Theta Sport

Relazione progetto Programmazione III

Attilio Di Vicino, Mario Vista, Lorenzo Pergamo

A.A. 2022/2023

Indice

Elenco delle figure

Capitolo 1

Presentazione

Theta Sport è un e-commerce di articoli sportivi. Divertiti a navigare e comprare i numerosi prodotti presenti all'interno del sito e, solo per te, verranno proposte offerte personalizzate.

1.1 Analisi dei Requisiti

L'utente una volta aperto il sito potrà subito navigare tra gli articoli proposti, per poter usufruire di tutti i servizi dovrà però eseguire l'accesso al proprio account o, in caso non ne avesse uno, crearlo. Una volta fatto ciò l'utente potrà liberamente aggiungere o rimuovere prodotti dal carrello e, una volta soddisfatto, procedere con l'ordine. Al momento dell'ordine potrà autonomamente scegliere il metodo di pagamento tra bancomat, contati o carta di credito e, una volta inserite tutte le informazioni necessarie per la spedizione e la consegna, procedere con l'ordine. È possibile anche effettuare un accesso come admin, tramite apposite credenziali,la cui home page avrà un' interfaccia completamente diversa e più complessa, nella quale avrà la possibilità di tenere d'occhio ogni informazione utile alla gestione del sito come ad esempio i guadagni dell'ultimo mese. Tramite la sidebar potrà accedere ad un menu nel quale potrà aggiungere o modificare dei prodotti per categoria all'interno del catalogo, visualizzare aggiornamenti sulle vendite per categoria e inviare offerte personalizzate agli utenti.

1.2 Glossario

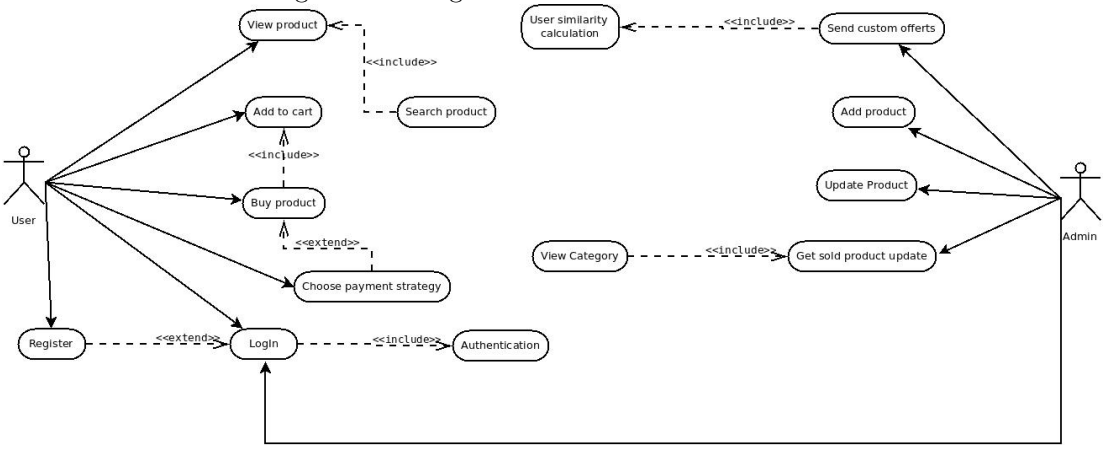
In questa sezione vengono chiariti alcuni termini utilizzati nel corso della relazione.

Glossario	
Termine	**Significato**
Categoria	Categoria di appartenenza del prodotto(es: Football, Tennis).
Sotto categoria	Sotto categoria di appartenenza del prodotto(es: t-shirt, shoes).
Ruolo	Admin o User.

1.3 Use Case

Di seguito è possibile osservare lo use case della web app, con tutte le possibili azioni che user e admin possono compiere.

Figura 1.1: Diagramma Use case



1.4 Database

Per il seguente progetto è stato creato un database in MySql per conservare alcuni dati utili per l'utilizzo del sito. In particolare sono presenti le tabelle Utente, Prodotto e Ordine, contenente le rispettive informazioni. Tutte le informazioni sui prodotti, sugli ordini e sugli utenti di cui è possibile usufruire nell'utilizzo dell'e-commerce, sono tutte recuperate tramite query nel database, utilizzando il pattern architetturale DAO.

Figura 1.2: Diagramma EE/R del database

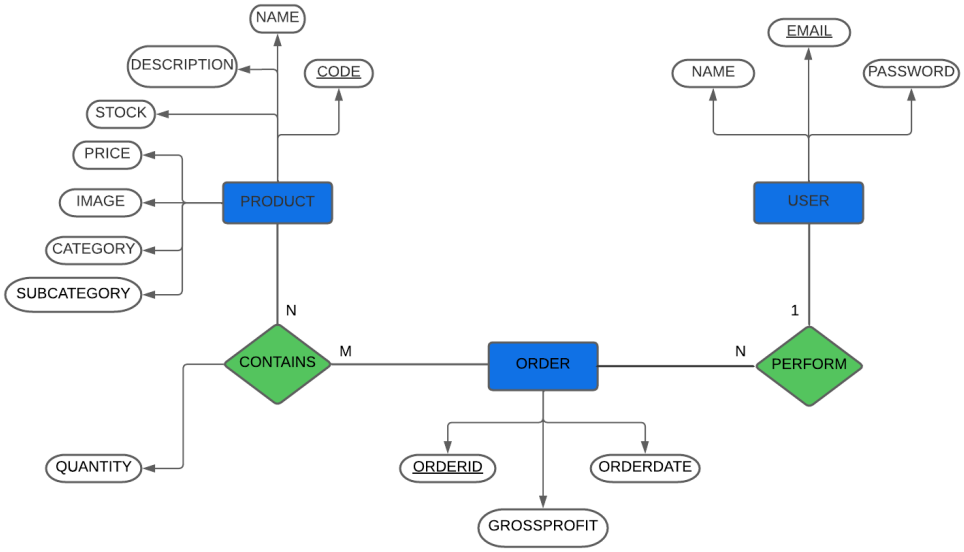
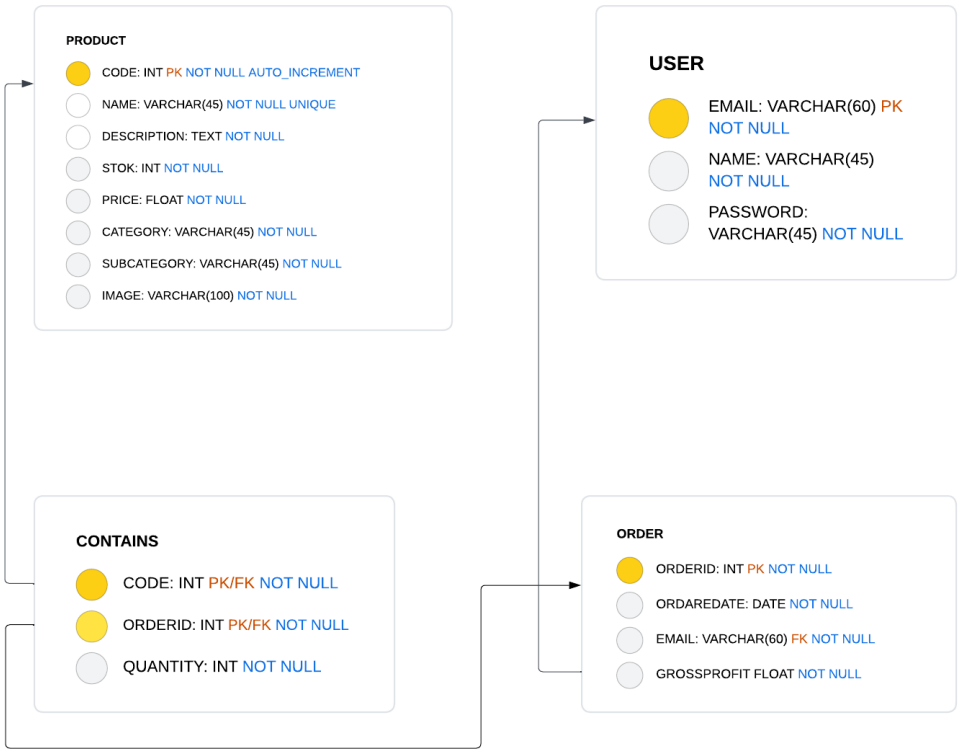


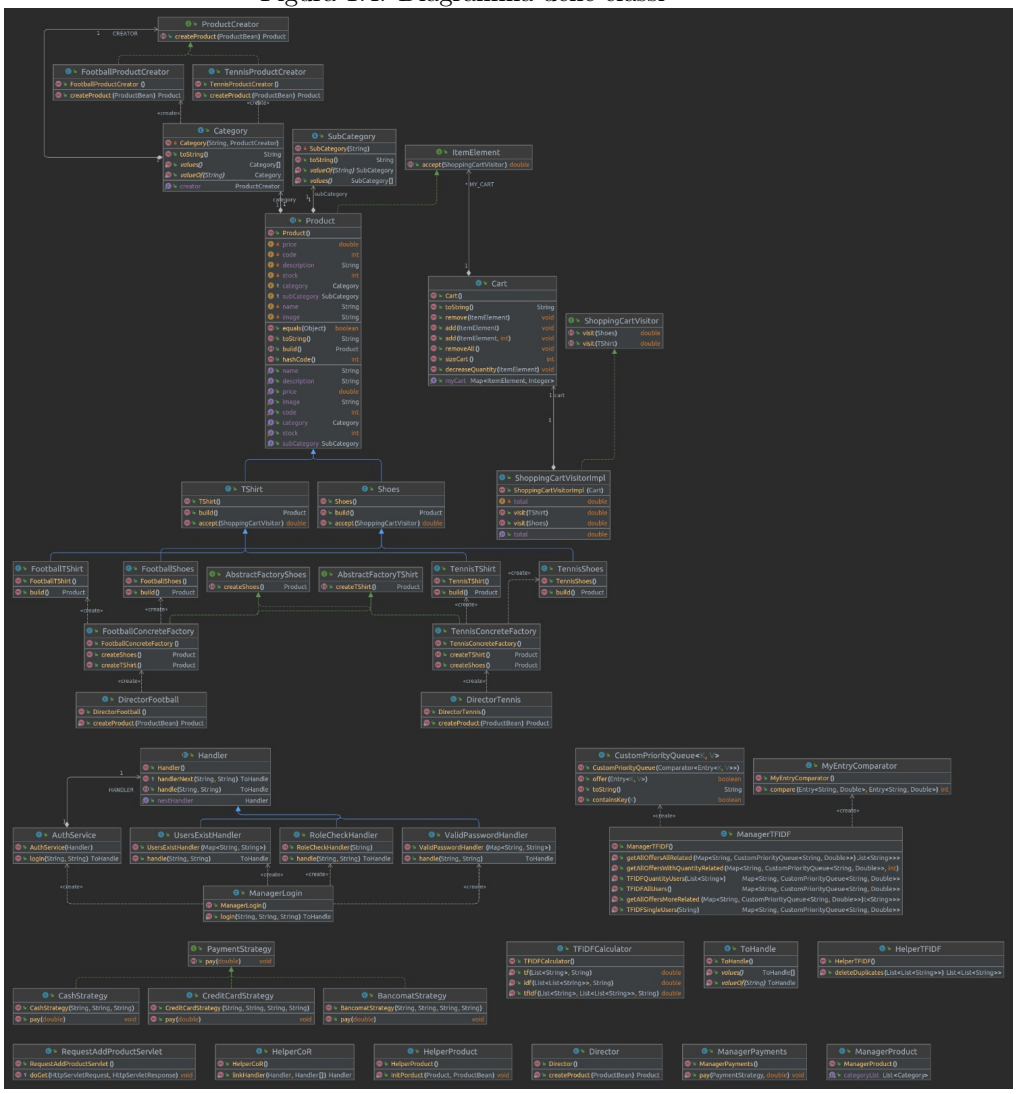
Figura 1.3: Diagramma relazionale del database



1.5 Diagramma delle classi

Dopo una breve introduzione del progetto di seguito è possibile vedere il diagramma delle classi, che verrà riproposto in vari pezzi nella prossima sezione in cui verranno spiegati i design pattern utilizzati nel progetto.

Figura 1.4: Diagramma delle classi



Capitolo 2

Design Pattern

Nella seguente sezione vengono descritti e presentati i design pattern presenti all'interno del progetto.

2.1 Abstract Factory

Abstract Factory è un pattern creazionale che permette di creare famiglie di prodotti senza specificare le classi concrete.

Il pattern Abstract Factory è stato utilizzato, per la gestione dei prodotti. Ciascun prodotto disponibile, tra i vari attributi, ha una categoria, per cui abstract factory si presta al meglio.

Nel seguente progetto è stato utilizzato con alcuni accorgimenti.

- L' Abstract Factory è stata divisa creando un'interfaccia apposita per ogni sotto categoria di prodotto, questa scelta è stata fatta per delle ragioni specifiche. In primis, per rispettare il principio della segregazione delle interfacce. Inoltre se in futuro ci dovesse essere la necessità di aggiungere una sotto categoria non bisogna modificare l'interfaccia Abstract Factory e tutte le classi che la implementano ma basta aggiungere un ulteriore interfaccia funzionale in modo da rispettare anche il principio open-closed. Infine le concrete factory vanno a implementare solo le interfacce necessarie, senza essere costrette ad implementare tutti i metodi.
- La classe astratta Product è unica poichè tutti i prodotti hanno esattamente gli stessi attributi.
- Viene utilizzata l'idea del builder per semplificare la creazione dei prodotti:
    1. All'interno della classe astratta Product è presente un metodo astratto build per la creazione di prodotti di una categoria e sotto categoria, che sarà implementato da tutte le classi che la estendono.
    2. Sono stati utilizzati dei direttori per la creazione di specifici prodotti di una definita categoria e sotto categoria.

3. Sono stati creati più direttori per ogni categoria scegliendo di avere del codice duplicato per rispettare il principio open-closed. Così facendo se c'è la necessità di aggiungere una categoria, bisognerà creare un nuovo direttore che si occupa singolarmente di essa.

- Inoltre è stata utilizzato un'interfaccia ProductCreator la quale viene implementata dalle varie categorie. Queste ultime vanno a richiamare il direttore dedicato, così che ogni classe che implementa ProductCreator avrà una sola responsabilità. Inoltre nel momento in cui si vuole andare ad aggiungere una categoria non bisogna modificare il direttore generale, rispettando dunque i principi open-closed e single responsibility. Questa architettura richiama il design pattern Strategy.

2.1.1 Spiegazione classi

Di seguito la spiegazione delle classi che formano il pattern:

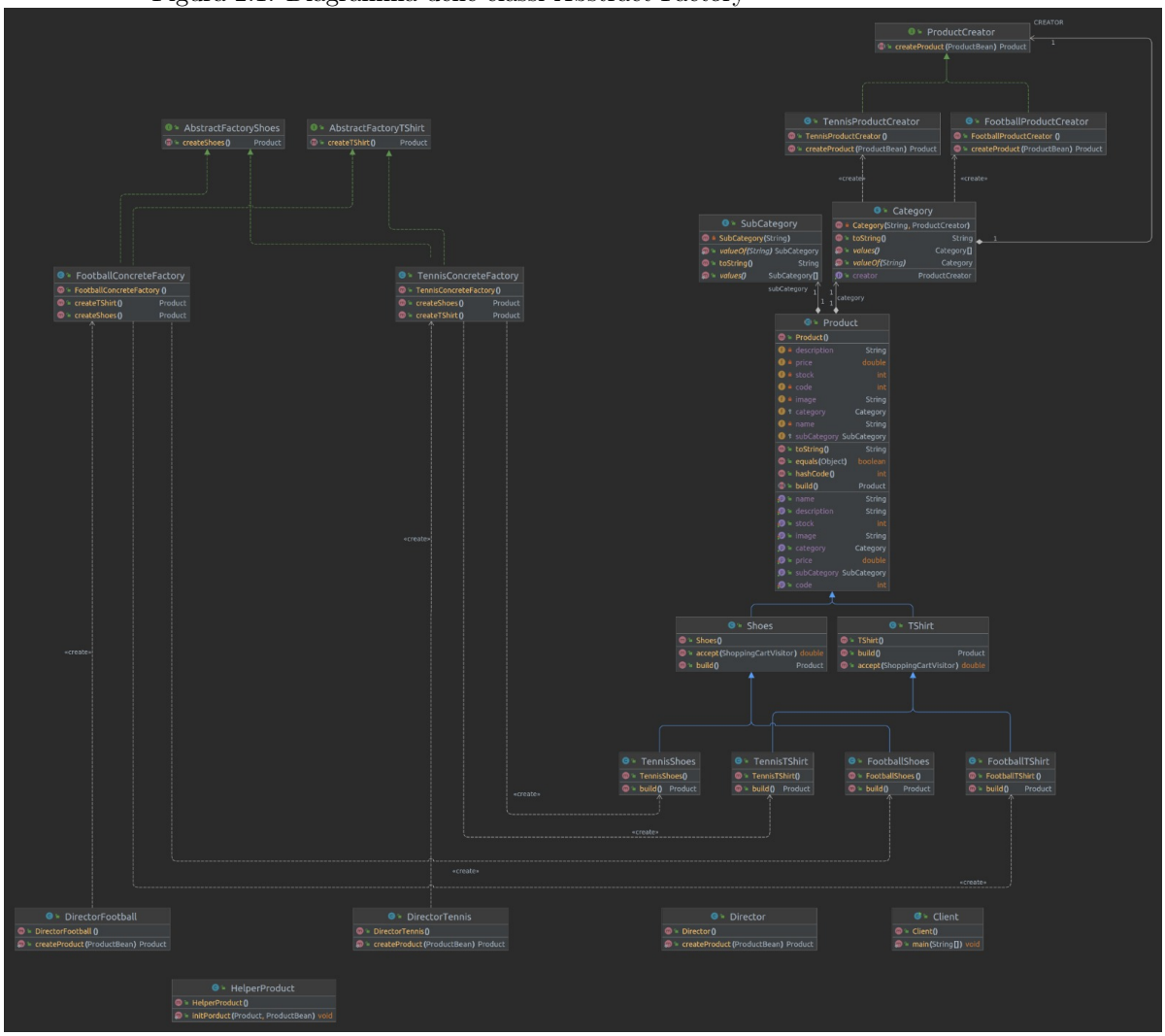
- *AbstractFactoryTShirt*: interfaccia che definisce un'abstract factory per la creazione di prodotti di tipo tShirt
- *AbstractFactoryShoes*: interfaccia che definisce un'abstract factory per la creazione di prodotti di tipo shoes
- *Category*: classe enum che rappresenta le categorie di prodotto e ad ognuna associa un nome e un product creator.
- *Director*: classe che crea e restituisce un prodotto tramite il creatore associato alla categoria.
- *DirectorFootball*: rappresenta il direttore per la creazione di prodotti di tipo Football.
- *DirectorTennis*: rappresenta il direttore per la creazione di prodotti di tipo Tennis.
- *FootballConcreteFactory*: implementazione delle interfacce *AbstractFactoryTShirt, AbstractFactoryShoes* per la creazione di prodotti di tipo*FootballShoes, FootballTShirt.*
- *FootballProductCreator*: implementazione dell'interfaccia *ProductCreator* per la creazione di prodotti di tipo Football.
- *FootballShoes*: rappresenta una classe concreta che estende la classe astratta *Shoes* e che rappresenta scarpe da calcio.
- *FootballTShirt*: rappresenta una classe concreta che estende la classe astratta *TShirt* e che rappresenta magliette da calcio.
- *HelperProduct*: inizializza il prodotto tramite builder impostando i valori del *ProductBean*

- *ManagerProduct*: ritorna la lista delle categorie in maniera dinamica(all'aumentare delle categorie aumenta anche la lista) se si aggiunge una categoria non si modifica il codice
- *Product*: classe astratta con gli attributi e metodi base di ogni prodotto
- *ProductCreator*: interfaccia che definisce un metodo per la creazione di un oggetto
- *Shoes*: prodotto di tipo scarpa, che estende product e tramite metodo build imposta la sotto categoria come shoes
- *SubCategory*: classe enum che rappresenta tutte le possibili sotto-categorie dei prodotti
- *TennisConcreteFactory*: implementazione delle interfacce *AbstractFactoryTShirt, AbstractFactoryShoes* per la creazione di prodotti di tipo *TennisShoes, TennisTShirt*.
- *TennisProductCreator*: implementazione dell'interfaccia *ProductCreator* per la creazione di prodotti di tipo Tennis.
- *TennisShoes*: rappresenta una classe concreta che estende la classe astratta *Shoes* e che rappresenta scarpe da tennis.
- *TennisTShirt*: rappresenta una classe concreta che estende la classe astratta *TShirt* e che rappresenta magliette da tennis.
- *TShirt*: prodotto di tipo maglietta, che estende product e tramite metodo build imposta la sotto categoria come tShirt.

2.1.2 Diagramma delle classi

Di seguito il diagramma delle classi

Figura 2.1: Diagramma delle classi Abstract Factory



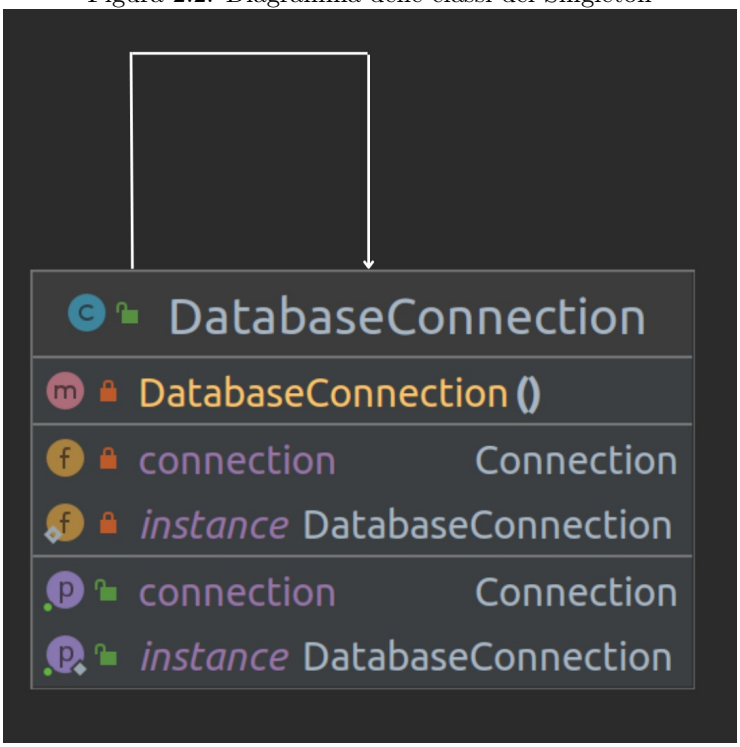
2.2 Singleton

Il singleton è pattern creazionale che serve ad assicurare che una classe abbia una sola istanza e fornire un punto globale di accesso ad essa.

Nel seguente progetto il singleton è stato utilizzato per gestire l'accesso al database.

Di seguito è possibile osservare il diagramma delle classi

Figura 2.2: Diagramma delle classi del Singleton



2.3 Strategy

Il pattern Strategy viene utilizzato per definire una famiglia di algoritmi, incapsularli e renderli intercambiabili.

Questo pattern viene utilizzato all'interno del progetto per la gestione dei metodi di pagamento poiché ci sono modi diversi di effettuare il pagamento e l'utente può scegliere quale usare a runtime.

2.3.1 Spiegazione classi

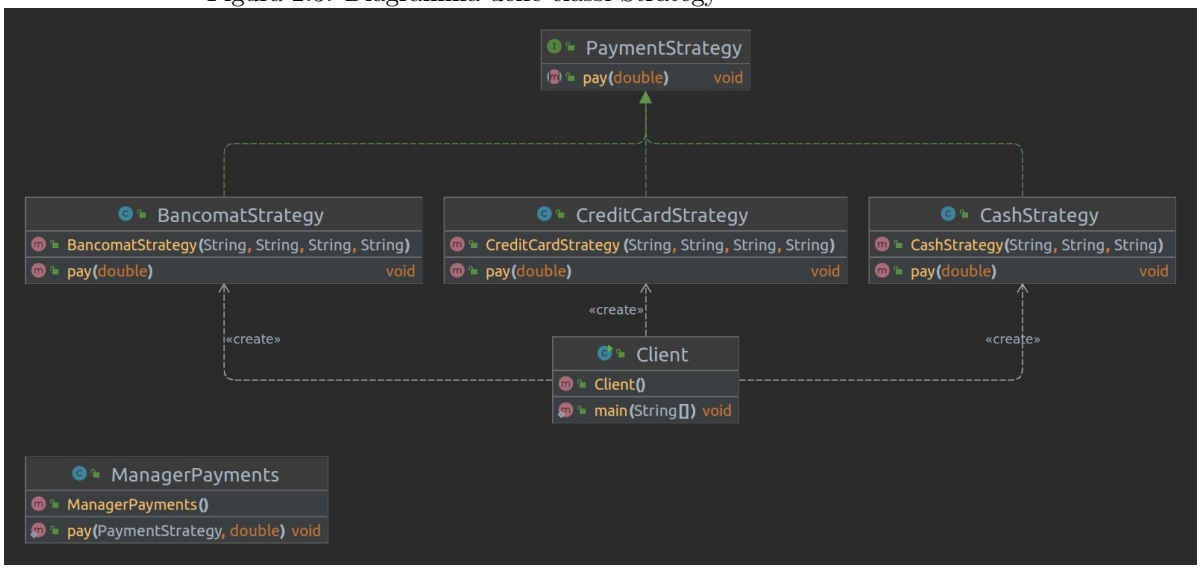
Di seguito la spiegazione delle classi che formano il pattern:

- *PaymentStrategy*: un' interfaccia che contiene il metodo pay e definisce il contratto per le strategie di pagamento.
- *ManagerPayments*: una classe nella quale è presente un metodo statico che permette di effettuare il pagamento tramite una delle possibili modalità.
- *CreditCardStrategy*: una classe che implementa la strategia di pagamento tramite carta di credito.
- *CashStrategy*: una classe che implementa la strategia di pagamento tramite contanti.
- *BancomatStrategy*: una classe che implementa la strategia di pagamento tramite bancomat.

2.3.2 Diagramma delle classi

Di seguito il Diagramma delle classi

Figura 2.3: Diagramma delle classi Strategy



2.4 Chain of Responsibility

Il pattern Chain of Responsibility viene utilizzato per separare il mittente di una richiesta dal destinatario, in modo da consentire anche a più handler diversi, che formano una catena, di gestire la richiesta. In questo modo la richiesta viene trasmessa fino all'handler in grado di gestirla.

Il modo in cui è stato applicato la Chain of Responsibility nel seguente progetto è per la gestione del login al sito. In questo caso, a differenza della definizione data sopra, la richiesta non viene trasmessa all'handler in grado di gestirla, bensì ogni handler serve ad effettuare una verifica diversa e, in caso nessuno di questi dia errore, il login avviene con successo.

2.4.1 Priorità delle responsabilità

Nella catena di handler sono presenti, in ordine di priorità: il controllo dell'esistenza della mail nel database, la correttezza della password inserita e infine il ruolo dell'utente che effettua il login.

2.4.2 Spiegazione classi

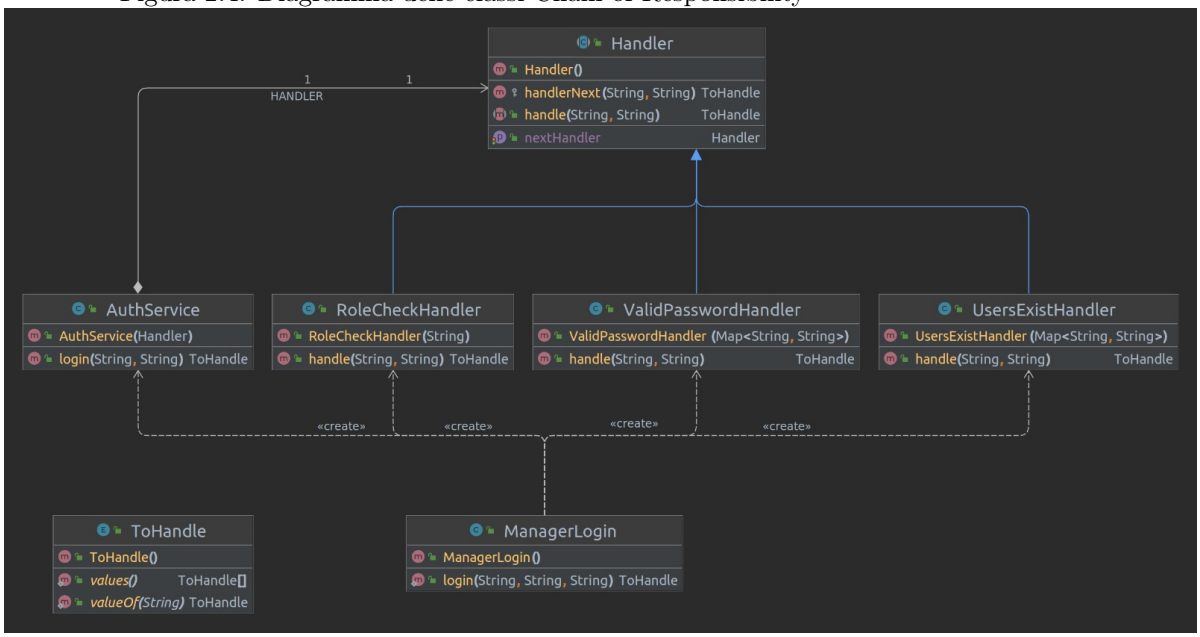
Di seguito la spiegazione delle classi che formano il pattern:

- *Handler*: classe astratta che rappresenta un gestore di accesso, ogni handler ha un riferimento all'handler successivo in modo da creare la catena di responsabilità.
- *AuthService*: classe che rappresenta il servizio di autenticazione degli utenti.
- *RoleCheckHandler*: handler per controllare il ruolo dell'utente che effettua il login.
- *UserExistHandler*: handler che verifica se l'account dell'utente esiste.
- *ValidPasswordHandler*: handler che verifica se la password inserita dall'utente è corretta.
- *ManagerLogin*: crea la chain di handler con le priorità prestabilite.
- *ToHandle*: classe enum per la rappresentazione delle possibili operazioni successive ad una richiesta di login.
- *HelperCoR*: classe che contiene metodi per il collegamento tra gli handler.

2.4.3 Diagramma delle classi

Di seguito il diagramma delle classi:

Figura 2.4: Diagramma delle classi Chain of Responsibility



2.5 Visitor

Il design pattern Visitor permette di separare l'algoritmo dagli oggetti su cui esso opera. In questo modo è possibile aggiungere operazioni e comportamenti senza modificare la struttura.

Nel seguente progetto il pattern visitor è stato utilizzato per la gestione del carrello, poiché al suo interno ci sono diverse categorie di prodotti e tramite il visitor la logica operativa è stata spostata in una classe differente. Dunque applicando questo pattern è stato possibile separare le operazioni che si possono effettuare sul carrello dal carrello stesso

2.5.1 Spiegazione classi

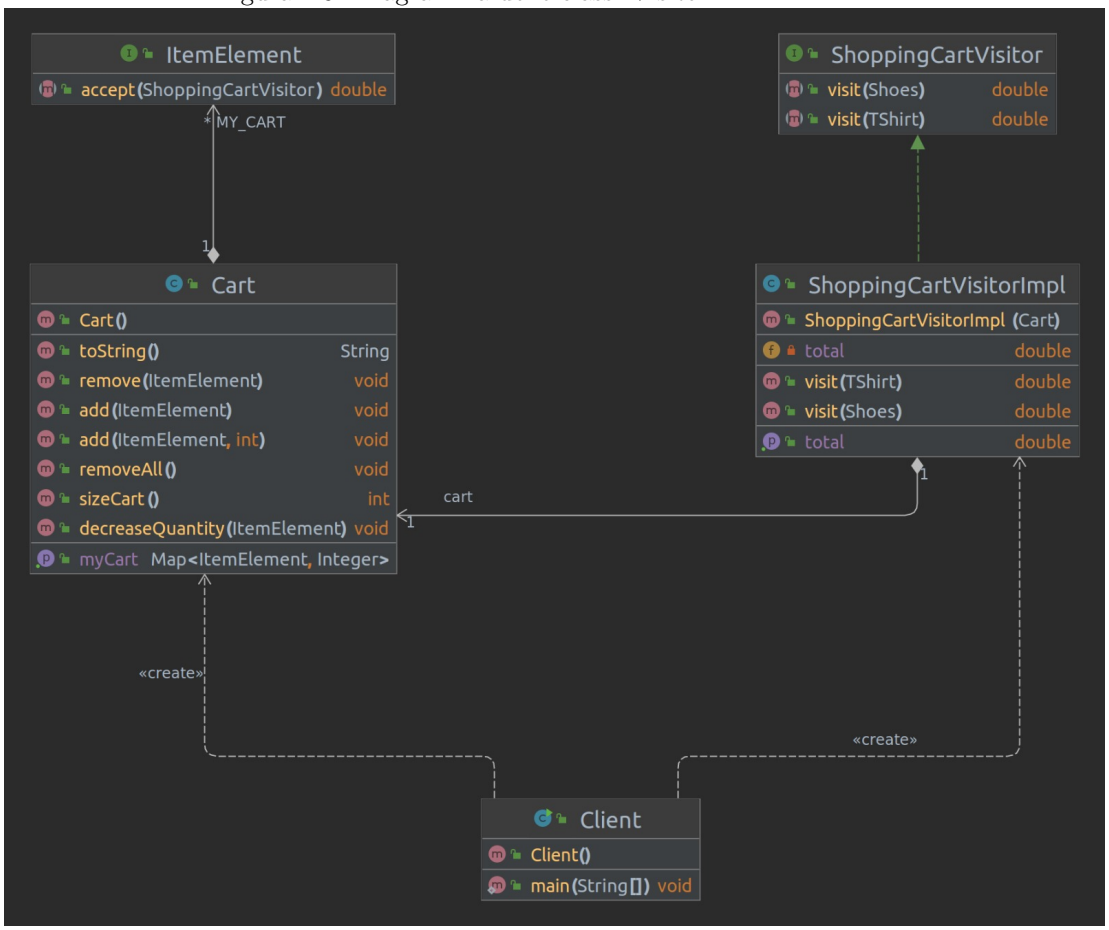
Di seguito la spiegazione delle classi che formano il pattern:

- *ShoppingCartVisitor*: definisce il contratto per i visitatori del carrello della spesa. Un visitatore può calcolare il prezzo totale dei prodotti nel carrello in base alle diverse implementazioni dei metodi visit.
- *ShoppingCartVisitorImpl*: classe che visita dei prodotti del carrello.
- *Cart*: classe che aggiunge e rimuove i prodotti dal carrello
- *ItemElement*: interfaccia che rappresenta l'elemento all'interno del carrello che fornisce il valore dell'elemento visitato ed espone un singolo metodo accept.

2.5.2 Diagramma delle classi

Di seguito il diagramma delle classi:

Figura 2.5: Diagramma delle classi Visitor



2.6 Model View Controller

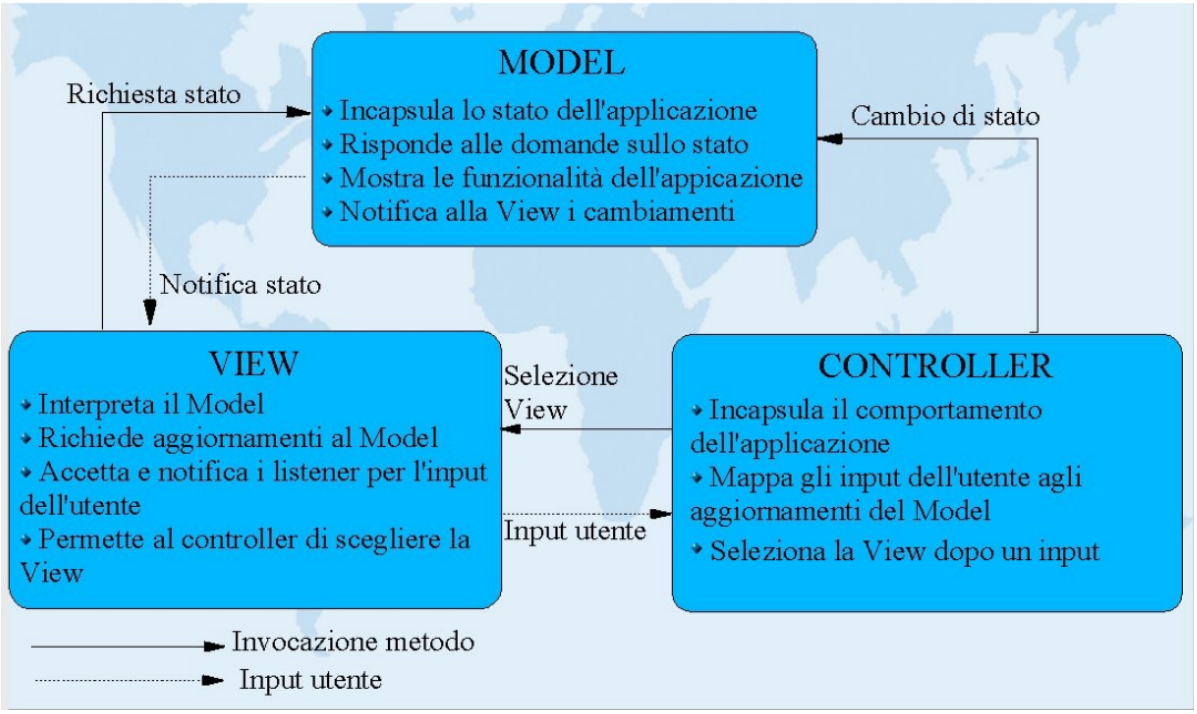
Il pattern Model View Controller (o MVC) viene utilizzato per separare i componenti software. È un pattern di sviluppo che serve per organizzare i file all'interno del progetto in 3 sezioni differenti.

- *Model*: contiene tutte le logiche di business.Comprende tutti gli oggetti con cui possono interagire sia gli utenti che l'admin e vengono implementati tramite Java Bean
- *View*: contiene tutta la logica di presentazione e l'interfaccia e viene implementata tramite JSP e ricavata tramite servlet
- *Controller*: contiene tutti i controlli delle funzionalità facendo da mediatore tra il model e la view, prendendo ad esempio le informazioni dalla view e passandolo al model tramite servlet o, viceversa passando delle informazioni presenti nel model all'interfaccia.

2.6.1 Diagramma delle classi

Di seguito il diagramma delle classi:

Figura 2.6: Diagramma delle classi Chain of Responsibility



Capitolo 3

TF-IDF

3.1 Funzionamento

Il **TF-IDF**(*term frequency-inverse document frequency*) è una tecnica di intelligenza artificiale utilizzato per determinare quanto sia importante una parola all'interno di un documento, generalmente dando uno score ad ogni parola.

3.1.1 Term frequency

La term frequency, come suggerisce il nome, si occupa di calcolare un termine quante occorrenze ha all'interno del documento. Vengono divise le occorrenze di ogni parola per la quantità totale di parole presenti all'interno del documento, in modo da normalizzare il peso che ha un termine anche in base a quante parole totali ci sono all'interno del documento.

$$TF = \frac{numero\_di\_occorrenze\_del\_termine}{numero\_totale\_di\_parole}$$

3.1.2 Inverse document frequency

Calcola l'inverso della frequenza di un termine nell'intero set di documenti. Il motivo per cui si usa l'inverso è che una parola molto comune, come ad esempio un articolo, è meno importante di una parola che invece compare molto poco che di conseguenza avrà uno score maggiore.

$$IDF = \log(\frac{numero\_di\_documenti}{numero\_di\_documenti\_in\_cui\_compare\_il\_termine})$$

Una volta trovati questi valori per calcolare la TF-IDF viene eseguita la moltiplicazione:

$$TFIDF = TF \cdot IDF$$

3.2 Applicazione di TF-IDF

Il concetto alla base è quello di trovare l'utente più simile a quello a cui bisogna inviare l'offerta, osservando il suo comportamento, caratterizzato dagli ordini effettuati, e creare un'offerta personalizzata sulla base di questo.

Dal punto di vista implementativo ciò viene realizzato andando a considerare come singleDocument la lista dei prodotti ordinati del cliente a cui sottoporre l'offerta (che per convenzione chiameremo **ricevente**), dove ogni prodotto viene utilizzato come **term**.
A questo punto bisogna trovare un secondo utente con acquisti simili al nostro cliente ricevente.
Calcolando la *term frequency* per ogni prodotto nel term set otteniamo l'importanza del prodotto per il cliente esaminato.
Successivamente andiamo a calcolare l'*inverse document frequency* sulla lista degli ordini degli altri utenti, presi uno alla volta, rispetto ai term del term set del cliente ricevente.
Il prodotto tra le TF e le IDF ci darà come risultato un indicatore TF-IDF per ogni term del ricevente.
La somma di questi valori fornirà uno score di somiglianza del cliente ricevente verso tutti gli altri clienti.
Selezionando l'utente con lo score più alto otterremo il cliente più simile al nostro ricevente e effettuando la differenza insiemistica tra la lista ordini dell'utente più simile e quella del ricevente otterremo la lista di prodotti candidati ad essere offerti.

A livello implementativo ciò viene fatto attraverso una coda di massima priorità custom la quale è composta da una map con chiave string e valore double. Quindi per ogni utente avremo associato una coda di massima priorità la quale conterrà una mappa composta da una chiave email di ogni singolo utente ed un valore score rappresentato da un double.

Infine la testa della coda di massima priorità rappresenterà l'utente più simile all'utente al quale vogliamo fare l'offerta, sul quale fare la differenza insiemistica sopra citata per trovare gli articoli da offrire.

3.2.1 Diagramma delle classi

Di seguito il diagramma delle classi della TF-IDF.

Figura 3.1: Diagramma delle classi TF-IDF