Assignment 5 Bust-a-Move

By
Calvin Nhieu
Jason Xie
Winer Bao
Justin Segond
Maurice Willemsen

TI2206 Software Engineering Methods

Supervisor:
Dr. A. Bacchelli

Teaching Assistants:
F. C. A. Abcouwer
R. van Bekkum
M. M. Beller
T. Boumans
A. Ferouge
J.E.Giesenberg

Exercise 1 - Anonymous peer suggestions

The following suggestions are implemented:

App.java

- Make all screens an attribute for consistency.
  All screens are now attributes for the App class

Bubble.java

- Let randomColor() method of Bubble return a Color.
- In the move method, the switch statements contains cases which do not change to state. Remove the NEW, LANDED and POPPED cases.

Collision.java

- You could divide this into the two classes; a collision and a pop class.
  All pop and drop related methods have been moved to a PopBehaviour class.

PowerUp.java

- You do not have to create all the methods again which have already been created in the superclass. Remove those methods.

Bubble.java

- There are two constructors, one with and one without x coordinate. In the one without that coordinate, you set x to zero. Remove the constructor without x coordinate.

NameScreen.java and NamesScreen.java

- We suggest you make a super class for this.

VictoryScreen.java, DefeatScreen.java and PausedScreen.java

- We suggest you make a super class for this as well.

Player.java

- Validate the input for the setName method.

The following suggestions are not implemented:

PowerUp.java

- *The apply method in line 66 of the PowerUp class contains an if-statement. This if-statement uses the "lazy and" (&&). You could easily convert this into an if-statement without the lazy and.*
  This is already done in the latest release.

Game.java

- *Instead you could use only one javadoc for all the attributes.*
  This is already done in the latest release.

GameState.java

- *We do not see the purpose of using the abstract class GameState.*
  It contains the names to makes the programming easier to understand instead of a bunch of number. Especially useful when switching to states with enterState()

SoundHandler.java

- *The different play sound methods can be refactored to one method with two parameters: pitch and name of sound.*

This does not improve anything in our opinion. We keep it like this to focus on other more serious problems.

Log.java

- *Make the logger a singleton.*
  This is already done in the latest release.

Iterator.java

- *Use the Iterator interface that is already provided by Java.*
  To be filled in by Calvin.

Formatting

This is already improved a lot in the latest release.

Cannon.java

- *The main thing that is remarkable while taking a first look at the cannon class is a huge method of about 70 lines of code., named update(). Try splitting this method into multiple methods based on its functionality.*
  This was indeed a big point, but it was already fixed in the latest release.

GameData.java

- *The two methods that must be implemented in the GameData class have not been implemented. One of these two methods is empty. We do suggest finishing the implementation of this design pattern.*

This is because a GameData object only contains one player info so the amount of players is not necessary to be known. That's why one method is empty. The screens, which implements the Observer as well, needs to know the amount of players so that's why the method is needed.

Exercise 2 - Software Metrics - inCode

InCode .result file is located under the /res/ folder in git.

Design flaw 1

Message chains in Game

To be filled in by Calvin

Design flaw 2

Arena is a data class. The Arena class "glues" a Collision and a BubbleStorage object/class together and provides getter and setter methods. The only unique functionality the Arena class provides is drawing a rectangle on the screen. It also

contains several unused methods: getHeight(), setxPos() and setyPos(). Another problem the class has is the fact that there is a lot of external uses of public data. Cannon.getNextBubble(), GameData.update(), BubbleStorage.removeBubble(), Collision.checkWallCollision(), Collision.landBubble(), OBomb.pop and RowBomb.pop() uses getBubbleStorage(), getCollision(), getWidth(), getxPos(), getyPos() methods of the Arena class.

In short, the Arena class contains too much data and too many classes try to access them.

To solve this problem, the Arena class will be removed. Data will be redistributed among the Collision and BubbleStorage classes. The unique functionality of Arena is moved to the GameData class.

Design flaw 3
App is a data class. The App class gave static access to several different other classes. The classes it gave static access to where PauseScreen, VictoryScreen and DefeatScreen. This is bad because any class can then access these classes and change things in them or call methods.
To fix this, the getter methods were removed from these screens from the App class. Then the rest of the code was refactored to not need the methods anymore. The Game object now holds a list of PlayerObserver objects, which it registers to the players whenever a GameData object (which hold a player) is created. When this was refactored the design flaw was no longer there.

Exercise 3 - Option menu feature

To start off we created a new requirements document in which we highlighted every noun.

RDD requirements
To change certain settings of the game, a option menu is needed. This menu is accessed by clicking a button called “option” located in the main menu. When the option button is pressed, a clicking sound is played. The program then shows the OptionScreen with two buttons; one mute button and one back button. The text displayed in the mute button depends on the current sound state of the game. When the game is already muted, the text reads unmute and vice versa. The back button contains the text “Back” and returns to the main menu screen. A clicking sound is played when on of the buttons are pressed.

Later, we went on to divide every noun in a category.

Conceptual entities:
OptionScreen

We also removed some of the nouns because they meant the same as others or they already exist:

One word for one concept:

Option menu -> OptionScreen
Option Button -> Button
Clicking sound -> SoundHandler
Mute Button -> Button
Back button -> Button
Text -> TextField
Main menu screen -> MainMenu

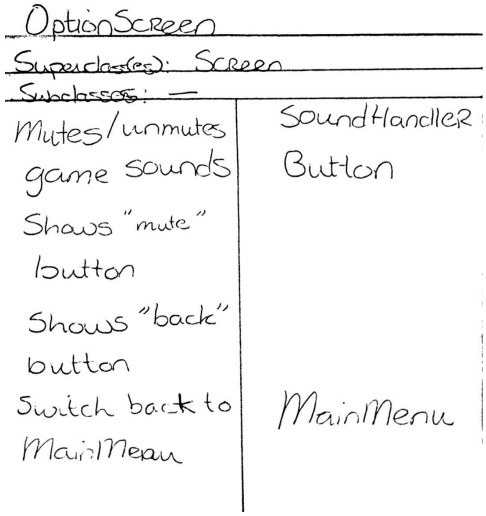
Misleading subjects:

Settings

Already existing classes:

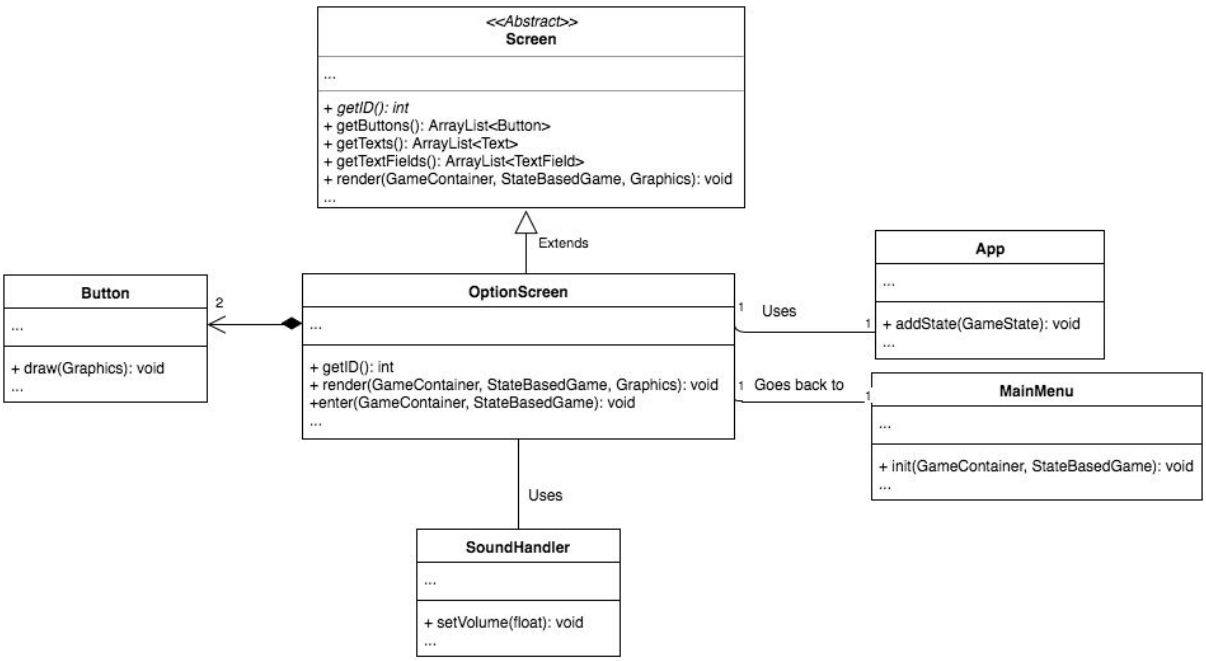
Game, MainMenu, Button, SoundHandler, TextField

The next step is to define the responsibility of each class and the collaborations between each class. All the derived classes from this RDD redesign session are documented. The CRC cards show each classes with its corresponding responsibilities and collaborations.

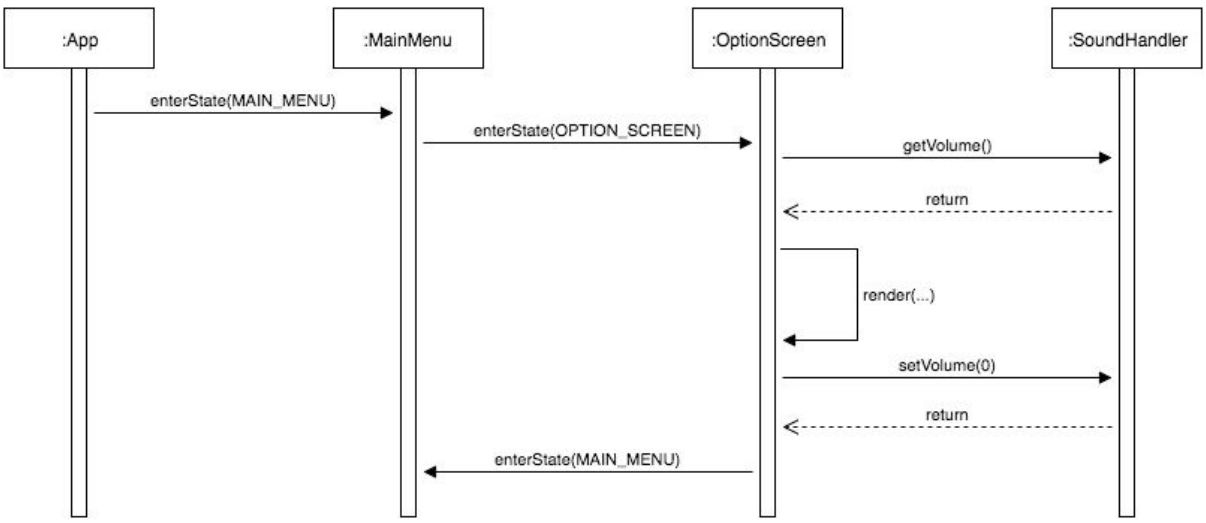


The OptionScreen extends the Screen class to inherent its methods and implement its abstract methods. The Screen class contains algorithms for rendering buttons and other elements. The OptionScreen call also uses to Button objects to mute sounds and to go back to the main menu

The following design was implemented:



When the Option button in the main menu is clicked, the game state is changed to the the option screen. The option screen then calls enter(...) to get the sound state of the game. Render(...) will then draw the buttons and texts on the screen. When the mute button is pressed, the setVolume(...) of the Soundhandler is called. When the back button is pressed, the enterState(...) is called to return to the main menu. The sequence diagrams below show the methods calls show and return from the OptionScreen:



The following requirements are set for the Option menu implementation:

Option menu

Functional requirements:

Must have:

- A clickable button in the main menu allows the user to enter the option menu.

- The button contains a text of “Options”.
- After the button is clicked, the OptionScreen is displayed.
- A mute/unmute button is displayed in the OptionScreen.
- The text inside the button changes from mute to unmute and vice versa depending on the state (whether the game is muted or not).

Should have:

- The user is able to change the color theme of the bubbles in the OptionScreen.

Could have:

- The user is able to change the volume level in the OptionScreen.

Won’t have:

- The user won’t be able to invert all colors of the game.

Non-functional requirements:

The test coverage of the OptionScreen must be at least 60%.

The implementation and testing must be completed before Friday Oct 28th 2016 5.55PM.