Digital Music Store

Davide Del Bimbo
davide.delbimbo@edu.unifi.it

Abstract

Il progetto consiste in uno store musicale digitale che permette agli utenti di gestire playlist, inclusa la creazione e l'eliminazione di playlist, nonché l'aggiunta e la rimozione di canzoni da esse. L'obiettivo principale è implementare l'applicazione utilizzando tecniche di build automation e seguendo i principi del Test-Driven Development (TDD) e del Behavior-Driven Development (BDD). Questo approccio garantisce un codice di alta qualità, riducendo al minimo i bug e migliorando la manutenibilità del software. Il sistema utilizza MongoDB per la gestione dei dati musicali e sfrutta una GUI Swing per l'interazione con l'utente.

Tale report vuole descrivere il design dell'applicazione con riferimenti ai pattern e agli strumenti utilizzati.

1. Introduzione

L'elaborato presenta un prototipo di applicazione desktop per uno store musicale digitale, focalizzato sulla gestione di playlist musicali. Gli utenti possono creare nuove playlist, eliminare quelle esistenti e aggiungere o rimuovere canzoni esplorando una lista di brani disponibili. Questo strumento è progettato per offrire un controllo semplice e intuitivo delle raccolte musicali, migliorando l'esperienza di gestione della musica personale.

L'applicazione è basata su un'architettura Model-View-Presenter (MVP) che consente di separare le responsabilità tra i componenti dell'applicazione, garantendo modularità e manutenibilità del codice (figura 1). In particolare:

- **Model**: il modello dell'applicazione comprende la rappresentazione dei dati, come canzoni e playlist. Questi dati sono gestiti in un database. Per l'implementazione è stato scelto un database non relazionale MongoDB.
- **View**: la vista dell'utente è rappresentata dall'interfaccia grafica dell'applicazione. Nell'implementazione è stato utilizzato Swing per creare una GUI, che include un menù di scelta delle playlist create dall'utente, una lista delle canzoni presenti nello store ed una lista delle canzoni aggiunte alla playlist. Inoltre, sono disponibili vari bottoni utili per effettuare operazioni come la creazione, eliminazione e modifica delle playlist.
- **Presenter** (o **Controller**): il Presenter, chiamato anche Controller, funge da intermediario tra la vista ed il modello. Contiene la logica di business dell'applicazione, inclusa la gestione degli eventi utente e l'interazione con il database.

Per garantire la qualità e la correttezza del software, il progetto è stato sviluppato combinando i principi di Test-Driven Development (TDD) e Behavioral-Driven Development (BDD). Il TDD ci ha permesso di scrivere test automatici prima di implementare il codice, assicurando che ogni unità del software soddisfacesse i requisiti specificati. Il BDD, invece, ci ha aiutato a scrivere scenari di test in un linguaggio naturale, facilitando la comunicazione tra sviluppatori e stakeholder.

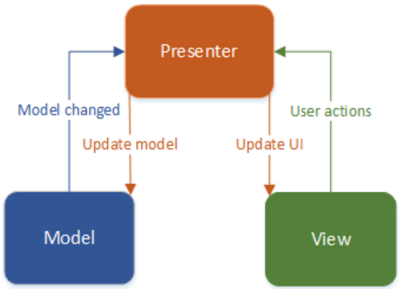


Figure 1: Architettura MVP.

2. Strumenti utilizzati

Il progetto è stato sviluppato come un'applicazione Java 8 utilizzando Maven come strumento di gestione delle dipendenze e di automazione della build. Inoltre, è stato adottato Eclipse come ambiente di sviluppo (IDE). Durante lo sviluppo sono stati impiegati diversi strumenti e framework, tra cui:

- **JUnit**: framework per l'esecuzione dei test.
- **AssertJ**: libreria che fornisce metodi di asserzione per rendere i test più leggibili e comprensibili.

- **Mockito**: framework di mocking che permette di simulare il comportamento di oggetti reali durante i test.
- **MongoDB**: database non relazionale per la memorizzazione e la gestione dei dati.
- **Picocli**: libreria per il parsing della riga di comando.
- **Log4J**: framework di logging per la gestione dei log dell'applicazione.
- **Cucumber**: framework per il Behavioral-Driven Development (BDD).
- **Docker**: framework per la creazione, la distribuzione e l'esecuzione di container.
- **JaCoCo** e **Coveralls**: strumenti per l'analisi della Code Coverage e per il monitoraggio della copertura dei test.
- **SonarCloud**: strumento per l'analisi statica del codice.
- **PIT**: framework per l'esecuzione di Mutation Testing.
- **GitHub Actions**: servizio di Continuous Integration (CI).

3. Behavioral-Driven Development

Il Behavioral-Driven Development (BDD) è una metodologia di sviluppo del software che si concentra sulla definizione del comportamento desiderato del sistema attraverso scenari di utilizzo, scritti in linguaggio naturale. Questi scenari sono utilizzati per definire i requisiti dell'applicazione e guidare lo sviluppo del software. Tale approccio facilita la comunicazione tra il team di sviluppo e gli stakeholder, garantendo una comprensione comune delle aspettative del software.

In questo progetto, è stato utilizzato il framework Cucumber per seguire l'approccio BDD. Tale framework consente la scrittura di scenari in linguaggio naturale tramite la sintassi Gherkin. Gli scenari scritti in Gherkin descrivono il comportamento atteso dell'applicazione in modo chiaro e comprensibile.

Durante lo sviluppo, gli scenari BDD sono stati tradotti in test automatizzati che verificano il comportamento dell'applicazione rispetto ai requisiti specificati. Questi test, insieme ai test unitari e di integrazione, forniscono una copertura completa dei requisiti funzionali dell'applicazione e contribuiscono alla creazione di un software solido e affidabile.
Il ciclo di sviluppo BDD include diverse fasi:

- Definizione dei casi d'uso: inizialmente sono stati identificati e definiti i casi d'uso principali che rappresentano le interazioni dell'utente con l'applicazione.
- Scrittura delle specifiche: per ogni caso d'uso, sono state scritte specifiche utilizzando la sintassi Gherkin che definiscono il comportamento atteso dell'applicazione in termini di "features" e "scenari".
- Scrittura dei test BDD: le specifiche Gherkin sono state utilizzate per automatizzare i test BDD tramite uno strumento compatibile, come Cucumber. Questi test servono come documentazione e insieme di test automatizzati per verificare che l'applicazione risponda correttamente ai comportamenti definiti nelle specifiche.

Un esempio di applicazione di questa metodologia è rappresentato dal seguente scenario che verifica la corretta selezione di una playlist dal menù:

```
Feature: Music Store View
  Specifications of the behavior of the
  ↪  Music Store View

  Background:
    Given the database contains a few songs
    And the database contains a playlist
    ↪  with a few songs in it
    And the Music Store view is shown

  Scenario: Select an existing playlist
    Given the user wants to select a
    ↪  playlist from the drop-down list
    When the playlist is selected
    Then all songs in the playlist are
    ↪  shown
```

Quindi, sono stati implementati i test che verificano tale scenario in classi Java dedicate come `DatabaseSteps.java` e `MusicStoreSwingSteps.java`.

Osserviamo che questo approccio comporta un tempo maggiore di completamento di un ciclo rispetto al semplice approccio TDD. Infatti, per soddisfare uno scenario, è necessario implementare tutte le funzionalità ed i requisiti richiesti da esso.

4. Domain Model

Il Domain Model è composto da due entità principali:

- **`Song`**. Questa classe rappresenta una singola canzone ed è caratterizzata da due attributi:
  - `title`: il titolo della canzone.
  - `artist`: l'artista che ha interpretato la canzone.

- **`Playlist`**. Questa classe rappresenta una raccolta di canzoni ed è definita da due attributi:
  - `name`: il nome della playlist.
  - `songs`: una lista di canzoni aggiunte alla playlist.

Entrambe le classi forniscono i metodi standard getter e setter per l'accesso e la gestione degli attributi, oltre ai metodi equals, hashCode e toString per supportare la comparazione e la rappresentazione delle istanze di queste classi. Inoltre, la classe `Playlist` implementa due semplici metodi per aggiungere e rimuovere canzoni dalla lista.

Osserviamo che, poiché si tratta di codice privo di logica, può essere escluso dall'analisi della Code Coverage e del Mutation Testing. Infatti, queste classi sono semplici contenitori di dati. Invece, la logica di business dell'applicazione sarà implementata altrove.

5. Unit Test

Il primo passo nel ciclo di sviluppo è stato quello di definire gli Unit Test per i singoli componenti. Seguendo l'approccio TDD, ogni componente è stato implementato riflettendo le specifiche fornite dai test. Questo approccio assicura una maggiore modularità e garantisce una migliore manutenibilità del codice, testando ogni componente in modo isolato.

5.1. Controller

La classe **`MusicStoreController`** è stata implementata a partire dalla relativa classe di test **`MusicStoreControllerTest`**. Questa classe funge da Controller dell'applicazione ed è responsabile di elaborare le richieste provenienti dalla View e di delegare le operazioni sul database al Repository, presentando successivamente i risultati all'utente.

Per garantire che il Controller venga testato in modo isolato, abbiamo utilizzato il framework Mockito per simulare il comportamento delle sue dipendenze. A tal fine, abbiamo definito le interfacce `MusicStoreRepository` e `MusicStoreView`, che stabiliscono i contratti da rispettare. Utilizzando Mockito, abbiamo creato istanze fittizie di queste interfacce (ancora non implementate) e abbiamo configurato i loro metodi tramite stubbing. Inoltre, l'API di Mockito mette a disposizione il metodo `Mockito.inOrder()` per verificare che alcune operazioni vengano eseguite in ordine (ad esempio, per verificare che il Repository venga aggiornato prima della View).

Al termine del ciclo di Unit Test, la classe `MusicStoreController` presenta i seguenti metodi:

- `allSongs()`: recupera tutte le canzoni dal Repository e le visualizza nella View.
- `allPlaylists()`: recupera tutte le playlist dal Repository e le visualizza nella View.
- `createPlaylist(Playlist playlist)`: crea la playlist nel Repository e la visualizza nella View. Se la playlist è già presente, mostra un messaggio di errore e aggiunge la playlist alla View.
- `deletePlaylist(Playlist playlist)`: elimina la playlist nel Repository e la nasconde dalla View. Se la playlist non è presente, mostra un messaggio di errore e rimuove la playlist dalla View.
- `allSongsInPlaylist(Playlist playlist)`: recupera tutte le canzoni dalla playlist specificata e le visualizza nella View.
- `addSongToPlaylist(Playlist playlist, Song song)`: verifica se la playlist esiste e se la canzone non è già presente in essa. Se la canzone è già presente, mostra un messaggio di errore ed aggiorna la View. Altrimenti, aggiunge la canzone alla playlist, aggiorna la playlist nel Repository e visualizza tutte le canzoni nella View.
- `removeSongFromPlaylist(Playlist playlist, Song song)`: verifica se la playlist esiste e se la canzone è presente in essa. Se la canzone non è presente, mostra un messaggio di errore ed aggiorna la View. Altrimenti, rimuove la canzone dalla playlist, aggiorna la playlist nel Repository e visualizza tutte le canzoni nella View.

Da notare che questa classe contiene tutta la logica di business dell'applicazione, pertanto è fondamentale sottoporla al Mutation Testing.

5.2. Repository

L'implementazione dell'interfaccia Repository è stata realizzata con il database non relazionale MongoDB. Questo tipo di database è efficace poiché semplice da configurare ed efficiente per operazioni CRUD come quelle presentate nel progetto. La classe **`MusicStoreMongoRepository`** è stata implementata a partire dalla relativa classe di test **`MusicStoreMongoRepositoryIT`**.

Poiché è necessario eseguire le operazioni su un database reale, che rappresenta un servizio di terze parti, è stato utilizzato un container Docker contenente l'immagine di MongoDB. Questo container deve essere avviato prima di eseguire i test. A tal fine, è stato utilizzato il plugin Docker Maven, che avvia il container una sola volta prima di eseguire i test durante la build con Maven, mappando una porta casuale dell'host per consentire l'interazione con il database.

In alternativa, per implementare il codice ed eseguire i test

localmente, è possibile avviare un container Docker con il seguente comando. Questo espone la porta `27017` per consentire l'interazione con il database dall'host e rimuove il container una volta terminato l'uso:

```
docker run -p 27017:27017 -rm
    mongo:6.0.14
```

Nonostante si utilizzi un servizio di terze parti, i test implementati possono essere considerati Unit Test, poiché una volta avviato il container, l'esecuzione dei test è piuttosto rapida. Pertanto, l'overhead maggiore è dovuto all'avvio e alla chiusura del container. Inoltre, questi test sono considerati Unit Test poiché sono necessari per implementare la classe concreta seguendo l'approccio TDD.

La scelta di utilizzare un container Docker anziché un database in-memory è motivata dal fatto che, poiché si sta implementando una classe che deve interagire con il database, utilizzare una rappresentazione meno potente del database potrebbe essere rischioso. Pertanto, si preferisce utilizzare il database reale per garantire l'affidabilità dei test.

Per realizzare dei test coerenti, prima di ogni test configuriamo il database in una situazione predefinita, ovvero un database vuoto. Questa configurazione iniziale assicura che ogni test parta da uno stato pulito e consenta di isolare i test, garantendo che i risultati non siano influenzati da dati residui di test precedenti. Il database MongoDB utilizza i `Document` come unità di memorizzazione dei dati, consentendo una struttura flessibile e schemaless.

Al termine del ciclo di Unit Test, la classe `MusicStoreMongoRepository` espone i seguenti metodi, i quali implementano le operazioni CRUD sfruttando l'API fornita dalla dipendenza di MongoDB:

- `findAllSongs()`: recupera tutte le canzoni dalla collezione `songCollection` nel database e le converte in una lista di oggetti `Song`.
- `initializeSongs(List<Song> songs)`: inizializza il database con una lista di canzoni. Osserviamo che il progetto si concentra sulla gestione delle playlist, assumendo che sia disponibile una lista di canzoni. Quindi, non viene implementata la gestione completa delle canzoni nello store musicale.
- `findAllPlaylists()`: recupera tutte le playlist dalla collezione `playlistCollection` nel database e le converte in una lista di oggetti `Playlist`.
- `findPlaylistByName(String playlistName)`: cerca una playlist per nome dalla collezione `playlistCollection` utilizzando un pattern case-insensitive. Se la playlist viene trovata, allora viene convertita in un oggetto `Playlist` e restituita. Altrimenti, viene restituito `null`.
- `createPlaylist(Playlist playlist)`: inserisce una nuova playlist nella collezione `playlistCollection`.
- `updatePlaylist(Playlist playlist)`: sostituisce una playlist esistente nella collezione `playlistCollection` con una nuova versione della stessa playlist, utilizzando il nome come identificatore con un pattern case-insensitive.
- `deletePlaylist(Playlist playlist)`: Rimuove una playlist dalla collezione `playlistCollection`, utilizzando il nome come identificatore con un pattern case-insensitive.

Osserviamo che questa classe non contiene particolare logica di business. Inoltre, a causa dell'overhead derivante dall'interazione con un servizio di terze parti, potrebbe risultare onerosa l'analisi tramite Mutation Testing. Di conseguenza, tale classe è stata esclusa da quest'ultimo.

5.3. View

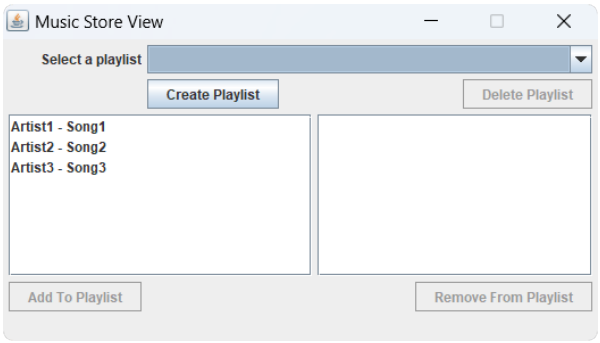
L'implementazione dell'interfaccia View è stata realizzata utilizzando il framework Swing insieme allo strumento Window Builder, disponibile per Eclipse. Tale strumento permette di costruire l'interfaccia grafica posizionando gli elementi desiderati e generando automaticamente il codice corrispondente. Successivamente, il codice generato viene personalizzato aggiungendo eventi ai pulsanti e alle selezioni di liste.

La GUI (figura 2) è costituita da una vista principale (figura 2a) per la gestione delle playlist e da una finestra di dialogo (figura 2b) per la creazione di una nuova playlist.

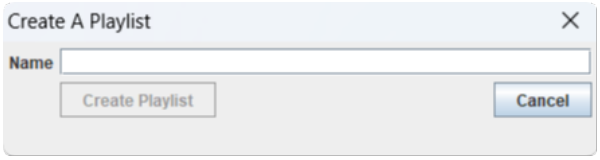
5.3.1 Main view

La classe **`MusicStoreSwingView`** è stata implementata seguendo l'approccio TDD, partendo dalla relativa classe di test **`MusicStoreSwingViewTest`**. Inizialmente, abbiamo scritto i test per garantire la presenza e la corretta configurazione degli elementi dell'interfaccia grafica, utilizzando la libreria AssertJ Swing. Successivamente, abbiamo proceduto con l'implementazione della logica degli eventi che gestiscono il comportamento dei pulsanti, la popolazione delle liste e l'invocazione dei metodi del Controller. Queste funzionalità sono state testate utilizzando il metodo `GuiActionRunner.execute()`, fornito da AssertJ Swing, che esegue una funzione lambda nell'Event Dispatch Thread (EDT).

Di seguito sono elencate le principali caratteristiche dell'interfaccia grafica:



(a) Vista principale dello store musicale.



(b) Finestra di dialogo per creare una nuova playlist.

Figure 2: Screenshot della GUI Desktop implementata con Swing.

- Il menù a discesa permette di selezionare una delle playlist create. Al momento dell'apertura della finestra, nessuna playlist viene selezionata per default. Una volta selezionata una playlist dalla lista, viene chiamato il metodo `allSongsInPlaylist` del Controller.
- Il pulsante "Create Playlist" è sempre abilitato ed apre la finestra di dialogo per la creazione di una nuova playlist.
- Il pulsante "Delete Playlist" è disabilitato di default e diventa attivo solo quando viene selezionata una playlist dal menù a discesa. Una volta premuto, chiama il metodo `deletePlaylist` del Controller passando la playlist selezionata dal menù.
- La lista delle canzoni nello store visualizza tutte le canzoni disponibili.
- La lista delle canzoni nella playlist mostra tutte le canzoni aggiunte alla playlist selezionata. Quando viene selezionata una playlist dal menù, qui vengono visualizzate le canzoni contenute in essa.
- Il pulsante "Add To Playlist" è disabilitato di default e diventa attivo solo quando viene selezionata una playlist dal menù a discesa ed una canzone dalla lista dello store. Una volta premuto, viene chiamato il metodo `addSongToPlaylist` del Controller passando la playlist selezionata dal menù e la canzone selezionata dalla lista delle canzoni nello store.
- Il pulsante "Remove From Playlist" è disabilitato di default e diventa attivo solo quando viene selezionata una playlist dal menù a discesa ed una canzone dalla lista della playlist. Una volta premuto, viene chiamato il metodo `removeSongFromPlaylist` del Controller passando la playlist selezionata dal menù e la canzone selezionata dalla lista delle canzoni nella playlist.
- L'etichetta di errore mostra un messaggio quando si verificano errori o condizioni non valide.

In questo contesto, il Controller è stato simulato per testare le interazioni con l'interfaccia. Inoltre, la finestra di dialogo è stata simulata per verificare che venga correttamente visualizzata quando richiesto.

Al termine del ciclo di Unit Test, la classe `MusicStoreSwingView` espone i seguenti metodi:

- `displayAllSongsInStore(List<Song> songs)`: aggiunge tutte le canzoni dello store musicale alla lista delle canzoni nello store.
- `displayAllPlaylists(List<Playlist> playlists)`: aggiunge tutte le playlist disponibili al menù di selezione delle playlist.
- `displayPlaylist(Playlist playlist)`: chiude la finestra di dialogo ed aggiunge una playlist al menù, selezionandola e resettando eventuali messaggi di errore. Osserviamo che, nel caso in cui la playlist creata venga nello stesso momento eliminata, potrebbe verificarsi una duplicazione di questa nella lista. Pertanto, se la playlist esiste già nel menù di selezione (con pattern case-insensitive), questa viene soltanto selezionata dal menù.
- `hidePlaylist(Playlist playlist)`: rimuove una playlist dal menù, resettando la selezione ed eventuali messaggi di errore.
- `displayAllSongsInPlaylist(List<Song> songs)`: visualizza tutte le canzoni nella playlist selezionata, sostituendo quelle precedentemente visualizzate e resettando eventuali messaggi di errore.
- `displayErrorAndAddPlaylist(String message, Playlist playlist)`: se la playlist è già presente nel menù di selezione, mostra

un messaggio di errore nella finestra di dialogo. Altrimenti, nel caso in cui la playlist che si vuole aggiungere è già stata creata ma non è presente nel menù di selezione (con pattern case-insensitive), aggiunge la playlist alla lista, chiude la finestra di dialogo, seleziona la playlist e mostra un errore nella finestra principale.

- `displayErrorAndRemovePlaylist(String message, Playlist playlist)`: visualizza un messaggio di errore e rimuove una playlist dal menù, resettando la selezione.
- `displayErrorAndUpdatePlaylist(String message, Song song, Playlist playlist)`: aggiorna la lista delle canzoni nella playlist chiamando il metodo `allSongsInPlaylist` del Controller e visualizza un messaggio di errore.

Osserviamo alcuni casi particolari:

- Quando viene selezionata una playlist, l'etichetta di errore viene resettata.
- La selezione di una canzone dallo store comporta un'eventuale deselezione della canzone nella playlist e viceversa.
- Quando si seleziona una playlist che è stata nello stesso momento rimossa, viene mostrato un messaggio di errore e la playlist viene rimossa correttamente dalla lista, resettando la selezione.

Poiché la classe segue una natura event-driven della GUI, l'analisi del Mutation Testing potrebbe risultare problematica. Inoltre, non rappresenta una classe progettata per implementare la logica di business dell'applicazione. Pertanto, questa classe viene esclusa dall'analisi.

5.3.2 Dialog

Analogamente, la classe **`CreatePlaylistDialog`** è stata implementata seguendo lo stesso approccio TDD, a partire dalla classe di test **`CreatePlaylistDialogTest`**. Di seguito sono elencate le principali caratteristiche della finestra di dialogo:

- La finestra viene sempre visualizzata in uno stato di default, sovrascrivendo il metodo `setVisible` per resettare la casella di testo, abilitare i pulsanti e ripristinare il messaggio di errore
- Il pulsante "Create Playlist" è abilitato solo quando la casella di testo non è vuota (a tale scopo si sfrutta la funzione `trim()` offerta dalla classe `String`). Una volta premuto, viene chiamato il metodo `createPlaylist` del Controller passando il nome fornito nella casella di testo.
- Il pulsante "Cancel" annulla l'operazione chiudendo la finestra di dialogo.
- L'etichetta di errore mostra un eventuale messaggio di errore.

6. Integration Test

Il secondo passo nel ciclo di sviluppo è stato definire gli Integration Test per i componenti precedentemente testati in isolamento, al fine di verificare la corretta integrazione tra di essi. Seguendo il principio della Pyramid Test, scriviamo un numero inferiore di test rispetto agli Unit Test, poiché questi test di integrazione possono essere ragionevolmente più lenti da eseguire, coinvolgendo l'integrazione di più componenti o di servizi terzi. Inoltre, si presuppone che i casi speciali testati durante questa fase siano già coperti positivamente dagli Unit Test.

6.1. Controller e Repository

In questa fase, verifichiamo l'integrazione del Controller con il Repository definendo la classe **`MusicStoreControllerIT`**. Durante questa verifica, la View viene simulata, poiché non siamo ancora interessati a verificare la sua integrazione con il Controller. Infatti, potrebbe non essere ancora stata implementata.
Inoltre, utilizziamo un vero database MongoDB per assicurarci che il Controller deleghi correttamente le operazioni da eseguire sul database al Repository e che il database risulti coerente con tali operazioni. A tale scopo, possiamo utilizzare un container Docker come descritto in precedenza. Il database viene configurato per avere uno stato predefinito prima di ogni test. In particolare, il dataset viene inizializzato con una lista di canzoni predefinita e la collezione delle playlist viene resettata.

In questi test, verifichiamo che il Controller aggiorni correttamente il database. Ad esempio, controlliamo che:

- Una playlist venga creata o eliminata correttamente.
- Una canzone venga aggiunta o rimossa correttamente ad una playlist.

6.2. View

Durante questa verifica, vogliamo testare la corretta integrazione tra la finestra principale e la finestra di dialogo definendo la classe **`MusicStoreSwingViewAndCreatePlaylistDialogIT`**. In questa classe, il Controller viene simulato poiché siamo interessati a verificare che gli eventi nella View comportino una corretta integrazione tra i diversi componenti.

In particolare, per verificare che la corretta creazione di una playlist comporti l'aggiornamento appropriato dell'interfaccia grafica, effettuiamo lo stubbing del metodo `createPlaylist` del Controller sfruttando le answer di Mockito. In questo modo, possiamo simulare il comportamento del metodo `createPlaylist` in modo che, quando viene chiamato, aggiunga la playlist alla casella di selezione delle playlist nella View, emulando così l'effetto di una reale interazione utente.

6.3. View e Controller

La verifica dell'integrazione tra la View e il Controller attraverso la classe **`MusicStoreViewIT`** ha lo scopo di garantire che il Controller aggiorni correttamente la View. In questo contesto, non ci interessa verificare l'integrità del database, quindi possiamo utilizzare un database MongoDB in-memory. Il database viene configurato per avere uno stato predefinito prima di ogni test. In particolare, il dataset viene inizializzato con una lista di canzoni predefinita e la collezione delle playlist viene resettata.

In questi test, verifichiamo che la GUI si comporti correttamente seguendo la logica del Controller reale. Ad esempio, controlliamo che:

- Le canzoni vengano correttamente visualizzate nella lista dello store.
- Le playlist vengano correttamente mostrate nel menù di selezione.
- La selezione di una playlist dal menù comporti la corretta visualizzazione delle canzoni nella lista.
- La selezione di una playlist non esistente comporti la rimozione di questa dalla lista e la visualizzazione di un messaggio di errore.

Osserviamo che, per evitare un ciclo nella dependency injection, la View viene passata nel costruttore del Controller ed il Controller viene impostato nella View tramite un metodo setter.

6.4. View, Controller e Repository

Questo test di integrazione, definito dalla classe **`ModelViewControllerIT`**, ha lo scopo di verificare la corretta integrazione tra tutti i componenti del sistema. In particolare, utilizziamo un database MongoDB reale per assicurarci che le azioni effettuate sulla View modifichino correttamente lo stato del database attraverso il Controller. A tale scopo, il database viene configurato per avere uno stato predefinito prima di ogni test. In particolare, il dataset viene inizializzato con una lista di canzoni predefinita e la collezione delle playlist viene resettata.

Ad esempio, controlliamo che:

- La creazione e l'eliminazione di una playlist avvenga correttamente sul database.
- L'aggiunta e la rimozione di una canzone da una playlist venga registrata correttamente nel database.

7. End-to-End Test

L'ultimo passo nel ciclo di sviluppo è stato definire gli End-to-End Test per tutti i componenti del sistema, verificando l'applicazione nella sua interezza. Seguendo il principio della Pyramid Test, scriviamo soltanto pochi test che interagiscono con l'interfaccia utente, senza dover chiamare direttamente i metodi delle classi sottostanti. A tale scopo, utilizziamo la GUI dell'applicazione per simulare il comportamento dell'utente.

Per eseguire l'applicazione da testare, definiamo un metodo di ingresso `main()` che lancia la finestra principale. Utilizziamo la libreria Picocli, un parser della riga di comando per Java, che ci permette di gestire gli argomenti passati dalla riga di comando. Nei test E2E, è necessario l'utilizzo di un database MongoDB reale, quindi ricorriamo ancora una volta ad un container Docker per fornire un ambiente di test isolato e riproducibile. Il database viene configurato in modo coerente per tutti i test, presentando una lista di canzoni disponibili nello store ed una collezione di playlist vuote.

I test E2E implementati adottano un approccio di sviluppo basato sul Behavioral-Driven Development (BDD) e mirano a verificare i requisiti specificati all'inizio dell'implementazione. Pertanto, questi test possono essere soddisfatti solo al termine dell'implementazione completa dell'applicazione. Ciò rende il ciclo di sviluppo più lungo e impegnativo rispetto al tradizionale approccio Test-Driven Development (TDD), dove i test sono scritti e soddisfatti man mano che i singoli componenti vengono implementati.

8. Version Control System

Abbiamo scelto GitHub come Version Control System per gestire lo sviluppo dell'applicazione. Ogni nuova funzionalità o modifica al codice è stata gestita attraverso la creazione di nuovi branch, successivamente integrati nel branch `main` tramite Pull Requests (PR). Le Issues sono state utilizzate per tenere traccia e risolvere problemi specifici. Il progetto è disponibile come repository GitHub all'indirizzo: Music Store Repository.

9. Continuous Integration Server

Abbiamo adottato GitHub Actions come server di Continuous Integration per eseguire build automatizzate e test nel nostro progetto. Per configurare GitHub Actions, è necessario creare uno o più file YAML di configurazione che

definiscono specifici workflow. Nel nostro caso, abbiamo implementato due diversi workflow:

- **`maven.yml`**: questo workflow viene eseguito ogni volta che avviene un push o una PR sul repository GitHub, escludendo i casi in cui le modifiche riguardano solo file non relativi al processo di build. Il workflow comprende le seguenti azioni:
  - Build del progetto su Java 8, Java 11 e Java 17.
  - Analisi della Code Coverage con Coveralls durante la build su Java 8, attivando il profilo Maven `jacoco`.
  - Analisi della Code Quality con SonarCloud durante la build su Java 17, attivando il profilo Maven `jacoco` ed il goal `sonar:sonar`.
- **`pitest.yml`**: questo workflow viene eseguito ogni volta che viene aperta una PR sul repository GitHub. Si occupa di condurre l'analisi del Mutation Testing prima dell'integrazione dei branch.

10. Code Coverage

Grazie all'approccio TDD, è stato possibile raggiungere una copertura del 100% in modo efficace. Tuttavia, per raggiungere questo obiettivo, sono state escluse dall'analisi della Code Coverage le classi `Song` e `Playlist`, poiché non contengono logica da testare, e la classe con il metodo `main()`, in quanto non ha senso effettuare il controllo della Code Coverage su di essa.

11. Mutation Testing

Nell'analisi del Mutation Testing, l'attenzione si concentra sul codice che racchiude la logica di business dell'applicazione, ossia il Controller. Di conseguenza, tutte le altre classi possono essere escluse dall'analisi. In particolare, attraverso la configurazione del plugin PIT di Maven, abbiamo abilitato il tipo di mutatori "Stronger".

12. Code Quality

Per ottenere un'analisi completa della qualità del codice, abbiamo dovuto escludere alcuni falsi positivi:

- Le classi `MusicStoreSwingView` e `CreatePlaylistDialog` presentano alcuni "code smell" associati alla regola `java:S117`, che riguarda i nomi delle variabili locali. Tuttavia, le variabili vengono generate automaticamente dal tool Window Builder, pertanto abbiamo escluso queste classi dalla regola.
- Le classi `MusicStoreSwingViewTest`, `CreatePlaylistDialogTest` e `MusicStoreSwingViewAndCreatePlaylistDialogIT` presentano alcuni "code smell" associati alla regola `java:S2699`, che riguarda la mancanza di asserzioni nei test. In realtà, questi test contengono delle asserzioni poiché verificano la presenza dei controlli utilizzando i metodi di AssertJ Swing. Tuttavia, SonarCloud non riconosce le asserzioni di AssertJ Swing, pertanto abbiamo escluso queste classi dalla regola.

13. Utilizzo

Dalla cartella contenente il file `pom.xml`, possiamo avviare la build dell'applicazione ed eseguire tutti i test con il seguente comando Maven:

```
mvn clean verify
```

Possiamo anche utilizzare i seguenti profili nei test:

- **Code Coverage**: `jacoco`.
- **Mutation Test**: `pitest`.

Inoltre, è possibile evitare specidici test utilizzando le seguenti variabili:

- **Unit Test**: `skipUT`.
- **Integration Test**: `skipIT`.
- **BDD Test**: `skipBDD`.

Prima di eseguire l'applicazione, dobbiamo avviare un container Docker per MongoDB con il seguente comando:

```
docker run -p 27017:27017 –rm mongo:6.0.14
```

Quindi, eseguiamo l'applicazione con il seguente comando:

```
java -jar target/music-store-0.0.1-jar-with-dependencies.jar
```

Inoltre, possiamo configurare la connessione a MongoDB con i seguenti parametri:

- `-mongo-host`: indirizzo host di MongoDB (default: `localhost`).
- `-mongo-port`: porta di MongoDB (default: `27017`).
- `-db-name`: nome del database (default: `music_store`).
- `-song-collection`: nome della collezione di canzoni (default: `songs`).
- `-playlist-collection`: nome della collezione di playlist (default: `playlists`).