.bubbleManager = bubbleManager;
                …
        }

         //Class constructor used when initializing a new level.
        public BubbleSprite(… , BubbleManager bubbleManager, …) {
                super(area);
                …
                this.bubbleManager = bubbleManager;
                …
                addToManager();
        }

        public void addToManager() {
                bubbleManager.addBubble(bubbleFace);
        }
        public void removeFromManager() {
                bubbleManager.removeBubble(bubbleFace);
        }
}

public class BubbleManager {
        …
        public void addBubble(BmpWrap bubble) {
                countBubbles[findBubble(bubble)]++;
                bubblesLeft++;
        }

        public void removeBubble(BmpWrap bubble) {
                countBubbles[findBubble(bubble)]--;
                bubblesLeft--;
        }
        …

}
```

Code Smells and Possible Refactorings

The FrozenBubble class has a **Bloater code smell**. We notice a lot of the Bloaters' symptoms such as Large Class, Data Clumps and Primitive Obsession. For instance, there's a Data Clumps code smell from the fields adsOn to targetMode. We can use Extract Class and Move Method to replace these fields by a single field representing the Preferences class.

1. We select the fields from adsOn to targetMode and extract a class that we call Preferences.
2. In the Preference class, we change the attributes' protection level to private.
3. Since there are set and get methods for each attribute that has been extracted into the Preference class, we get create a get and a set method (i.e. getPreference(Preference preference), setPreference(Preference preference)) in the FrozenBubble class.
4. We then move the getters and setters of each attribute into the Preference class.

In the FrozenBubble class, a **Message Chain code smell** can be observed in the onPreparingOptionsMenu() method. Using Extract Method and Move Method will allow us to isolate the chain and move it the class from which the chain starts.

1. We are going to use Extract Method to extract the chain findItem(x).setVisible(y) into a method called setItemVisible(item x, bool y);
2. We are going to use Move Method to move the method into the Menu class. This way, the FrozenBubble class only calls one Menu class method.

Still in the FrozenBubble class, we notice that there is a **Switch Statement code smell** in the onOptionsItemSelected(MenuIntem item) method. We can use Replace Type Code with State/Strategy and Replace Conditional with Polymorphism to remedy to this.

1. Using Replace Type Code with State/Strategy, we create a class EditorType and subclasses of the EditorType class because it is an object of the class Editor that acts differently in the switch statement. Each subclass will correspond to a case in the switch statement.

```
class EditorType {}
class EditorNewGame extends EditorType {}
```

2. In the EditorType class, we create an abstract getTypeCode() method.
3. In the Editor class, we create an instance variable type of belonging to the EditorType class and different constant values for each switch statement case. We also create a setType() method which will call a factory method from the EditorType class to create the appropriate

subclass object.

```
class Editor {
	EditorType  type;
	public final static int MENU_COLORBLIND_ON 1;
	public final static int MENU_NEW_GAME 8;
	int setType(int value) {
		type = EmployeeType.newType(value);
	}
}

class EditorType {
	newType(int value) {
		switch(value) {
			case MENU_NEW_GAME:
				return new EditorNewGame();
				break:
			…
		}
	}
}
```

4. In each subclass, we override the getTypeCode() method to return one of the constant values respective to the appropriate type of subclass.

```
class EditorNewGame extends EditorType {
	int getTypeCode () {
		return Editor.MENU_NEW_GAME;
	}
}
```

5. In the Editor class, we create a getType() method which returns one of the constant values respective to the appropriate type of subclass by calling getTypeCode() of the variable type.

```
int getType() {
	type.getTypeCode();
}
```

6. Using Replace Conditional with polymorphism, we create an abstract method onEventItemSelected() in the EdirtorType.
7. In each subclass, we override the onEventItemSelected() method with the appropriate class behavior.
8. We change the switch statement method to call the onEventItemSelected() of the type variable of the Editor object, and it will call the appropriate subclass method.

```
editor.type.onEventItemSelected;
```

9. Since we now have a Message Chain code smell, we will use Extract Method to create a class in the Editor class as follows:

```
void onEventItemSelected {
      this.type.onEventItemSelected;
}
```

The BubbleSprite class also has a **Bloater code smell**. We can observe Data Clumps, Large Class, Long Parameter List, Long Method and Primitive Obsession. We notice the Data Clumps for the variables moveX and moveY, which are always used together, as well as realX and realY. We will use Extract Class to replace the fields with proper Coordinates object.

1. We select the moveX and moveY fields and extract a class that we call Coordinates with the fields as attributes. The new field in the BubbeSprite class is called moveCoordinates.
2. We replace the fields realX and realY with the field realCoordinates and replace the variables realX and realY with realCoordinates.x and realCoordiantes.y respectively.

Similarly, in FrozenBubble, there's a **Data Clumps code smell** from the fields adsOn to targetMode. We can use Extract Class to replace these fields by a single field representing the Preferences class.

1. We select the fields from adsOn to targetMode and extract a class that we call Preferences.
2. In the Preference class, we change the attributes' protection level to private.
3. Since there are set and get methods for each attribute that has been extracted into the Preference class, we get create a get and a set method (i.e. getPreference(Preference preference), setPreference(Preference preference)) in the FrozenBubble class.
4. We then move the getters and setters of each attribute into the Preference class.

In the "SplashScreen.java", we believe the addHomeButtons() can be considered to have a **Long Method code smell**. Since the method is rather long, it can create a lot of confusion when trying to understand exactly what the code is doing. The addHomeButtons method is actually creating various home screen buttons, and also adding them to the main splash screen for the android app. To make this method smaller and the button creation task easier, we believe it would help to use a factory pattern and Replace Type Code with Subclass as a refactoring.

To create a button factory we would follow these steps:

1. We create an abstract class named "Button"
2. We would put a single method (called "create") in this class
3. Create a class corresponding to each button type. In this case there would be a class

2pGame, 1pGame and Options

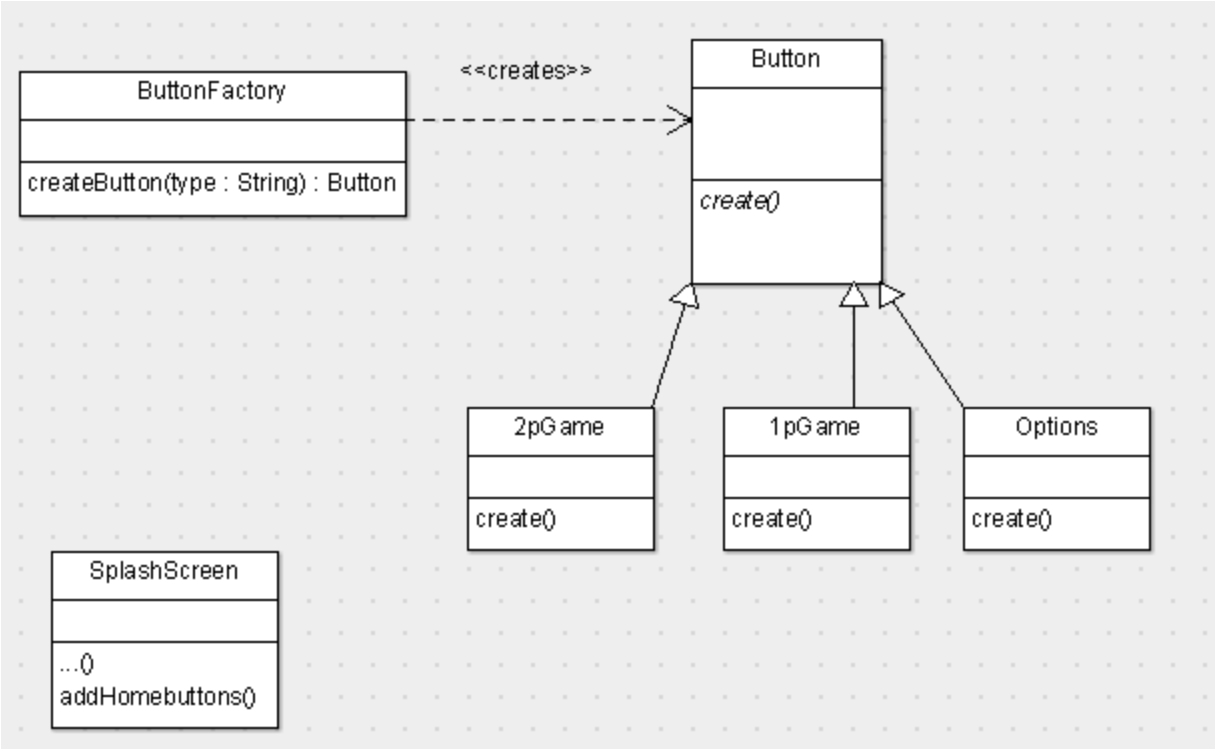
4. Have each of the classes from step 3 extend the button class
5. For the 2pGame, 1pGame and Options classes, we would add a create method (public void create()) and within it we take all the creation functions dealing with that button from addHomeButtons and move them into the newly made create method.
6. Create a class ButtonFactory
7. In ButtonFactory, make a method which uses an if statement to compare a string passed from driver (later step) to a declared string (corresponding to each of the buttons, so a string named “2p”, “1p” and “Options” for example). Depending on which statement gives a true value, we would return an object of that button type.
8. In addHomeButtons(), we add a declaration which will call the button factory to create the specific button object, and then we add a line to call the specific buttons create method.

   E.g.: Button 2pGame= ButtonFactory.createButton(”2P”)

   2pGame.create()

By following these steps, we break up the large method addHomeButtons in SplashScreen.java and create smaller classes doing the same job of creating buttons. This makes it much easier to locate and edit any code dealing with buttons, as each button has its own specific class and is not just clumped together in one method. Adding a new button would be simple as well since all that would need to be done is to add a new class and a few declarations/functions.

UML of ButtonFactory



Original Button Creation Code

```
private void addHomeButtons() {
    // Construct the 2 player game button.
    Button start2pGameButton = new Button(this);
    start2pGameButton.setOnClickListener(new Button.OnClickListener(){

      public void onClick(View v){
        buttonSelected = BTN2_ID;
        // Process the button tap and start/resume a 2 player game.
        startFrozenBubble(2);
      }
    });
    start2pGameButton.setText("Player vs. CPU");
    start2pGameButton.setTextSize(TypedValue.COMPLEX_UNIT_PT, 8);
    start2pGameButton.setWidth((int) (start2pGameButton.getTextSize() * 10));
    start2pGameButton.setHorizontalFadingEdgeEnabled(true);
    start2pGameButton.setFadingEdgeLength(5);
    start2pGameButton.setShadowLayer(5, 5, 5, R.color.black);
    start2pGameButton.setId(BTN2_ID);
    start2pGameButton.setFocusable(true);
    start2pGameButton.setFocusableInTouchMode(true);
    LayoutParams myParams1 = new LayoutParams(LayoutParams.WRAP_CONTENT,
                                              LayoutParams.WRAP_CONTENT);
    myParams1.addRule(RelativeLayout.CENTER_HORIZONTAL);
    myParams1.addRule(RelativeLayout.CENTER_VERTICAL);
    myParams1.topMargin = 15;
    myParams1.bottomMargin = 15;

  // Add view to layout.
    myLayout.addView(start2pGameButton, myParams1);
```

(This code was edited, showing the creation of just one button. We would multiply this amount of code by the amount of buttons needed).

Long class -- There exist only a few classes doing most of the work and this resulted in classes with large amounts of attributes, several of which are public constants, lots of unrelated methods, and several very long methods. This also leads to several other smells which will be described below and refactoring for those should help reduce the length of several classes.

Primitive Obsession -- The reason for the large amount of attributes comes from a large number of primitives being used, mostly of type **int** to store ID numbers, or type codes, to be used when setting up the environment. For example, the *FrozenBubble* class contains 30 such attributes while the FrozenGame class contains 16 plus several other attributes needed by each class. By using Replace Type Code with Class, the amount of primitive type attributes being used can be greatly reduced as

well as reducing the overall length of the classes.

Inappropriate Intimacy -- A side effect from building classes that try to do everything is that they can sometimes end up retaining data used primarily by other classes. In this case, *FrozenBubble,* the main class responsible for setting up the environment in which the game runs, also holds the set of ID numbers that will be used by *SoundManager*, the class responsible for playing the sounds in the game. Therefore, whenever a sound needs to be played a called has to be made to *SoundManager* and the ID held by *FrozenBubble* is passed as a parameter. This creates an unnecessary coupling between the two classes.

```
From FrozenBubble.java:
  public final static int SOUND_WON     = 0;
  public final static int SOUND_LOST    = 1;
  public final static int SOUND_LAUNCH  = 2;
  public final static int SOUND_DESTROY = 3;
  public final static int SOUND_REBOUND = 4;
  public final static int SOUND_STICK   = 5;
  public final static int SOUND_HURRY   = 6;
  public final static int SOUND_NEWROOT = 7;
  public final static int SOUND_NOH     = 8;
  public final static int SOUND_WHIP    = 9;
  public final static int NUM_SOUNDS    = 10;


Sample call - from FrozenGame.java:
  soundManager.playSound(FrozenBubble.SOUND_NOH);
```

```
From SoundManager.java - Constructor:
  public SoundManager(Context context)
  {
    this.context = context;
    soundPool    = new SoundPool(4,
AudioManager.STREAM_MUSIC, 0);
    sm = new int[ FrozenBubble.NUM_SOUNDS ];

    sm[ FrozenBubble.SOUND_WON ] = _;
    sm[ FrozenBubble.SOUND_LOST ] = _;
    sm[ FrozenBubble.SOUND_LAUNCH ] = _;
    sm[ FrozenBubble.SOUND_DESTROY ] = _;
    sm[ FrozenBubble.SOUND_REBOUND ] = _;
    sm[ FrozenBubble.SOUND_STICK ] = _;
    sm[ FrozenBubble.SOUND_HURRY ] = _;
    sm[ FrozenBubble.SOUND_NEWROOT ] = _;
    sm[ FrozenBubble.SOUND_NOH ] = _;
    sm[ FrozenBubble.SOUND_WHIP ] = _;
  }
```

The main refactoring that needs to be applied here is *Move Field* to move the IDs into the *SoundManager* class so that it may have all the data it needs available from within itself.