|  |
| --- |
| Note musicali con riempimento a tinta unita  EMOTIONAL  SONGS  Manuale tecnico |
| Progetto Laboratorio Interdisciplinare A – A.A. 2021/22Università degli Studi dell’InsubriaLaurea triennale in Informatica **Sviluppato da:**  • Erik Gurzau, 749400, Varese  • Sara Biavaschi, 748698, Varese  • Alessia Metaj, 738945, Varese |



Versione 1.0

## Indice

Introduzione 3

Struttura generale delle classi 3

Scelta delle strutture dati 4

Gestione delle eccezioni 4

Architettura dell’applicazione 5

Formato e gestione dei file 5

Song 6

SongsManager 6

* Struttura dati 6
* Metodi principali 6
* Complessità stimate 7

Playlist 7

PlaylistsManager 8

* Struttura dati 8
* Metodi principali 8
* Complessità stimate 8

Emotion 9

EmotionsManager 10

* Struttura dati 10
* Metodi principali 10
* Complessità stimate 10

Person 10

User 11

* Metodi principali 11
* Complessità stimate 12

SecurePassword 12

* Complessità stimate 12

UsersManager 12

* Struttura dati 12
* Metodi principali 12
* Complessità stimate 13

Feedback 13

* Metodi principali 13
* Complessità stimate 13

FeedbackItem 14

FeedbacksManager 14

* Struttura dati 14
* Metodi principali 14
* Complessità stimate 15

FileManager 15

CommandManager 16

EmotionalSongs 16

* Funzionalità per utenti non registrati 17
* Funzionalità per utenti registrati 17
* Complessità stimate 17

Display 17

* Complessità stimate 18

**Introduzione**

Emotional Songs è un progetto sviluppato per il corso di Laboratorio Interdisciplinare A del corso di laurea triennale in Informatica dell’Università degli Studi dell’Insubria.

Esso è stato sviluppato in Java 17 e testato sul sistema operativo Windows 10.

**Struttura generale delle classi**

L’applicazione è composta dalle seguenti classi:

**Classi principali**

* EmotionalSongs (main)
* Display

**Managers**

* CommandManager
* EmotionsManager
* FeedbacksManager
* FileManager
* PlaylistsManager
* SongsManager
* UsersManager

**Classi per la definizione di oggetti**

* Emotion
* Feedback
* FeedbackItem
* Person
* Playlist
* Song
* User

**Altre classi ausiliarie**

* Address
* DisplayColors
* Input
* InputException
* SecurePassword
* TypeStreet
* UserException

Nei paragrafi seguenti verranno approfondite nel dettaglio tutte le classi, con i relativi metodi principali e l’analisi della loro complessità.

**Scelta delle strutture dati**

Sono state utilizzate come strutture dati principali **Vector** e **HashMap**.

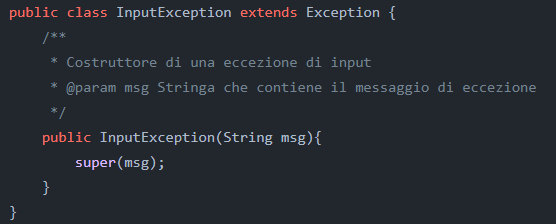
* Vector permette l'iterazione degli elementi contenuti in maniera molto efficiente e veloce, tuttavia l'inserimento e la cancellazione sono più costose in termine di tempo.
* HashMap è ideale per memorizzare coppie di chiave e valore e permette l'accesso rapido agli elementi grazie alla struttura basata sugli Hash. Inoltre, garantisce un buon funzionamento di operazioni come inserzioni, cancellazioni, ricerca e modifica.

Gli svantaggi sono invece la possibilità che possano verificarsi delle collisioni e il fatto che a volte, al crescere della tabella, le prestazioni peggiorano. Inoltre non garantisce un ordine degli elementi.

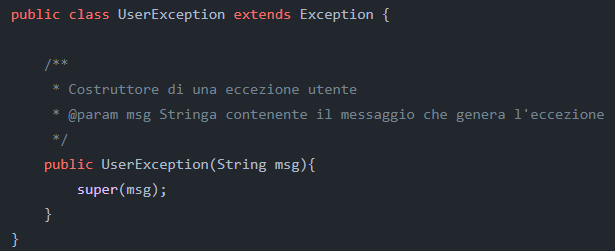
**Gestione delle eccezioni**

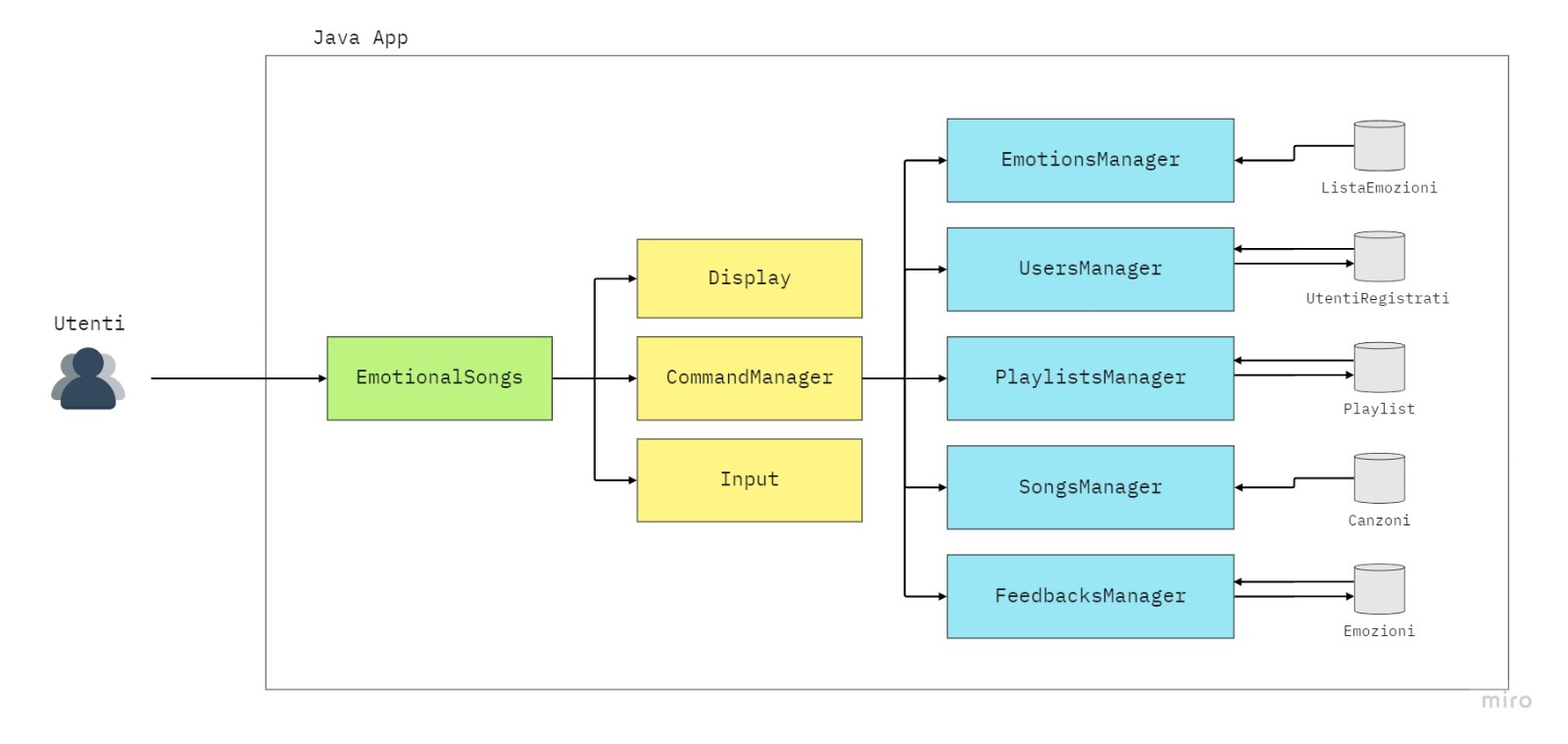
All’interno del programma sono state create delle classi per la gestione di alcune eccezioni. In particolare:

* **InputException**: definisce una eccezione riguardante la sezione di input.



* **UserException**: definisce una eccezione riguardante la sezione utente.



**Architettura dell’applicazione**

**Formato e gestione dei file**

Tutti i dati relativi alle canzoni, alle playlist, agli utenti registrati, alle emozioni e alle recensioni vengono salvati all’interno di file di testo in formato *.txt*.

Tutti i file *.txt* si trovano all’interno della cartella denominata *data*. Essi vengono utilizzati all’interno delle classi Manager, che prelevano i dati presenti e li elaborano.

**Song**

Questa classe è utilizzata per rappresentare gli oggetti di tipo Song, costituiti dalle singole canzoni presenti nel file Canzoni.txt.

Un oggetto di tipo *Song* possiede i seguenti campi:

* *id* di tipo primitivo int, che rappresenta l’id del brano
* *title* di tipo String, che rappresenta il titolo del brano
* *author* di tipo String, che rappresenta l’autore del brano
* *genre* di tipo String, che rappresenta il genere del brano
* *year* di tipo int di 4 cifre, che rappresenta l’anno di produzione del brano
* *duration\_ms* di tipo int di 6 cifre, che rappresenta la durata del brano in millisecondi

All’interno della classe è presente il metodo *millisToTime()*, utilizzato per la conversione della durata del brano da millisecondi al formato *minuti:secondi*.

**SongsManager**

Questa classe è responsabile della gestione degli oggetti di tipo Song, che sono salvati all’interno del file Canzoni.txt nel seguente modo:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1; | Titolo; | Autore; | Genere; | Anno; | Durata |

**Struttura dati**

Anche in questo caso viene utilizzata la struttura dati HashMap (**mapSongs**), che possiede come chiave l’ID della canzone e come valore l’oggetto Song.

**Metodi principali**

La classe contiene i metodi di ricerca delle canzoni, in particolare:

* *findSongsByTitle()*: per ricercare una canzone partendo dal titolo;
* *findSongsByAuthorAndYear()*: per ricercare una canzone partendo da autore ed anno.

**Complessità stimate**

* *parseData()*
* Permette di convertire le stringhe lette nel file Canzoni.txt in una HashMap. La complessità del ciclo for è quindi riconducibile ad O(n), dove n è il numero di stringhe contenute all’interno del file. Siccome il file di testo delle canzoni non verrà mai modificato durante l’utilizzo del programma, la complessità potrebbe ricondursi ad un costo costante.
* I metodi *put()* per aggiungere un nuovo elemento alla mappa e *add()* per aggiungere una canzone all’interno del vettore hanno costo O(1).
* *findSongsByTitle() e findSongsByAuthorAndYear()*
* Il ciclo for ha un costo ammortizzato costante, poiché scorre la lista di tutte le canzoni presenti nel programma, avente una lunghezza fissa.
* I metodi *contains()* per verificare se la stringa inserita come parametro è contenuta all’interno del titolo del brano analizzato e *add()* per aggiungere la canzone nel vettore rappresentante il risultato hanno costo O(1).

**Playlist**

Questa classe è utilizzata per la creazione di oggetti di tipo Playlist, creati a partire dagli input inseriti dall’utente richiesti a terminale, ovvero: nome della playlist e ID delle canzoni da inserire.

Un oggetto di tipo *Playlist* possiede i seguenti campi:

* *userId* di tipo primitivo int
* *name* stringa alfanumerica che rappresenta il nome della playlist scelto dall’utente
* *listSongs* vettore di tipo Integer contenente la lista delle canzoni

L’inserimento delle canzoni all’interno della playlist è gestito dal metodo *addSong()*, presente all’interno della classe.

**PlaylistsManager**

Questa classe è responsabile della gestione degli oggetti di tipo Playlist, i quali vengono salvati all’interno del file Playlist.txt nel seguente modo:

|  |  |  |
| --- | --- | --- |
| Playlist; | 1; | 45;34;56;67 |

*Nome playlist*

*ID utente*

*ID canzone*

**Struttura dati**

Le playlist sono salvate utilizzando come struttura dati una HashMap (**playlistMap**), che possiede come chiave l’ID utente e come valore la lista di playlist create dall’utente.

**Metodi principali**

Tra i metodi principali di questa classe troviamo:

* *isNameAvailable()*: controlla che il nome della playlist che si vuole creare non esista già per un determinato utente (il cui ID viene fornito come parametro).
* *getPlaylistByUserId()*: fornisce la lista delle playlist create da un utente in base al suo ID specificato nei parametri.
* *getPlaylistByName()*: fornisce una playlist, di un determinato utente, che ha il nome esattamente uguale al nome specificato nei parametri.

**Complessità stimate**

* *parseData()*
* Permette di convertire le stringhe lette nel file Playlist.txt in una HashMap. La complessità del ciclo for è quindi riconducibile ad O(n), dove n è il numero di stringhe contenute all’interno del file.
* Il metodo *containsKey()* possiede complessità O(1), così come i metodi *put()* per aggiungere un nuovo elemento alla mappa e *add()* per aggiungere una playlist all’interno del vettore.

Il metodo ha quindi una complessità stimata di O(n).

* *isNameAvailable()* e *getPlaylistByName()*

In questo metodo è necessario scorrere tutte le playlist create dall’utente che possiede l’Id fornito come parametro, e avrà perciò complessità O(p), dove p è il numero delle playlist da analizzare.

**Emotion**

Questa classe è utilizzata per rappresentare oggetti di tipo Emotion, costituiti dalle emozioni contenute nel file ListaEmozioni.txt.

Un oggetto di tipo *Emotion* possiede i seguenti campi:

* *id* di tipo int, che rappresenta l’id dell’emozione
* *category* di tipo String, che rappresenta il nome dell’emozione
* *explanation* di tipo String, che rappresenta la spiegazione dell’emozione

Le emozioni presenti in totale sono 9, ovvero:

|  |  |  |
| --- | --- | --- |
| 1 | **Amazement** | Feeling of wonder or happiness |
| 2 | **Solemnity** | Feeling of transcendence, inspiration. Thrills |
| 3 | **Tenderness** | Sensuality, affect, feeling of love |
| 4 | **Nostalgia** | Dreamy, melancholic, sentimental feelings |
| 5 | **Calmness** | Relaxation, serenity, meditativeness |
| 6 | **Power** | Feeling strong, heroic, triumphant, energetic |
| 7 | **Joy** | Feels like dancing, bouncy feeling, animated, amused |
| 8 | **Tension** | Feeling nervous, impatient, irritated |
| 9 | **Sadness** | Feeling depressed, sorrowful |

**EmotionsManager**

Questa classe è responsabile della gestione degli oggetti di tipo Emotion, che sono salvati all’interno del file ListaEmozioni.txt nel seguente modo:

|  |  |  |
| --- | --- | --- |
| 1; | Nome emozione; | Spiegazione emozione |

**Struttura dati**

In questo caso la lista di emozioni viene salvata all’interno di un oggetto di tipo *Vector<Emotion>*, ovvero array dinamico contenente oggetti di tipo Emotion.

**Metodi principali**

È presente il metodo *getListEmotions(),* che restituisce un vettore contenente tutte le possibili emozioni per la recensione di un brano.

**Complessità stimate**

* *parseData()*
* Permette di convertire le stringhe lette nel file ListaEmozioni.txt in una lista di emozioni salvata all’interno di un vettore. La complessità del ciclo for è quindi riconducibile ad O(n), dove n è il numero di stringhe contenute all’interno del file. Siccome il file di testo delle emozioni non verrà mai modificato durante l’utilizzo del programma, la complessità potrebbe ricondursi ad un costo costante.
* Il metodo *add()* per aggiungere un’emozione all’interno del vettore ha costo O(1).

**Person**

Questa classe viene utilizzata per rappresentare oggetti di tipo Person, che definiscono una persona prima che diventi un utente dell'applicazione tramite il completamento della registrazione.

Un oggetto di tipo *Person* possiede i seguenti campi:

* *name*di tipo String, che rappresenta il nome della persona
* *surname* di tipo String, che rappresenta il cognome della persona
* *cf* di tipo String, che rappresenta il codice fiscale della persona e che deve avere una lunghezza di 16 caratteri
* *address* di tipo Address, che rappresenta l’indirizzo della persona

L’oggetto di tipo *Address* è definito nella classe Address, ed è costituito dai campi typeStreet, nameStreet, houseNumber, postalCode, city e province.

Il campo typeStreet è un’oggetto di tipo TypeStreet definito all’interno di una classe enumerativa contenente i seguenti valori: “via”, “largo”, “piazza”, “corso”, “vicolo”, “viale”.

**User**

Questa classe estende la classe Person e viene utilizzata per rappresentare oggetti di tipo User, che definiscono gli utenti registrati all’interno dell’applicazione assegnandogli uno userId.

Un oggetto di tipo *User* possiede i seguenti campi:

* *name*, *surname*, *cf*, *address*, che vengono ereditati da Person
* *userId* di tipo int, che rappresenta l’ID dell’utente
* *email* di tipo String, che rappresenta l’email dell’utente
* *psw* di tipo String, limitato ad una lunghezza compresa tra 6 e 16 caratteri definiti dai campi final MIN\_LENGTH\_PSW e MAX\_LENGTH\_PSW

**Metodi principali**

All’interno della classe sono presenti i metodi per il controllo della validità dell’e-mail e della password, che devono rispettare determinati requisiti, ovvero:

* *isPswValid()*: lunghezza compresa tra i limiti, presenza di un carattere maiuscolo, di un numero e di un carattere speciale.
* *isEmailValid()*: il nome utente deve contenere lettere o numeri, non deve contenere solo numeri e non può iniziare con un numero; deve essere presente il DNS.
* *containsAllNumers()*: controlla che la stringa passata come parametro contenga solamente numeri. È utilizzata all’interno di *isEmailValid().*

**Complessità stimate**

* *isPswValid()*

Il metodo analizza tutti i caratteri della stringa fornita come parametro: dato che la password non può superare la lunghezza di 16 caratteri, possiamo considerare il metodo come avente costo costante.

* *containsAllNumbers()*

Possiede complessità O(n), dove n è il numero di caratteri della stringa da controllare.

**SecurePassword**

Questa classe viene utilizzata per criptare la password associata ad un utente al momento della registrazione.

Possiede inoltre il metodo *genPsw()*, che consente la generazione di una password casuale che verrà visualizzata solamente una volta al momento della registrazione.

**Complessità stimate**

* *genPsw()*

Possiede costo costante, poiché all’interno del metodo il ciclo for è implementato per scorrere al massimo 16 caratteri.

**UsersManager**

Questa classe è responsabile della gestione degli oggetti di tipo User, che sono salvati all’interno del file UtentiRegistrati.txt nel seguente modo:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ID utente; | e-mail; | password criptata; | nome; | cognome; | CF; | indirizzo |

**Struttura dati**

Anche in questo caso viene utilizzata la struttura dati HashMap (**mapUsers**), che possiede come chiave l’e-mail e come valore l’oggetto User.

**Metodi principali**

All’interno di questa classe troviamo i seguenti metodi:

* *nextUserId()*, che assegna ad un nuovo utente che si vuole registrare l'ultimo ID disponibile.
* *register()*, che registra un nuovo utente nell'applicazione.
* *login()*, che effettua i controlli per l’autenticazione dell’utente a partire da e-mail e password.

**Complessità stimate**

* *parseData()*
* Permette di convertire le stringhe lette nel file UtentiRegistrati.txt in una HashMap di utenti. La complessità del ciclo for è quindi riconducibile ad O(n), dove n è il numero di stringhe contenute all’interno del file.
* Il metodo *put()* per aggiungere un nuovo elemento alla mappa ha costo O(1).
* *getUserById()*

In questo metodo è necessario scorrere tutti gli utenti che hanno effettuato la registrazione all’applicazione e avrà perciò complessità O(u), dove u è il numero di utenti registrati.

* Tutti gli altri metodi possiedono complessità O(1) o comunque inferiore a *getUserById()*.

**Feedback**

Questa classe è utilizzata per definire oggetti di tipo Feedback per permettere la gestione dell’inserimento di recensioni emozionali di un brano da parte dell’utente. Tutte le recensioni vengono salvate nel file Emozioni.txt.

Un oggetto di tipo feedback possiede i seguenti campi:

* *namePlaylist* di tipo String, che rappresenta il nome della playlist in cui è contenuto il brano che l’utente desidera recensire
* *userId* di tipo int, che rappresenta l’id dell’utente che crea la recensione
* *songId* di tipo int, che rappresenta l’id del brano da recensire

**Metodi principali**

Tra i metodi di questa classe troviamo:

* *addItem()*, che aggiunge nella lista dei feedback il dettaglio recensione fornito come parametro (di tipo FeedbackItem);
* *getFeedbackItemByEmotionId()*, che restituisce un dettaglio della recensione in base all’ID dell'emozione fornita come parametro.

**Complessità stimate**

* *addItem()*

Ha complessità O(1), poiché aggiunge un elemento in un vettore.

* *getFeedbackItemByEmotionId()*

Ha complessità O(f), dove f rappresenta il numero di singole recensioni di ogni emozione presenti all’interno dell’applicazione.

**FeedbackItem**

Questa classe definisce il dettaglio di una recensione, riferita ad una emozione, per una determinata canzone.

Un oggetto di tipo FeedbackItem possiede i seguenti campi:

* *emotionId* di tipo int, che rappresenta l’Id dell’emozione
* *score* di tipo int, che rappresenta il punteggio da 1 a 5 assegnato all’emozione

Il campo *note* di tipo String, che rappresenta l’eventuale commento rilasciato dall’utente per una determinata emozione, è gestito in maniera differente in base al costruttore utilizzato, poiché viene data possibilità all’utente di non inserire alcun commento.

**FeedbacksManager**

Questa classe è responsabile della gestione degli oggetti di tipo Feedback, che sono salvati all’interno del file Emozioni.txt nel seguente modo:

Recensione senza nota

Recensione con nota

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| NomePlaylist; | ID utente; | ID Canzone | 1,5, commento; | 2,4; |

1 2

1 2

*1 – ID emozione*

*2 – punteggio emozione*

**Struttura dati**

Anche in questo caso viene utilizzata la struttura dati HashMap (**mapFeedback**), che possiede come chiave con l’ID della canzone e come valore l’oggetto Feedback.

**Metodi principali**

All’interno della classe sono presenti i metodi:

* *countFeedback()*, che conta il numero di recensioni di una canzone all'interno dell'applicazione, e *totScoreFeedback()*, che esegue la somma totale dei punteggi di una emozione X rispetto ad una canzone Y; essi vengono successivamente utilizzati per il calcolo del punteggio medio associato ad ogni emozione;
* *getFeedbackIfHasNote()*, che permette di salvare all’interno di un vettore la lista di tutti i commenti rilasciati dagli utenti riguardo ad un’emozione di un brano;
* *saveFeedback()*, che salva le informazioni della recensione emozionale di una canzone della playlist nel formato prestabilito;
* *hasFeedback()*, che verifica che la canzone con l'ID specificato come parametro abbia almeno una recensione in tutta l'applicazione oppure che l'utente con l'ID specificato abbia recensito una specifica canzone in una playlist;
* *addFeedback()*, che aggiorna la mappa aggiungendo una nuova recensione alla lista di recensioni di una canzone il cui Id è fornito come parametro.

**Complessità stimate**

* *parseData()*
* Permette di convertire le stringhe lette nel file Emozioni.txt in una lista di recensioni. La complessità del ciclo for è quindi riconducibile ad O(n), dove n è il numero di stringhe contenute all’interno del file.
* Il metodo *addItem()* per aggiungere un dettaglio all’interno di una recensione ha costo O(1).
* Il metodo *addFeedback()* ha complessità O(p), dove p è il numero di recensioni da aggiungere
* *hasFeedback(), countFeedback(), addFeedback(), totScoreFeedback() e getFeedbackIfHasNote()*

In questi metodi è necessario scorrere tutte le recensioni create dagli utenti, e avrà perciò complessità O(r), dove r è il numero delle recensioni presenti nell’applicazione.

* *saveFeedback()*

Questo metodo ha complessità O(f), dove f è il numero di recensioni che devono essere salvate.

**FileManager**

Questa classe si occupa dell’importazione e dell’elaborazione dei file di testo forniti come input.

Viene infatti utilizzata all’interno degli altri manager presenti nell’applicazione per salvare oppure leggere i dati all’interno dei file .txt presenti nella cartella “data”, creando oggetti FileManager che ricevono come input il percorso del file necessario.

**CommandManager**

Questa classe consente il controllo e la gestione di tutti i manager dell’applicazione.

Il costruttore della classe contiene infatti tutti i gestori e permette l’utilizzo dei relativi metodi in essi definiti.

All’interno della classe sono inoltre implementati i metodi *login()* e *logout()*, che permettono agli utenti registrati di accedere all’applicazione oppure di scollegarsi dall’account.

**EmotionalSongs**

La classe EmotionalSongs contiene il metodo *main()*, che permette l’avvio e l’esecuzione dell’applicazione e definisce il menu principale.

All’interno del menu sono presenti nove possibili opzioni, che possono essere selezionate dall’utente digitando il numero corrispondente.

Questo è reso possibile grazie all’utilizzo dell’istruzione **Switch Case**, che permette di eseguire le diverse funzionalità tramite il richiamo dei metodi statici definiti all’interno della classe.

**Funzionalità per utenti non registrati**

In particolare, gli utenti non registrati possono usufruire solamente di alcune funzionalità, definite dai seguenti metodi:

* *registrazione()*, per registrarsi all’applicazione (2);
* *cercaBranoMusicale()*, per ricercare brani tramite titolo oppure autore ed anno (3);
* *reportBrano()*, per visualizzare un report delle recensioni associate alla canzone selezionata (7);
* *visualizzaCatalogoCanzoni()*, per visualizzare tutte le canzoni (9);
* uscire dall’applicazione (0).

**Funzionalità per utenti registrati**

Gli utenti registrati che effettuano l’accesso possono usufruire di ulteriori funzionalità, definite dai seguenti metodi:

* *login()*, per effettuare il login/logout (1);
* *creaPlaylist()*, per creare le proprie playlist (4);
* *inserisciEmozioniBrano()*, per recensire le canzoni inserendo l’intensità delle emozioni provate durante l’ascolto insieme ad eventuali commenti (6);
* *reportPlaylist()*, per visualizzare un report emozionale di tutte le canzoni presenti all’interno della playlist selezionata (8).

**Complessità stimate**

Questa classe presenta al suo interno metodi che sono già stati analizzati precedentemente poiché implementati in altre classi.

**Display**

È la classe che gestisce l’**interfaccia utente**.

Permette di visualizzare il menu all’avvio dell’applicazione e di formattare gli output nel terminale in diversi colori e stili, avvalendosi della classe *DisplayColors*, contenente i possibili colori.

Questa classe dispone inoltre di metodi per stampare sotto forma di tabella le liste delle canzoni, delle playlist e dei report emozionali e per visualizzare a terminale il nome e l’e-mail dell’utente che ha effettuato l’accesso.

**Complessità stimate**

* *printListSongs()*

Questo metodo viene utilizzato per stampare una lista di canzoni all’interno di una tabella. La complessità in generale sarebbe quindi O(n), dove n rappresenta il numero di canzoni da stampare, ma esse hanno un limite massimo (5000), quindi il costo può essere considerato costante.

* *printPlaylist()*

Questo metodo viene utilizzato per stampare una lista di playlist, perciò possiede una complessità pari a O(p)+O(b), dove p è rappresenta il numero di playlist da stampare e b rappresenta il numero di brani presenti nella playlist contenente il maggior numero di canzoni.

* *printReportSong()*
* all’interno del ciclo for viene fatta scorrere la lista delle emozioni, ottenendo quindi un costo costante poiché il numero di emozioni presenti nell’applicazione non varia.
* il metodo *hasFeedback()*, come anche il metodo *countFeedback()* e *totScoreFeedback()*, hanno complessità O(r), dove r è il numero delle recensioni presenti nell’applicazione.

Il metodo ha quindi complessità O(r).

* *printReportPlaylist()*
* Primo ciclo for: viene fatta scorrere la lista delle emozioni (costo costante).
* Secondo ciclo for: O(c), dove c rappresenta il numero dei brani all’interno della playlist di cui si vuole stampare il report.
* I metodi *countFeedback()* e *totScoreFeedback()*, hanno complessità O(r), dove r è il numero delle recensioni presenti nell’applicazione.
* *printComments()*
* Primo ciclo for: viene fatta scorrere la lista delle emozioni (costo costante).
* Secondo ciclo for: costo O(f), dove f è il numero di recensioni ricevuti dal brano fornito come parametro.
* *getFeedbackIfHasNote*() ha costo O(r), dove r è il numero delle recensioni presenti nell’applicazione.