TI 2206 Software Engineering Methods Assignment 1, Exercise 2. September 20, 2016
Calvin Nhieu, Winer Bao, Jason Xie, Justin Segond, Maurice Willemsen

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, BubbleFactory, 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, BubbleFactory, 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, BubbleFactory, 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.

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.

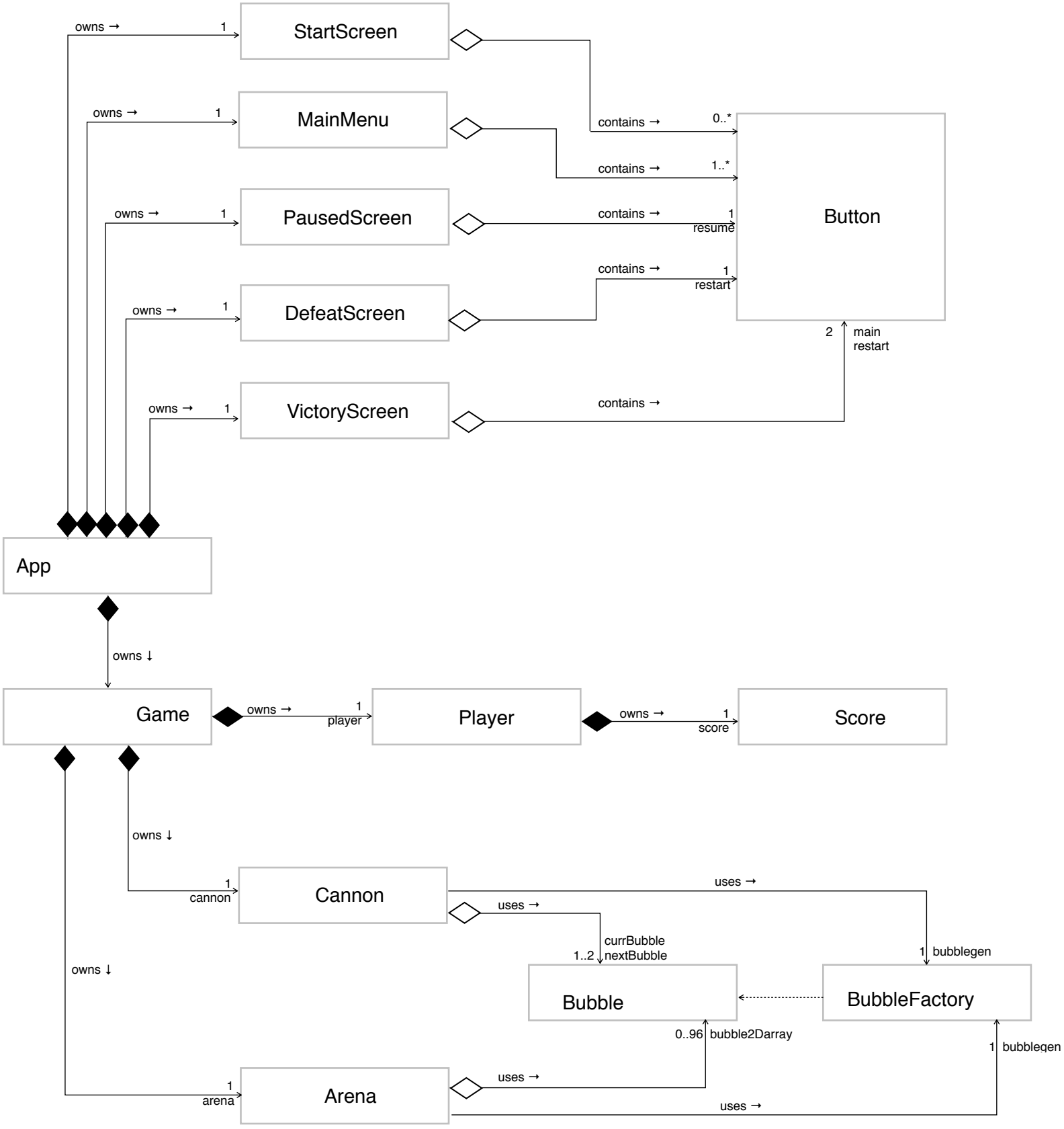
3.

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 also share a composition relationship with 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.

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 (-)*