Exercise 2: DESIGN PATTERS (30 pts)

This exercise will be split into 2 parts, according to the 2 chosen design patters. During sprint 3 we used the strategy and decorator pattern. This time we will use the observer and singleton pattern.

PART 1: Observer pattern (15 pts)

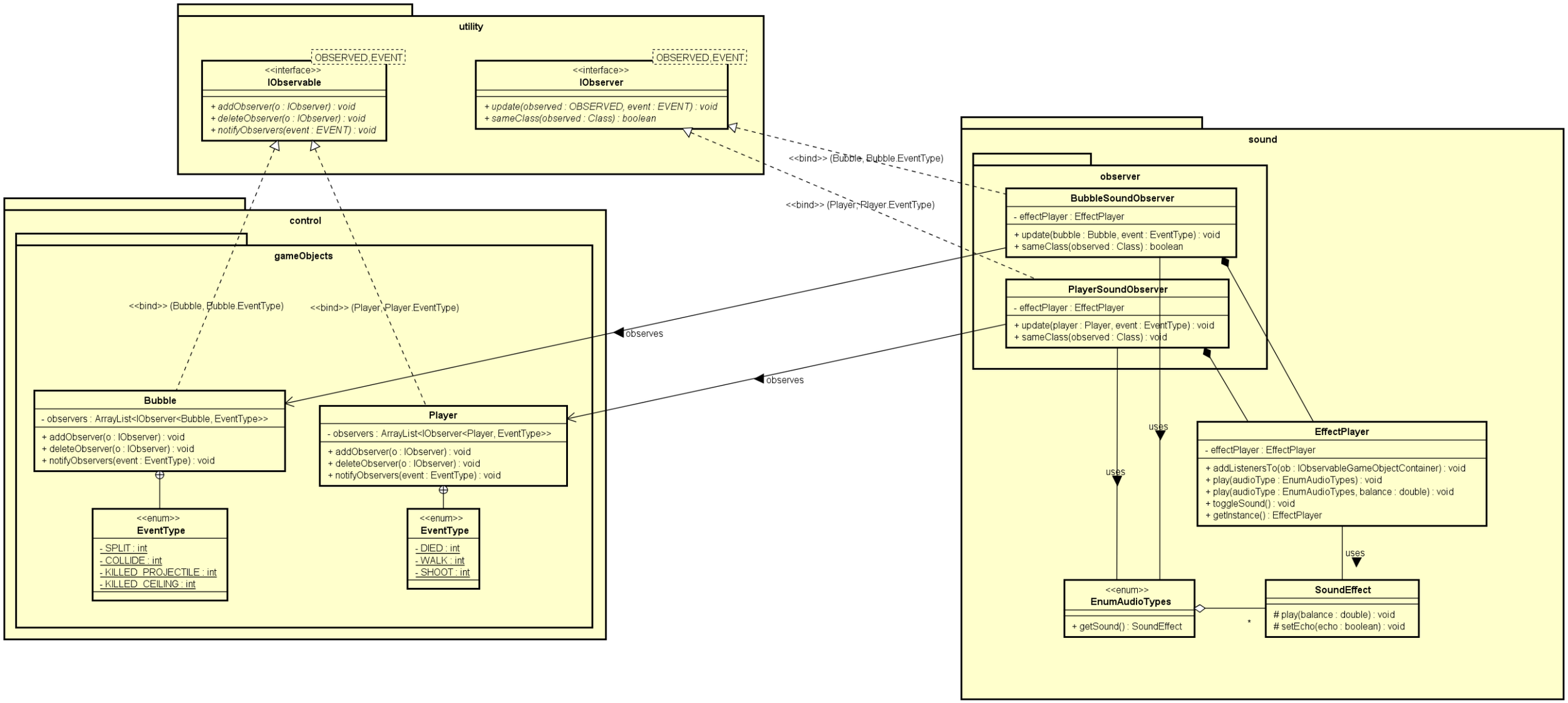
1. Write a natural language description of why and how the pattern is implemented in your code (5 pts)

In our game we have recently added sound effects. These sound effects are played when certain events occur. There is for example a sound effect when the player hits a bubble and when the player dies. To implement this we wanted to create a class which handles the sound effects when an event occurs. The objects which generate the events (for example the player object) shouldn't have to know anything about which sounds have to be played, or even that a sound is played at all. In other words: we don't want the player and the sound effect generator class to be tightly coupled.
We also wanted to determine the balance of the sound based on the position on the screen where the event occurred. Because of this the sound effect player has to be able to request information from the game objects. The game objects still don't have to know anything about the sound generator though.
To make this possible we have decided to use the observer pattern. With the observer pattern we can let the sound effect player class add a listener to each observable game object and then handle all the observed events. For this to be possible the game object classes (i.e. player and bubble) only have to notify their observers of the events which occur.
We will describe how this is implemented in 2 part:
1. The observer:
We have created 2 observer classes: BubbleSoundObserver and PlayerSoundObserver. They will observe the events of the bubble and player and let the EffectPlayer play sounds according to the observed events.
2. The observable:
To make the player and bubble classes observable, we have created an IObservable interface. Every time an event occurs in one of those classes, they will notify all of their observers. As argument they will pass an enum (which indicates the event) and an instance of themselves to the observers. With the event enum the observers can determine which sound to play and with the instance of the bubble/player which is passed they can get extra information if needed. Extra information is for example needed when determining the balance of the sound (for example when an event occurs on the left side of the screen, the left sound should be louder).

An additional nice feature of using the observer pattern is that it will be very easy to add listeners for other purposes. A few example of other cases where a listener could be used are:

- Keep track of the score of the player by listening to the bubble split/bubble destroyed events (needs an additional listener for coin pickups)
- Keep track of the lives of the player by listening to the player died events.
- Log all events with a logger class by listening to all events and printing a log for every event.

2. Make a class diagram of how the pattern is structured statically in your code (5pts)



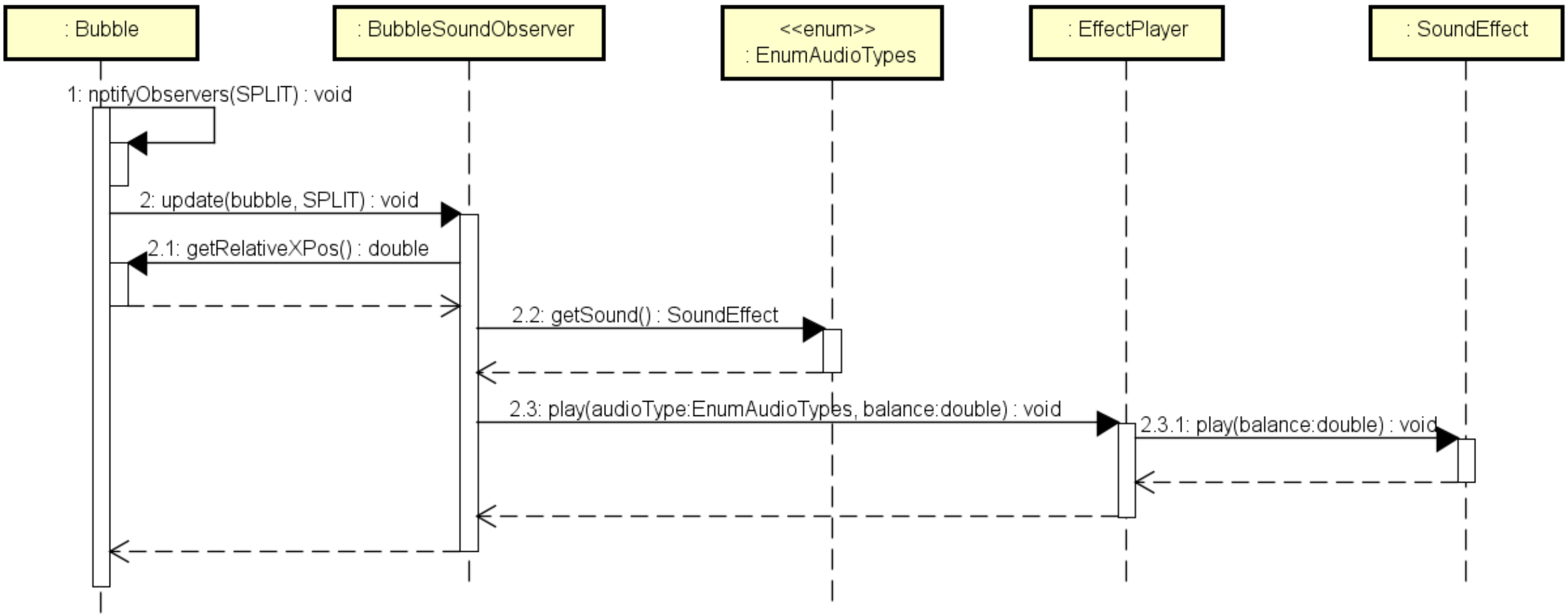
Classes, methods, variables and relations between classes which don't contribute to the observer pattern are omitted to avoid cluttering the diagram.

The generics of the IObservable and IObserver interfaces are used to improve efficiency and for cleaner code:
- Efficiency: each observable class first checks if the observer which is being added is of the correct type (so no player observers are added to the bubbles for example). This is checked by using the sameClass method of the observable interface. We are doing it this way because now the GameObjects class can just add all observers to all observable objects, without checking if it is the right kind of observer. If the GameObjects class tried to add an invalid observer to an observable object, the observable object will simply not add it to its observer list.
- Cleaner code: because we can ensure the type of events which the observer classes will receive with the generics, we will not have to check and cast the type of the received argument in the update method.

When the BubbleSoundObserver or PlayerSoundObserver receives an update event they will use the EnumAudioTypes to get the required audio file and tell the EffectPlayer to play the audio.

3. Make a sequence diagram of how the pattern works dynamically in your code (5 pts)

We will create a sequence diagram of the “bubble is split” event. All other events (including the player events) are handled the same way.



First we notify the observers in the bubble class. After that an update is send to the BubbleSoundObserver. The BubbleSoundObserver now requests the relative x position of the bubble, to determine the balance of the sound. After that the BubbleSoundObserver gets the correct sound and lets the EffectPlayer play the sound.

PART 2: Singleton pattern (15 pts)

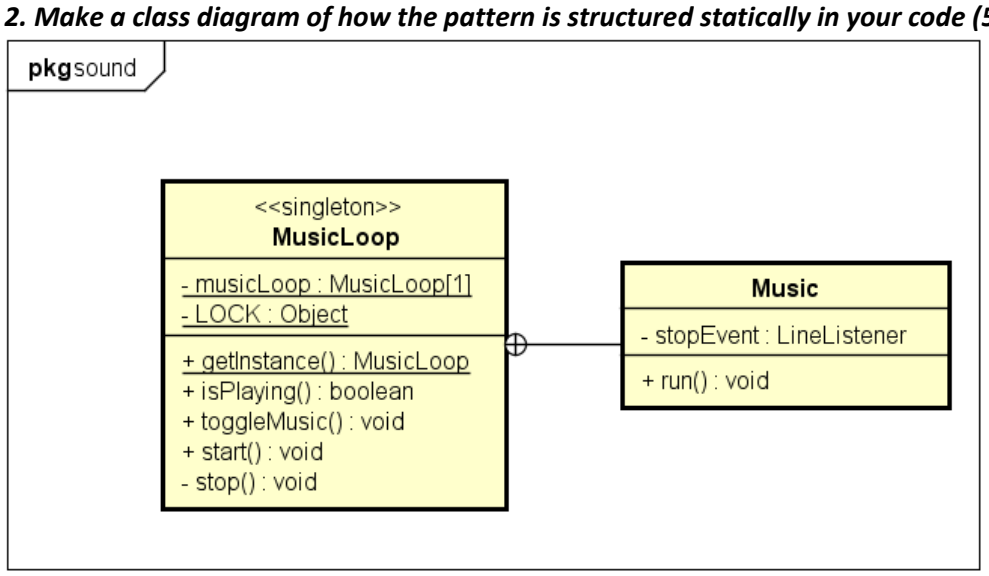
1. Write a natural language description of why and how the pattern is implemented in your code (5 pts)

In our game we have recently added a music player which plays background music throughout the whole game. It must be possible to toggle the music at any given time and there may only be one instance of the music player: there can only be one background music clip playing at a time.

Because there may only be one instance of the music player, we have decided to use the singleton pattern. The singleton pattern also makes it easy toggle the music at any given time. Currently we can only toggle the music in the options menu of the main menu screen, but we might add more places from where you can turn off the music in the future, so the global access point of the singleton pattern is very useful.

The singleton pattern is implemented in the code by making the constructor of the music player class private. This ensures that no other class can make an instance of the music player. In the music player class we have ensured that there will be at most one music player instance and this instance can be retrieved with a static getter method. We have also synchronized all of the music players public methods to ensure that the music player can be safely accessed by multiple threads (this doesn't currently happen, but we have prepared this for future development).

2. Make a class diagram of how the pattern is structured statically in your code (5pts)

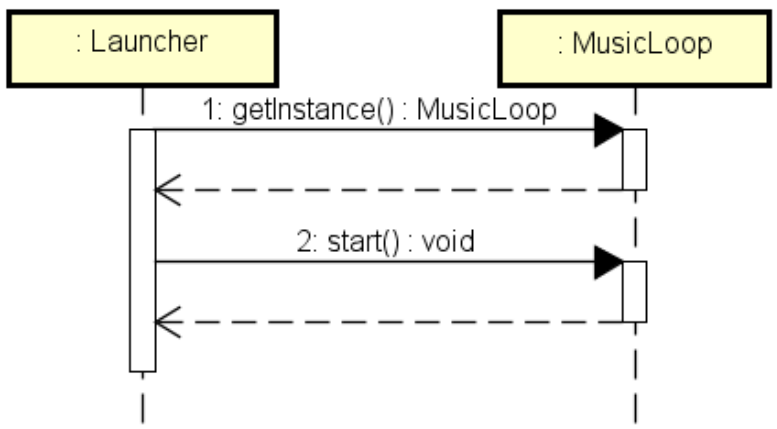


Only the most important variables and methods are displayed. The Music class is a private inner class of MusicLoop for better encapsulation.

3. Make a sequence diagram of how the pattern works dynamically in your code (5 pts)

Because the diagrams are so small we have made 2 sequence diagrams to show all of the functionality of the music loop class.

The first sequence diagram shows how the music is started when the game starts:



The second sequence diagram shows how the music is toggled from the main menu options screen: