Laurea Triennale in informatica-Università di Salerno
Corso di *Ingegneria del Software*- Prof. C.Gravino

ODD
OBJECT DESIGN DOCUMENT

UniPass

Riferimento	
Versione	1.6
Data	17/02/2023
Destinatario	Prof C.Gravino
Presentato da	Barretta Alessandro Di Lauro Antonio Malafronte Sabato Zampaglione Gerardo
Approvato da	

Revision History

Data	Versione	Cambiamenti	Autori
27/12/2022	1.0	Prima Stesura	Alessandro Barretta
02/01/2023	1.1	Aggiunta Class Interfaces Package registrazione	Gerardo Zampaglione
03/01/2023	1.2	Aggiunta Class Interfaces Package autenticazione	Sabato Malafronte
04/01/2023	1.3	Aggiunta Class Interfaces Package viaggio e prenotazione	Antonio Di Lauro
04/01/2023	1.4	Aggiunta Class Interfaces Package veicolo e valutazione	Alessandro Barretta
11/02/2023	1.5	Stesura sezione Design Pattern	[Gruppo]
17/02/2023	1.6	Aggiunta Class Diagram	[Gruppo]

Sommario

1. Introduzione

Il sistema UniPass si prefigge come obiettivo quello di aiutare gli studenti dell'Università di Salerno nel far ritorno a casa, assicurando loro una soluzione a quelli che sono i problemi dei trasporti pubblici tradizionali.

In questa prima parte del documento saranno descritti i trade-offs e le linee guida per la fase di implementazione, riguardanti la nomenclatura, la documentazione e le convenzioni sui formati.

1.1 Linee guida per la scrittura del codice

Le linee guida per l'implementazione del codice raggruppano un insieme di regole che gli sviluppatori devono seguire durante la progettazione delle interfacce. Per la loro costruzione si è fatto riferimento alla convenzione java nota come Sun Java Coding Conventions [Sun, 2009].

Link a documentazione ufficiale sulle convenzioni
Di seguito una lista di link alle convenzioni usate per definire le linee guida:
• Java Sun: https://checkstyle.sourceforge.io/sun_style.html
• HTML: https://www.w3schools.com/html/html5_syntax.asp

1.2 Definizioni, acronimi e abbreviazioni

Vengono riportati di seguito alcune definizioni presenti nel documento:

• **Package**: riunisce classi, interfacce o file correlati;
• **Design pattern**: soluzione generale a problemi ricorrenti che si incontrano durante le fasi di implementazione o sviluppo di sistema software al fine di ottenere riuso e flessibilità;
• **Interfaccia**: lista di signature delle operazioni fornite dalla classe;
• **lowerCamelCase**: notazione usata per scrivere parole composte o frasi unendo tutte le parole tra loro con la prima lettera delle parole intermedie in maiuscolo;
• **UpperCamelCase**: notazione usata per scrivere parole composte o frasi unendo tutte le parole tra loro con la prima lettera di ciascuna parola in maiuscolo;
• **Javadoc**: strumento offerto da Java che genera pagine di documentazione in formato HTML raccogliendo informazioni dai commenti che gli sviluppatori lasciano sul codice sorgente.

1.3 Riferimenti e Link Utili

È stato preso come riferimento il libro di testo: Object-Oriented Software Engineering (Using UML, Patterns, and Java) Third Edition che tratta la progettazione e l'analisi orientata agli oggetti.

2. Packages

In questa parte del documento si vanno ad esplicitare i vari packages che compongono il sistema, avendo come riferimento il documento di System Design.
Come la regola impone, ogni classe del sistema sarà disposta nel file ad essa dedicato, e i file saranno collocati nei package assegnati, che saranno disposti in modo gerarchico. Le scelte dei vari packages sono state prese in accordo sia con le scelte architetturali intraprese durante i precedenti documenti, che con la struttura standard delle cartelle di lavoro fornita da Maven.

La suddivisione è la seguente:

- **.idea**
- **src**, contiene i file sorgente
    - **main**
        - **java**
            - **com**
                - **saga**
                    - **unipass**, contiene le classi Controller, Model e Service
        - **resources**
            - **static**
                - **css**, contiene i file CSS
                - **img**, contiene le immagini
                - **js**, contiene gli script JS
            - **templates**, contiene i file HTML
    - **test**, contiene le classi per il Testing di unità

Package UniPass
Il package principale di UniPass contiene i package di tutti i sottosistemi. Si è deciso di creare un package per ogni sottosistema e creare un package a parte per il model, dove si andranno a mettere i Java Beans. I package contenuti nel package unipass saranno quindi:

1. Registrazione: gestisce la registrazione degli utenti;
2. Autenticazione: gestisce le funzioni di Login, Logout e gestione del profilo;
3. Viaggio: gestisce le funzioni relative ai Viaggi;
4. Prenotazione: gestisce le funzioni relative alle prenotazioni ai Viaggi;
5. Veicolo: gestisce l'aggiunta e la modifica di Veicoli;
6. Valutazione: gestisce le funzioni relative alla valutazione degli utenti.
7. Dao: gestisce gli accessi al Database.
8. Model: contiene i Java Beans.

3. Interfacce delle classi

Di seguito vengono presentate le interfacce dei package di UniPass.

Nome classe	RegistrazioneService
Descrizione	Questa classe permette di gestire le operazioni relative alla registrazione.
Metodi	+registrazione (String email, String password, String nome, String cognome, String telefono): Utente
Invariante di classe	/

Nome Metodo	+registrazione (String email, String password, String nome, String cognome, String telefono): Utente
Descrizione	Questo metodo permette la registrazione da parte di un utente
Pre-condizione	/
Post-condizione	**context:** RegistrazioneService::registrazione(String email, String password, String nome, String cognome, String telefono) **post:** AutenticazioneDAO.save(utente) == true

Nome classe	PrenotazioneService
Descrizione	Questa classe permette di gestire le operazioni relative alle prenotazioni.
Metodi	- prenotaViaggio(int idViaggio): void - cancellaPrenotazione(int idViaggio): void - chiudiPrenotazioni(int idViaggio): void
Invariante di classe	/

Nome Metodo	+prenotaViaggio(int idViaggio): void
Descrizione	Questo metodo permette la prenotazione di un viaggio da parte di un utente
Pre-condizione	/
Post-condizione	context: PrenotazioneService::prenotaViaggio (int idViaggio) post: visualizzaStorico().contains(viaggioDAO.getViaggioById(idViaggio))

Nome Metodo	+cancellaPrenotazione(int idViaggio): void
Descrizione	Questo metodo permette la cancellazione di una prenotazione fatta da un utente per un viaggio
Pre-condizione	context: PrenotazioneService::cancellaPrenotazione(int idViaggio) **pre:** visualizzaStorico().contains(viaggioDAO.getViaggioById(idViaggio))
Post-condizione	context: PrenotazioneService::cancellaPrenotazione(int idViaggio) **post:** not(visualizzaStorico().contains(viaggioDAO.getViaggioById(idViaggio)))

Nome Metodo	+chiudiPrenotazioni(int idViaggio): void
Descrizione	Questo metodo permette la chiusura delle prenotazioni per un viaggio da parte del guidatore che lo ha creato
Pre-condizione	
Post-condizione	context: PrenotazioneService:chiudiPrenotazioni (String idViaggio) post: not(listaViaggi.contains(viaggioDAO.getViaggioById(idViaggio)))

Nome classe	AutenticazioneService
Descrizione	Questa classe permette di gestire le operazioni relative all'autenticazione.
Metodi	+login(String email, String password) : UtenteRegistrato +logout() : void +modificaProfilo(String nome, String cognome, String email, String password, String telefono) : Boolean +visualizzaStorico() : List<Viaggio>
Invariante di classe	/

Nome Metodo	+login(String email, String password) : UtenteRegistrato
Descrizione	Questo metodo consente di accedere alla piattaforma.
Pre-condizione	/
Post-condizione	**context:** AutenticazioneService::login(email, password) **post:** if(utenteRegistratoDAO.getElementByEmailPassword(email, password) == null) {utenteRegistrato == null} else utenteRegistrato.getEmail.equals(email) && utenteRegistrato.getPassword.equals(password)
Nome Metodo	+logout() : void
Descrizione	Questo metodo si utilizza per disconnettersi dalla piattaforma.
Pre-condizione	**context:** AutenticazioneService::logout() **pre:** session.getAttribute("utenteRegistrato") != null

Post-condizione	**context:** AutenticazioneService::logout() **post:** session.getAttribute("utenteRegistrato") == null
Nome Metodo	+modificaProfilo(String nome, String cognome, String email, String password, String telefono) : Boolean
Descrizione	Questo metodo permette ad un utente di modificare le informazioni personali.
Pre-condizione	/
Post-condizione	**context:** AutenticazioneService::modificaProfilo(String nome, String cognome, String email, String password, String telefono) **post:** utente.getNome().equals(nome) && utente.getCognome.equals(cognome) && utente.getEmail.equals(email) && utente.getPassword.equals(password) && utente.getTelefono.equals(telefono)
Nome Metodo	+visualizzaStorico() : List<Viaggio>
Descrizione	Questo metodo fornisce all'utente una lista di tutti i viaggi che ha creato e a cui ha partecipato.
Pre-condizione	/
Post-condizione	/

Nome classe	ViaggioService
Descrizione	Questa classe permette di gestire le operazioni relative al viaggio.
Metodi	+creaViaggio(String destinazione, Date dataPartenza, int posti, int prezzo) : Viaggio +eliminaViaggio(Viaggio viaggio) : Boolean +cercaViaggi(String destinazione, Date dataPartenza, int prezzo) : List<Viaggio> +escludiPasseggero(Utente passeggero) : Boolean
Invariante di classe	/

Nome Metodo	+creaViaggio(String destinazione, Date dataPartenza, int posti, int prezzo) : Viaggio
Descrizione	Questo metodo permette ad un Guidatore di creare un nuovo viaggio.
Pre-condizione	**context:** ViaggioService::creaViaggio(destinazione, dataPartenza, posti, prezzo) **pre:** isGuidatore(UtenteRegistrato)==true
Post-condizione	**context:** ViaggioService::creaViaggio(destinazione, dataPartenza, posti, prezzo) **post:** viaggioDAO.save(Viaggio) == true
Nome Metodo	+eliminaViaggio(Viaggio viaggio) : Boolean
Descrizione	Questo metodo permette ad un Guidatore di eliminare un viaggio da lui creato.
Pre-condizione	**context:** ViaggioService::eliminaViaggio(viaggio) **pre:** visualizzaStorico().includes(viaggio)
Post-condizione	**context:** ViaggioService::eliminaViaggio(viaggio) **post:** not(visualizzaStorico().includes(viaggio))

Nome Metodo	+cercaViaggi(String destinazione, Date dataPartenza, int prezzo) : List<Viaggio>
Descrizione	Questo metodo permette ad un utente di cercare un viaggio in base a dei filtri
Pre-condizione	/

Post-condizione	Foreach(Viaggio viaggio: listaViaggi) {
	
	viaggio.isPrenotabile() && viaggio.getDestinazione.equals(destinazione) && viaggio.getDataPartenza >= max(dataPartenza, dataCorrente + *40 minuti*) && viaggio.getPrezzo <= prezzo && viaggio.getPostiDisponibili >= 1
	
	}

Nome Metodo	+escludiPasseggero(UtenteRegistrato passeggero) : Boolean
Descrizione	Questo metodo permette ad un Guidatore di escludere un passeggero prenotato ad un suo Viaggio.
Pre-condizione	/
Post-condizione	**context:** ViaggioService::escludiPasseggero(passeggero) **post:** not(listaPasseggeriViaggio.contains(passeggero))

Nome classe	VeicoloService
Descrizione	Questa classe fornisce i metodi necessari alla gestione dei veicoli da parte dei guidatori (aggiunta, modifica, rimozione).
Metodi	+aggiungiVeicolo(String targa, String marca, String modello, String colore, int postiDisponibili): void +modificaVeicolo(String targa, String marca, String modello, String colore, int postiDisponibili): void +rimuoviVeicolo(Veicolo veicolo): void
Invariante di classe	

Nome Metodo	+aggiungiVeicolo(String targa, String marca, String modello String colore, int postiDisponibili): void
Descrizione	Questo metodo permette di associare un veicolo ad un guidatore
Pre-condizione	**context:** VeicoloService::aggiungiVeicolo(String targa, String marca, String modello, String colore, int postiDisponibili) **pre:** utente.getVeicolo() == null
Post-condizione	**context:** VeicoloService::aggiungiVeicolo(String targa, String marca, String modello, String colore, int postiDisponibili) **post:** utente.getVeicolo().getTarga().equals(targa) && utente.getVeicolo().getMarca().equals(marca) && utente.getVeicolo().getModello().equals(modello) && utente.getVeicolo().getColore().equals(colore) && utente.getVeicolo().getPostiDisponibili() == postiDisponibili

Nome Metodo	+modificaVeicolo(String targa, String marca, String modello String colore, int postiDisponibili): void
Descrizione	Questo metodo permette di modificare il veicolo associato ad un guidatore
Pre-condizione	**context:** VeicoloService::modificaVeicolo(String targa, String marca, String modello, String colore, int postiDisponibili) **pre:** utente.getVeicolo() != null
Post-condizione	**context:** VeicoloService::modificaVeicolo(String targa, String marca, String modello, String colore, int postiDisponibili) **post:** utente.getVeicolo().getTarga().equals(targa) && utente.getVeicolo().getMarca().equals(marca) && utente.getVeicolo().getModello().equals(modello) && utente.getVeicolo().getColore().equals(colore) && utente.getVeicolo().getPostiDisponibili() == postiDisponibili

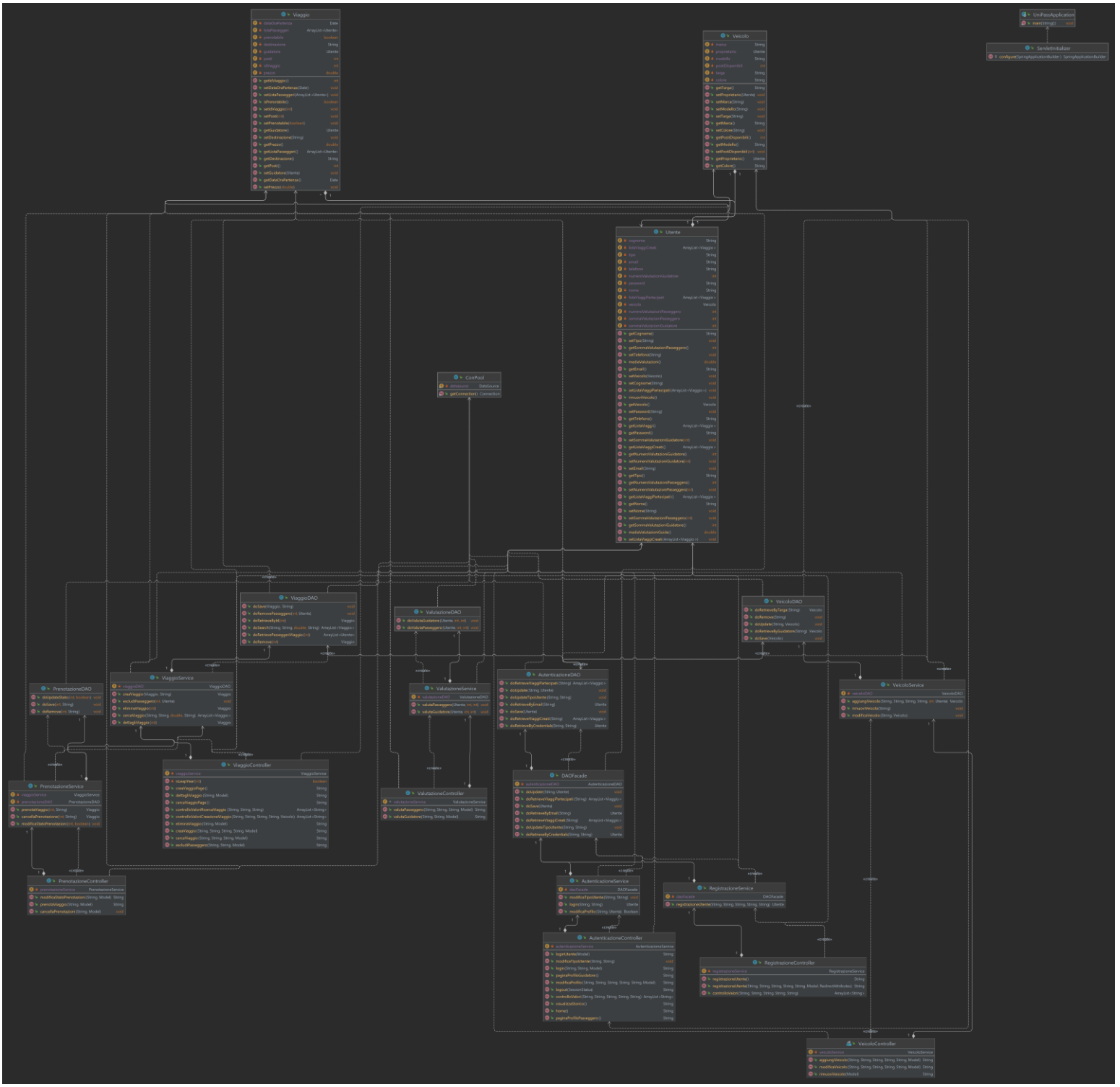
Nome Metodo	+rimuoviVeicolo(): void
Descrizione	Questo metodo permette di rimuovere il veicolo associato ad un guidatore
Pre-condizione	/
Post-condizione	**context:** VeicoloService::rimuoviVeicolo(Veicolo veicolo) **post:** utente.getVeicolo() == null

Nome classe	ValutazioneService
Descrizione	Questa classe permette la registrazione delle valutazioni utente
Metodi	+valutaGuidatore(Utente guidatore): void +valutaPasseggero(Utente passeggero): void
Invariante di classe	

Nome Metodo	+valutaGuidatore(Utente guidatore)
Descrizione	Questo metodo permette ad un passeggero di valutare il guidatore di un viaggio a cui ha partecipato
Pre-condizione	/
Post-condizione	**context:** ValutazioneService::valutaGuidatore(Utente guidatore) **post:** guidatore.numValutazioni = @pre.guidatore.numValutazioni + 1

Nome Metodo	+valutaPasseggero(Utente passeggero)
Descrizione	Questo metodo permette ad un guidatore di valutare un passeggero che ha partecipato ad un suo viaggio
Pre-condizione	/
Post-condizione	**context:** ValutazioneService::valutaGuidatore(Utente passeggero) **post:** passeggero.numValutazioni = @pre.passeggero.numValutazioni + 1

4. Class Diagram



5. Elementi di riuso

In questa sezione si discuterà il design pattern utilizzato per adottare il riuso.

5.1 Design Pattern usati

Il Facade Pattern è un design pattern strutturale che fornisce un'interfaccia unificata semplificata per un insieme di interfacce complesse all'interno di un sottosistema, al fine di rendere più semplice l'utilizzo del sottosistema stesso. In questo progetto, il Facade Pattern è stato utilizzato per fornire una classe di interfaccia, chiamata "DAOFacade", che nasconde la complessità dell'accesso ai dati e semplifica l'utilizzo del sottosistema implementando i metodi di interfaccia con i metodi della classe DAO.