UNIVERSITÀ DEGLI STUDI DI FIRENZE
Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Relazione elaborato SWE

Niccolò Arati

Maggio 2021

Indice

1 Introduzione e Requisiti

1.1 Problem Statement

Il progetto riguarda la gestione di un catalogo musicale.
Il catalogo contiene canzoni, playlist, e informazioni sugli autori delle canzoni e sui generi musicali presenti.
Le canzoni hanno un titolo, una durata, un autore e un genere musicale.
Le playlist presentano informazioni su numero di canzoni contenute e durata complessiva, e possono rappresentare o una normale playlist, o un disco di un autore, e in quast' ultimo caso devono anche avere informazioni su chi sia l' autore.
Le categorie musicali hanno informazioni sul numero di canzoni nel catalogo di quella categoria, e non possono essere ascoltate.
Un autore viene identificato dal suo nome, ed ha la possibilità di produrre nuove canzoni o album. Ha anche una raccolta di tutte le sue canzoni e dei suoi dischi.
Il catalogo è organizzato da un gestore, che ha accesso diretto ad esso, e ha la possibilità di aggiungere nuove playlist.
Quando un autore produce una nuova canzone o un nuovo disco, il gestore deve provvedere ad aggiungerlo al catalogo.
All' interno del catalogo si possono effettuare 4 tipi di ricerche : o si ricerca in modo generico una canzone, una playlist o un autore, o si richiedono informazioni rispetto ad un determinato genere musicale, o si ricerca una canzone specificandone l' autore, oppure specificandone la categoria.
La stessa cosa può essere fatta per riprodurre la musica.
Si può scegliere di ascoltare tutte le canzoni di un autore, mentre non è possibile farlo per i generi musicali.
Gli utenti che possono accedere al sito sono di due tipi: iscritti e non.
Gli utenti non iscritti possono solo fare ricerche all' interno del sito e riprodurre la musica.
Gli utenti iscritti hanno anche la possibilità di salvare i brani che gli sono piaciuti in una loro raccolta personale, e se hanno pagato un abbonamento possono pure scaricarli.
Un utente non iscritto ha la possibilità di iscriversi.
Quando una canzone o una playlist vengono scaricate, viene data la possibilità all'utente di salvarle con un nome personalizzato.
Le raccolte di canzoni di un utente possono inoltre essere riprodotte casualmente.

1.2 Casi d'uso

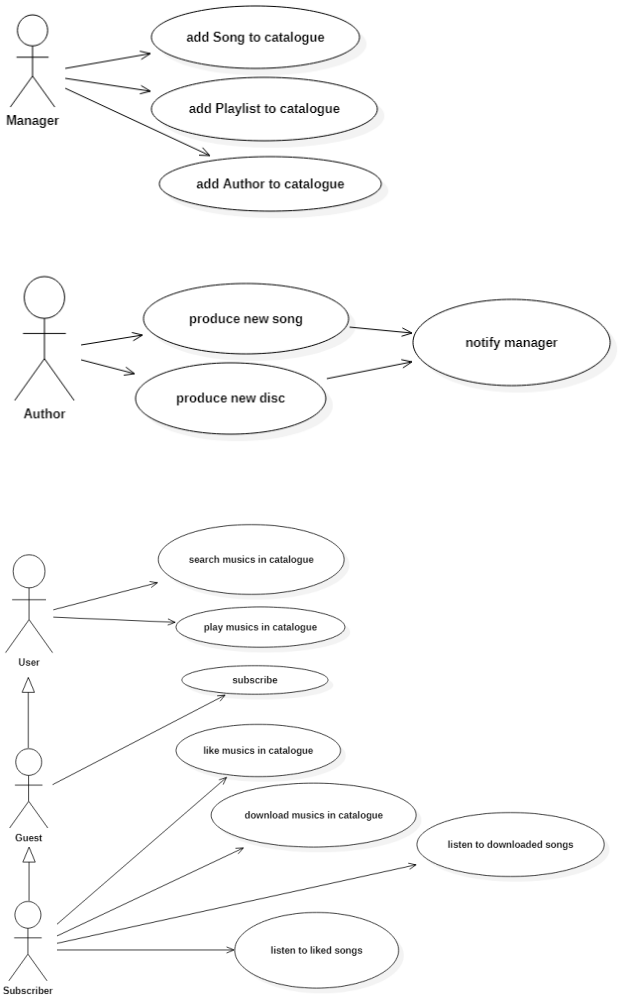

Figure 1: Use Case Diagram

1.3 Requisiti

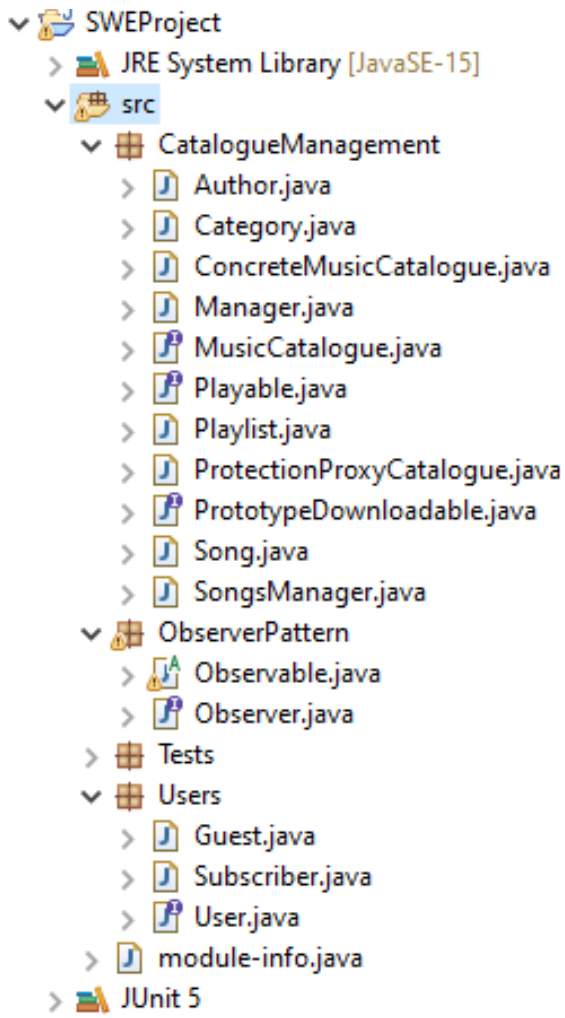

Figure 2: Organizzazione packages del progetto

2 Progettazione

2.1 Class Diagram

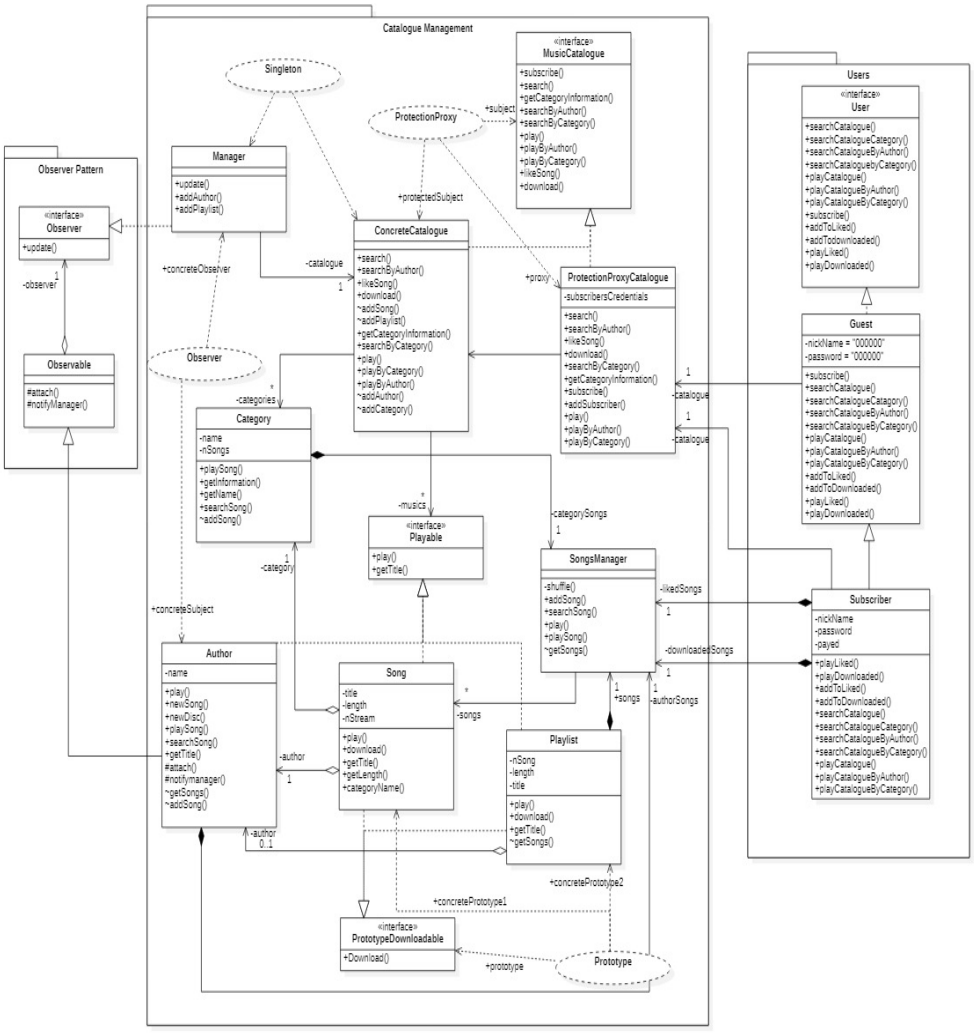

Figure 3: Class diagram

Ho utilizzato il pattern Observer perchè si richiedeva che ogni qualvolta un autore presente nel catalogo pubblicasse una nuova canzone o un nuovo disco, questo venga aggiunto al catalogo. Così facendo il gestore viene informato ogni volta che un autore modifica il proprio stato producendo un nuovo componimento, e provvede ad aggiornare il catalogo.

Il Prototype è stato implementato per permettere il download di canzoni e playlist, con la possibilità di personalizzarne il nome.
Il Protection Proxy infine è stato implementato per controllare l' accesso alle funzionalità del catalogo di download e di aggiunta alla lista delle canzoni personali da parte di un utente, in quanto è consentito solo agli utenti iscritti.

2.2 Sequence Diagram

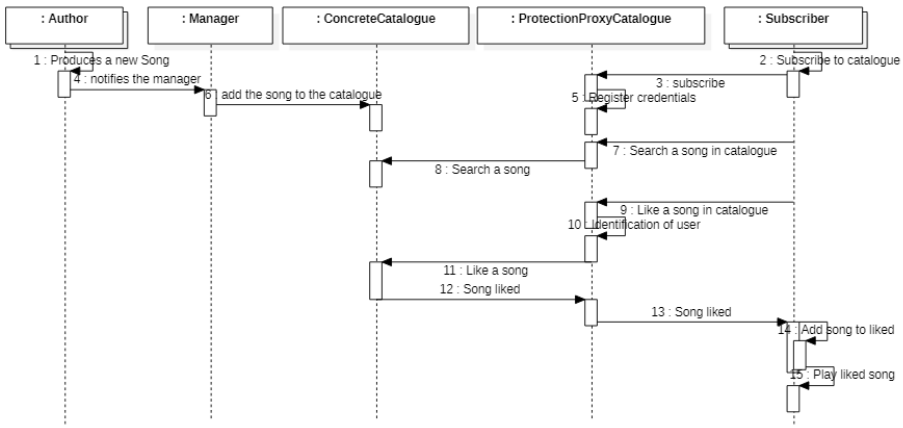

Figure 4: Sequence Diagram

2.3 Mockups

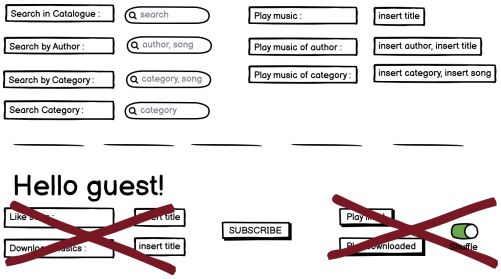

Figure 5: View per un Guest

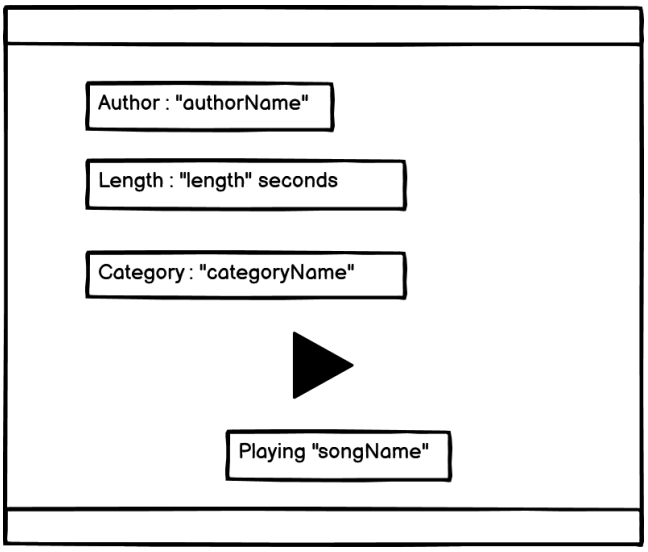

Figure 6: View per la riproduzione di una canzone

3 Implementazione

3.1 Classi

3.1.1 MusicCatalogue

E' un' interfaccia che espone i metodi necessari a soddisfare le responsabilità del catalogo. Le principali funzionalità richieste sono quelle di ricerca e di riproduzione della musica, ed anche la possibilità di poter mettere like o scaricare canzoni da parte degli utenti.
Rappresenta il Subject astratto del pattern Proxy.

```java
public interface MusicCatalogue {

    public Subscriber subscribe(Guest u, String nickName, String password, Boolean payed);

    public Boolean search (String songName);

    public Boolean getCategoryInformation (String category);

    public Boolean searchByAuthor(String authorName, String songName);

    public Boolean searchByCategory (String category, String songName);

    public Boolean play (String songName);

    public Boolean playByAuthor (String authorName, String songName);

    public Boolean playByCategory (String category, String songName);

    public Playable likeSong(User u, String songName);

    public PrototypeDownloadable download(User u, String songName, String Title);

}
```

Figure 7: Metodi esposti dall' interfaccia MusicCatalogue

3.1.2 ProtectionProxyCatalogue

Implementa l' interfaccia MusicCatalogue, ed ha un riferimento al ConcreteMusicCatalogue. Tiene traccia di tutte le credenziali degli utenti iscritti al catalogo, e per ogni operazione che essi vogliono effettuare sul catalogo prima verifica le credenziali, se necessario, e poi chiama il catalogo concreto.

```java
public PrototypeDownloadable download(User u, String songName, String title) {
    if (u instanceof Guest && !(u instanceof Subscriber)) {
        System.out.println("Error, you are not subscribed");
        return null;
    }
    else {
        if (! u.getPassword().equals(subscribersCredentials.get(u.getNickName()))) {
            System.out.println("Wrong password");
            return null;
        }
        else {
            if(! ((Subscriber)u).getPayed()) {
                System.out.println("Error, you have not a payed subscription");
                return null;
            }
            else
                return catalogue.download(u, songName, title);
        }
    }
}

public Subscriber subscribe(Guest u, String nickName, String password, Boolean payed) {
    int i = 0;
    while (subscribersCredentials.containsKey(nickName)) {
        nickName = nickName + "2";
        i ++;
    }
    if (i != 0)
        System.out.println("NickName already registered, we assign you " + nickName);
    subscribersCredentials.put(nickName, password);
    return new Subscriber(nickName, password, payed, this, true);
}
```

Figure 8: Controllo credenziali nel download e aggiunta di un nuovo subscriber

3.1.3 ConcreteMusicCatalogue

Contiene una raccolta di canzoni, autori e playlist ed una di generi musicali. Ha metodi per la ricerca su queste raccolte, per riprodurre musica e per aggiungere oggetti al catalogo.
In particolare, questi ultimi metodi sono dichiarati package protected, perchè solo il gestore del catalogo può invocarli per aggiungere elementi.
Le categorie musicali non vengono aggiunte direttamente, ma quando si aggiunge una canzone si verifica se il suo genere sia presente o meno nel catalogo. In caso non lo fosse, si provvede ad istanziarlo e ad aggiugnerlo.

```
void addSong (Song s) {
    Boolean added = false;
    if (musics.containsKey(s.getTitle())) {
        if(musics.get(s.getTitle()) instanceof Song) {
            if(((Song) musics.get(s.getTitle())).getAuthor().getTitle().compareTo(s.getAuthor().getTitle()) != 0) {
                Vector<Song> songs = new Vector<Song>();
                songs.add(s);
                songs.add((Song) musics.get(s.getTitle()));
                Playlist p = new Playlist(s.getTitle(), null, songs);
                musics.put(p.getTitle(), p);
                added = true;
            }
        }
        else {
            Vector<Song> songs = new Vector<Song>();
            songs.add(s);
            Vector<Song> oldsongs = ((Playlist) musics.get(s.getTitle())).getSongs();
            for (Song song : oldsongs)
                songs.add(song);
            Playlist p = new Playlist(s.getTitle(), null, songs);
            musics.put(p.getTitle(), p);
            added = true;
            }
        }
    else {
        musics.put(s.getTitle(), s);
        added = true;
    }
    if (added) {
        Boolean found = false;
        for (Category category : categories) {
            if (category.getName().equals(s.categoryName())) {
                category.addSong(s);
                found = true;
            }
        }
        if (!found) {
            Category c = new Category(s.categoryName());
            c.addSong(s);
            addCategory(c);
        }
        if (! s.getAuthor().searchSong(s.getTitle()))
            s.getAuthor().addSong(s);
    }
}
```

Figure 9: Metodo addSong() della classe ConcreteMusicCatalogue

3.1.4 Manager

Manager implementa l' interfaccia Observer, è responsabile dell' aggiunta al catalogo di playlist e autori.
Le playlist che aggiunge sono create da lui, non rappresentano dei dischi di un certo autore e quindi non hanno informazioni su di esso.
Ogni volta che un autore aggiunto al catalogo produrrà delle nuove canzoni o dischi, è responsabilità del Manager di aggiungerle al catalogo, attraverso il suo attributo privato che contiene un riferimento al catalogo concreto.

3.1.5 Playable e PrototypeDownloadable

Playable è un' interfaccia che espone il metodo play(), ed è implementata da tutti gli oggetti del catalogo che possono essere riprodotti. Per questioni pratiche espone anche il metodo getTitle(), che restituisce il titolo di un oggetto riproducibile.
PrototypeDownloadable invece è un' interfaccia che espone il metodo download(), che serve a clonare gli oggetti che implementano questa interfaccia con la possibilità di modificarne il titolo.

3.1.6 Author

Author è una classe che implementa l' interfaccia Playable ed estende Observable. Ha un nome e una collezione di tutti i suoi componimenti, sia canzoni che dischi. Può produrre nuove canzoni e nuovi dischi, e ha un metodo per ricercare canzoni e dischi nella sua raccolta.
Presenta anche un attributo di tipo Observer, in quanto è anche il Subject del pattern Observer.
Ha anche un metodo package protected getSongs(), nel quale restituisce la sua collezione sottoforma di Playlist, nel caso in cui un utente scelga di mettere like all' autore.

```
public Author(String name) {
    this.name = name;
    this.songs = new SongsManager<Playable>();
}

public void play () {
    System.out.println("Playing songs of " + name);
    songs.play(false);
}

public void playSong (String songName) {
    songs.playSong(songName);
}

public String getTitle () {
    return this.name;
}
```

```java
public Boolean searchSong (String songName) {
    return songs.searchSong(songName);
}

Playlist getSongs() {
    Vector<Playable> authorSongs = songs.getSongs();
    Vector<Song> song = new Vector<Song>();
    for (Playable p : authorSongs) {
        if (p instanceof Playlist) {
            Vector<Song> temp = ((Playlist) p).getSongs();
            for (Song s : temp)
                song.add(s);
        }
        else
            song.add((Song) p);
    }
    return new Playlist(name + "songs", null, song);
}
```

Figure 10: Frammenti di codice della classe Author

3.1.7 Category

Category è una classe che rappresenta un possibile genere musicale associato ad una canzone. Ha un nome, un valore che tiene conto di tutte le canzoni contenute all' interno del catalogo che sono della sua categoria, ed una collezione di esse.
Presenta dei metodi per ricercare canzoni del suo genere musicale e riprodurle, ma non è possibile riprodurre tutte le canzoni della sua collezione.

3.1.8 Song e Playlist

Song è una classe che rappresenta una canzone contenuta nel catalogo. Ha un titolo, una lunghezza in secondi, un genere musicale ed un autore.
Playlist invece rappresenta una raccolta di canzoni, ha un titolo, una lunghezza complessiva, un valore che indica quante canzoni contiene, e un riferimento ad un oggetto di tipo SongsManager che gestisce le sue canzoni. Può anche avere un autore se rappresenta un disco.
Entrambe le classi implementano le interfacce Playable e PrototypeDownloadable, e quindi espongono anche i loro metodi.

```java
public void play () {
    System.out.println("Playing " + title + ", length: " + length + ", author: " + author.getTitle());
}

public PrototypeDownloadable download (String title) {
    return new Song(title, this.length, this.category, this.author);
}

public String getTitle () {
    return title;
}
```

```java
public void play () {
    if(author == null)
        System.out.println("Playing playlist " + title + ", " + nSongs + " songs, total length: " + length);
    else
        System.out.println("Playing disc" + title + " of " + author.getTitle() + ", " + nSongs + " songs, total length:
    songs.play(false);
}

public PrototypeDownloadable download (String title) {
    return new Playlist(title, this.author, this.songs.getSongs());
}

public String getTitle () {
    return title;
}
```

Figure 11: Frammenti di codice delle classi Song e Playlist

3.1.9 SongsManager

SongsManager è una classe dipendente da un tipo parametrizzato, presenta una collezione di Playable o di Downloadable a seconda del contesto in cui è utilizzata.
Serve per gestire una collezione di canzoni o playlist, ha la possibilità di riprodurle, oppure di restituirle per essere scaricate attraverso un metodo package protected getSongs().
Inoltre ha anche un metodo shuffle() per mischiarle, in modo da avere una riproduzione casuale.

3.1.10 User, Guest e Subscriber

User è un' interfaccia che espone i metodi eseguibili sul catalogo dagli utenti. Ha metodi per fare ricerche sul catalogo, per riprodurre musica, per iscriversi, ed espone anche quelli per mettere like e per scaricare le canzoni.
Guest implementa l' interfaccia User, ha un riferimento al MusicCatalogue, ma non può riprodurre le proprie raccolte personali di musica perchè non è iscritto. Ha un nickname ed una password di default, ma non può cambiarle.
Subscriber estende Guest, ha la possibilità di cambiare il proprio attributo payed per poter anche scaricare le canzoni, e può anche riprodurre le proprie raccolte, in modo anche casuale. Ha anche la possibilità di variare le proprie credenziali con cui si è iscritto al catalogo.

```java
public interface User {

    public Subscriber subscribe(String nickName, String password, Boolean payed);

    public Boolean searchCatalogue (String songName);

    public Boolean searchCategoryCatalogue (String category);

    public Boolean searchCatalogueByAuthor (String authorName, String songName);

    public Boolean searchCatalogueByCategory (String category, String songName);

    public Boolean playCatalogue (String songName);

    public Boolean playCatalogueByAuthor (String authorName, String songName);

    public Boolean playCatalogueByCategory (String category, String songName);

    public void addToLiked (String songName);

    public void addToDownloaded(String songName, String title);

    public void playLiked(Boolean shuffle);

    public void playDownloaded(Boolean shuffle);

    public String getNickName();

    public String getPassword();

    public void changeNickName (String newNick);

    public void changePassword (String newPassword);

}
```

Figure 12: Interfaccia User

3.1.11 Observable e Observer

Observable è una classe astratta, mentre Observer è un' interfaccia. Entrambe servono a poter realizzare il pattern Observer, e contengono i metodi e gli attributi ad esso necessari.

3.2 Pattern usati

3.2.1 Singleton

Singleton è un pattern creazionale, serve per assicurare che una classe possa avere solo un' istanza, e inoltre fornisce un accesso globale a questa istanza. Viene implementato ponendo il costruttore della classe privato, e attraverso un metodo pubblico statico, getInstance(), si accede all' unica istanza della classe. Nel progetto ConcreteMusicCatalogue e Manager sono Singleton, e quando viene invocato per la prima volta getInstance(), un oggetto della classe viene creato e associato all' attributo privato e statico instance, che poi viene restituito. Se invece getInstance() era già stato chiamato, restituirà l' istanza della classe precedentemente istanziata.

```
private ConcreteMusicCatalogue () {
    this.musics = new java.util.HashMap<String, Playable>();
    this.categories = new Vector<Category>();
}

public static ConcreteMusicCatalogue getInstance () {
    if (instance == null)
        instance = new ConcreteMusicCatalogue ();
    return instance;
}
```

Figure 13: Implementazione Singleton in ConcreteMusicCatalogue

```
private Manager () {
    catalogue = ConcreteMusicCatalogue.getInstance();
}

public static Manager getInstance () {
    if (instance == null)
        instance = new Manager ();
    return instance;
}
```

Figure 14: Implementazione Singleton in Manager

3.2.2 Observer

Observer è un pattern comportamentale utilizzato quando si ha necessità da parte di alcuni oggetti, detti observers, di monitorare le variazioni dello stato di altri oggetti, detti subjects.
Per implementarlo ho creato un package ObserverPattern con dentro una classe astratta Observable e un' interfaccia Observer. Observer espone il metodo update() in modo push, ovvero richiede come input un oggetto che notifichi il cambiamento dello stato avvenuto nel Subject. Observable espone i metodi per il Subject, ovvero attach(), che serve a registrare l' Observer al quale notificare i propri cambiamenti, e notifyManager(). Ha un attributo privato di tipo Observer e non una lista come di consueto, perchè avendo definito Manager come Singleton, ed essendo Manager l' unica classe che rappresenta un Observer concreto, la lista sarebbe comunque stata composta da un solo elemento.
Nel progetto la classe Manager implementa Observer e la classe Author estende Observable. Ogni volta che il manager aggiunge un autore al catalogo, l' attributo Observer di Author viene assegnato col manager attraverso il metodo attach().

```
public void update (Playable s) {
    if (s instanceof Song)
        catalogue.addSong((Song) s);
    else {
        if (s instanceof Playlist)
            catalogue.addPlaylist((Playlist) s);
    }
}

public void addAuthor (Author a) {
    a.attach(this);
    catalogue.addAuthor(a);
}
```

Figure 15: Implementazione Observer push in Manager

```
protected void attach (Observer o) {
    observer = o;
}

protected void notifyManager (Playable s) {
        observer.update(s);
}
```

Figure 16: Implementazione Observable in Author

Ogni volta che un autore produce una nuova canzone o un nuovo disco, invoca notifyManager(), passando l'oggetto prodotto al Manager che poi lo aggiungerà al catalogo.

```java
public void newSong (String title, int length, String category) {
    songs.addSong(new Song(title, length, category, this));
    notifyManager(new Song(title, length, category, this));
}

public void newDisc (String name, Vector<Song> s) {
    songs.addSong(new Playlist(name, this, s));
    notifyManager((Playable) new Playlist(name, this, s));
    for (Song song : s)
        songs.addSong(song);
}
```

Figure 17: Chiamate a notifymanager() in Author

3.2.3 Protection Proxy

Il Proxy è un pattern strutturale che viene utilizzato per gestire l' accesso ad un oggetto complesso, il ConcreteSubject, attraverso un oggetto più semplice, il Proxy. In particolare nella sua variante Protection Proxy, esso fornisce un controllo sull' accesso dell' oggetto protetto attraverso un proxy intermedio. Per implementarlo, si definisce un' interfaccia Subject che espone i metodi per ottenere il comportamento atteso dal client dell' oggetto. Quindi ci saranno due classi che implementano l' interfaccia: l' oggetto concreto ed il proxy, che mantiene un riferimento all' oggetto concreto.
Viene utilizzato nel progetto per impedire agli utenti non iscritti al catalogo di poter scaricare o mettere like alle canzoni o alle playlist. Viene anche impedito il download agli utenti iscritti che non abbiano pagato l' abbonamento.
Il modo per impedire queste funzionalità è quello di richiedere un' autenticazione, dato che la classe ProtectionProxyCatalogue contiene una raccolta di tutte le credenziali degli utenti iscritti. Se l' utente risulta effettivamente iscritto il proxy consente l' accesso al catalogo, altrimenti no.

```java
public Playable likeSong (User u, String songName) {
    if (u instanceof Guest && !(u instanceof Subscriber)) {
        System.out.println("Error, you are not subscribed");
        return null;
        }
    else {
        if (u.getPassword().compareTo(subscribersCredentials.get(u.getNickName())) != 0) {
            System.out.println("Wrong password");
            return null;
        }
        else {
            return catalogue.likeSong(u, songName);
        }
    }
}
```

Figure 18: Proxy controlla le credenziali di chi richiede la funzionalità likeSong()

3.2.4 Prototype

Il Pattern Prototype è un design pattern creazionale che permette di copiare degli oggetti esistenti, senza rendere il codice dipendente dalle loro classi.
Si realizza definendo un interfaccia, detta Prototype, che espone un metodo per clonare l' oggetto che lo invoca. Le classi che la implementano sono i ConcretePrototype.
Nel progetto l' ho utilizzato per implementare il download delle canzoni e delle playlist all' interno del catalogo da parte degli utenti che ne hanno il permesso.
Il metodo che clona gli oggetti è download(), e richiede in input una stringa per poter personalizzare il nome dell' oggetto clonato.

```
public PrototypeDownloadable download (String title) {
    return new Song(title, this.length, this.category, this.author);
}
```

Figure 19: Metodo download() nella classe Song

```
public PrototypeDownloadable download (String title) {
    return new Playlist(title, this.author, this.songs.getSongs());
}
```

Figure 20: Metodo download() nella classe Playlist

4 Testing

Ogni classe di testing contiene un metodo setUp(), dove si inizializzano degli oggetti statici che poi serviranno nei vari test.
Solitamente vengono istanziati un Manager, un ConcreteMusicCatalogue, due Autori, che produrranno gli elementi che verranno inseriti nel catalogo, ed alcuni utenti, ma solo nelle classi che li interessano.
Gli autori non sono dichiarati statici, perchè vengono aggiunti al catalogo dai manager. Inoltre, sono stati considerati alcuni casi particolari per verificare la loro corretta gestione, ad esempio il fatto che due autori possano produrre canzoni con lo stesso nome.
Il catalogo gestisce questa situazione creando una playlist col titolo delle canzoni dal nome comune, ed in alcune classi di test viene verificato che la playlist sia effettivamente presente nel catalogo.

```
@Before
public void setUp() {
    manager = Manager.getInstance();
    catalogue = ConcreteMusicCatalogue.getInstance();
    ((ConcreteMusicCatalogue) catalogue).reset();
    Author a1 = new Author("Author1");
    Author a2 = new Author("Author2");
    manager.addAuthor(a1);
    manager.addAuthor(a2);
    a1.newSong("Song1", 135, "Category1");
    a2.newSong("Song2", 125, "Category2");
    a1.newSong("Song3", 100, "Category1");
    Vector<Song> songs = new Vector<Song>();
    songs.add(new Song("Song4", 300, "Category1", a1));
    songs.add(new Song("Song5", 300, "Category3", a1));
    a1.newDisc("Disc1", songs);
    a2.newSong("Song6", 320, "Category3");
    Vector<Song> songs2 = new Vector<Song>();
    songs2.add(new Song("Song6", 200, "Category3", a1));
    songs2.add(new Song("Song7", 340, "Category3", a2));
    manager.addPlaylist("Playlist1", songs2);
}
```

Figure 21: Metodo setUp() nella classe di testing TestSearching

4.0.1 TestSearching

Contiene dei metodi per testare le varie funzioni di ricerca del catalogo.
Ogni metodo di ricerca del catalogo restituisce un valore Booleano, così per sapere se la ricerca ha avuto successo o meno basterà eseguire la funzione assertTrue() o assertFalse(), e come argomento usiamo uno dei metodi di ricerca.
In particolare ho testato la ricerca normale, quella per autore e quella per genere musicale.

4.0.2 TestPlaying

Contiene dei metodi per testare le funzioni per riprodurre musica.
Come per i metodi di ricerca, anche quelli per riprodurre musica restituiscono un valore Booleano, e quindi anche qui ho usato le funzioni asserTrue() e assertFalse().
Ho testato la riproduzione normale, quella specificando l' autore della canzone o del disco, e quella specificando la categoria.

4.1 TestManager

E' una classe di test molto semplice, l' ho creata solo per verificare il corretto funzionamento dell' aggiunta di una playlist al catalogo creata dal Manager. Infatti potrebbe accadere che all' interno della playlist siano presenti canzoni di un autore non ancora presente nel catalogo, o che delle canzoni non siano ancora state aggiunte alla raccolta di un autore.
Con questa classe ho testato che la situazione sia gestita correttamente, aggiungendo l' autore al catalogo nel primo caso, e aggiungendo la canzone alla raccolta dell' autore nel secondo caso.

4.1.1 TestUsers

In questa classe ho testato le funzioni di ricerca, riproduzione e cambio credenziali per gli utenti. In particolare, le prime due funzionalità semplicemente richiamano i metodi già testati, mentre per il cambio di credenziali sono stati presi alcuni casi particolari gestiti nei metodi, ovvero quando due utenti scelgono lo stesso nickname.
Ho scelto di gestire questo caso aggiungendo un 2 come ultimo carattere del nickname richiesto, ed ho verificato che effettivamente il nickname assegnato all' utente era diverso rispetto a quello richiesto.
Per i cambi di password invece ho controllato che la password venisse aggiornata correttamente nella collezione di credenziali del ProtectionProxyCatalogue.

```
@Test
public void changeCredentials() {
    u3.changeNickName("n3");
    assertTrue(u3.getNickName().compareTo("n3") != 0);
    u3.changePassword("p3");
    assertTrue(u3.getPassword().compareTo("p2") != 0);
    u4.changeNickName("n32");
    assertTrue(u4.getNickName().compareTo("n32") != 0);
    u4.changePassword("p4");
    assertTrue(u4.getPassword().compareTo("p4") == 0);
}
```

Figure 22: Test changeCredentials() della classe di testing TestUsers

4.1.2 TestSubscribing

Ho testato il corretto funzionamento del processo di iscrizione al catalogo da parte di un Guest.
Per verificare che fosse avvenuto correttamente, ho provato ad aggiungere una canzone del catalogo alla sua collezione di canzoni piaciute, ed ho verificato che effettivamente essa non fosse vuota.
Nel metodo doubleSubscribe() invece, ho verificato che, se due Guest si iscrivono indicando uno stesso nickname, il secondo ad essersi iscritto avrà nickname pari a quello del primo concatenato ad un "2".

```
@Test
public void subscribe() {
    assertFalse(u1 instanceof Subscriber);
    u1 = ((Guest)u1).subscribe("n1", "p1", false);
    assertTrue(u1 instanceof Subscriber);
    ((Subscriber) u1).addToLiked("Song1");
    assertFalse(((Subscriber) u1).isEmpty(true));
    ((Subscriber) u1).playLiked(false);
}
```

Figure 23: Test subscribe() della classe TestSubscribing

4.1.3 TestLikeAndDownload

Infine, in questa classe ho testato i metodi per aggiungere canzoni alle collezioni private degli utenti.
Ho verificato che, dopo aver aggiunto canzoni alla lista di quelle piaciute o di quelle scaricate, la lista non risultasse vuota, e poi ho chiamato i metodi per riprodurre le liste.

```
@Test
public void download() {
    assertTrue(u3 instanceof Subscriber);
    assertTrue(((Subscriber) u3).isEmpty(false));
    ((Subscriber) u3).addToDownloaded("Song1", "song1");
    ((Subscriber) u3).addToDownloaded("Song4", "song4");
    assertFalse(((Subscriber) u3).isEmpty(false));
    ((Subscriber) u3).playDownloaded(false);
    u2 = u2.subscribe("n2", "p2", true);
    assertTrue(u2 instanceof Subscriber);
    assertTrue(((Subscriber) u2).isEmpty(false));
    ((Subscriber) u2).addToDownloaded("Song2", "song2");
    ((Subscriber) u2).addToDownloaded("Disc1", "disc");
    assertFalse(((Subscriber) u2).isEmpty(false));
    ((Subscriber) u2).playDownloaded(false);
}
```

Figure 24: test download() della classe TestLikeAndDownload