Assignment 1 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

1.1

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

RDD requirements

The bust-a-move game is a shooting game where the player controls a cannon on the arena. The objective of the game is to remove all bubbles off the arena by connecting 3 or more bubbles of the same color.

The game starts up with a start screen. The player can click anywhere inside the game window to enter the main menu. The main menu contains buttons to start the game and to see the high scores.

When the start button is pressed, the game screen is displayed. The game screen consists of an arena with some random bubbles, a cannon, player name, player score, combo and level number.

The arena is a hexagonal grid that can hold a maximum of 8 by 12 bubbles. A border indicates the outer limits of the arena. Bubbles cannot exist outside of this border. When the arena becomes empty, the player wins the game. However, the player loses the game when the arena is full. The player may also pause the game at any time during the gameplay.

The cannon is located at the bottom center of the game screen. This cannon shoots a bubble out of its barrel when the player presses the UP arrow key. The cannon can be angled < 90 degrees left and right by pressing the LEFT and RIGHT arrow key. If the player remains idle for 5 seconds, the cannon will automatically shoot a bubble to continue game's progress. The bubble currently loaded in the cannon is rendered at the bottom of the cannon. The next bubble is displayed at the right of the cannon. The color of the next bubble can only be one of the colors still present on the arena. A new row of randomly generated bubbles is added at the top of the arena when the 10 bubbles are launched.

Bubbles travel in a straight line and bounce off the borders of the arena. Bubbles hold their position when they collide with other bubbles or the top of the arena. Bubbles of the same color will pop when a shot bubble forms a connection of 3 or more bubbles of the same color. Any bubbles hanging from these popped bubbles drop off the arena.

The player's name and score are displayed on the left top of the game window. Along with the current combo and level number.

Each level increases in difficulty by introducing tricky patterns and color combinations.

The game displays a win screen when the level is completed. This win screen shows the score of the player. The game displays the lose screen when the player fails the level. The lose screen also show the player's score. Both screen contain a button to restart the game or to return to the main menu.

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

Physical objects:
bubbles, cannon

Conceptual entities:
Game, player, arena, button, border, grid

Adjectives:
start screen, main menu, win screen, lose screen, game screen, start button

Attribute value:
color, position, name, score, combo, level

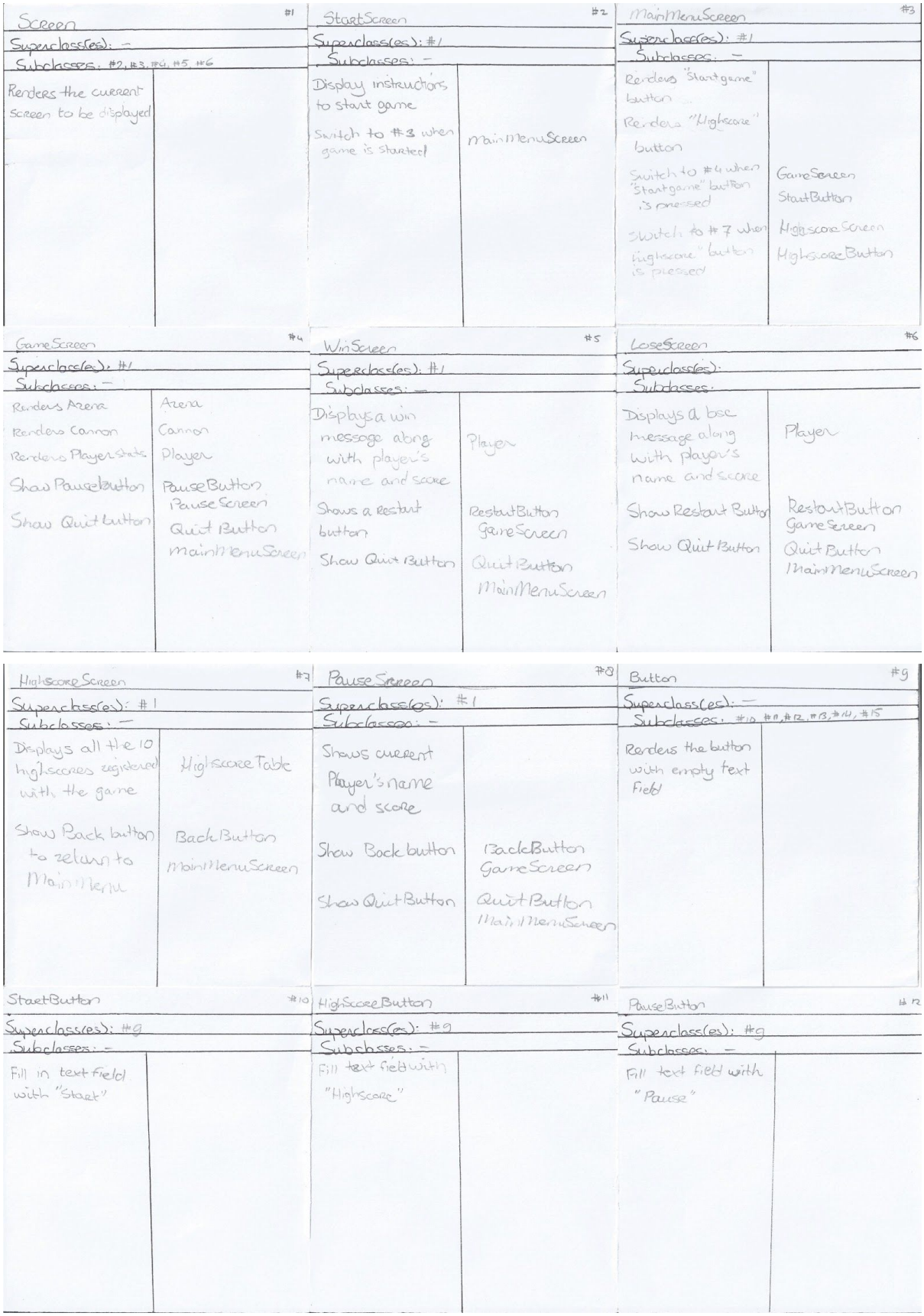
We also removed some of the nouns because they meant the same as others:
One word for one concept:
Outer limits > border
Barrel > cannon
Center > position
Patterns > level
Difficulty > level
Window > screen
Connection > collision
Highscores > score

Misleading subjects:
Objective, UP arrow key, LEFT arrow key, RIGHT arrow key, color combinations, row, progress

The next step is to define the responsibility of each class and the collaborations between each class. During this process, new classes came to light when it felt like they are missing or when a class to many responsibilities. These classes are Screen, HighscoreScreen, HighscoreButton, PauseButton, RestartButton, QuitButton, BackButton, HighScoreTable, Angle, NextBubble, Collision, Speed, Point, Level1, Level2, Level3.

All the derived classes from this RDD redesign session are documented. The CRC cards show each classes with its corresponding responsibilities and collaborations.

Derived classes with their respective responsibilities and collaborations



Screen #1	
Superclass(es): –	
Subclasses: #2, #3, #4, #5, #6	
Renders the current screen to be displayed	

StartScreen #2	
Superclass(es): #1	
Subclasses: –	
Display instructions to start game	
Switch to #3 when game is started	MainMenuScreen

MainMenuScreen #3	
Superclass(es): #1	
Subclasses: –	
Renders "Startgame" button	
Renders "Highscore" button	
Switch to #4 when "Startgame" button is pressed	GameScreen StartButton
Switch to #7 when "highscore" button is pressed	HighscoreScreen HighscoreButton

GameScreen #4	
Superclass(es): #1	
Subclasses: –	
Renders Arena	Arena
Renders Cannon	Cannon
Renders Player state	Player
Show PauseButton	PauseButton PauseScreen
Show Quit button	QuitButton MainMenuScreen

WinScreen #5	
Superclass(es): #1	
Subclasses: –	
Displays a win message along with player's name and score	Player
Shows a restart button	RestartButton GameScreen
Show Quit Button	QuitButton MainMenuScreen

LoseScreen #6	
Superclass(es):	
Subclasses:	
Displays a lose message along with player's name and score	Player
Show Restart Button	RestartButton GameScreen
Show Quit Button	QuitButton MainMenuScreen

HighscoreScreen #7	
Superclass(es): #1	
Subclasses: –	
Displays all the 10 highscores registered with the game	HighscoreTable
Show Back button to return to MainMenu	BackButton MainMenuScreen

PauseScreen #8	
Superclass(es): #1	
Subclasses: –	
Shows current Player's name and score	
Show Back button	BackButton GameScreen
Show QuitButton	QuitButton MainMenuScreen

Button #9	
Superclass(es): –	
Subclasses: #10, #11, #12, #13, #14, #15	
Renders the button with empty text Field	

StaetButton #10	
Superclass(es): #9	
Subclasses: –	
Fill in text field with "Staet"	

HighScoreButton #11	
Superclass(es): #9	
Subclasses: –	
Fill text field with "Highscore"	

PauseButton #12	
Superclass(es): #9	
Subclasses: –	
Fill text field with "Pause"	

RestartButton #13

Superclass(es): #9

Subclasses: —

Fill text field with "Restart"	

QuitButton #14

Superclass(es): #9

Subclasses: —

Fill in text field with "Quit"	

BackButton #15

Superclass(es): #9

Subclasses: —

Fill in text field with "Back"	

Player #16

Superclass(es): —

Subclasses: —

Contains the name and score of the player playing	Name Score
If this player has a top 10 scores, this player is stored in #19	HighscoresTable

Name #17

Superclass(es): —

Subclasses: —

Stores a string that represents a name of e.g player	Player

Score #18

Superclass(es):

Subclasses:

Stores a double value that represents a score of e.g player	Player

HighScoreTable #19

Superclass(es): —

Subclasses: —

Stores the top 10 scores + name	Player

Arena #20

Superclass(es): —

Subclasses: —

Renders the bubbles in the Bubble Storage	BubbleGrid
Renders the background of the Arena	
Renders the Border of the arena	Border

Border #21

Superclass(es): —

Subclasses: —

Ensures that the bubble stays inside of the arena	Bubble Arena

BubbleGrid #22

Superclass(es) —

Subclasses: —

Stores bubbles in the arena in a hexagonal grid	Bubble Arena

Collision #23

Superclass(es): —

Subclasses:

Detect a collision between bubbles in the arena	Bubble Arena BubbleGrid
Report back the location of the collision	Point

Cannon #24

Superclass(es): —

Subclasses: —

Renders the barrel	Barrel
Get the next Bubble	NextBubble
Stores and renders the current bubble in the cannon	Bubble
Fires a bubble	
Rotates the Barrel	
Automatic fire after idle	

Angle	#25
Superclass(es): –	
Subclasses: –	
Stores the angle of the barrel	Barrel

Barrel	#26
Superclass(es):	
Subclasses:	
Draws the barrel Calculate the angle	Angle

NextBubble	#27
Superclass(es): #28	
Subclasses: –	
Determines the color of the bubble should be based on the colores available in bubble storage	BubbleGrid

Bubble	#28
Superclass(es): –	
Subclasses: –	
Renders the bubble according to its color, speed and position	Color Speed Point

Color	#29
Superclass(es): –	
Subclasses: –	
Contains the Color of the Bubble	Bubble

Speed	#30
Superclass(es):	
Subclasses:	
Contains the speed of bubble	Bubble

Point	#31
Superclass(es):	
Subclasses:	
Contains the x and y position of bubble. e.g. Can be used by other classes aswell	Bubble Arena

Level	#32
Superclass(es): –	
Subclasses: #33, #34, #35	
load a preset bubbles in BubbleGrid	BubbleGrid

Level 1	#33
Superclass(es): #32	
Subclasses: –	
Preset bubble config of Level1	

Level2	#34
Superclass(es): #32	
Subclasses: –	
Preset bubble config of Level2	

Level3	#35
Superclass(es): #32	
Subclasses: –	
Preset bubble config of Level3	

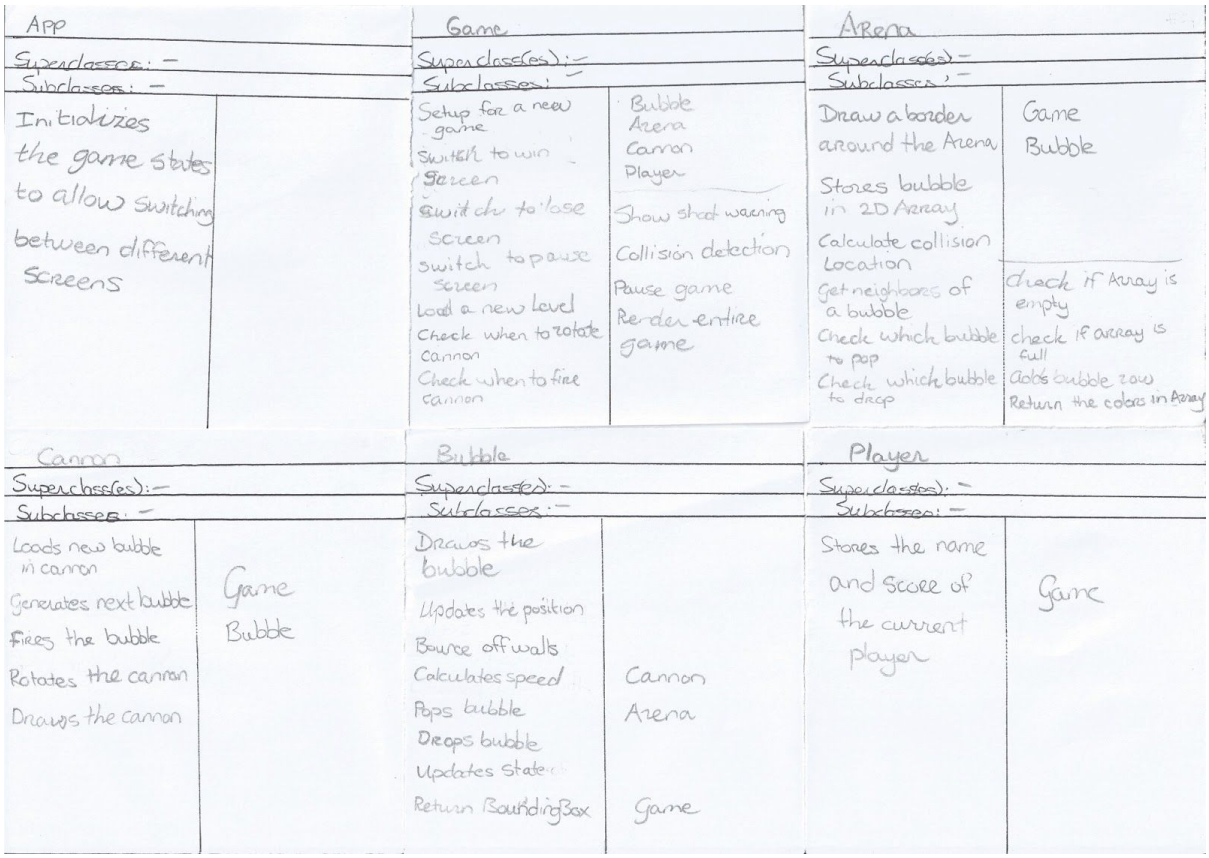
Combo	#36
Superclass(es): –	
Subclasses: –	
Calculates the combo	Score

RDD classes compared to working game classes

Many more classes are derived using the Responsibility Driven Design compared to the classes in the working game. Classes in the working game have multiple responsibilities and the code structure can become chaotic at times. The RDD classes are limited to one responsibility as much as possible which keep the code organized and easy to understand.

A HighscoreScreen class is added in the RDD classes to display the highscores of the game. The Button class is subcategorized into more specific classes. The name and score of a Player class is stripped out into it's own classes. The new HighscoreTable classes hold 10 highscores values that are displayed on the HighscoreScreen. The Arena class keeps the rendering methods but loses the bubble storage. This has become its own class. A Border class is created to perform bouncing of bubbles. The collision mechanism is also moved into a separate class. The Cannon class is divide into three classes now: Cannon, Barrel and NextBubble. The Color, Speed and Point class are associated to the Bubble class. The new Level class allows new levels to be added later. The Combo class was not available before.

1.2



The program works on a state based mechanism. Each state has its own init(), update(), render() and getID() methods. The App class tells the program which states are available and with which state to start the program. One of these state is Game.

The Game class controls the information flow to Bubble, Arena, Cannon and Player. It sets up a new game when a new level is started. When the player wins, loses or pauses the game, the Game class switches the state of the program. The class checks for user input to

determine if certain cannon actions need to be performed. If the player remains idle for too long, it will notify the cannon to fire once. Whilst a bubble is traveling, a collision detection algorithm is engaged. Once a collision is detected, the action is passed onto the Arena class to handle this collision. Last but not least, it signals all the objects associated with this class to render itself.
The Arena class has a 2D array in the form of a LinkedList of Bubble arrays to store all the bubbles on the Arena. When a collision occur, the location of the bubble is determined and translate to a location in the 2D array. This bubble is then saved at this location. Next, the Arena determines if any bubbles need to be popped or dropped. If so, the pop() or drop() method of these bubbles are called. The Arena class also has a function to return neighbors of an bubble needed for the pop/drop algorithm. Methods to check if the array is empty or full help determine if the player has lost or won. Another responsibility is to add a new row bubble at the top of the arena when the player has shot 10 times. To determine which color of bubble to load in the cannon, the class has a method to return the colors present in the 2D array. Finally, the class also draws a border around the arena.
The cannon class must load a new bubble after each time it has shot the previous bubble. The color of the next bubbles is generated based on the colors present in the arena. Of course, the cannon can shoot a bubble and rotates its cannon. The draw() method renders the cannon on the screen.
The Bubble class renders a bubble on the screen, updates the position of a moving bubble, detect a wall and bounces off of it, calculates the speed of the bubble, update its state and returns a bounding box for collision detection. There are pop and drop method to start its respective animation.
The Player class stores the name and the score of the current player and reports these back to the game class.

1.3
These class are less important since they do not directly contribute to the gameplay or are used to enhance the code readability and/or user experience.

The BubbleFactory class

- This is a utility class to facilitate the creation of bubbles. This class basically facilitates creation of random color bubbles. It's functionality can be easily duplicated in the Bubble class itself.
- This class should be removed

The Score class

- The class is currently simply storing an int value with methods to modify this value.
- Later in the development, a scoring algorithm will be added to this class.
- This class is therefore not modified.

The Log class

- This is a utility class for logging all actions to the console.
- This class is required by the TA and should therefore remain unmodified.

The Button class

- This class draws a standard button that can be reused in many situations where a simple button is needed. The text inside the button can be easily changed.
- This reduces the amount of copy-pasting and keep the program easy to maintain.
- This class is therefore not modified.

The GameState class
- The class maps String constants to integer numbers of the screens.
- This prevents magic number violation and the code is more human-readable.
- This class is therefore not modified.

The PausedScreen class
- The class shows the player that the game has been paused
- This will stay unchanged unless an alternate design is available.

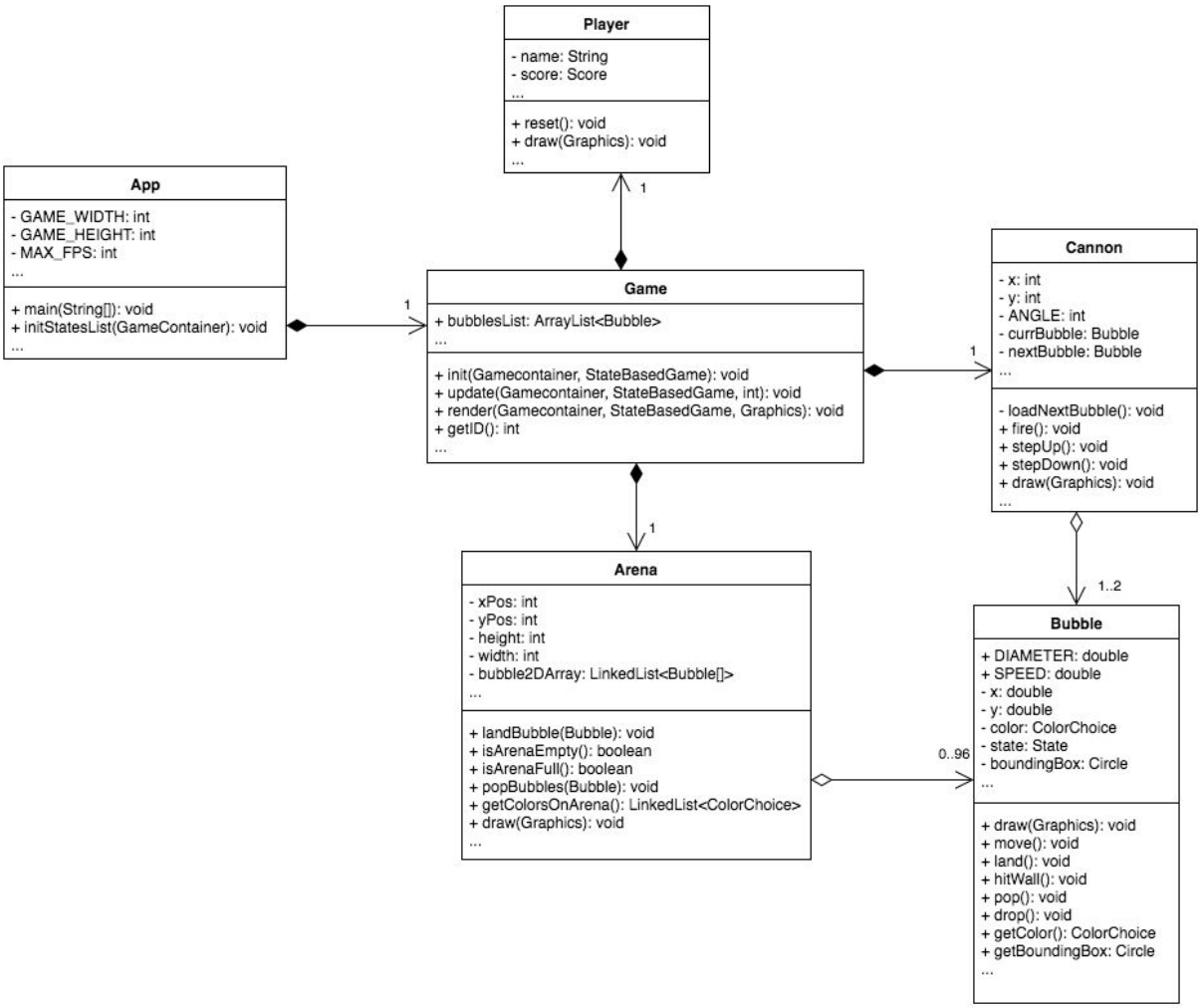
The VictoryScreen class
- The class shows that the player has won the game
- This will stay unchanged unless an alternate design is available.

The DefeatScreen class
- The class shows that the player has lost the game.
- No better replacement are designed yet. The message must be displayed to the player.
- This class is therefore not modified

The MainMenu class
- This class allows the player to choose between different menu options including start game and show highscores.
- This will stay unchanged unless an alternate design is available.

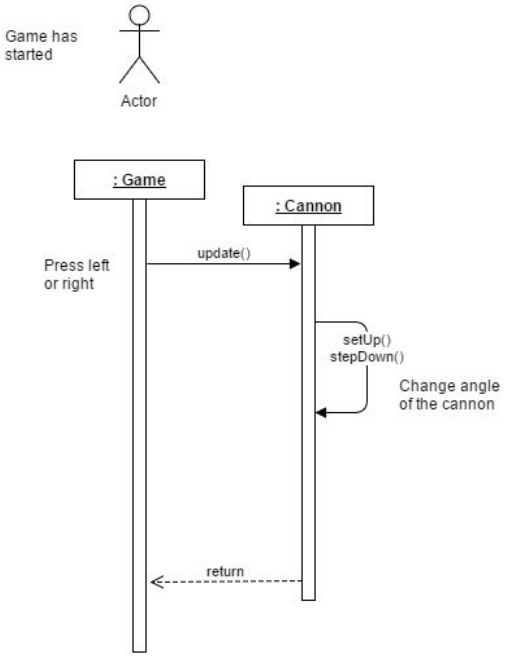
The StartScreen class
- This screen is displayed when the player start the program.
- This will stay unchanged unless an alternate design is available.

1.4

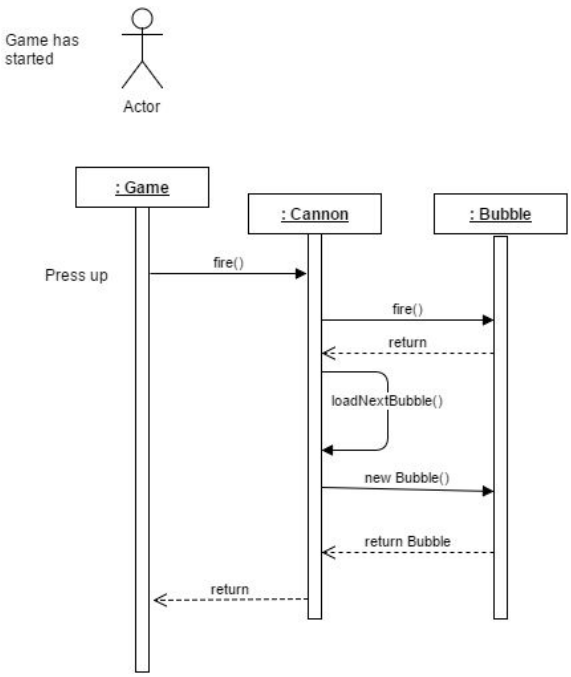

Player
- name: String
- score: Score
...
+ reset(): void
+ draw(Graphics): void
...
1
App
- GAME_WIDTH: int
- GAME_HEIGHT: int
- MAX_FPS: int
...
+ main(String[]): void
+ initStatesList(GameContainer): void
...
1
Game
+ bubblesList: ArrayList<Bubble>
...
+ init(Gamecontainer, StateBasedGame): void
+ update(Gamecontainer, StateBasedGame, int): void
+ render(Gamecontainer, StateBasedGame, Graphics): void
+ getID(): int
...
1
Cannon
- x: int
- y: int
- ANGLE: int
- currBubble: Bubble
- nextBubble: Bubble
...
- loadNextBubble(): void
+ fire(): void
+ stepUp(): void
+ stepDown(): void
+ draw(Graphics): void
...
1
Arena
- xPos: int
- yPos: int
- height: int
- width: int
- bubble2DArray: LinkedList<Bubble[]>
...
+ landBubble(Bubble): void
+ isArenaEmpty(): boolean
+ isArenaFull(): boolean
+ popBubbles(Bubble): void
+ getColorsOnArena(): LinkedList<ColorChoice>
+ draw(Graphics): void
...
0..96
1..2
Bubble
+ DIAMETER: double
+ SPEED: double
- x: double
- y: double
- color: ColorChoice
- state: State
- boundingBox: Circle
...
+ draw(Graphics): void
+ move(): void
+ land(): void
+ hitWall(): void
+ pop(): void
+ drop(): void
+ getColor(): ColorChoice
+ getBoundingBox: Circle
...

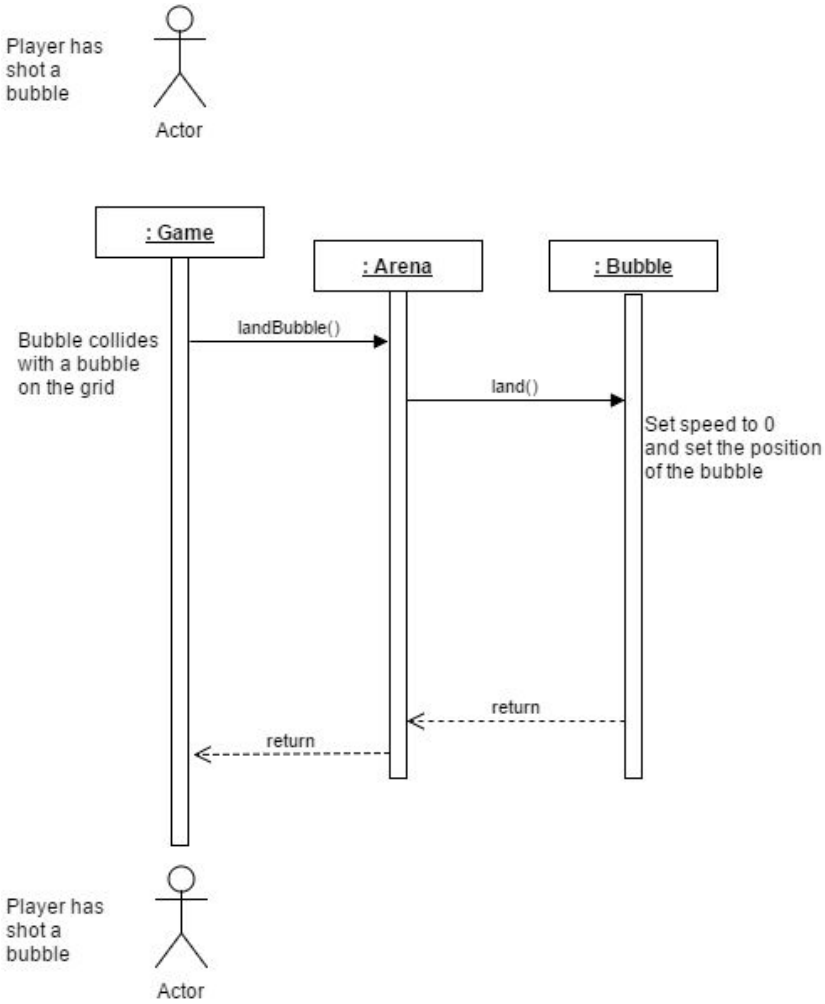
1.5

Scenario: user presses left or right during the game

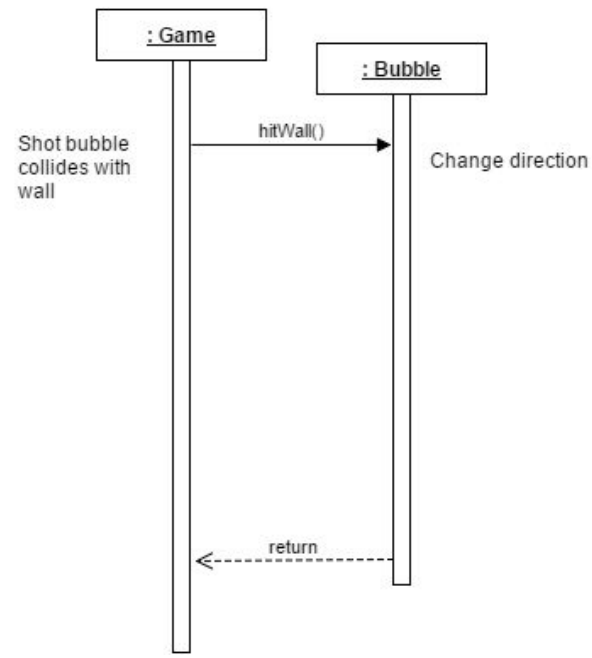


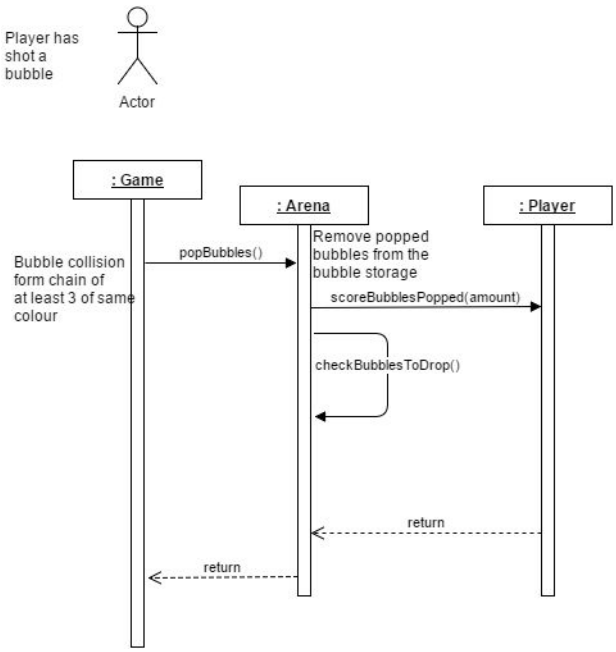
Scenario: user presses up




Player has shot a bubble
Actor
: Game
: Arena
: Bubble
Bubble collides with a bubble on the grid
landBubble()
land()
Set speed to 0 and set the position of the bubble
return
return

Player has shot a bubble
Actor


: Game
: Bubble
Shot bubble collides with wall
hitWall()
Change direction
return



Exercise 2

2.1

Aggregation is a type of association between two classes, class A and class B, where A uses B and B may conceptually exist without the existence of A. For example, a room may contain people, however a person exists without the concept of a room. In contrast, composition is a type of aggregation where A has a (and hence also uses) B, however B can not conceptually exist without the existence of A. For example, the association between a house and a room is a composition since there is no concept of a room without the existence of a house.

In the project, the following class pairs demonstrate an aggregation relationship: Arena uses Bubble, Cannon uses Bubble, and StartScreen, MainMenu, Game, PausedScreen, DefeatScreen, VictoryScreen use Button. These aggregation relationships are displayed in Table 2.1.

Table 2.1: Bust-A-Move Aggregation Class Associations

Class A	uses	Class B
Arena		Bubble
Cannon		Bubble
StartScreen, MainMenu, Game, PausedScreen, DefeatScreen, VictoryScreen		Button

Arena uses Bubble to store the pattern of bubbles the player needs to clear, and Cannon uses Bubble to allow the user to shoot bubbles at the arena. Since Bubble may exist independent of the existence of Arena and/or Cannon, these are a aggregation relationships. The classes representing the game's states (StartScreen, MainMenu, Game, PausedScreen, DefeatScreen, VictoryScreen) use Button to create menus and options to interface the user with the game. The current hierarchy suggests that these relationships are aggregations, however, it may be favorable to create an abstract superclass representing a Screen entity. This would introduce a composition relationship between Screen and Button since Screen uses Button to create menus, and the concept of a Button only exists within the context of a Screen.

The following class pairs demonstrate a composition relationship: Game has an Arena, Player, Bubble, and Cannon, and Player has a Score. These relationships are illustrated in Table 2.2.

Table 2.2: Bust-A-Move Composition Class Associations

Class A	has a	Class B
Game		Arena, Player, BubbleFactory, Bubble, Cannon
Player		Score

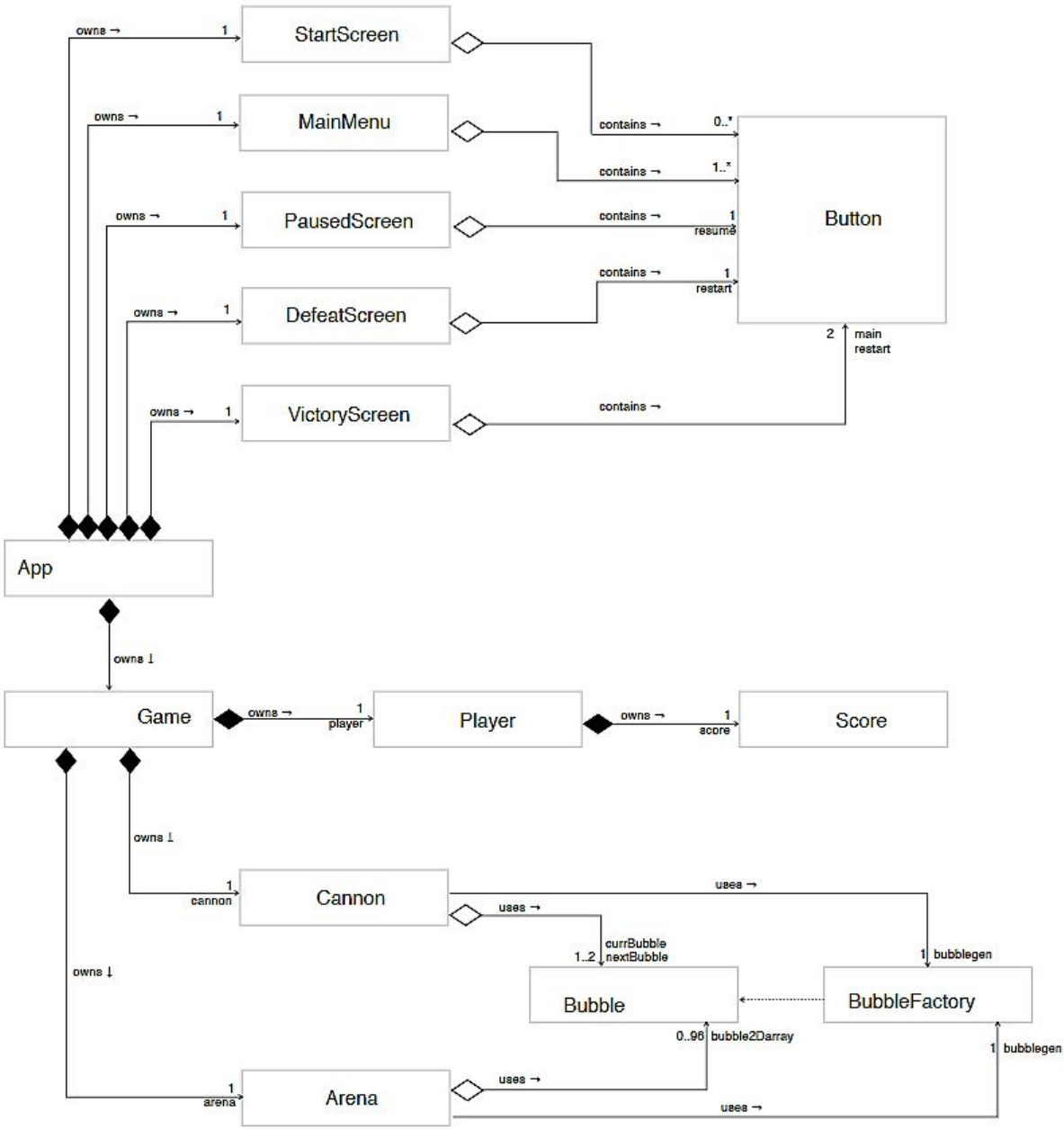
The Game class represents an instance of the game and manages the entities required to play the game. These entities are the Arena, Player, Bubble, and Cannon classes. These classes are owned by the Game class and do not exist outside of the context of a Game instance. Thus, the Game class exhibits a composition relationship with the game entities Arena, Player, Bubble, and Cannon.
The Player maintains a Score object that exists in particular within the context of a Player. A Score object represents a Player's score accumulated by playing the game. Therefore, the Player and Score class demonstrate a composition association as well.

2.2
There are no parameterized classes in the source code. A parameterized class is a class that accepts a type variable(s) on instantiation that can be used throughout the class. This allows the type of the object to be used within the class to be determined at runtime. Parameterized classes should be used when the type of an object used within a class should be abstracted from the implementation of the class. For example, Java's LinkedList class is a parameterized class that abstracts the type of elements stored in the LinkedList. This provides a single interface that allows developers to create different LinkedLists that each store a different object type. Parameterized classes program to an interface not an implementation.

2.3
As mentioned in question 1, an abstract Screen class may be introduced to encapsulate the behaviors of the StartScreen, MainMenu, PausedScreen, DefeatScreen, and VictoryScreen. This class would introduce a composition relationship between Screen and Button as highlighted in question 1.

Furthermore, based on the Factory Pattern discussed in class, the Bubble class may be extracted into multiple classes that would be instantiated by BubbleFactory. The current BubbleFactory implementation uses overloaded constructors exposed in the Bubble class to instantiate Bubble variants. By implementing this change, the factory implementation is realized and the resulting Bubble classes will be closed to modification, and open to extension. However, due to the lack of Bubble variants, we may reconsider the Factory pattern entirely. The Bubbles produced by BubbleFactory vary only in initialization parameters and not
in class structure. Thus, only one Bubble class exists and the use of the Factory pattern is unnecessary for the scope of the app. It is preferable that the Factory pattern be removed from the class hierarchy.
Figure 2.1 illustrates the UML class diagram representing the app's current implementation.

Figure 2.1 Bust-A-Move UML class diagram



** all fields are private (-)*

Exercise 3

3.1
Refer to program code and log requirements documentation

3.2

This is the document.