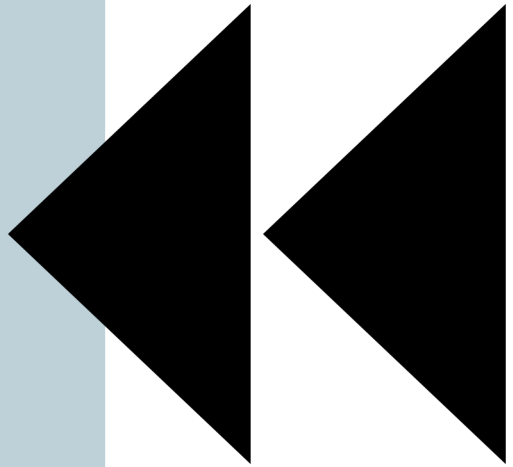


COMP 303 Review Session

February 20, 2024

Avinash Bhat

With contributions from Divya Kamath and Billy Exarhakos

I would like to have a review of the state diagrams (how to make sure that our diagram doesn't include useless states and intuition on the transitions) and the implementation of interfaces (like the ones from some of the lab tests). Thanks!

5 Reply Edit Delete Endorse ···

Below I'll list the topics which I found to be suitable for coverage:

- the UML Class Diagram, with the definitions and relations of the annotations/arrows
- the standard/rules to follow when drawing State/Class/Object diagrams
- flyweight vs. singleton design pattern

Thanks for your consideration.

12 Reply Edit Delete Endorse ···

Agenda

- Singleton
- Flyweight
- Strategy
- Unit Testing
- UML Diagrams
  - Class Diagram
  - State Diagram

A typical lab test question…

1a) Create a MusicType interface, which defines a play() method. Song and Podcast classes implement MusicType. The Song class should have *songName* (String) and *artistName* (String) attributes. The Podcast class should have an attribute called *podcastName* (String). The play() method should print the values of each class.

1b) Create a Library class which maintains a list of Song objects. Ensure that only one instance of Library can be created throughout the application.

2a) Implement a mechanism to ensure that when adding songs to the Library, check for an existing instance against a global repository with the same name and artist. If it exists, reuse it; otherwise, create a new one.

2b) Implement sorting functionality within the Library class to sort the list using the song name in ascending order and the artist name in descending order.

3a) Test the functionality when existing Songs are added to the Playlist; they are reused.

3b) Draw Class and State Diagrams.

1a) Create a **MusicType** interface, which defines a *play()* method. **Song** and **Podcast** classes implement MusicType. The Song class should have *songName* (String) and *artistName* (String) attributes. The Podcast class should have an attribute called *podcastName* (String). The play() method should print the values of each class.

```
public interface MusicType {
    public void play();
}
```

```java
public class Song implements MusicType{

    private final String aSongName;
    private final String aArtistName;

    @Override
    public void play(){
        System.out.println("Song: " + aSongName + "\t Artist:" + aArtistName);
    }

    Song(String sname, String aname){
        aSongName = sname;
        aArtistName = aname;
    }

    public String getaSongName() { return aSongName; }

    public String getaArtistName() { return aArtistName; }
}
```

```java
public class Podcast implements MusicType{

    private String aPodcastName;

    @Override
    public void play(){
        System.out.println(aPodcastName);
    }
}
```

1b) Create a **Library** class which maintains a list of Song objects. Ensure that **only one instance** of Library can be created throughout the application.

Singleton

Used when you need **only one instance** of a given class during code runtime.

Objects can be mutable.

Guarantees a single instance of a class.

How do we implement Singleton?

```
private static Library instance = new Library();

private Library(){}

public static Library getInstance() { return instance; }
```

2a) Implement a mechanism to ensure that when adding songs to the Library, check for an existing instance against a global repository with the same name and artist. If it exists, reuse it; otherwise, create a new one.

Flyweight

It is used when the code uses **many objects** that occupy **large** storage space; we want to reuse the already created objects.

Flyweight objects are (preferably) immutable.

The application doesn’t depend on object identity.

How do we implement Flyweight?

```java
public class SongFactory {
    static final HashMap<Integer, Song> playList = new HashMap<>();

    public static Song getSong(Song pSong) {
        Integer hash = pSong.getaSongName().hashCode() + pSong.getaArtistName().hashCode();
        if (playList.containsKey(hash)) {
            return playList.get(hash);
        } else {
            playList.put(hash, pSong);
            return playList.get(hash);
        }
    }

    private SongFactory(){}
}
```

Is this pre-initialization?

```
public class SongFactory {
    static final HashMap<Integer, Song> playList = new HashMap<>();

    public static Song getSong(Song pSong) {
        Integer hash = pSong.getaSongName().hashCode() + pSong.getaArtistName().hashCode();
        if (playList.containsKey(hash)) {
            return playList.get(hash);
        } else {
            playList.put(hash, pSong);
            return playList.get(hash);
        }
    }

    private SongFactory(){}
}
```

2b) Implement sorting functionality within the Library class to sort the list using the song name in ascending order and the artist name in descending order.

```java
public class bySongName implements Comparator<Song> {
    public int compare(Song s1, Song s2){
        return s1.getaSongName().compareTo(s2.getaSongName());
    }
}
```

```java
public class byArtistName implements Comparator<Song> {
    @Override
    public int compare(Song s1, Song s2) {
        return s1.getaArtistName().compareTo(s2.getaArtistName());
    }
}
```

```
public void sortSongs(Comparator<Song> c, int reverseFlag){
    if(reverseFlag == 1)
        // reverseOrder because artist desc
        Collections.sort(songList, Collections.reverseOrder(c));
    else
        Collections.sort(songList, c);
    playSongs();
}
```

Enums?

```
public void sortSongs(Comparator<Song> c, int reverseFlag){
    if(reverseFlag == 1)
        // reverseOrder because artist desc
        Collections.sort(songList, Collections.reverseOrder(c));
    else
        Collections.sort(songList, c);
    playSongs();
}
```

3a) Test the functionality when existing Songs are added to the Playlist; they are reused.

3a) Test the functionality when existing Songs are added to the Playlist; they are reused.

What should we test?

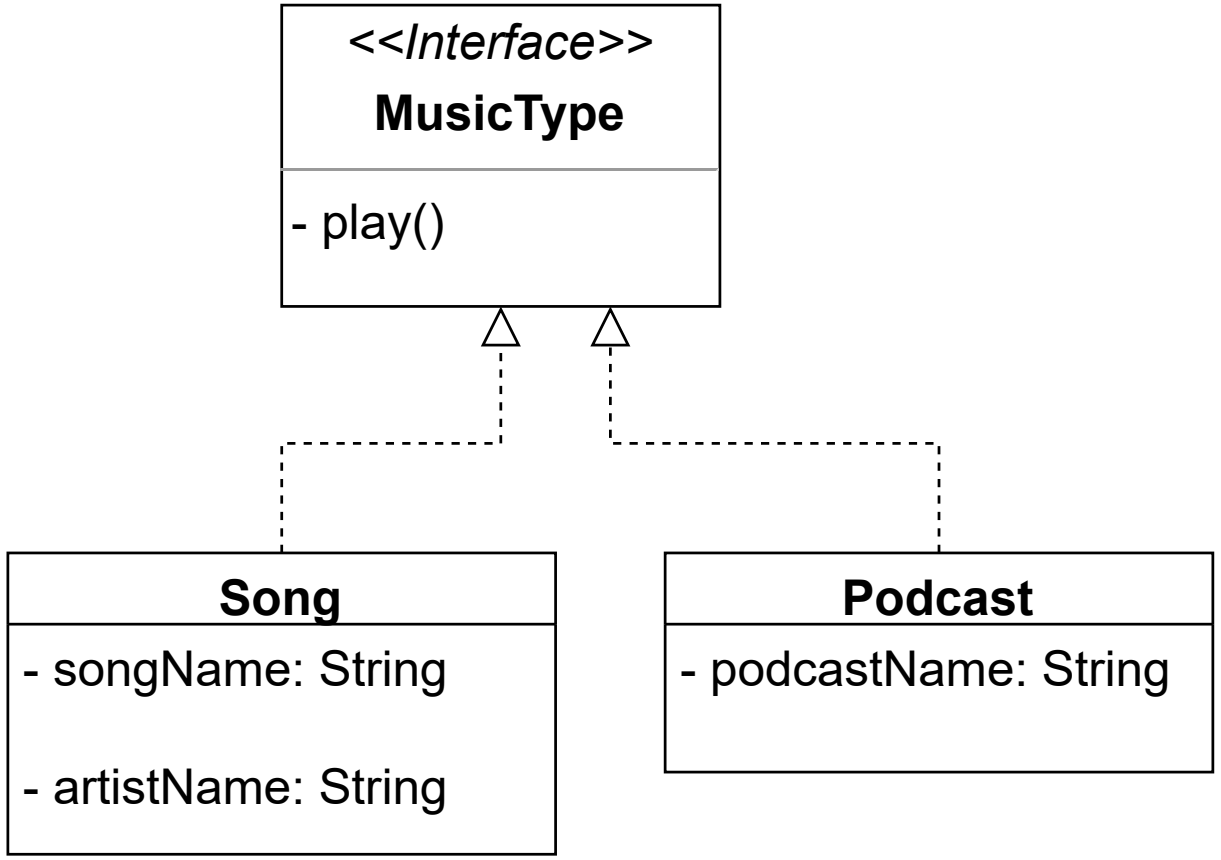
```java
public class FlyweightTest {
    @Test
    public void testSongFactoryNoSizeIncrease() {
        Song s1 = new Song( sname: "Kerala",  aname: "Bonobo");
        Library.getInstance().addSong(s1);
        Library.getInstance().addSong(s1);
        assertEquals( expected: 1, SongFactory.playList.size());
    }

    @Test
    public void testSongFactorySizeIncrease() {
        Song s1 = new Song( sname: "Kerala",  aname: "Bonobo");
        Song s2 = new Song( sname: "Navajo",  aname: "Masego");
        Library.getInstance().addSong(s1);
        Library.getInstance().addSong(s2);
        assertEquals( expected: 2, SongFactory.playList.size());
    }
}
```

3b) Draw Class and State Diagrams.

Class Diagram

1a) Create a **MusicType** interface, which defines a *play()* method. **Song** and **Podcast** classes implement MusicType. The Song class should have *songName* (String) and *artistName* (String) attributes. The Podcast class should have an attribute called *podcastName* (String). The play() method should print the values of each class.

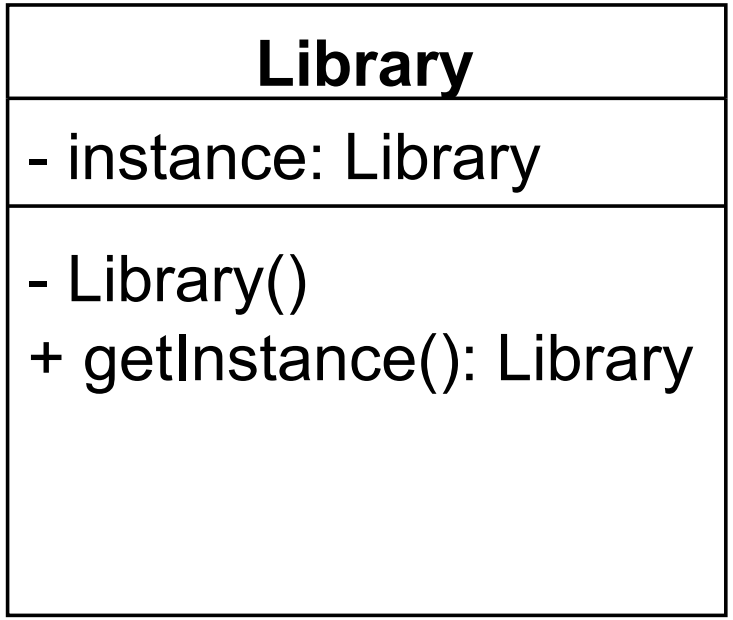
Class Diagram



Class Diagram

1b) Create a **Library** class which maintains a list of Song objects. Ensure that **only one instance** of Library can be created throughout the application.

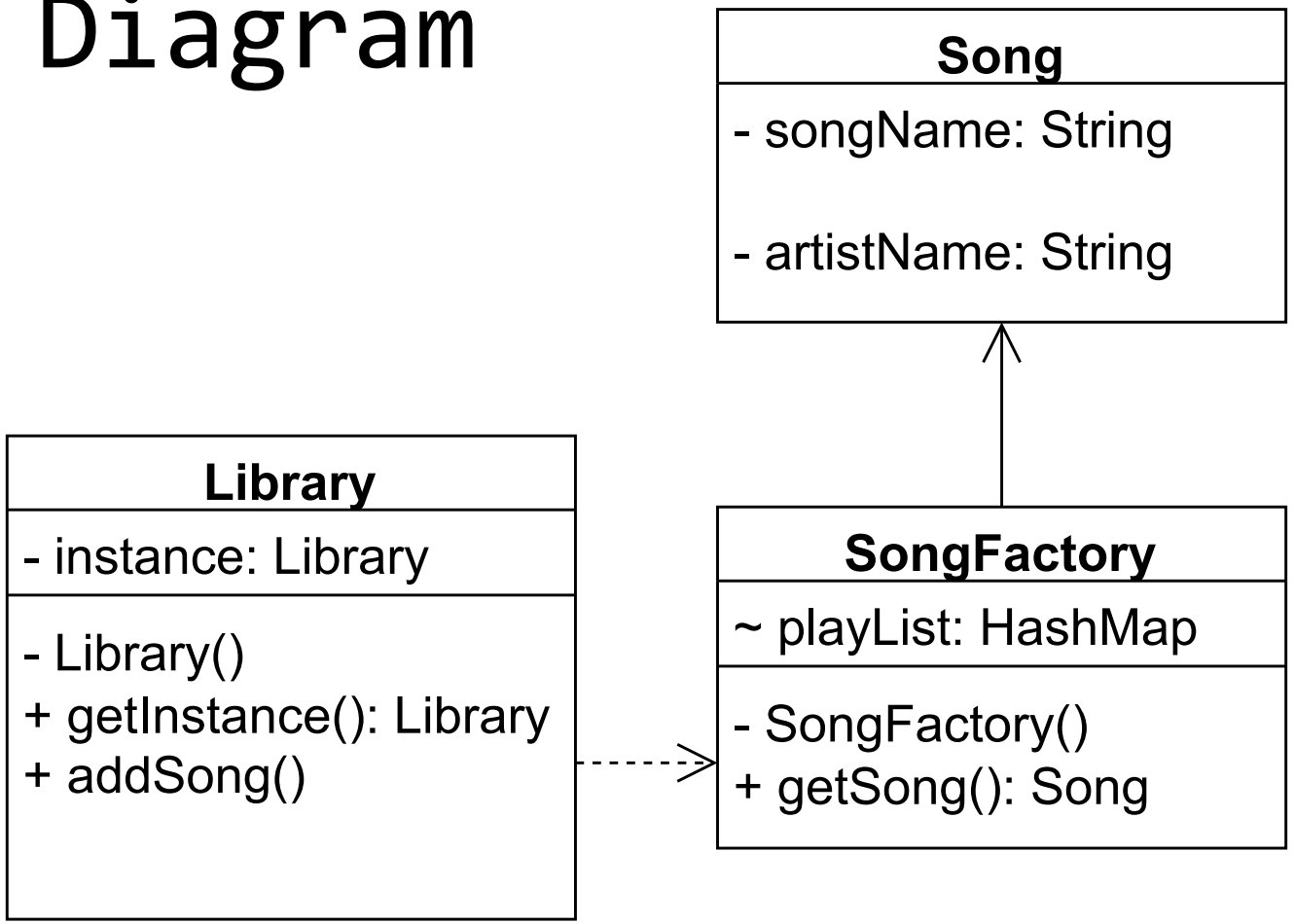
Class Diagram



Class Diagram

2a) Implement a mechanism to ensure that when adding songs to the Library, check for an existing instance against a global repository with the same name and artist. If it exists, reuse it; otherwise, create a new one.

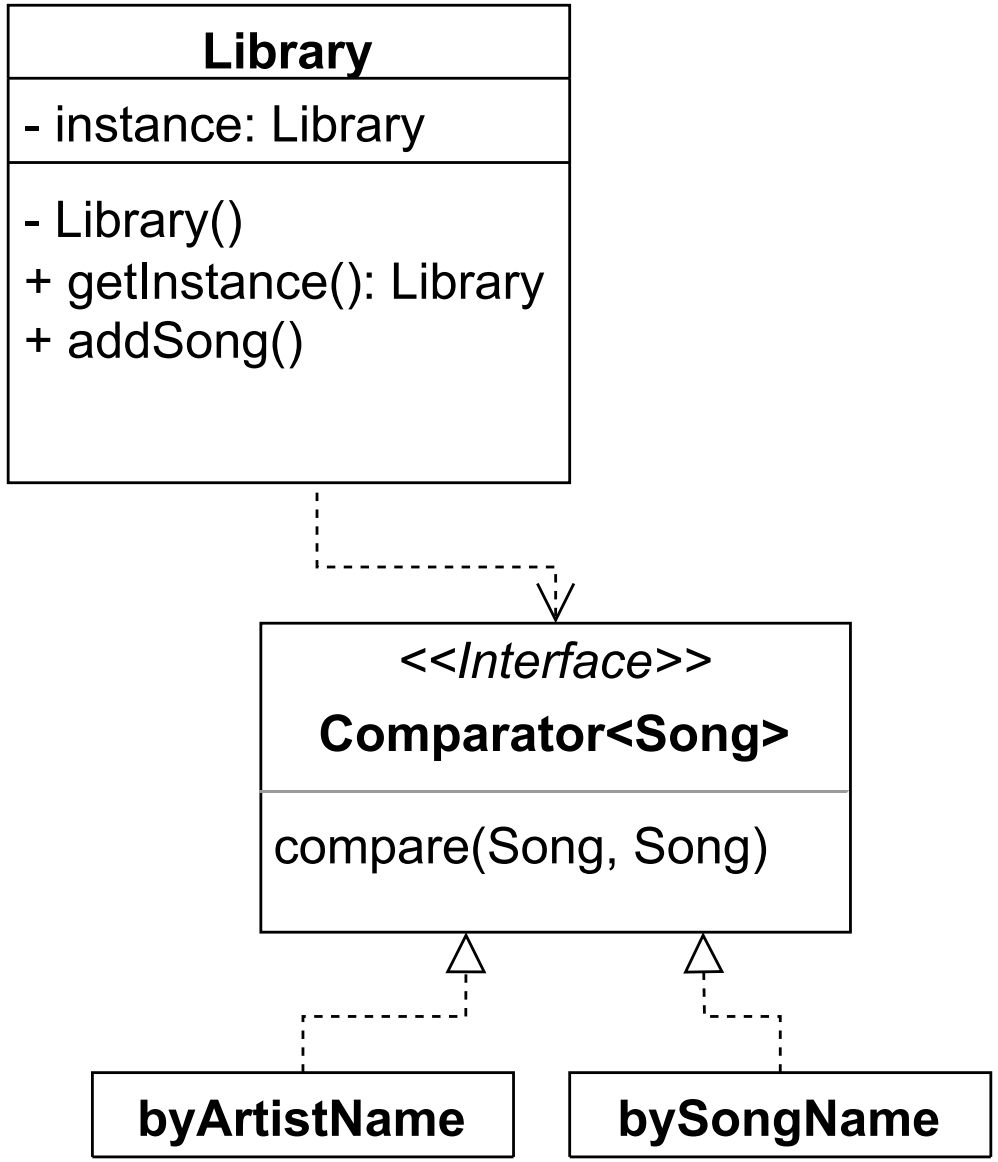
Class Diagram



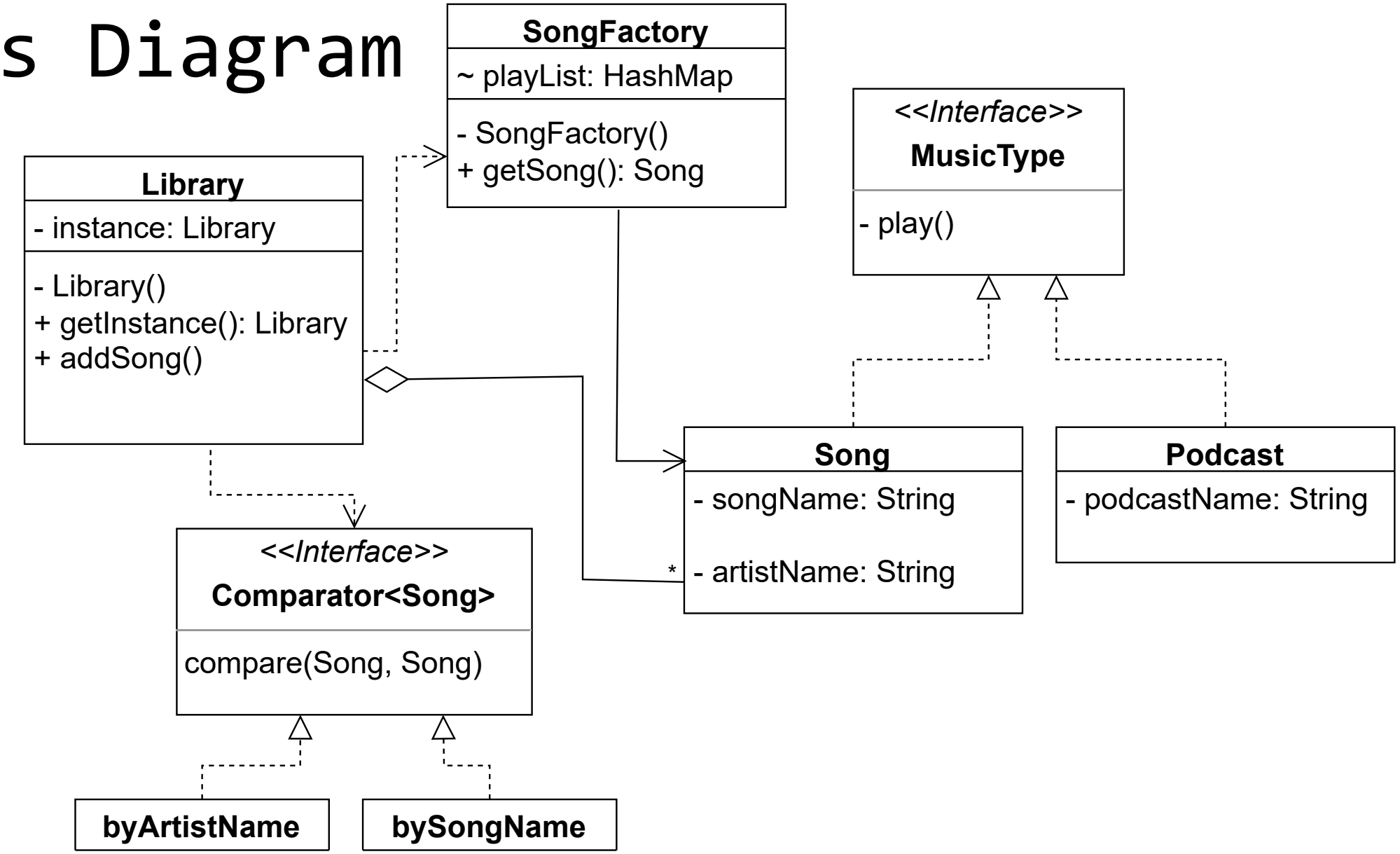
Class Diagram

2b) Implement sorting functionality within the Library class to sort the list using the song name in ascending order and the artist name in descending order.

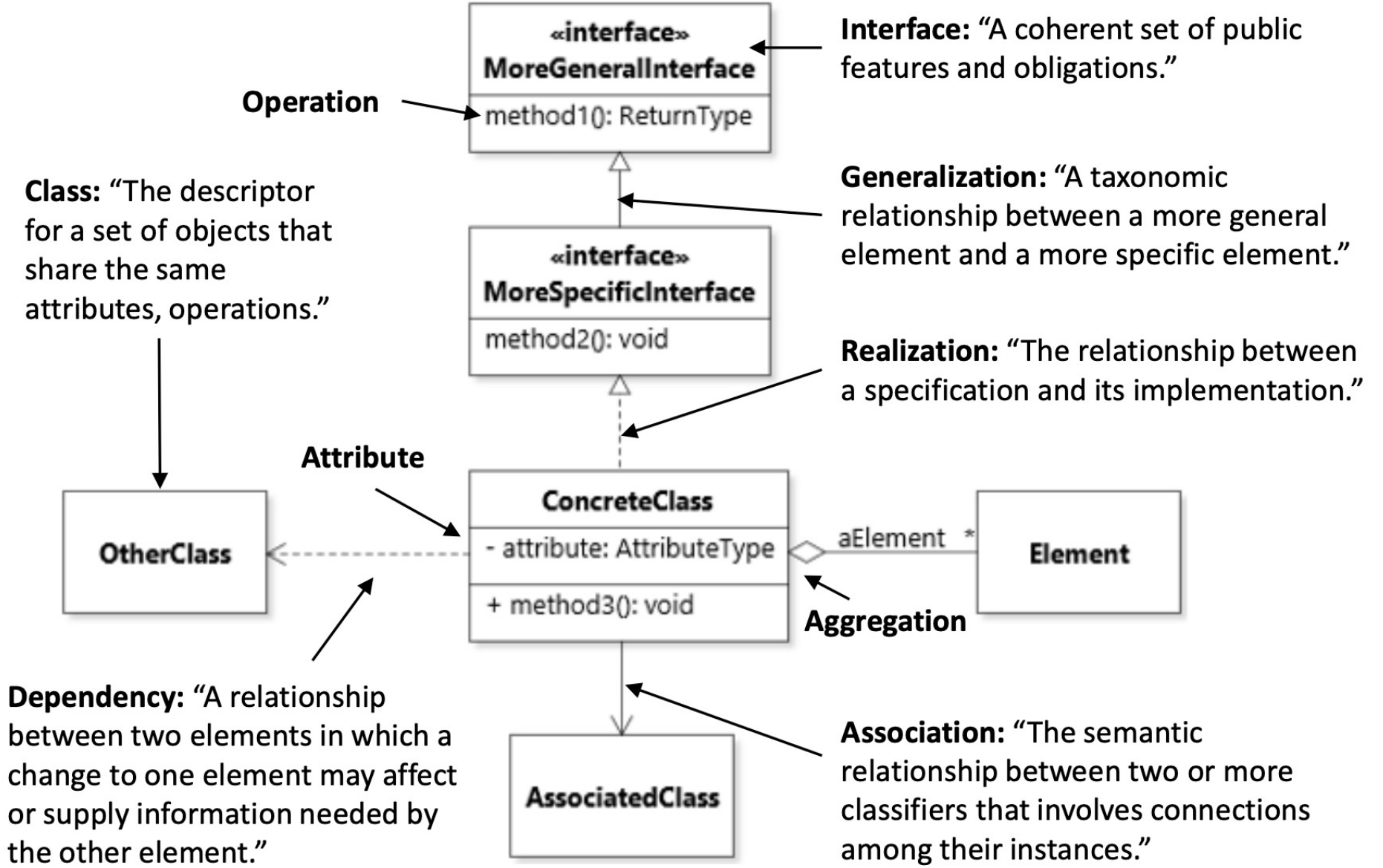
Class Diagram



Class Diagram



Class Diagram Reference



Source: Introduction to Software Design with Java. 2nd Edition. Martin P. Robillard

State Diagram

State Diagram

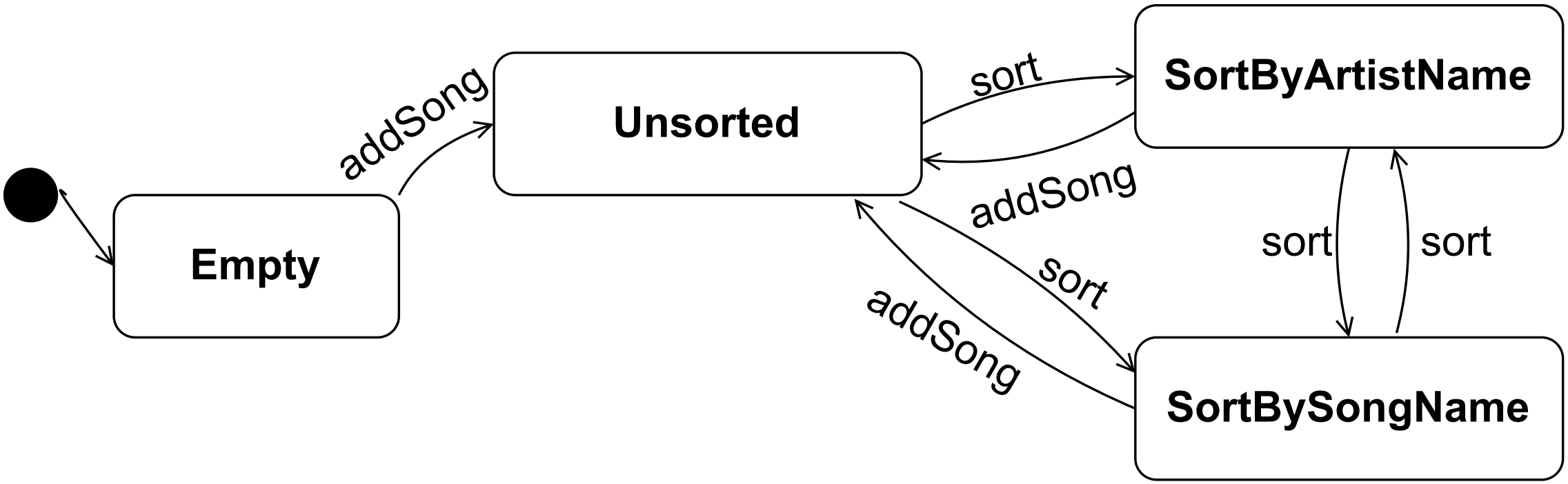
3b) Draw a State Diagram to represent various states of the **Library** class.

State Diagram

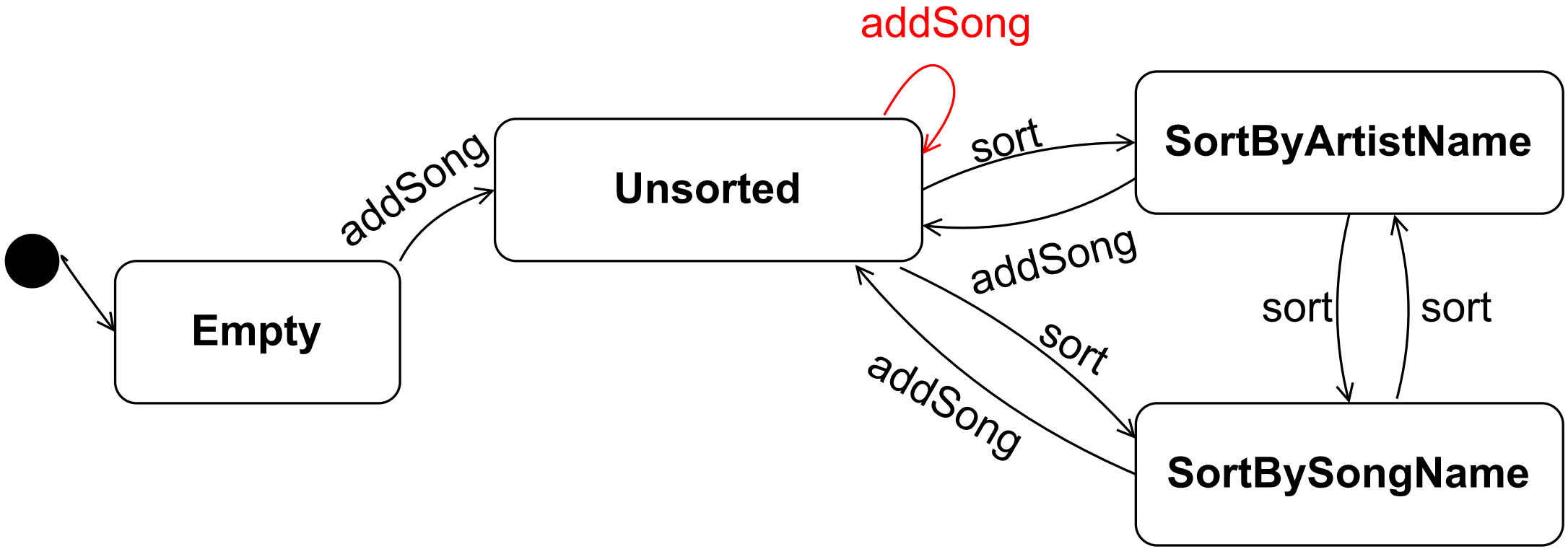
3b) Draw a State Diagram to represent various states of the **Library** class.

What are the states?

State Diagram



State Diagram



Thank you!