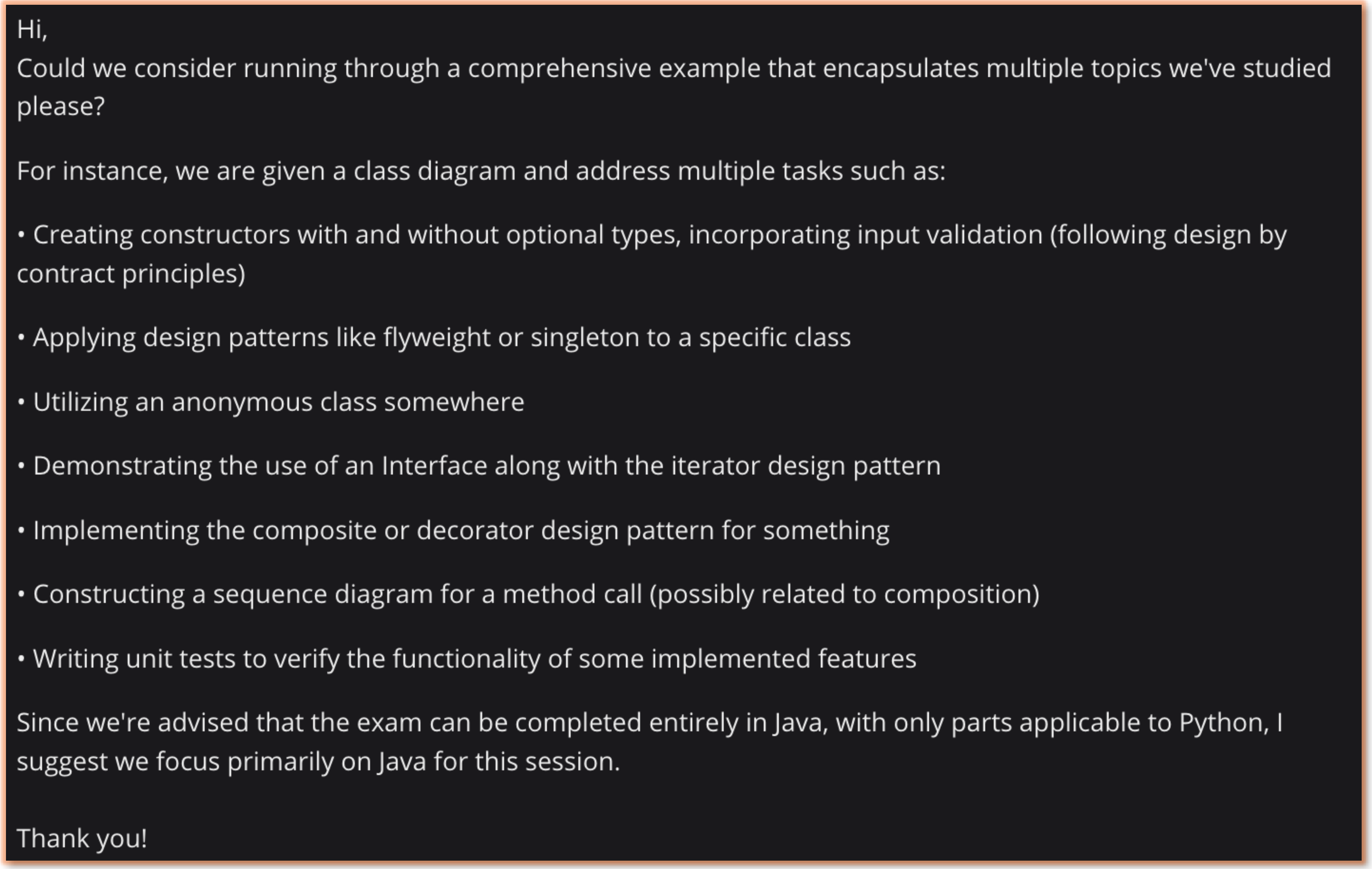
COMP 303 Review Session

February 28th, 2025

Avinash Bhat

Code Contribution by Divya Kamath

Requests



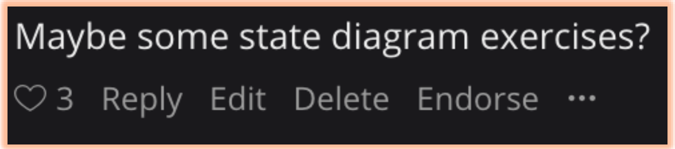
Hi,
Could we consider running through a comprehensive example that encapsulates multiple topics we've studied please?

For instance, we are given a class diagram and address multiple tasks such as:

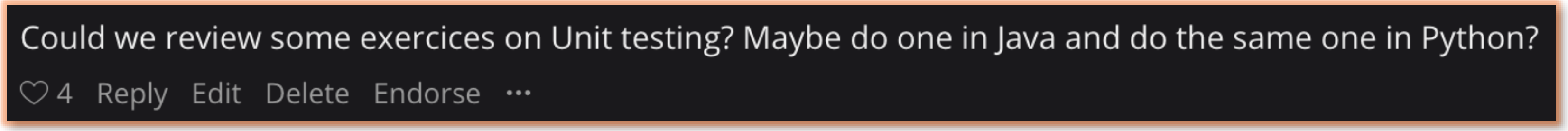
- Creating constructors with and without optional types, incorporating input validation (following design by contract principles)
- Applying design patterns like flyweight or singleton to a specific class
- Utilizing an anonymous class somewhere
- Demonstrating the use of an Interface along with the iterator design pattern
- Implementing the composite or decorator design pattern for something
- Constructing a sequence diagram for a method call (possibly related to composition)
- Writing unit tests to verify the functionality of some implemented features

Since we're advised that the exam can be completed entirely in Java, with only parts applicable to Python, I suggest we focus primarily on Java for this session.

Thank you!



Maybe some state diagram exercises?



Could we review some exercices on Unit testing? Maybe do one in Java and do the same one in Python?

Agenda

- Design by Contract
- Optional
- Singleton
- *Iterator (unintentional – but able to provide an example)*
- Flyweight
- Unit Testing
- State Diagram

a) Create a MusicType interface, which defines a play() method. Song and Podcast classes implement MusicType. The Song class should have songName (String), artistName (String) and genre (String) attributes. A song may or may not have genre. The Podcast class should have an attribute called podcastName (String). Podcast name cannot contain numbers or special characters. The play() method should print the values of each class.

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

c) 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.

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

e) Draw State Diagram to represent various states of **Library** class.

Code available at:

https://github.com/avinashbhat/comp303-midterm-review

a) Create a MusicType interface, which defines a play() method. Song and Podcast classes implement MusicType. The Song class should have songName (String), artistName (String) and genre (String) attributes. A song may or may not have genre.

The Podcast class should have an attribute called podcastName (String). Podcast name cannot contain numbers or special characters.

The play() method should print the values of each class.

Design by Contract

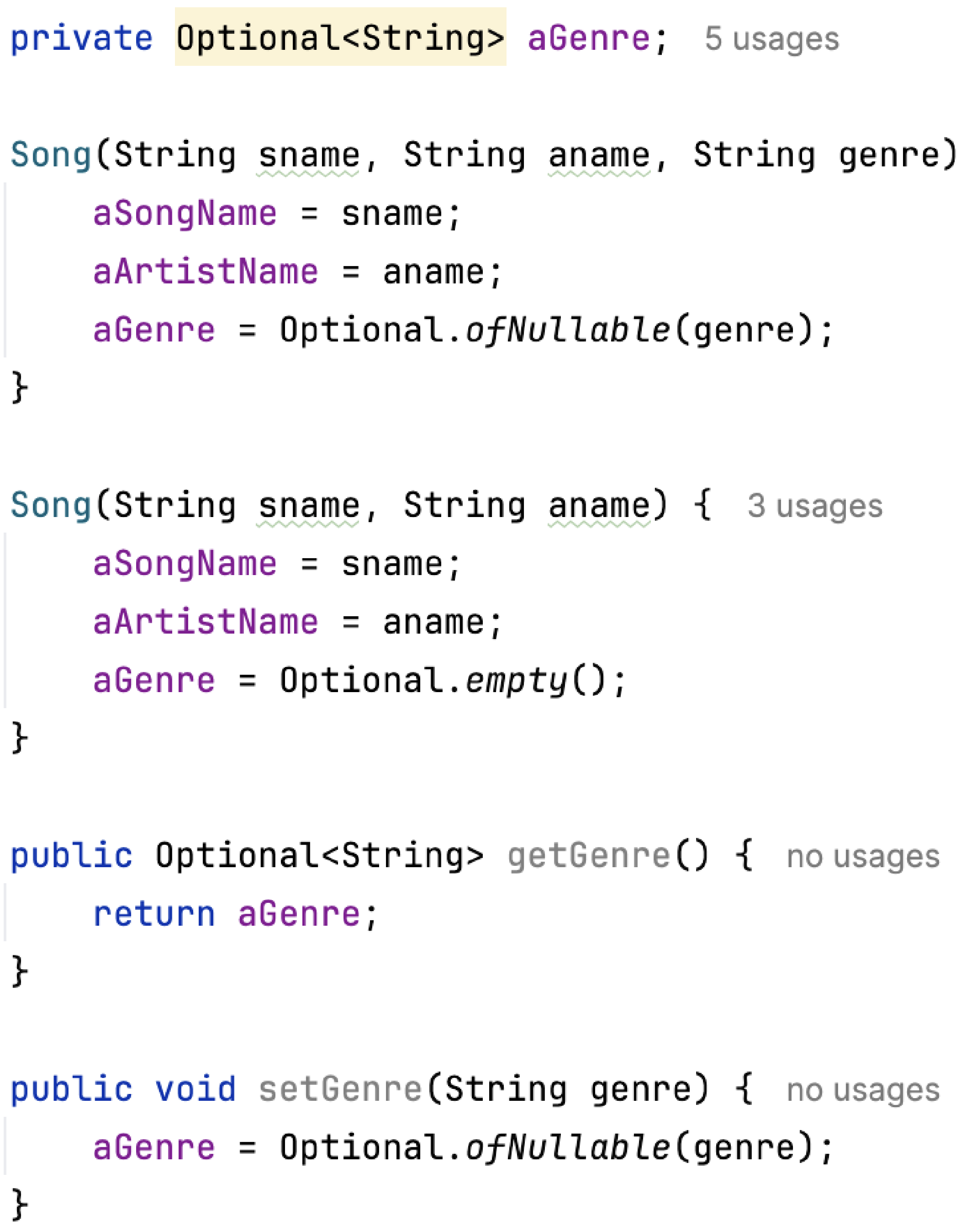
- Contract between the method and the caller.
- Implemented using assertions and documentation.

```java
/**
 * Creates a new Podcast
 * @pre podcastName != null && podcastName contains only letters and spaces
 * @param podcastName the name of the podcast (must not be null and must contain only letters and spaces)
 */
public Podcast(String podcastName) {  1 usage
    assert podcastName != null : "Podcast name cannot be null";
    assert isValidPodcastName(podcastName) : "Podcast name must contain only letters and spaces";

    aPodcastName = podcastName;
}

private boolean isValidPodcastName(String name) {  2 usages
    return name.matches( regex: "^[a-zA-Z ]+$");
}
```

Optional

- Cleaner alternative to null checks.

Optional API?



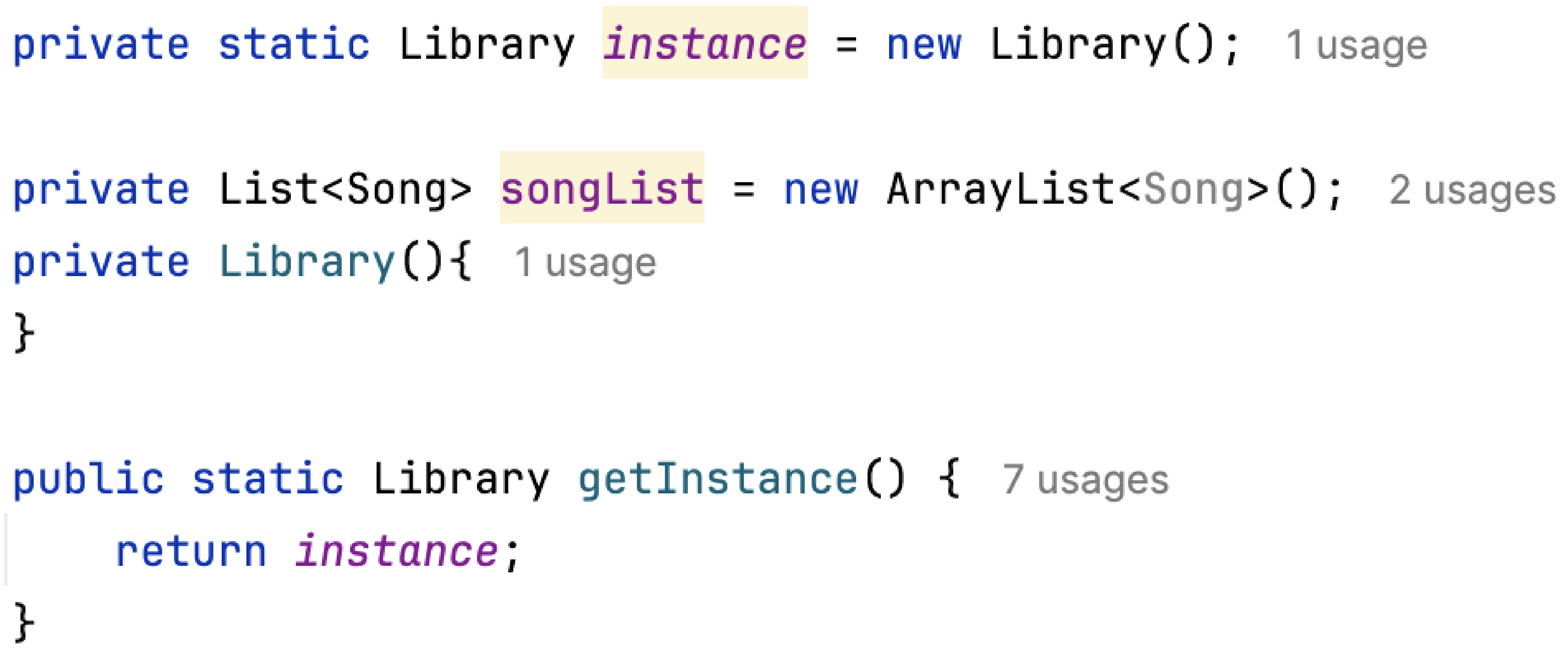
```
private Optional<String> aGenre;  5 usages

Song(String sname, String aname, String genre) {
    aSongName = sname;
    aArtistName = aname;
    aGenre = Optional.ofNullable(genre);
}

Song(String sname, String aname) {  3 usages
    aSongName = sname;
    aArtistName = aname;
    aGenre = Optional.empty();
}

public Optional<String> getGenre() {  no usages
    return aGenre;
}

public void setGenre(String genre) {  no usages
    aGenre = Optional.ofNullable(genre);
}
```

b) 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?



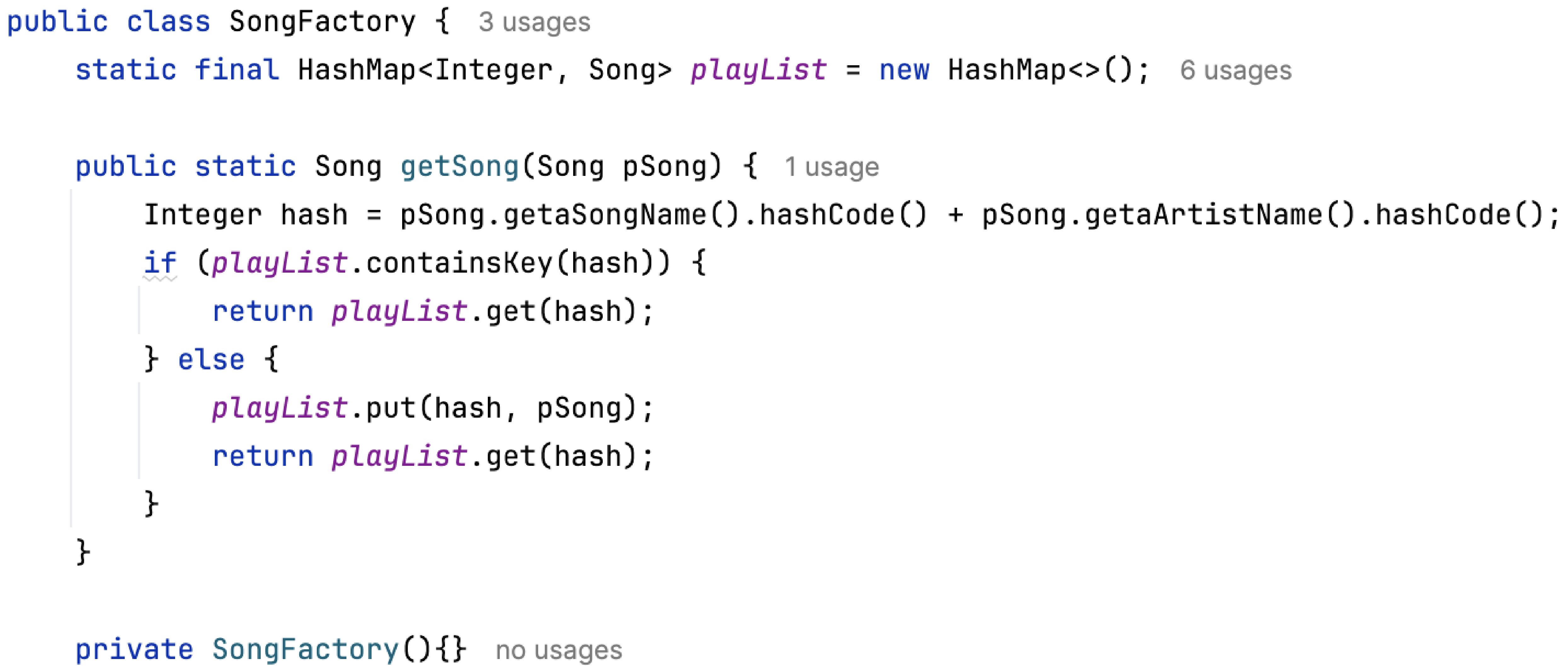
```
private static Library instance = new Library();  1 usage

private List<Song> songList = new ArrayList<Song>();  2 usages
private Library(){  1 usage
}

public static Library getInstance() {  7 usages
    return instance;
}
```

c) 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?



```
public class SongFactory {  3 usages
    static final HashMap<Integer, Song> playList = new HashMap<>();  6 usages

    public static Song getSong(Song pSong) {  1 usage
        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(){}  no usages
```

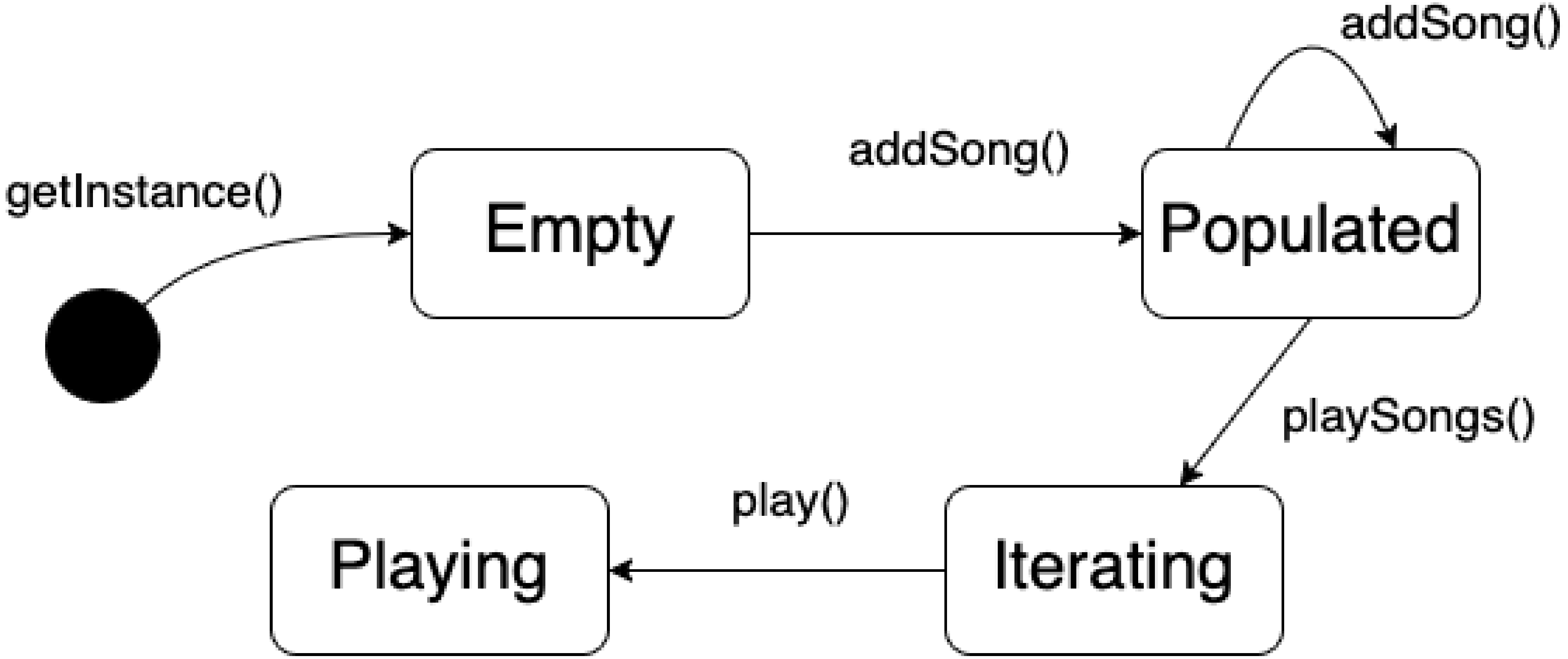
d) Test the functionality when existing Songs are added to the Playlist; they are reused.

What are we testing?

```
@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", genre: "Blues");
    Song s2 = new Song( sname: "Navajo", aname: "Masego", genre: "Jazz/House");
    Library.getInstance().addSong(s1);
    Library.getInstance().addSong(s2);
    assertEquals( expected: 2, SongFactory.playList.size());
}
```

e) Draw State Diagram to represent various states of **Library** class.

What are the states?


getInstance()
Empty
addSong()
Populated
addSong()
playSongs()
Iterating
play()
Playing

Questions?