Assignment 3 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 - Design Patterns

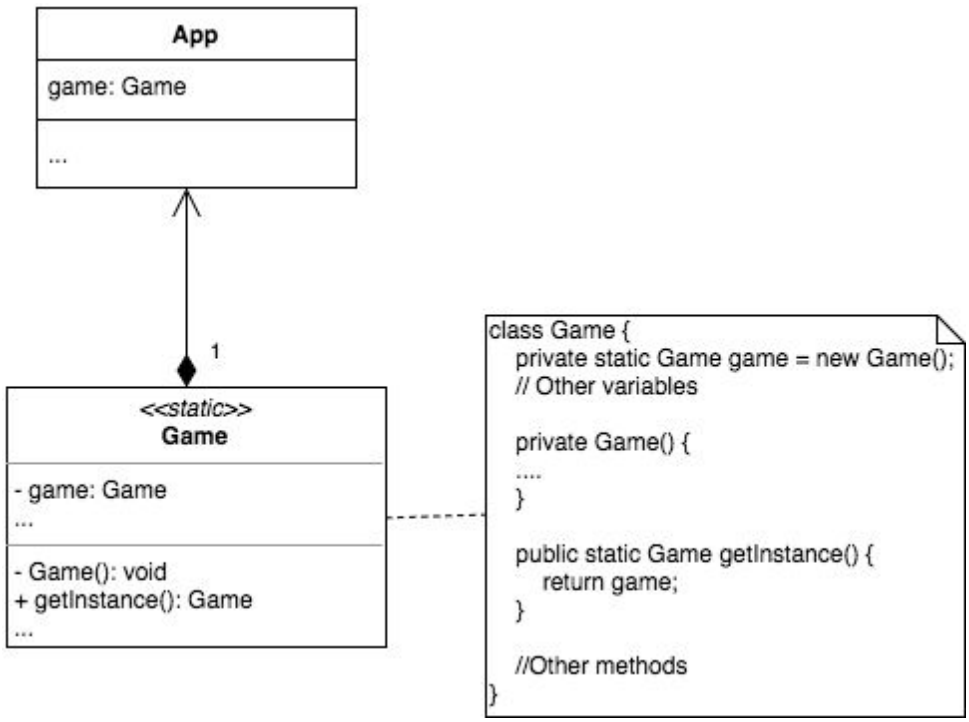
For exercise 1, the team must choose two design pattern discussed during the lectures. The team picked the following patterns: singleton and iterator design patterns.

Singleton pattern

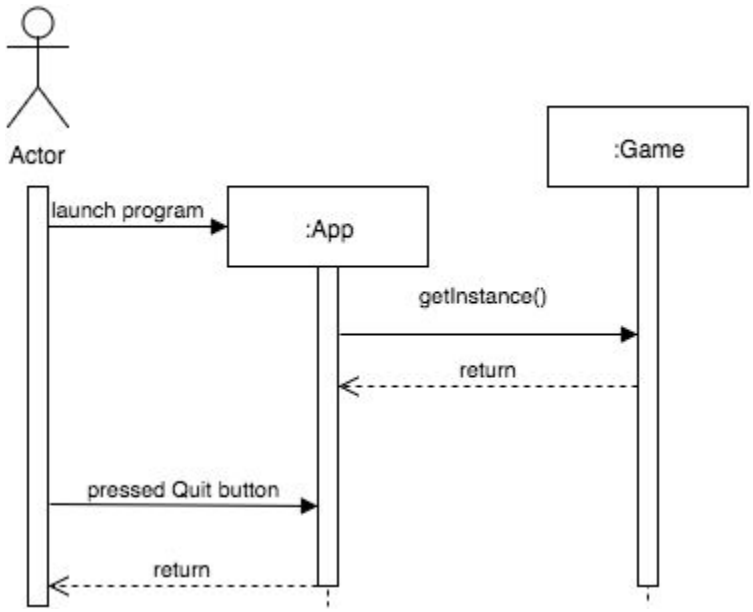
The singleton pattern ensures that only one instance of a Class can exist at a time. The current program creates one instance of the Game class. The design of the program relies on the idea that a game is an entity that contains the “playable” part of the program. The program only contains one game (bust-a-move). Therefore, it should contain one instance of Game. To ensure this, the team decided to use the singleton pattern.

The Game class will have a private constructor instead of the public constructor it has at the moment of writing. To hold the unique instance of the Game class, variable uniqueInstance is added in the class declaration. A static method called getInstance() is added which returns uniqueInstance to the caller. This static method is how other classes can get the unique Game object. The method is thread-safe for future-proofing and it is relatively simple to implement. The pattern uses the eagerly created instance solution because the getInstance() method is expected to be called frequently. The memory penalty can be neglected since the game object is used very early in the program.

The pattern is further explained in the next class diagram.



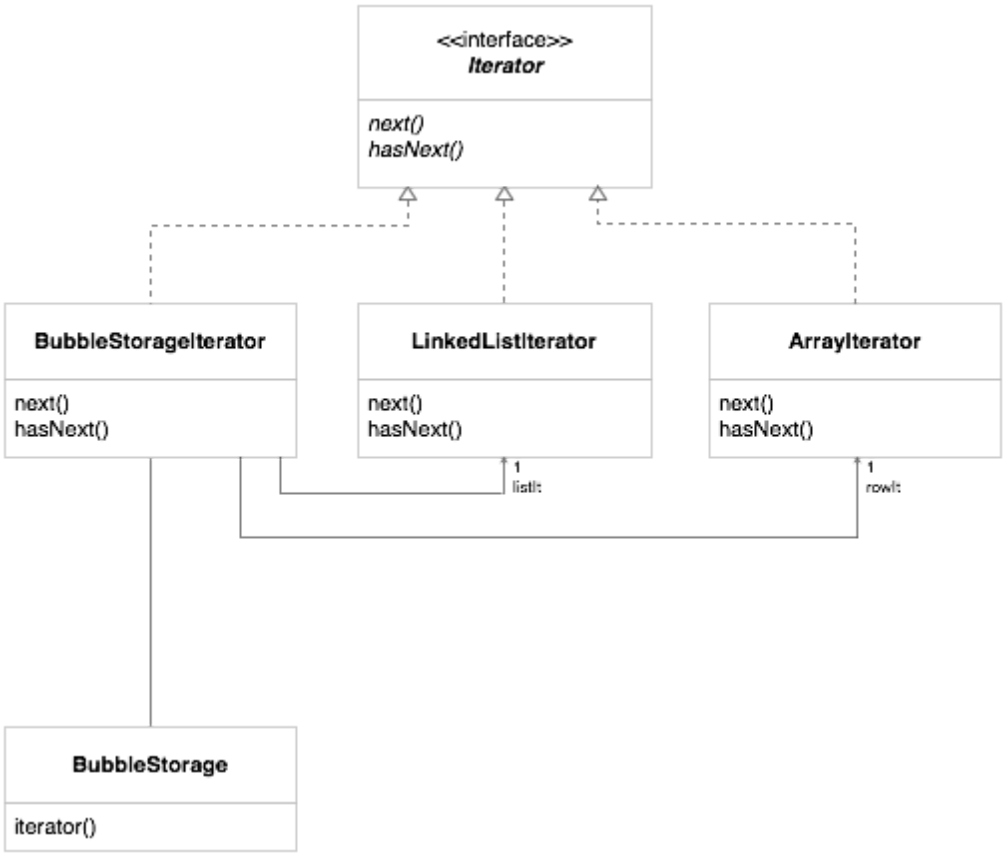
And also in a sequence diagram.



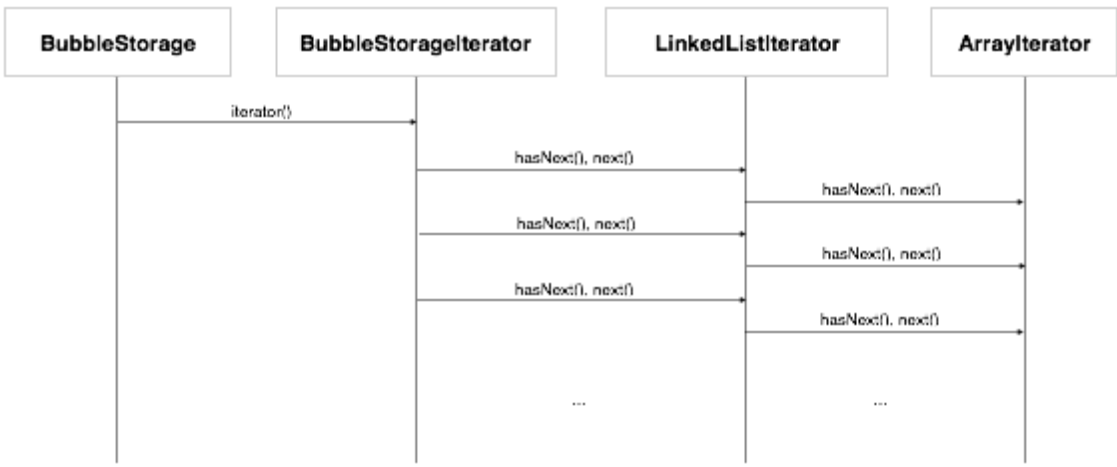
Design Pattern: Iterator Pattern

The iterator pattern provides an interface for accessing the members of an object sequentially, while also abstracting the object's implementation details. The pattern is realized in the Bust A Move game as an iterator used to iterate over Bubble elements that have landed on the arena. These bubbles are stored in the BubbleStorage, a 2D structure composed of a LinkedList of arrays. Iteration over BubbleStorage is required numerous times during a single game loop: collision detection of fired bubbles, detecting popped/dropped bubbles, rendering bubbles. To simplify this repeated process, a BubbleStorageIterator has been implemented that exposes methods to efficiently, and easily iterate over bubbles in a BubbleStorage. The iterator pattern itself is implemented by introducing an Iterator interface that is used to realize three concrete iterators, ArrayIterator, LinkedListIterator, and BubbleStorageIterator.

Class Diagram: Iterator Pattern



Sequence Diagram: Iterator Pattern



Exercise 2 - 2 Player Implementation

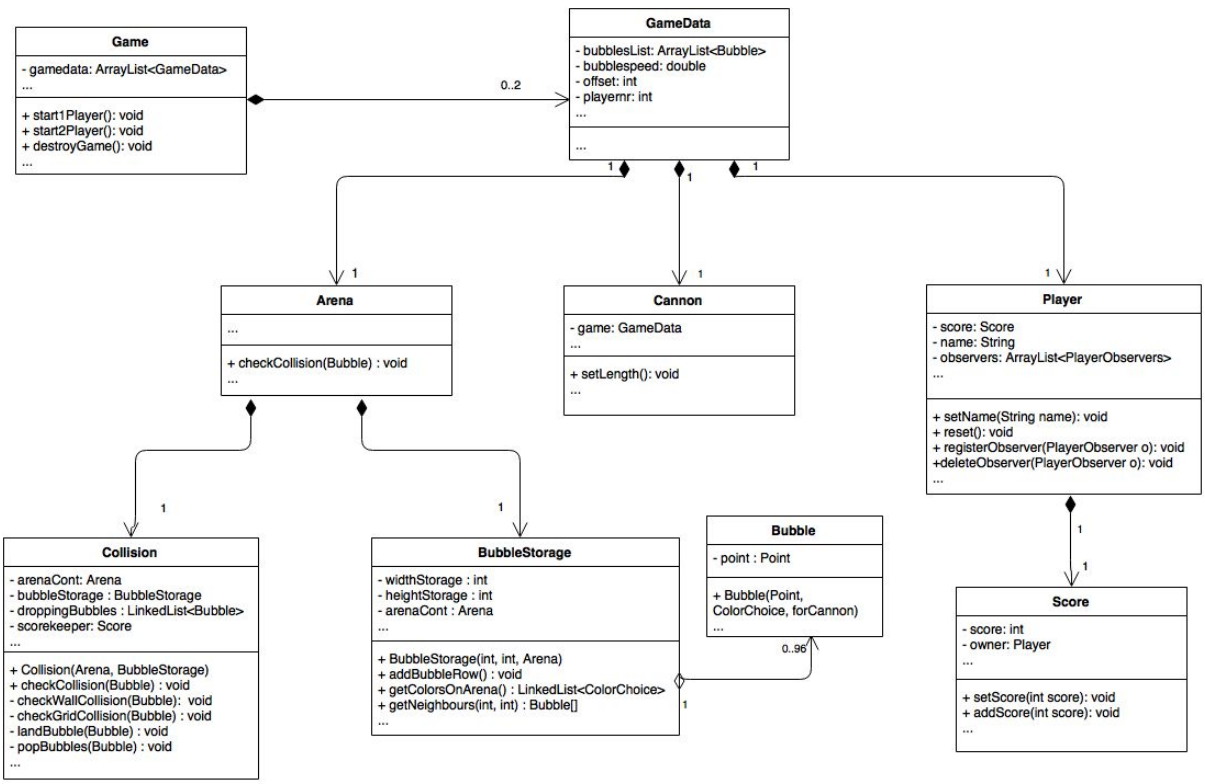
This exercise is about an implementation of the a 2 player mode in the Bustamove game.

The game class used to hold all classes needed for running a game. So the Game class used to hold Arena, Cannon and Player. The instance of the Game class was always available in the App class so there was always, even when not visible, a game running. The change for a multiplayer mode is made as follows: The hold of the game data and the classes needed for the running a game is now stored in the new Class GameData. So GameData represents 1 running game. Now the Game class can hold 0, 1 or 2 GameData objects to depending on whether you play and whether you play with 1 or 2 players. And now when not running a game. The GameData objects can be destroyed and then the Game class does not hold any GameData object. Functions in the Game class can start a 1 player game, a 2 player game or destroy a game.

Two screens are added to the game, these are two screens for adding the names. When selecting in the mainmenu to start a 1 player game. A screen is shown where 1 player name can be inserted. When selecting to start a 2 player game, a screen is shown where 2 player names can be inserted.

The player class is now observable and update to subscribers besides the score also the player name and the id of player so the observers now can know the name and score of every player. A second update method from the observable updates the amount of players which is selected. So the observers know how much players there are playing. GameData is now an observer instead of Game and it responds only to update with it's own playerid.

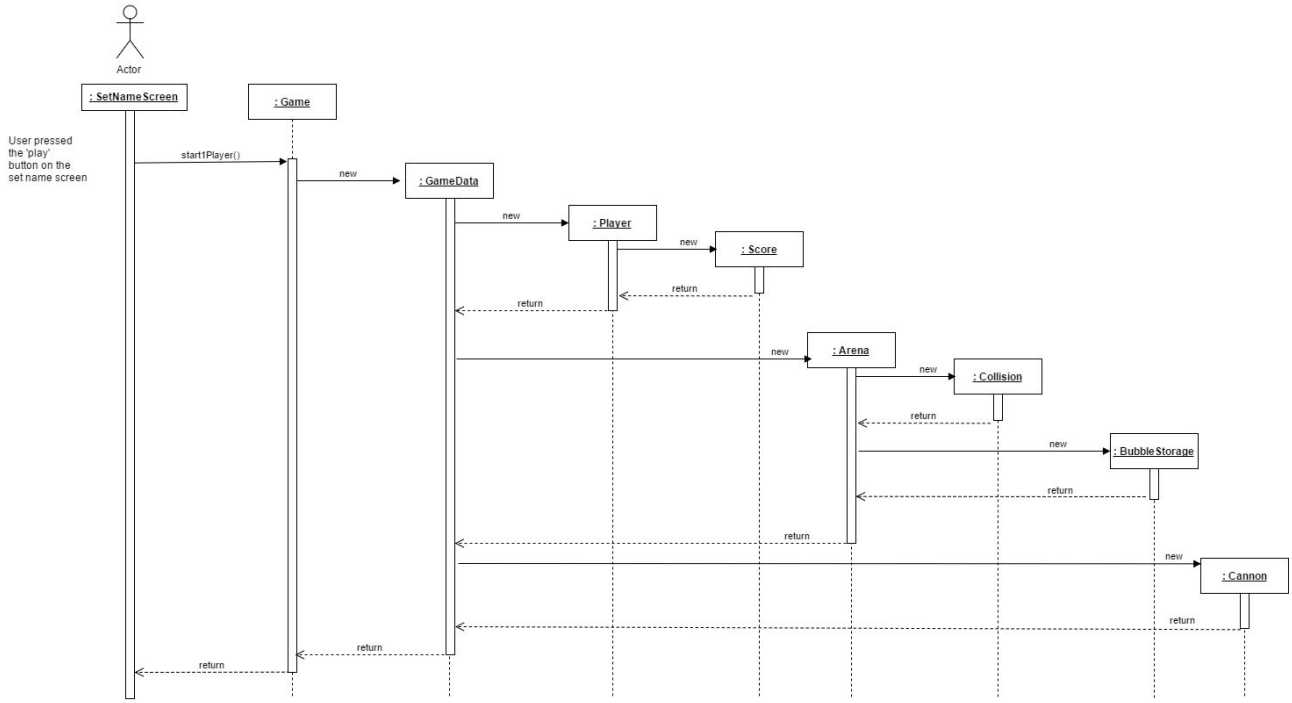
Class Diagram:



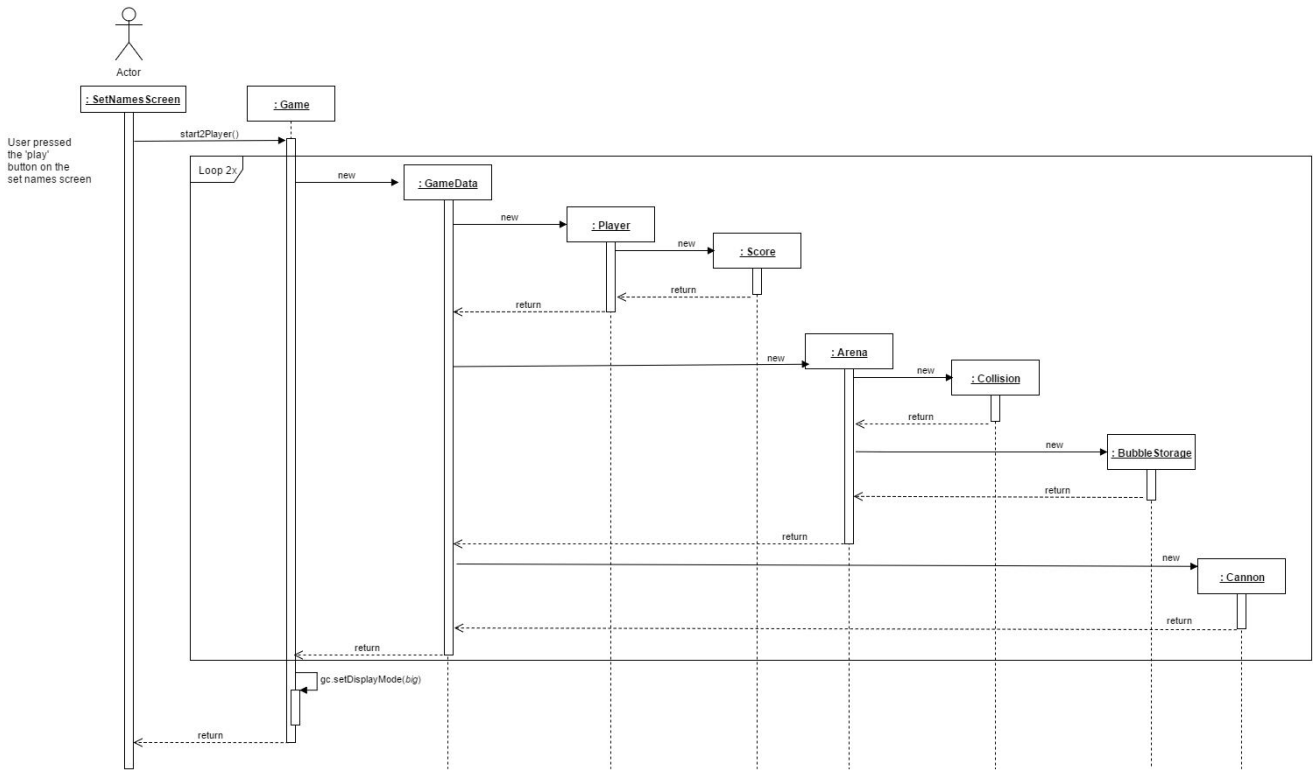
Sequence Diagram:

The sequence diagrams are given on the next page. Bigger sizes are uploaded to the project docs folder.

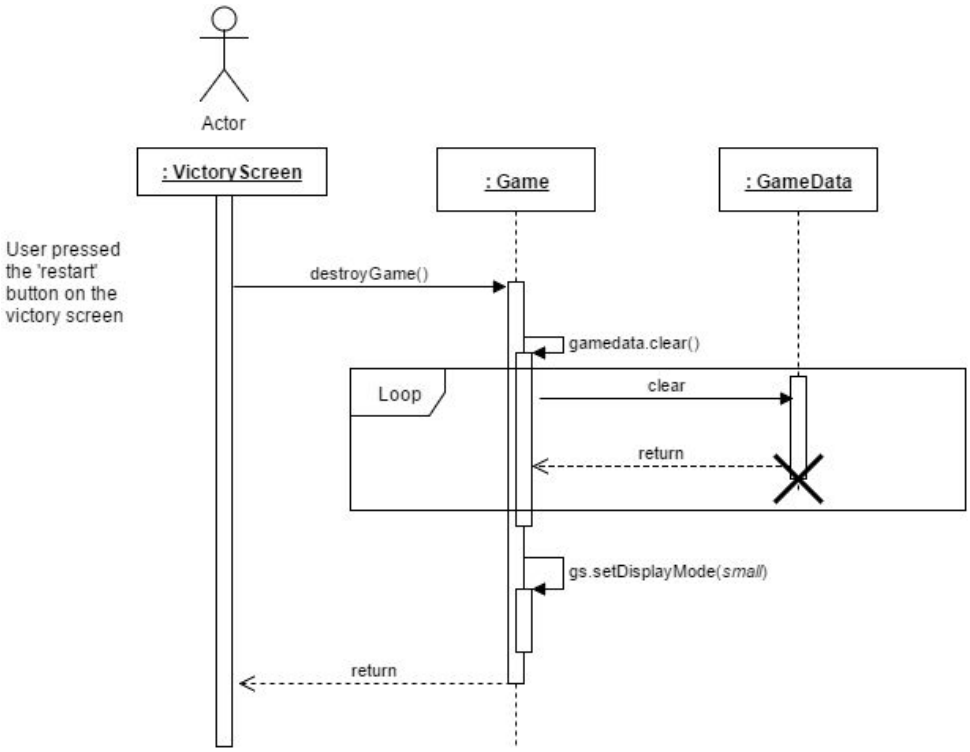
Create a 1 player mode:



Create a 2 player mode:



Destroy current game:



Exercise 3 - Sound Implementation

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

RDD requirements

To enhance the gameplay experience, we would like to add sound effects to the game. Once the program starts, the background music starts playing. A sound plays when any menu button is pressed to provide audible feedback to the user. The game also plays a sound effect when the player rotates or fire the cannon. The bubbles make a popping sound when they are popped from the arena. In addition to this, special bubbles can have a different pop sound compared to the normal bubbles. To congratulate the winning player, an "achievement" sound is played. In contrast, when the player loses, a "failure" sound is played. When a combo system is added to the game, a combo sound can be played when a combo is achieved. When a bubble is dropping, it will not make a sound since this will be very distracting to the player. This is especially true when a large amount of bubbles are dropping at once.

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

Conceptual entities:
Sound

Adjectives:
Gameplay

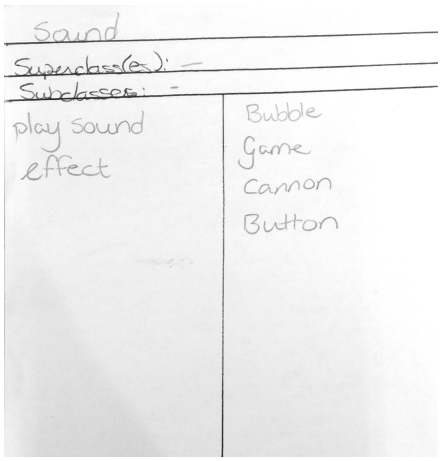
We also removed some of the nouns because they meant the same as others or they already exist:

One word for one concept:
Sound effect -> sound
Music -> sound

Misleading subjects:
Experience, program, feedback, user, combo

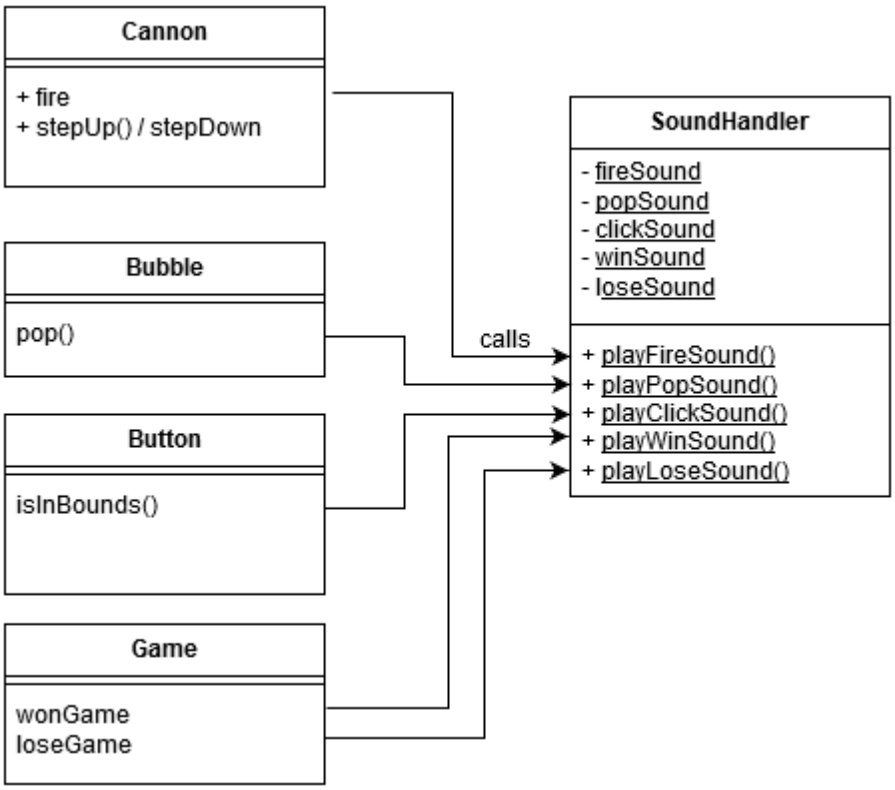
Already existing classes:
Game, Menu button, player, cannon, bubbles, arena

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.

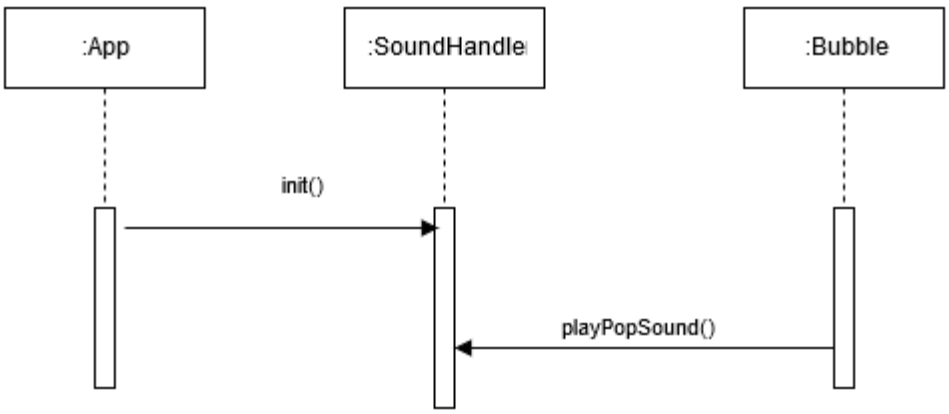


A lot of different classes need access to playing sounds. Therefore we chose to use a global sound handler to handle the playing of all sounds. And we renamed Sound to SoundHandler because there was a conflict with the name Sound of the Slick2D library, which we used for the actual loading and playing of sounds.

The following design was implemented:



First the SoundHandler needs to be initialized. That happens in the App class as shown below. After that the SoundHandler should be able to play all the sounds by calling the right method, as shown for the example of Bubble below;


:App
:SoundHandle
:Bubble
init()
playPopSound()

...