*Corso di Laurea in Informatica, prof. A. De Lucia,*

*a.a 2021/2022*

*Progetto di Ingegneria del Software*



*Object Design Document*

|  |  |
| --- | --- |
| Partecipanti | Matricola |
| Marta Coiro | 0512108154 |
| Katia Buonocore | 0512106528 |
| Rita Cuccaro | 0512109495 |
|  |  |
|  |  |
|  |  |

**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| Data | Versione | Descrizione | Autore |
| 16/11/2021 | 1.0 | Prima stesura del documento(Problem [Statement](#_top)) | Membri del team |
| 30/11/2021 | 1.0 | Requirement Analysis Document | Membri del team |
| 06/12/2021 | 1.0 | System Design Document | Membri del team |
| 20/12/2021 | 1.0 | Gestione Dati Persistenti\_MusicConsole | Membri del team |
| 27/12/2021 | 1.0 | Object Design Document | Membri del team |
|  |  | Test Plan | Membri del team |
|  |  |  |  |

Indice

1. INTRODUZIONE

1.1 Object Design Trade-Offs.

1.1.2 Design Pattern

1.1.2.1 Singleton

1.1.2.2 Observer Design Pattern

1.1.2.3 Data Access Object(DAO)

1.2 Linee Guida per la Documentazione delle Interfacce.

1.3 Definizioni, acronimi e abbreviazioni.

1.4 Riferimenti.

2. PACKAGES

3. CLASS INTERFACES

4. GLOSSARIO

**1. INTRODUZIONE**

Dopo la realizzazione dei documenti RAD e SDD abbiamo descritto, in linea di massima, quello che sarà il nostro sistema e quindi i nostri obiettivi, tralasciando gli aspetti dell’implementazione. Il seguente documento ha lo scopo di produrre un modello capace di integrare in modo coerente e preciso tutte le diverse funzionalità individuate nelle fasi precedenti. In particolare, in questo documento si vanno a descrivere i trade-offs generali realizzati dagli sviluppatori, le linee guida sulla documentazione delle interfacce e le convenzioni di codifica, una panoramica dei design pattern utilizzati dalle classi che compongono il sistema e dei packages che le contengono e una descrizione delle interfacce dei sottosistemi individuati.

**1.1 Object Design Trade-offs**

Seguono i trade-offs trovati nello sviluppo del sistema.

* **Comprensibilità vs Tempo**: Il codice deve essere al quanto più comprensibile per poter facilitare la fase di testing ed eventuali future modifiche del codice. A tale scopo, il codice sarà quindi accompagnato da commenti che ne semplifichino la comprensione. Questa caratteristica incrementerà il tempo di sviluppo, ma allo stesso tempo lo renderà più comprensibile.

* **Interfaccia vs Usabilità**: Il sistema verrà sviluppato con un’interfaccia grafica realizzata in modo da poter essere molto semplice, chiara ed intuitiva. Nell’interfaccia saranno presenti form, menù e pulsanti, disposti in maniera da rendere semplice l’utilizzo del sistema da parte dell’utente finale.

● **Sicurezza vs Efficienza**: La sicurezza, come descritto nei requisiti non funzionali, rappresenta uno degli aspetti importanti del sistema. A causa dei tempi di sviluppo molto limitati, ci limiteremo ad implementare un sistema di sicurezza basato sull’utilizzo di username e password degli utenti.

● **Prestazioni vs Costi**: Il sistema fa affidamento a tecnologie open source non richiedenti licenze a pagamento per l’utilizzo di queste, tuttavia ampiamente e notoriamente diffuse, come il sistema di gestione di basi di dati relazionali ​ MySQL​ e l’application server Apache TomCat.

**1.1.2 DESIGN PATTERN**

**Immagine che contiene tavolo

Descrizione generata automaticamente 1.1.2.1 Singleton**

Il design pattern del ​singleton​ permette di restringere il numero di istanze di una classe ad una singola istanza.

**Funzionalità**

Il pattern del singleton permette la riduzione della memoria utilizzata per le classi che non hanno bisogno di mantenere uno stato o che possono usufruire di uno stato condiviso tra i suoi utilizzatori, in quanto esiste una singola istanza per ogni classe che adotta l’utilizzo del pattern. Il design pattern del singleton viene ampiamente utilizzato per l’accesso centralizzato a funzionalità e componenti in un’unica entità condivisa dai suoi utilizzatori.

**Utilizzo**

Il sistema utilizza il design pattern del singleton per la maggior parte delle classi che non mantengono uno stato o con stato condiviso, che quindi possono essere accedute concorrentemente, come classi d’utilità, classi realizzanti i servizi che implementano le diverse funzionalità individuate (funzionalità riguardanti gli account utenti, le recensioni, il catalogo, ecc.), o classi che si occupano di mantenere in memoria i risultati delle richieste recenti sul catalogo.

**Nel nostro caso utilizziamo il Singleton Pattern per collegare le classi al nostro DB tramite la classe:**

**DriverManagerConnectionPool.java.**

**1.1.2.2 Observer Design Pattern(DAO)**

Il design pattern dell’​observer​ è un pattern in cui un oggetto detto ​subject​ notifica un insieme di ​osservatori​ riguardo i suoi cambi di stato o esecuzioni di azioni su questo.

**Funzionalità**

Il pattern observer viene generalmente usato in sistemi di gestione di eventi in cui il subject viene detto “sorgente del flusso di eventi”, mentre gli observer vengono denominati “pozzi di eventi”. Questo pattern risulta particolarmente utile per soddisfare dipendenze tramite accoppiamento debole.

**Utilizzo**

Il pattern observer viene utilizzato dal sistema per aggiornare le sessioni degli utenti autenticati alla piattaforma se lo stato del loro account utente viene aggiornato, per mantenere le sessioni consistenti con lo stato dell’account utente. Viene inoltre utilizzato per aggiornare il contenuto delle cache implementate dall’applicazione in maniera trasparente, senza richiedere alcuna azione esplicita dall’utilizzatore. Il pattern observer può inoltre essere utilizzato per risolvere problemi di logica trasversale, come il logging delle operazioni svolte dal sistema.

**1.2.2.3 Data Access Object**

Il design pattern del ​data access object​ (​DAO​) è un pattern utilizzato per fornire un’interfaccia ad una o più basi di dati o altri tipi di meccanismi di persistenza.

**Funzionalità**

Il pattern del DAO fornisce l’accesso a delle operazioni specifiche sui dati persistenti senza esporre dettagli sulla base di dati sottostante attraverso la mappatura di chiamate interne all’applicazione al layer di persistenza. Un data access object permette di separare l’insieme di dati di cui un’applicazione richiede l’utilizzo da come gli stessi vengono trattati ad un livello più basso, permettendo all’utilizzatore di ignorare la tipologia di base di dati sottostante e le strutture dati che utilizzate.

**Utilizzo**

Il sistema adotta il pattern DAO per le classi che lavorano con dati persistenti.

**1.2 Linee Guida per la Documentazione delle Interfacce**

Per assicurare un’alta leggibilità e di conseguenza una buona manutenibilità del codice, gli sviluppatori devono sottostare a delle linee guida per la stesura del codice.

**1.2.1 Code Style**

Gli sviluppatori devono attenersi quanto più possibile a delle linee guida ben precise dettate dalle convenzioni di stile del codice ​Google Java​. In particolare, il codice deve essere indentato utilizzando tab e non spazi, ed essere formattato seguendo uno stile conciso e comprensibile.

**1.2.2 Naming convention**

E’ buona pratica che i nomi da utilizzare siano:

* Descrittivi.
* Pronunciabili.
* Di uso comune.
* Non abbreviati(se non per variabili temporanee).
* Contenenti esclusivamente caratteri consentiti(A-Z,a-z,0-9).

**1.2.2.1 Variabili**

Anche i nomi delle variabili locali, istanze di variabili, e variabili di classe devono rispettare lo stile ​lower camel case​. I nomi delle variabili non dovrebbero iniziare per underscore (​\_) o per il simbolo del dollaro (​$), anche se entrambi sono permessi. È importante far sì quanto più possibile che i nomi delle variabili siano corti ma significativi; la scelta del nome di una variabile dovrebbe essere mnemonica. Nomi di variabili composte da un singolo carattere andrebbero evitati se non per variabili temporanee come contatori o variabili di buffer.

Esempi:

● int i;

● char c;

● float numeroAscoltatori;

**1.2.2.2 Metodi e funzioni**

I nomi dei metodi e delle funzioni devono rispettare lo stile ​lower camel case​ (o Dromedary case​), dove la prima lettera della prima parola è in minuscolo e la prima lettera di ogni parola successiva è in maiuscolo. I nomi di metodi e funzioni devono essere composti da verbi o da più parole che cominciano per un verbo.

Esempi:

● run();

● getName();

**1.2.2.3 Classi**

I nomi delle classi devono rispettare lo stile ​upper camel case​ (o​ Pascal case​), dove la prima lettera di ogni parola è in maiuscolo. I nomi delle classi non dovrebbero contenere acronimi o abbreviazioni, a meno che l’abbreviazione non sia molto più diffusa della forma normale, come ad esempio URL o HTML.

Esempi:

● class User {}

**1.2.2.4 Costanti**

I nomi delle costanti dovrebbero essere composte da parole in maiuscolo separate da underscore. I nomi delle costanti possono anche contenere cifre se appropriato, ma non come primo carattere.

Esempi:

● static final int MAX\_LENGTH = 10;

● const ERROR\_401 = “Unauthorized”;

**1.3 Definizioni, acronimi e abbreviazioni**

● RAD : Requirements Analysis Document

● SDD : System Design Document

● ODD : Object Design Document

**1.4 Riferimenti**

● Documento RAD del progetto MusicConsole.

● Docuemento Dati Persistenti del progetto MusicConsole.

**2. PACKAGES**

**4. GLOSSARIO**

* Componenti off-the-shelf: prodotti software sviluppati da terzi riutilizzabili.
* CSS: Linguaggio per la definizione degli stili delle pagine web.
* Framework: Software di supporto allo sviluppo.
* HTML: Linguaggio per la strutturazione delle pagine web.
* JavaScript: Linguaggio di scripting nato per dare dinamicità alle pagine HTML.