Università degli Studi di Salerno
Corso di Ingegneria del Software

SpeedScale
Object Design Document
Versione 1.0

Data: 08/01/2025

Coordinatore del progetto:

Nome	Matricola

Partecipanti:

Nome	Matricola
Sepe Gennaro	0512116971
La Marca Antonio	0512117826

Scritto da:	Sepe Gennaro

Revision History

Data	Versione	Descrizione	Autore
26/11/2024	0.1	Creazione del documento	La Marca Antonio
14/12/2024	0.2	Scrittura dei contenuti	Sepe Gennaro
16/12/2024	0.3	Prime correzioni post-consegna	Sepe Gennaro
08/01/2025	1.0	Aggiustamenti + Rilascio finale	Sepe Gennaro

Indice

1. Introduzione

Il design orientato agli oggetti è un approccio fondamentale per lo sviluppo di sistemi software moderni, poiché permette di tradurre esigenze complesse in soluzioni strutturate e scalabili. In questo documento, viene illustrato il processo di progettazione adottato, con particolare attenzione all'applicazione dei principi di orientamento agli oggetti, modularità e riusabilità. Il sistema è stato progettato attorno a oggetti che combinano dati e comportamenti, riflettendo fedelmente la logica del dominio e garantendo una forte connessione tra modello concettuale e implementazione tecnica. Inoltre, è stata posta grande enfasi sulla modularità, assicurando che ogni componente del sistema fosse autosufficiente e ben separato dagli altri, per semplificare l'individuazione e la risoluzione dei problemi e consentire interventi mirati senza influire negativamente sull'intero sistema. Un altro aspetto centrale del design è stato il focus sulla riusabilità, progettando classi e moduli flessibili. Questo approccio non solo migliora l'efficienza dello sviluppo, riducendo tempi e costi, ma contribuisce anche alla qualità e alla coerenza complessiva del software. L'integrazione di questi principi permette di sviluppare un sistema robusto, scalabile e facilmente mantenibile, capace di rispondere alle esigenze attuali senza compromettere la capacità di evoluzione futura.

1.1 Compromessi nella progettazione degli oggetti

La progettazione orientata agli oggetti offre potenti strumenti per creare sistemi modulari, riutilizzabili e scalabili. Tuttavia, come in qualsiasi approccio, è necessario affrontare una serie di compromessi. Nel bilanciare flessibilità, efficienza e manutenibilità, sono state prese delle decisioni che influenzano le prestazioni, la semplicità del design e la capacità del sistema di adattarsi a futuri cambiamenti. Di seguito vengono analizzate ulteriormente i compromessi decisi per il sistema, analizzando come scelte specifiche hanno inciso sul progetto.

1.1.1 Affidabilità vs Costo-efficacia

Nel contesto della progettazione di SpeedScale, uno dei requisiti fondamentali è stato garantire un'elevata affidabilità del sistema, essenziale per soddisfare le esigenze di performance e stabilità richieste dagli utenti finali. Tuttavia, raggiungere un alto livello di affidabilità senza trascurare la costo-efficacia rappresenta una sfida cruciale, che richiede scelte progettuali strategiche in ogni fase dello sviluppo. Per affrontare questo equilibrio, è stato adottato un approccio basato sul concetto di containerizzazione, in quanto l’intero applicativo viene gestito dalla piattaforma Java. Questo isolamento contribuisce direttamente all'affidabilità, riducendo i rischi di malfunzionamenti dovuti a configurazioni errate o incompatibilità. In aggiunta, questa scelta progettuale ben si presta come soluzione scalabile e immediata per un futuro deployment anche su infrastrutture cloud. Questa combinazione tra robustezza e economicità operativa ha permesso a SpeedScale di rispettare i rigorosi requisiti di affidabilità senza compromettere l'efficienza economica.

1.1.2 Scalabilità vs Sviluppo rapido

Uno dei vari obiettivi nella progettazione di SpeedScale è stato quello di garantire una scalabilità elevata, senza sacrificare la velocità di sviluppo del sistema. Questo equilibrio tra due esigenze apparentemente in competizione è stato raggiunto attraverso l'adozione di Java EE come piattaforma di sviluppo. Java EE offre un ecosistema consolidato e altamente performante che integra nativamente il supporto per ambienti containerizzati. Questa caratteristica consente di delegare una parte significativa delle complessità di gestione dell'applicazione al container Java EE, che si occupa di aspetti fondamentali come il ciclo di vita delle componenti, la gestione delle transazioni, il bilanciamento del carico e la persistenza. Tale delega permette agli sviluppatori di concentrarsi sulla logica applicativa, riducendo drasticamente i tempi di sviluppo e la possibilità di errori. Un altro elemento chiave che contribuisce alla velocità di produzione è stato l'uso delle annotazioni fornite da Java EE. Grazie a quest'ultime, è possibile implementare in modo rapido e semplice controlli di sicurezza e configurazioni avanzate, senza la necessità di scrivere codice aggiuntivo o gestire manualmente dettagli complessi. Questo approccio ha dimostrato di essere particolarmente efficace in un contesto in cui la scalabilità del sistema è cruciale. In sintesi, la scelta di Java EE permette di ottenere una soluzione scalabile senza rallentare il processo di sviluppo, offrendo strumenti che riducono la complessità per gli sviluppatori e garantiscono al contempo la robustezza e la sicurezza necessarie per un sistema moderno e performante.

1.1.3 Usabilità vs Prestazioni

Nella progettazione del sistema SpeedScale, è necessario bilanciare correttamente la usabilità dell'interfaccia utente con le esigenze di prestazioni elevate. Nel caso in cui alcune interfacce risultino troppo pesanti o compromettano la velocità del sistema, sono previste operazioni di ottimizzazione. Tali interventi includono la semplificazione delle interfacce, rimuovendo effetti visivi e transizioni non essenziali, in modo da concentrare le risorse sulla performance pura. Per quanto riguarda il back-end, le prestazioni sono ulteriormente rafforzate da query SQL ottimizzate e dall'utilizzo di un database relazionale che garantisce un accesso rapido ed efficiente ai dati. Questo approccio consente di mantenere tempi di risposta competitivi anche in presenza di carichi elevati, senza sacrificare la qualità dell'esperienza utente. Questo compromesso dinamico permette di adattarsi alle esigenze del progetto, garantendo un'esperienza soddisfacente per gli utenti senza compromettere la velocità e la scalabilità del sistema.

1.2 Linee guida per la documentazione dell'interfaccia

Questa sezione descrive le linee guida fondamentali da seguire durante lo sviluppo del sistema.

1.2.1 Convenzioni di nomenclatura

Per garantire coerenza e leggibilità nel codice, si adottano le convenzioni di nomenclatura standard Java. Questo approccio promuove uno stile uniforme che facilita la collaborazione tra sviluppatori, semplifica la manutenzione e migliora la comprensibilità del sistema. Le principali regole includono:

- Classi: i nomi delle classi seguono lo stile PascalCase, iniziando con una lettera maiuscola (es. UserManager, OrderService).
- Variabili e metodi: i nomi utilizzano il formato camelCase, iniziando con una lettera minuscola (es. userList, calculateTotal).
- Costanti: per le costanti, si utilizza il formato UPPER_SNAKE_CASE, separando le parole con un underscore (es. MAX_RETRIES, DEFAULT_TIMEOUT).
- Pacchetti: i nomi dei pacchetti sono interamente in minuscolo e riflettono la struttura gerarchica del progetto, tipicamente iniziando con il dominio aziendale (es. it.speedscale.example).

Queste convenzioni non solo assicurano uniformità all'interno del codice, ma favoriscono anche l'integrazione con strumenti e librerie Java che si basano su tali standard.

1.2.2 Convenzioni di nominazione delle classi

Per rendere il codice immediatamente comprensibile e agevolare la manutenibilità, la scelta dei nomi delle classi devono riflettere in modo diretto il loro ruolo e la loro responsabilità nel sistema, consentendo a uno sviluppatore di comprenderne il funzionamento già dalla prima lettura. Le linee guida principali per la nomenclatura delle classi includono:

- Descrizione del ruolo principale: il nome della classe deve esprimere chiaramente la funzione svolta. Ad esempio, una classe che gestisce operazioni sugli utenti sarà chiamata UserManager, mentre una classe che rappresenta un ordine sarà Order.
- Evitare abbreviazioni: usare nomi completi e significativi, evitando abbreviazioni criptiche o non standard (es. preferire CustomerService a CustSvc).
- Specificità e ambito: quando una classe è specializzata o legata a un contesto specifico, il nome deve rifletterlo. Ad esempio, una classe che gestisce l'autenticazione sarà chiamata AuthenticationHandler, distinguendola da altre classi relative agli utenti.
- Uso di suffissi e prefissi standard: per indicare il tipo o la funzione della classe, si possono utilizzare suffissi comuni come Manager, Service, Factory, o prefissi come Abstract per classi astratte (es. AbstractParser).

Queste convenzioni assicurano che ogni classe nel sistema non solo sia coerente con lo stile globale, ma fornisca anche un'indicazione chiara e immediata del suo scopo e del contesto di utilizzo.

1.2.3 Stile delle parentesi e spaziatura

Per garantire coerenza e leggibilità del codice, si adottano le convenzioni relative allo stile delle parentesi e spaziatura stabilite nello standard Java. Seguendo queste linee guida, il codice risulta uniforme, facilmente comprensibile e conforme alle best practice Java, favorendo una manutenzione più semplice e riducendo il rischio di errori. Le principali convenzioni includono:

- **Posizione delle parentesi:**
  - Le parentesi di apertura per i blocchi di codice (come metodi, classi, cicli, condizioni) devono essere posizionate sulla stessa riga della dichiarazione, senza andare a capo (stile K&R);
  - La parentesi di chiusura deve essere allineata con la riga della dichiarazione del blocco;
- **Spaziatura**
  - Uno spazio deve essere utilizzato dopo le parole chiave come if, for, while, return e altre dichiarazioni simili, per migliorare la leggibilità del codice;
  - Non devono essere usati spazi extra all'interno delle parentesi;
  - Uno spazio deve essere presente dopo ogni virgola in una lista di parametri o variabili.

1.2.4 Struttura del progetto

Il progetto è suddiviso in due principali moduli: un **modulo EJB** e un **modulo Web Application**. Questa struttura è stata progettata per favorire una gestione chiara e logica delle classi e delle pagine web, organizzandole in due principali pacchetti: **admin** e **common**. Tale organizzazione consente di mantenere un sistema ben strutturato e facilmente navigabile, separando le funzionalità in base alla loro rilevanza e accessibilità. Nel pacchetto admin vengono collocate tutte le classi e le componenti considerate "importanti", che richiedono specifici ruoli o permessi per poter essere utilizzate. Questo pacchetto include funzionalità critiche, come la gestione degli utenti, la configurazione avanzata e l'accesso a dati sensibili. Queste operazioni necessitano di un controllo rigoroso sui privilegi e sulle autorizzazioni per garantire la sicurezza del sistema. Nel pacchetto common, invece, si trovano tutte le classi e i moduli sempre accessibili e visualizzabili, indipendentemente dai ruoli degli utenti. Queste componenti comprendono funzionalità generali, come la gestione delle interfacce utente di base, le operazioni comuni e le classi di utilità. Lo stesso principio di organizzazione è applicato alla suddivisione delle pagine web. Le pagine appartenenti alla sezione admin sono progettate per essere accessibili solo agli utenti con privilegi amministrativi, mentre le pagine della sezione common possono essere visualizzate da tutti gli utenti. Questo garantisce una chiara separazione tra le aree riservate del sistema e quelle pubbliche. La suddivisione in un modulo EJB e un modulo Web Application, combinata con l'organizzazione in pacchetti admin e common, aiuta a mantenere il progetto ben organizzato, semplifica la gestione dei permessi e ottimizza l'uso delle risorse. Questo approccio migliora la sicurezza, la manutenibilità e la scalabilità del sistema nel lungo periodo.

1.2.5 Gestione degli errori

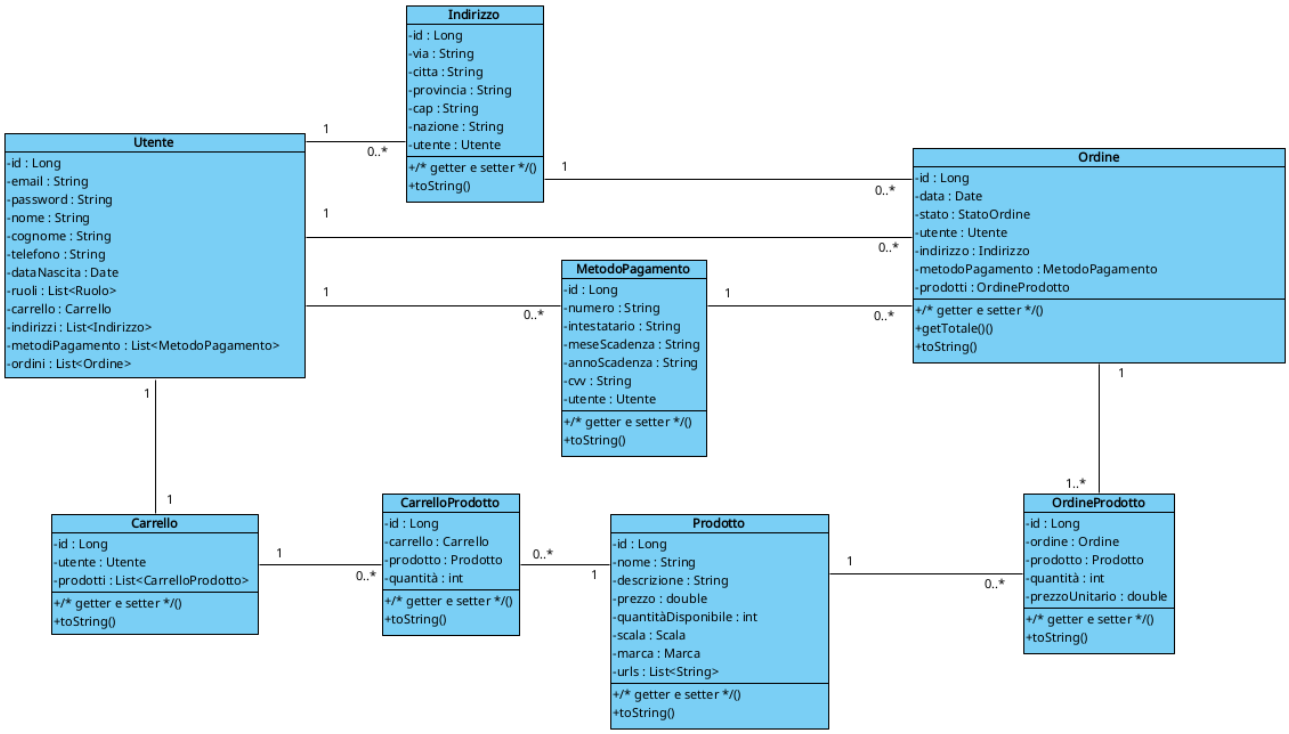
La gestione degli errori si basa su un approccio strutturato che utilizza le eccezioni per affrontare gli errori più significativi, mentre per quelli di minore rilevanza si farà affidamento su controlli sui tipi di ritorno. Gli errori critici o eccezioni che interrompono il flusso normale dell'applicazione saranno gestiti tramite le eccezioni Java. Questo approccio è ideale per trattare situazioni anomale o errori imprevisti che richiedono una gestione esplicita e dettagliata, come problemi di accesso ai dati, errori di connessione al database o violazioni di sicurezza. Le eccezioni saranno catturate utilizzando i meccanismi di try-catch, permettendo una gestione centralizzata e la registrazione degli errori, nonché l'adozione di strategie di recupero appropriate. Per gli errori secondari o le anomalie meno gravi, che non richiedono un intervento immediato o che possono essere facilmente risolte, la gestione avverrà attraverso il controllo dei valori di ritorno dei metodi. In questi casi, l'analisi del tipo di ritorno (ad esempio, un valore null o un codice di errore specifico) permetterà di gestire la situazione senza ricorrere a meccanismi complessi. Per esempio, se un metodo restituisce un valore null o un oggetto con uno stato di errore, il flusso di esecuzione potrà essere indirizzato in modo appropriato per evitare il fallimento del sistema. Questa separazione tra gestione delle eccezioni gravi e controllo dei ritorni per gli errori minori contribuisce a mantenere il codice semplice, leggibile e manutenibile, mentre allo stesso tempo garantisce una risposta tempestiva e adeguata a entrambi i tipi di errore.

1.3 Aggiornamento del modello a oggetti

Il modello a oggetti del sistema è stato aggiornato per migliorare la flessibilità, la manutenibilità e la scalabilità del codice, seguendo principi consolidati come **KISS** (Keep It Simple, Stupid) e **Composition over Inheritance**. Le modifiche apportate si concentrano su tre aspetti principali:

- **Semplificazione delle specializzazioni della classe Utente tramite un attributo ruolo:** Le precedenti sottoclassi di Utente, utilizzate per rappresentare diverse tipologie di utenti sono state sostituite da un unico attributo ruolo. Questo approccio consente di identificare il tipo di utente attraverso valori predefiniti come "CLIENTE" o "AMMINISTRATORE", eliminando la necessità di una gerarchia complessa di classi. Tale scelta riduce la proliferazione di classi nel modello, semplificando sia la struttura che la logica di gestione. In caso di nuove tipologie di utenti, sarà sufficiente aggiungere un nuovo valore per il ruolo senza modificare la struttura del modello.
- **Trasformazione delle relazioni in attributi nelle classi:** Le relazioni tra classi, che in precedenza erano rappresentate attraverso associazioni dirette, sono state trasformate in attributi strutturati. Ad esempio, la relazione tra la classe Utente e la classe Ordine è ora gestita tramite una lista di ordini all'interno della classe Utente. Questo approccio migliora l'accesso e la gestione degli oggetti associati, semplificando le operazioni comuni come l'aggiunta o la rimozione di elementi. Inoltre, consente una maggiore coesione all'interno delle singole classi, riducendo la complessità complessiva del modello.

Questi aggiornamenti non solo semplificano il modello a oggetti ma sono anche in linea con i principi di progettazione moderna, garantendo una maggiore modularità e flessibilità. L'eliminazione di strutture complesse come le gerarchie di classi e l'adozione di approcci più diretti e intuitivi contribuiscono a rendere il sistema più scalabile e adattabile alle esigenze future.


Indirizzo
-id : Long
-via : String
-citta : String
-provincia : String
-cap : String
-nazione : String
-utente : Utente
+/* getter e setter */()
+toString()
Utente
-id : Long
-email : String
-password : String
-nome : String
-cognome : String
-telefono : String
-dataNascita : Date
-ruoli : List<Ruolo>
-carrello : Carrello
-indirizzi : List<Indirizzo>
-metodiPagamento : List<MetodoPagamento>
-ordini : List<Ordine>
Ordine
-id : Long
-data : Date
-stato : StatoOrdine
-utente : Utente
-indirizzo : Indirizzo
-metodoPagamento : MetodoPagamento
-prodotti : OrdineProdotto
+/* getter e setter */()
+getTotale()()
+toString()
MetodoPagamento
-id : Long
-numero : String
-intestatario : String
-meseScadenza : String
-annoScadenza : String
-cvv : String
-utente : Utente
+/* getter e setter */()
+toString()
Carrello
-id : Long
-utente : Utente
-prodotti : List<CarrelloProdotto>
+/* getter e setter */()
+toString()
CarrelloProdotto
-id : Long
-carrello : Carrello
-prodotto : Prodotto
-quantità : int
+/* getter e setter */()
+toString()
Prodotto
-id : Long
-nome : String
-descrizione : String
-prezzo : double
-quantitàDisponibile : int
-scala : Scala
-marca : Marca
-urls : List<String>
+/* getter e setter */()
+toString()
OrdineProdotto
-id : Long
-ordine : Ordine
-prodotto : Prodotto
-quantità : int
-prezzoUnitario : double
+/* getter e setter */()
+toString()
1
0..*
1
0..*
1
0..*
1
0..*
1
0..*
1
0..*
1
0..*
1
0..*
1
1..*
1
0..*

1.4 Definizioni, acronimi e abbreviazioni

TERMINE	***DEFINIZIONE***
FR	Requisito funzionale
NFR	Requisito non funzionale
S	Scenario
UC	Caso d'uso
SD	Diagramma di sequenza
SCD	Diagramma di stati
MK	Mockup
TBD	Da definire

1.5 Riferimenti

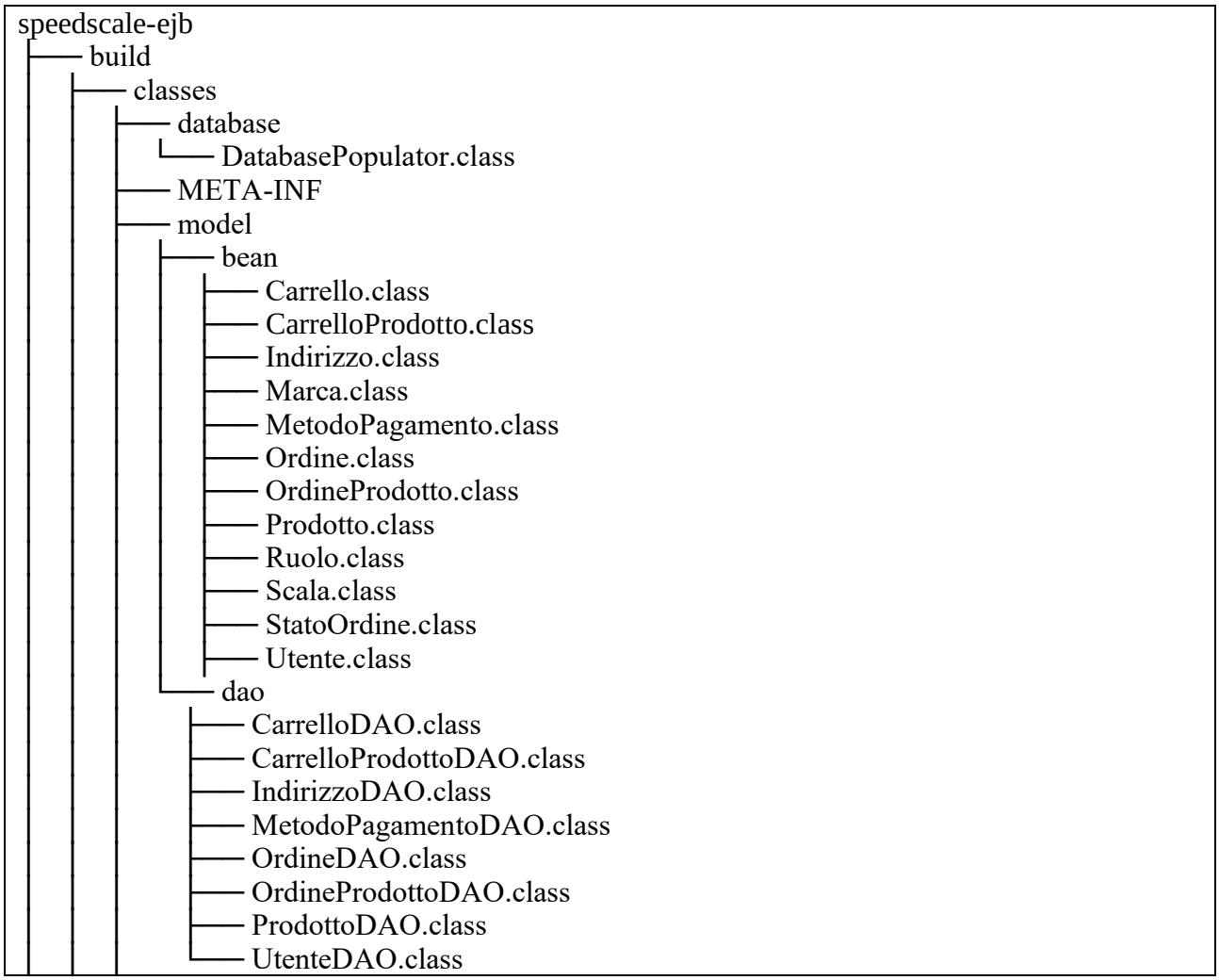
Sono stati utilizzati come base teorica il manuale "Object Oriented Software Engineering Using UML, Patterns, and Java™" di Bernd Bruegge e Allen H. Dutoit, il quale ha offerto le linee guida per la scrittura dei documenti e l'uso di UML. Oltre ai documenti realizzati precedentemente, il progetto è stato continuamente arricchito dagli insegnamenti del corso universitario "Ingegneria del Software" del professore De Lucia Andrea, che ha trattato le competenze per una gestione strutturata del ciclo di vita dello sviluppo software, applicate per garantire qualità e robustezza della piattaforma.

2. Pacchetti

Nel contesto del sistema SpeedScale, la strutturazione del progetto in pacchetti assume un ruolo cruciale per garantire una gestione efficiente del codice e una chiara separazione delle responsabilità. Il pacchetto principale di origine è denominato **it.unisa.speedscale**, e da esso derivano tutte le componenti del sistema, organizzate in due macroaree principali: **admin** e **common**.

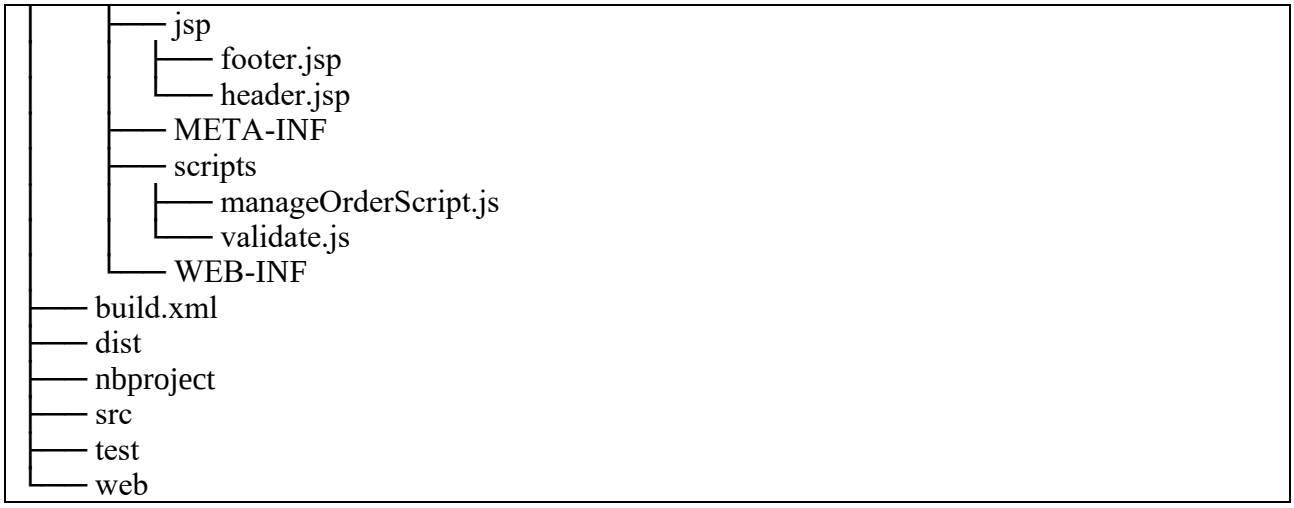
- Il pacchetto **admin** ospita tutte le componenti che gestiscono attività critiche e sensibili, come la gestione del catalogo prodotti e degli ordini. Queste funzionalità richiedono un accesso ristretto e un controllo rigoroso dei permessi, garantendo la sicurezza delle operazioni amministrative.
- Il pacchetto **common**, invece, include tutte le funzionalità accessibili a qualsiasi utente, come la gestione del profilo personale, il carrello e la visualizzazione del catalogo pubblico. Queste funzionalità sono progettate per essere intuitive e utilizzabili da tutti gli utenti senza restrizioni particolari.

Questa organizzazione modulare favorisce la scalabilità e la manutenibilità del progetto, consentendo di evolvere il sistema in modo ordinato. La chiara separazione delle responsabilità tra i pacchetti riduce la complessità del codice e ne facilita la comprensione, rendendo il sistema più robusto e adattabile alle esigenze future.



```
        ├── resouces.res
        └── service
            ├── CarrelloService.class
            ├── Catalogo.class
            ├── GestioneProfilo.class
            └── RegistroUtenti.class
    ├── empty
    └── generated-sources
├── build.xml
├── src
└── test
```

```
speedscale-war
├── build
│   ├── empty
│   ├── generated-sources
│   └── web
│       ├── admin
│       │   ├── addProduct.jsp
│       │   ├── homepage.jsp
│       │   ├── manageOrders.jsp
│       │   ├── manageProducts.jsp
│       │   └── updateProduct.jsp
│       ├── common
│       │   ├── addAddress.jsp
│       │   ├── addMetodoPagamento.jsp
│       │   ├── cart.jsp
│       │   ├── catalog.jsp
│       │   ├── homepage.jsp
│       │   ├── info.jsp
│       │   ├── login.jsp
│       │   ├── manageAddresses.jsp
│       │   ├── manageMetodiPagamento.jsp
│       │   ├── manageOrders.jsp
│       │   ├── productDetails.jsp
│       │   ├── registration.jsp
│       │   ├── selectAddress.jsp
│       │   ├── selectMetodoPagamento.jsp
│       │   ├── updateAccountDetails.jsp
│       │   ├── updateAddress.jsp
│       │   └── updateMetodoPagamento.jsp
│       ├── css
│       │   ├── style.css
│       │   └── styleError.css
│       ├── errors
│       │   └── 404.jsp
│       ├── images
│       │   ├── banner.png
│       │   └── logo.png
│       └── index.jsp
```


jsp
footer.jsp
header.jsp
META-INF
scripts
manageOrderScript.js
validate.js
WEB-INF
build.xml
dist
nbproject
src
test
web

3. Interfacce di classe

In questo capitolo vengono descritte le principali classi del sistema, in particolare le operazioni esposte. Le interfacce di classe forniscono un quadro chiaro di come le classi interagiscono tra loro all'interno del sistema e come gli sviluppatori possono utilizzarle.

Interfaccia	**RegistroUtenti**
Descrizione	Permette di verificare le credenziali dell'utente, registrare nuovi account, e inizializzare le sessioni per gli utenti autenticati.
Metodi	+ authenticate(String email, String pwd) : Utente + createUtente(String email, String pwd, String nome, String cognome, Date dataNascita) : Utente + inizializzaSessione(Utente utente, HttpSession session) : void
Invariante di classe	context RegistroUtenti inv: self.utenti->forAll(u | u.email.isNotEmpty() and u.pwd.isNotEmpty())

Metodo	**+ authenticate(String email, String pwd) : Utente**
Descrizione	Verifica le credenziali dell'utente per l'accesso.
Pre-condizione	email.isNotEmpty() and pwd.isNotEmpty()
Post-condizione	result = self.utenti->select(u | u.email = email and u.pwd = pwd)->any(true)

Metodo	**+ createUtente(String email, String pwd, String nome, String cognome, Date dataNascita) : Utente**
Descrizione	Crea un nuovo utente nel sistema con le informazioni fornite.
Pre-condizione	self.utenti->forAll(u | u.email <> email) and email.isNotEmpty() and pwd.isNotEmpty() and nome.isNotEmpty() and cognome.isNotEmpty() and dataNascita <> null
Post-condizione	self.utenti->exists(u | u.email = email and u.pwd = pwd and u.nome = nome and u.cognome = cognome and u.dataNascita = dataNascita)

Metodo	**+ inizializzaSessione(Utente utente, HttpSession session) : void**
Descrizione	Inizializza la sessione per l'utente autenticato, creando un oggetto Sessione.
Pre-condizione	self.utenti->includes(utente) and session <> null
Post-condizione	session.utente = utente

Interfaccia	**GestioneProfilo**
Descrizione	Permette di modificare i dati personali, gestire gli indirizzi di spedizione e i metodi di pagamento e visualizzare lo storico degli ordini effettuati.
Metodi	+ modificaDatiPersonali(Utente utente, String newNome, String newCognome, Date newDataNascita, String newTelefono) : void + addIndirizzoSpedizione(Utente utente, Indirizzo newIndirizzo) : void + modificaIndirizzoSpedizione(Utente utente, Indirizzo indirizzo, Indirizzo newIndirizzo) : void + removeIndirizzoSpedizione(Utente utente, Indirizzo indirizzo) : void + addMetodoPagamento(Utente utente, MetodoPagamento newMetodoPagamento) : void + modificaMetodoPagamento(Utente utente, MetodoPagamento metodoPagamento, MetodoPagamento newMetodoPagamento) : void + removeMetodoPagamento(Utente utente, MetodoPagamento metodoPagamento) : void + visualizzaStoricoOrdini(Utente utente) : List<Ordine>
Invariante di classe	-

Metodo	**+ modificaDatiPersonali(Utente utente, String newNome, String newCognome, Date newDataNascita, String newTelefono) : void**
Descrizione	Modifica i dati personali dell'utente.
Pre-condizione	self.utenti->includes(utente) and newNome.isNotEmpty() and newCognome.isNotEmpty() and newDataNascita <> null and newTelefono.isNotEmpty()
Post-condizione	utente.nome = newNome and utente.cognome = newCognome and utente.dataNascita = newDataNascita and utente.telefono = newTelefono

Metodo	**+ addIndirizzoSpedizione(Utente utente, Indirizzo newIndirizzo) : void**
Descrizione	Aggiunge un nuovo indirizzo di spedizione all'utente.
Pre-condizione	self.utenti->includes(utente) and newIndirizzo.isValid()
Post-condizione	utente.indirizzi->includes(newIndirizzo)

Metodo	**+ modificaIndirizzoSpedizione(Utente utente, Indirizzo indirizzo, Indirizzo newIndirizzo) : void**
Descrizione	Modifica un indirizzo di spedizione collegato all'utente.
Pre-condizione	self.utenti->includes(utente) and utente.indirizzi->includes(indirizzo) and newIndirizzo.isValid()
Post-condizione	utente.indirizzi->excludes(indirizzo) and utente.indirizzi->includes(newIndirizzo)

Metodo	**+ removeIndirizzoSpedizione(Utente utente, Indirizzo indirizzo) : void**
Descrizione	Rimuove un indirizzo di spedizione collegato all'utente.
Pre-condizione	self.utenti->includes(utente) and utente.indirizzi->includes(indirizzo)
Post-condizione	utente.indirizzi->excludes(indirizzo)

Metodo	**+ addMetodoPagamento(Utente utente, MetodoPagamento newMetodoPagamento) : void**
Descrizione	Aggiunge un metodo di pagamento all'utente.
Pre-condizione	self.utenti->includes(utente) and newMetodoPagamento.isValid()
Post-condizione	utente.metodiPagamento->includes(newMetodoPagamento)

Metodo	**+ modificaMetodoPagamento(Utente utente, MetodoPagamento metodoPagamento, MetodoPagamento newMetodoPagamento) : void**
Descrizione	Modifica un metodo di pagamento esistente collegato all'utente.
Pre-condizione	self.utenti->includes(utente) and utente.metodiPagamento->includes(metodoPagamento) and newMetodoPagamento.isValid()
Post-condizione	utente.metodiPagamento->excludes(metodoPagamento) and utente.metodiPagamento->includes(newMetodoPagamento)

Metodo	**+ removeMetodoPagamento(Utente utente, MetodoPagamento metodoPagamento) : void**
Descrizione	Rimuove un metodo di pagamento collegato all'utente.
Pre-condizione	self.utenti->includes(utente) and utente.metodiPagamento->includes(metodoPagamento)
Post-condizione	utente.metodiPagamento->excludes(metodoPagamento)

Metodo	**+ visualizzaStoricoOrdini(Utente utente) : List<Ordine>**
Descrizione	Recupera lo storico degli ordini dell'utente.
Pre-condizione	self.utenti->includes(utente)
Post-condizione	result = utente.getOrdini()

Interfaccia	**Catalogo**
Descrizione	Permette di aggiungere, modificare e rimuovere prodotti dal catalogo, oltre a poterlo visualizzare interamente.
Metodi	+ addNewProdotto(Prodotto prodotto) : boolean + modificaProdotto(Prodotto prodotto, Prodotto nuovoProdotto) : boolean + rimuoviProdotto(Prodotto prodotto) : boolean + getProdotti() : List<Prodotto> + getLatestProducts(int limit) : List<Prodotto> + getBestSellingProducts(int limit) : List<Prodotto> + getUpcomingProducts(int limit) : List<Prodotto>
Invarante di classe	-

Metodo	**+ addNewProdotto(Prodotto prodotto) : boolean**
Descrizione	Aggiunge un nuovo prodotto al catalogo.
Pre-condizione	prodotto.isValid() and self.prodotti->forAll(p | p.id <> prodotto.id)
Post-condizione	if result = true then self.prodotti->includes(prodotto) endif

Metodo	**+ modificaProdotto(Prodotto prodotto, Prodotto nuovoProdotto) : boolean**
Descrizione	Modifica un prodotto esistente.
Pre-condizione	self.prodotti->includes(prodotto) and nuovoProdotto.isValid()
Post-condizione	if result = true then self.prodotti->excludes(prodotto) and self.prodotti->includes(nuovoProdotto) endif

Metodo	**+ rimuoviProdotto(Prodotto prodotto) : boolean**
Descrizione	Rimuove un prodotto del catalogo.
Pre-condizione	self.prodotti->includes(prodotto)
Post-condizione	if result = true then self.prodotti->excludes(prodotto) endif

Metodo	**+ getProdotti() : List<Prodotto>**
Descrizione	Restituisce la lista di tutti i prodotti.
Pre-condizione	-
Post-condizione	result = self.prodotti

Metodo	**+ getLatestProducts(int limit) : List<Prodotto>**
Descrizione	Restituisce la lista dei prodotti più recenti.
Pre-condizione	limit > 0
Post-condizione	result->size() <= limit and result->forAll(p | self.prodotti->includes(p)) and result = self.prodotti->sortedBy(p | p.dataInserimento.descending())->first(limit)

Metodo	**+ getBestSellingProducts(int limit) : List<Prodotto>**
Descrizione	Restituisce la lista dei prodotti più venduti.
Pre-condizione	limit > 0
Post-condizione	result->size() <= limit and result->forAll(p | self.prodotti->includes(p)) and result = self.prodotti->sortedBy(p | p.numeroVendite.descending())->first(limit)

Metodo	**+ getUpcomingProducts(int limit) : List<Prodotto>**
Descrizione	Restituisce la lista dei prodotti in arrivo.
Pre-condizione	limit > 0
Post-condizione	result->size() <= limit and result->forAll(p | self.prodotti->includes(p) and p.dataRilascio > Date::now()) and result = self.prodotti->select(p | p.dataRilascio > Date::now())->sortedBy(p | p.dataRilascio.ascending())->first(limit)

Interfaccia	**CarrelloService**
Descrizione	Permette di aggiungere e rimuovere prodotti dal carrello, visualizzarne il contenuto e finalizzare l'acquisto creando un ordine.
Metodi	+ addProdottoCarrello(Carrello carrello, Prodotto prodotto, int quantita) : void + removeProdottoCarrello(Carrello carrello, Prodotto prodotto) : void + creaOrdine(Carrello carrello, Indirizzo indirizzo, MetodoPagamento pagamento) : void
Invariante di classe	-

Metodo	**+ addProdottoCarrello(Carrello carrello, Prodotto prodotto, int quantita) : void**
Descrizione	Aggiunge un prodotto specificato al carrello dell'utente con una determinata quantità.
Pre-condizione	prodotto.isValid() and quantita > 0
Post-condizione	if carrello.prodotti->exists(cp | cp.prodotto = prodotto) then carrello.prodotti->select(cp | cp.prodotto = prodotto)->first().quantita = carrello.prodotti->select(cp | cp.prodotto = prodotto)->first().quantita@pre + quantita else carrello.prodotti->exists(cp | cp.prodotto = prodotto and cp.quantita = quantita) endif

Metodo	**+ removeProdottoCarrello(Carrello carrello, Prodotto prodotto) : void**
Descrizione	Rimuove un prodotto dal carrello dell'utente.
Pre-condizione	carrello.prodotti->exists(cp | cp.prodotto = prodotto)
Post-condizione	carrello.prodotti->forAll(cp | cp.prodotto <> prodotto)

Metodo	**+ creaOrdine(Carrello carrello, Indirizzo indirizzo, MetodoPagamento pagamento) : void**
Descrizione	Finalizza il carrello, crea un ordine e lo salva nel sistema.
Pre-condizione	not carrello.prodotti->isEmpty() and indirizzo.isValid() and pagamento.isValid()
Post-condizione	let ordine : Ordine = Ordine.allInstances()->any(o | o.carrello = carrello) in ordine.prodotti = carrello.prodotti and ordine.indirizzo = indirizzo and ordine.metodoPagamento = pagamento and carrello.prodotti->isEmpty()

4. Glossario

Non sono stati introdotte nuove terminologie particolari.