

2024/25

Documentazione

Versione 2.0



Maffeis Riccardo – Mat. 1085706

Zanotti Matteo – Mat. 1085443



Indice

Sommario

Software Life Cycle	2
Configuration Management	5
People Management and Team Organization	8
Modelling	14
Software Architecture	16
Software Design	19
Design Pattern	20
Software Testing	23
Software Maintenance.....	24

Software Life Cycle

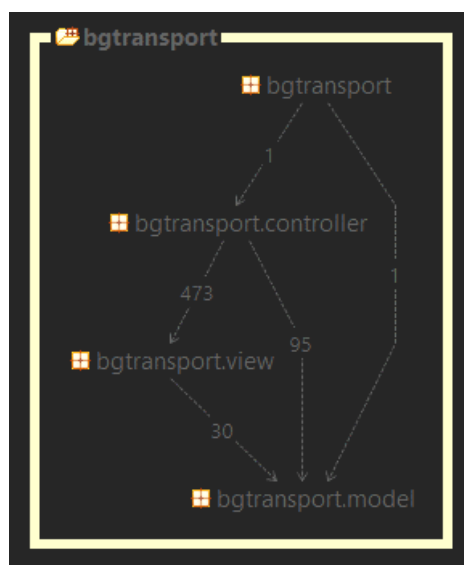
Nel nostro progetto di sviluppo software, abbiamo adottato il framework SCRUM per gestire in maniera agile ed efficiente l'intero ciclo di sviluppo. Questo approccio ci ha permesso di organizzare e suddividere il lavoro in modo iterativo e incrementale, favorendo una collaborazione costante tra i membri del team e una continua revisione delle funzionalità implementate.

In una prima fase, lo sviluppo del software è stato strutturato in tre principali pacchetti:

1. **bgtransport.controller**: responsabile della gestione della logica di controllo e dell'interazione tra gli altri moduli;
2. **bgtransport.model**: dedicato alla gestione dei dati e della logica di business del sistema;
3. **bgtransport.view**: incaricato della presentazione e dell'interfaccia utente.

In seguito, è stato introdotto un quarto pacchetto, denominato **bgtransport**, il quale svolge una funzione centrale nel progetto. Questo pacchetto contiene il **Main**, ossia il punto di ingresso principale dell'applicazione, che ha il compito di orchestrare il funzionamento complessivo del software. Inoltre, il pacchetto bgtransport si occupa di gestire due componenti fondamentali: il **MainController**, collocato all'interno del pacchetto bgtransport.controller, e il **MainModel**, situato nel pacchetto bgtransport.model. Questi due elementi rappresentano le strutture principali per il coordinamento delle operazioni e l'integrazione tra i diversi moduli.

Per analizzare e rappresentare visivamente le relazioni tra i vari pacchetti del progetto, abbiamo utilizzato lo strumento **STAN4J**. La seguente immagine, generata tramite STAN4J, illustra chiaramente le connessioni e le dipendenze esistenti tra i diversi pacchetti del software, fornendo una visione d'insieme della sua architettura modulare.



Inoltre, abbiamo inserito delle tabelle che mostrano come sono stati gestiti gli sprint e i daily:

Sprint	Data	Modalità	Decisioni
0	18/10/2024	Riunione in presenza	Miglioramenti al Project Plan e creazione del file che contiene i requisiti
1	25/10/2024	Riunione in presenza	Miglioramenti al Project Plan, miglioramenti dei requisiti
2	01/11/2024	Riunione in presenza	Termine prima versione ufficiale Project Plan e creazione primi schemi UML
3	08/11/2024	Riunione a distanza	Avanzamento creazione schemi UML e inizio scrittura codice (creazione JOOQ, database e file associati e prime parti della GUI)
4	15/11/2024	Riunione a distanza	Nuove implementazioni nella GUI (mappa, meteo, login) e avanzamento creazione file per i database
5	22/11/2024	Riunione a distanza	Avanzamento creazione GUI (signUp, database, trova la fermata)
6	29/11/2024	Riunione a distanza	Avanzamento della GUI e interconnessione con i database
7	06/12/2024	Riunione a distanza	Miglioramenti alla GUI e inserimento commenti e JavaDoc nel codice
8	13/12/2024	Riunione a distanza	Miglioramenti alla GUI e sistemazione del codice attraverso SonarLint/SonarQube e STAN4J
9	20/12/2024	Riunione a distanza	Miglioramenti alla GUI e sistemazione del codice attraverso SonarLint/SonarQube e STAN4J
10	27/12/2024	Riunione a distanza	Miglioramenti alla GUI e sistemazione del file relativo ai requisiti
11	03/01/2025	Riunione a distanza	Revisione diagrammi UML
12	10/01/2025	Riunione a distanza	Termine realizzazione e revisione della documentazione e creazione Presentazione per esposizione

Sprint	Data inizio	Data fine	Durata
1	21/10/2024	25/10/2024	1 settimana
2	28/10/2024	01/11/2024	1 settimana
3	04/11/2024	08/11/2024	1 settimana
4	11/11/2024	15/11/2024	1 settimana
5	18/11/2024	22/11/2024	1 settimana
6	25/11/2024	29/11/2024	1 settimana
7	02/12/2024	06/12/2024	1 settimana
8	09/12/2024	13/12/2024	1 settimana
9	16/12/2024	20/12/2024	1 settimana
10	23/12/2024	27/12/2024	1 settimana
11	30/12/2024	03/01/2025	1 settimana
12	06/01/2025	10/01/2025	1 settimana

Daily	Data	Modalità	Sprint
1 - 5	21/10 – 25/10 (2024)	Riunioni in presenza giornaliere	1
6-10	28/10 – 1/11 (2024)	Riunioni in presenza e a distanza giornaliere	2
11-15	04/11 – 08/11 (2024)	Riunioni a distanza giornaliere	3
16-20	11/11 – 15/11 (2024)	Riunioni in presenza giornaliere	4
21-25	18/11 – 22/11 (2024)	Riunioni in presenza giornaliere	5
26-30	25/11 – 29/11 (2024)	Riunioni a distanza giornaliere	6
31-35	02/12 – 06/12 (2024)	Riunioni a distanza giornaliere	7
36-40	09/12 – 13/12 (2024)	Riunioni a distanza giornaliere	8
41-45	16/12 – 20/12 (2024)	Riunioni a distanza giornaliere	9
46-50	23/12 – 27/12 (2024)	Riunioni a distanza giornaliere	10
51-55	30/12 – 03/01 (2024/25)	Riunioni a distanza giornaliere	11
56-60	06/01 – 10/01 (2025)	Riunioni a distanza giornaliere	12

Configuration Management

Abbiamo utilizzato GitHub e i relativi comandi (add, commit, push, pull, fetch, clone, merge) e i tools offerti: Project Board, di tipo kanban, con il quale vengono gestiti gli elementi da implementare e i bug da sistemare; RoadMap, la quale possiede una linea del tempo sulla quale vengono distribuiti i vari bug/enhancement in base alla data di creazione; My items, il quale mostra un elenco, in base alle priorità, dei vari bug/ enhancement.

Il link per accedere al repository è il seguente: <https://github.com/ZanottiMatteo/BGTransports>

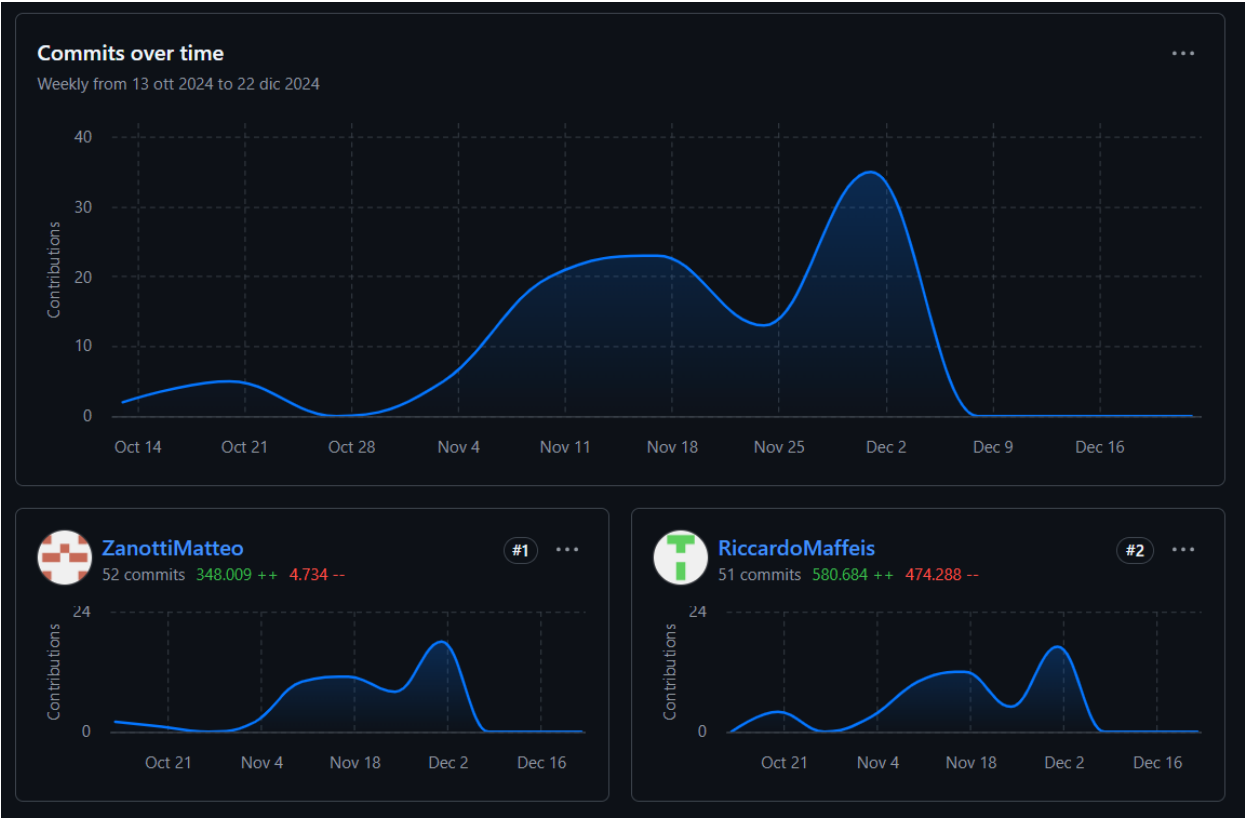
Issues:

43 Open ✓ 8 Closed		Author ▾	Label ▾	Projects ▾	Milestones ▾	Assignee ▾	Sort ▾
🕒	creazione DownloadDataDBController	enhancement					1
#54	opened 3 weeks ago by RiccardoMaffeis						
🕒	Standardizzazione dei Jpanel che si ripetono nei JFrame	enhancement					
#53	opened 3 weeks ago by ZanottiMatteo						
🕒	Creazione classe Utility	enhancement					
#51	opened 3 weeks ago by RiccardoMaffeis						
🕒	Gestione WaypointController	enhancement					
#50	opened 3 weeks ago by ZanottiMatteo						
🕒	Gestione UserInfoController	enhancement					
#49	opened 3 weeks ago by ZanottiMatteo						
🕒	Gestione AccountController	enhancement					
#48	opened 3 weeks ago by ZanottiMatteo						
🕒	Gestione Signup con collegamento a Database	enhancement					
#47	opened 3 weeks ago by ZanottiMatteo						
🕒	Gestione AccountIconView	enhancement					
#46	opened 3 weeks ago by ZanottiMatteo						
🕒	Creazione DownloadDataDBView	enhancement					
#45	opened 3 weeks ago by RiccardoMaffeis						
🕒	Gestione UserView						
#44	opened 3 weeks ago by ZanottiMatteo						

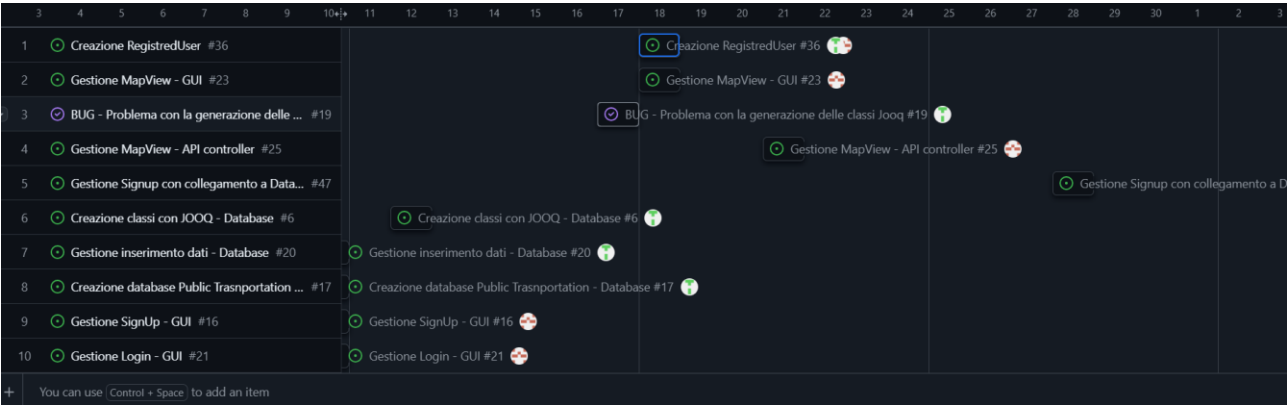
Pull Requests:

0 Open ✓ 2 Closed		Author ▾	Label ▾	Projects ▾	Milestones ▾	Reviews ▾	Assignee ▾	Sort ▾
🔗	Merge to public - Second release	Pull Request				2		
#43	by ZanottiMatteo was merged 3 weeks ago							
🔗	Merge to Main	Pull Request						
#39	by ZanottiMatteo was merged on Nov 22							

Commit:



RoadMap:



My Items:

New
This item hasn't been started
7

In progress
This is actively being worked on
3

Ready
This is ready to be picked up
4

Testing
This item is in review
10

Done
This has been completed
7

Show empty values

assignee:@me

31

Discard

Save

Title	Priority	Size
1 Creazione RegisteredUser #36	P1	-
2 BUG - Problema con la generazione delle classi Jooq #19	P1	-
3 Creazione classi con JOOQ - Database #6	P1	-
4 Gestione inserimento dati - Database #20	P1	-
5 Creazione database Public Transportation - Database #17	P1	-
6 Creazione Constant - Database #37	P1	-
7 Creazione database User - Database #18	P1	-
8 Gestione Login - Database #24	P1	-
9 Creazione tabelle - Database #26	P1	-
10 Creazione Cartella per gestione Risorse #27	P1	-
11 Project Plan #2	P0	-
12 UML #3	P0	-

+ You can use Control + Space to add an item

Priority Board:

New 7 Estimate: 0
This item hasn't been started

In progress 6 Estimate: 0
This is actively being worked on

Ready 10 Estimate: 0
This is ready to be picked up

Testing 18 Estimate: 0
This item is in review

Don
This has

P0 6 Estimate: 0

P1 45 Estimate: 0

BGTransports #9 UML: creazione class diagram	BGTransports #7 Creazione HomeView	BGTransports #25 Gestione MapView - API controller	BGTransports #36 Creazione RegisteredUser	BGTransports #19 BUG - Problema con la generazione delle classi Jooq
BGTransports #10 UML: creazione use case diagram	BGTransports #44 Gestione UserView	BGTransports #47 Gestione Signup con collegamento a Database	BGTransports #23 Gestione MapView - GUI	BGTransports #19 BUG - Problema con la generazione delle classi Jooq
BGTransports #11 UML: creazione state machine diagram	BGTransports #49 Gestione UserInfoController	BGTransports #27 Creazione Cartella per gestione Risorse	BGTransports #6 Creazione classi con JOOQ - Database	BGTransports #19 BUG - Problema con la generazione delle classi Jooq
BGTransports #12 UML: creazione sequence diagram		BGTransports #31 Creazione WeatherModel	BGTransports #20 Gestione inserimento dati - Database	BGTransports #19 BUG - Problema con la generazione delle classi Jooq

People Management and Team Organization

Per lo sviluppo del software, abbiamo adottato un approccio agile, seguendo il framework SCRUM. Questo metodo ci ha consentito di mantenere un'elevata efficienza, garantendo una collaborazione attiva, una pianificazione accurata, una comunicazione trasparente e una notevole flessibilità durante tutte le fasi del progetto.

Il team di sviluppo è composto da due persone, ciascuna con un ruolo ben definito all'interno del framework SCRUM:

1. **Maffei Riccardo**: riveste il ruolo di **Scrum Master**. È responsabile di assicurarsi che le pratiche SCRUM vengano correttamente applicate, facilitando il processo di sviluppo e risolvendo eventuali problematiche che potrebbero ostacolare il lavoro del team.
2. **Zanotti Matteo**: ricopre la posizione di **Product Owner**. Si occupa della definizione delle funzionalità e della loro priorità, garantendo che il prodotto finale risponda alle esigenze degli stakeholder e agli obiettivi prefissati.

Entrambi i membri del team sono attivamente coinvolti nella programmazione del software, contribuendo sia allo sviluppo tecnico che al miglioramento continuo del progetto. Questo approccio collaborativo ha permesso di coniugare competenze tecniche e gestionali, favorendo un equilibrio tra qualità del prodotto e tempi di consegna.

Software Quality

Il software sviluppato è stato progettato per garantire i seguenti fattori di qualità:

- **Correttezza:** il sistema deve consentire all'utente di registrarsi, effettuare il login e visualizzare tutti i propri dati personali. Inoltre, l'utente deve essere in grado di interagire con la mappa integrata e ricercare la fermata desiderata con facilità.
- **Affidabilità:** il software deve rispondere alle richieste dell'utente in tempi brevi, entro un massimo di 2 secondi. Non devono verificarsi perdite di dati o blocchi del sistema.
- **Efficienza:** il software deve ottimizzare l'uso delle risorse del sistema, garantendo prestazioni adeguate anche in condizioni di carico moderato. L'architettura del software è progettata per ridurre al minimo il consumo di memoria e processore, garantendo al contempo tempi di risposta rapidi per tutte le funzionalità.
- **Integrità:** ogni utente dispone di un proprio ruolo che definisce i permessi di accesso alle varie sezioni del software, assicurando un controllo rigoroso degli accessi e una separazione delle funzionalità sensibili.
- **Usabilità:** il software è dotato di un'interfaccia grafica semplice e intuitiva, che consente agli utenti di navigare facilmente tra le varie sezioni. Pulsanti con icone descrittive aiutano a identificare rapidamente la funzione desiderata, migliorando l'esperienza utente.
- **Manutenibilità:** la presenza di una documentazione JavaDoc completa e di commenti all'interno del codice facilita l'individuazione e la correzione di eventuali errori. Inoltre, valori ricorrenti, come stringhe, interi e booleani, sono stati organizzati in classi come costanti, semplificando la gestione e la modifica del codice.
- **Testabilità:** sono state implementate classi di test per verificare le logiche di business e l'integrità dei database. Per quanto riguarda l'interfaccia grafica (GUI), la testabilità è assicurata direttamente dall'utente, che può verificare visivamente il corretto funzionamento di menù, pulsanti e mappe.

- **Flessibilità:** il software è stato progettato per essere facilmente estendibile. Nuove funzionalità possono essere aggiunte senza compromettere il corretto funzionamento delle componenti esistenti. Particolare attenzione è stata posta nell'integrazione delle nuove logiche con la parte grafica.
- **Portabilità:** il trasferimento del software da un ambiente operativo a un altro richiede uno sforzo minimo, grazie alla presenza di istruzioni dettagliate contenute nel file **README** disponibile su GitHub.

Requirement Engineering

Requisiti funzionali

1 Autenticazione e gestione degli utenti

- **RF-1: Registrazione degli utenti**

Gli utenti potranno registrarsi attraverso un modulo di registrazione, che richiederà i seguenti dati: nome, cognome, data di nascita, email, password, username, indirizzo città e CAP.

- **RF-2: Login degli utenti**

Gli utenti potranno effettuare il login tramite email e password. In caso di errore, il sistema visualizzerà un messaggio di errore specificando l'inesistenza dell'utente o l'errore di scrittura della password.

- **RF-3: Gestione del profilo utente**

Gli utenti registrati potranno accedere a una sezione dedicata al proprio profilo dove potranno aggiornare i propri dati personali e scegliere un'immagine del profilo. Essi potranno anche effettuare il logout dalla stessa pagina.

- **RF-4: Differenti tipi di accesso**

Gli utenti base (non registrati) potranno accedere solo alla schermata iniziale e alle opzioni di login/registrazione. Gli utenti registrati avranno accesso completo a tutte le funzionalità, inclusa la personalizzazione del profilo.

2 Consultazione delle informazioni di trasporto

- **RF-5: Visualizzazione delle informazioni sui trasporti**

L'applicazione visualizzerà le informazioni relative agli autobus con dettagli su orari, percorsi e fermate

- **RF-6: Mappa interattiva**

Gli utenti potranno interagire con una mappa che mostrerà le fermate dei mezzi di trasporto. La mappa dovrà anche mostrare la posizione dell'utente in tempo reale (tramite GPS del proprio dispositivo).

- **RF-7: Ricerca per fermata**

Gli utenti potranno calcolare un percorso desiderato inserendo le fermate di partenza e arrivo, ottenendo tutte le informazioni sulla linea desiderata.

3 Gestione dei dati personali

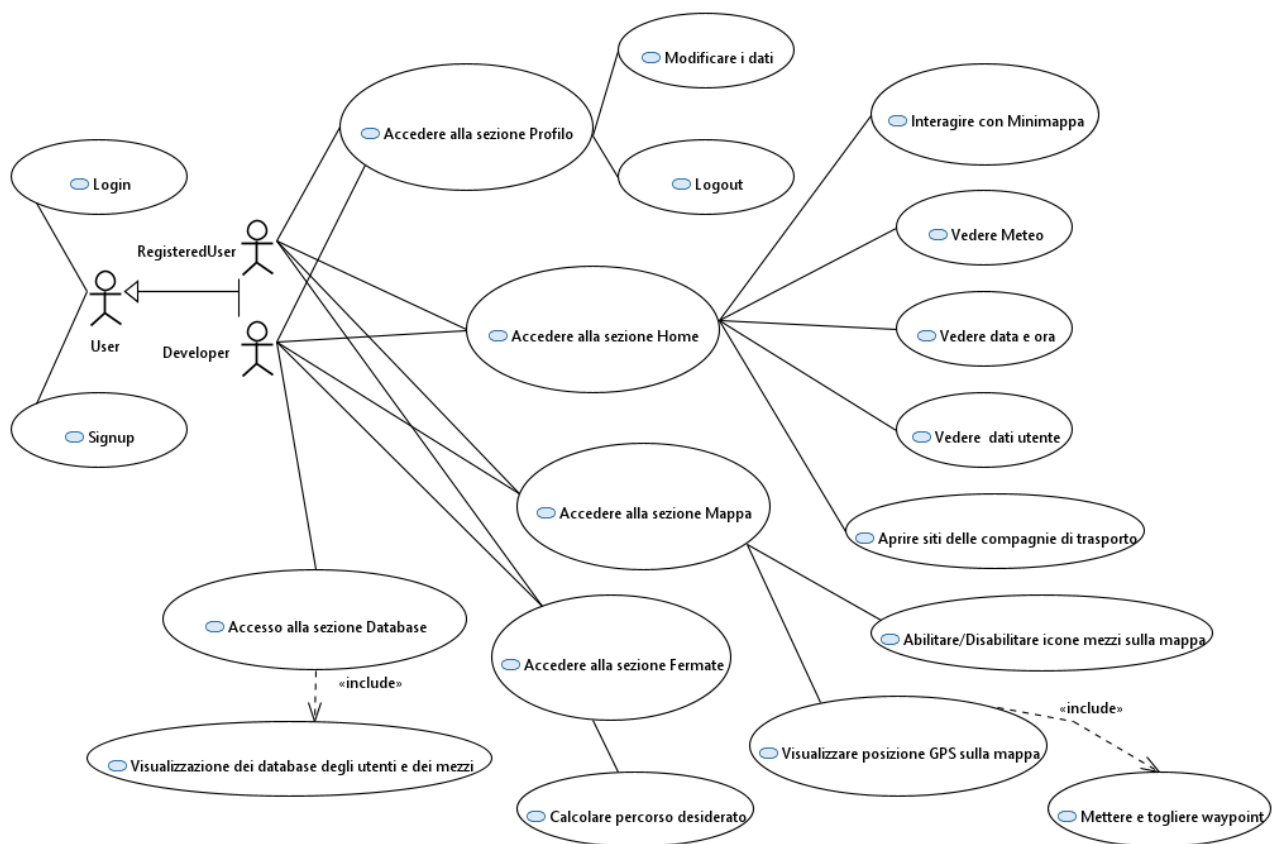
- **RF-8: Modifica dei dati utente**

Gli utenti registrati potranno modificare i propri dati personali attraverso una sezione di gestione del profilo.

4 Gestione dei permessi e amministrazione

- **RF-10: Accesso per gli sviluppatori**

Gli sviluppatori avranno accesso completo all'applicazione, inclusi i permessi per modificare le aree a loro dedicate.



Requisiti non funzionali

1 Prestazioni

- **RNF-1: Tempo di risposta**

Il sistema dovrà garantire un tempo di risposta inferiore a 3 secondi per le operazioni di caricamento delle informazioni di trasporto.

- **RNF-2: Scalabilità**

Il sistema dovrà essere progettato per gestire un numero crescente di utenti senza compromettere le prestazioni.

2 Sicurezza

- **RNF-3: Protezione dei dati**

I dati sensibili degli utenti, come password e informazioni personali, dovranno essere criptati sia durante la trasmissione che a riposo nel database.

3 Usabilità

- **RNF-4: Interfaccia utente semplice**

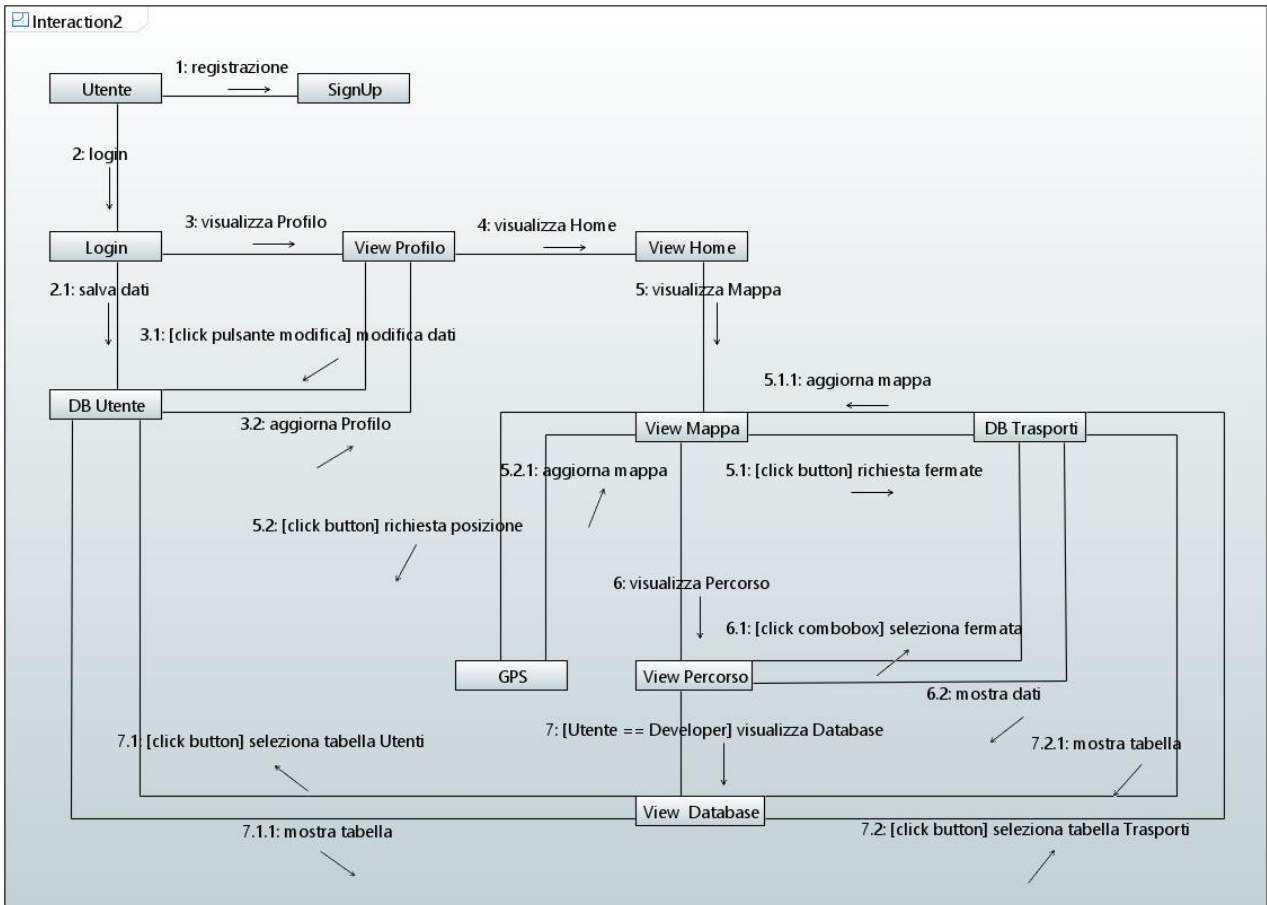
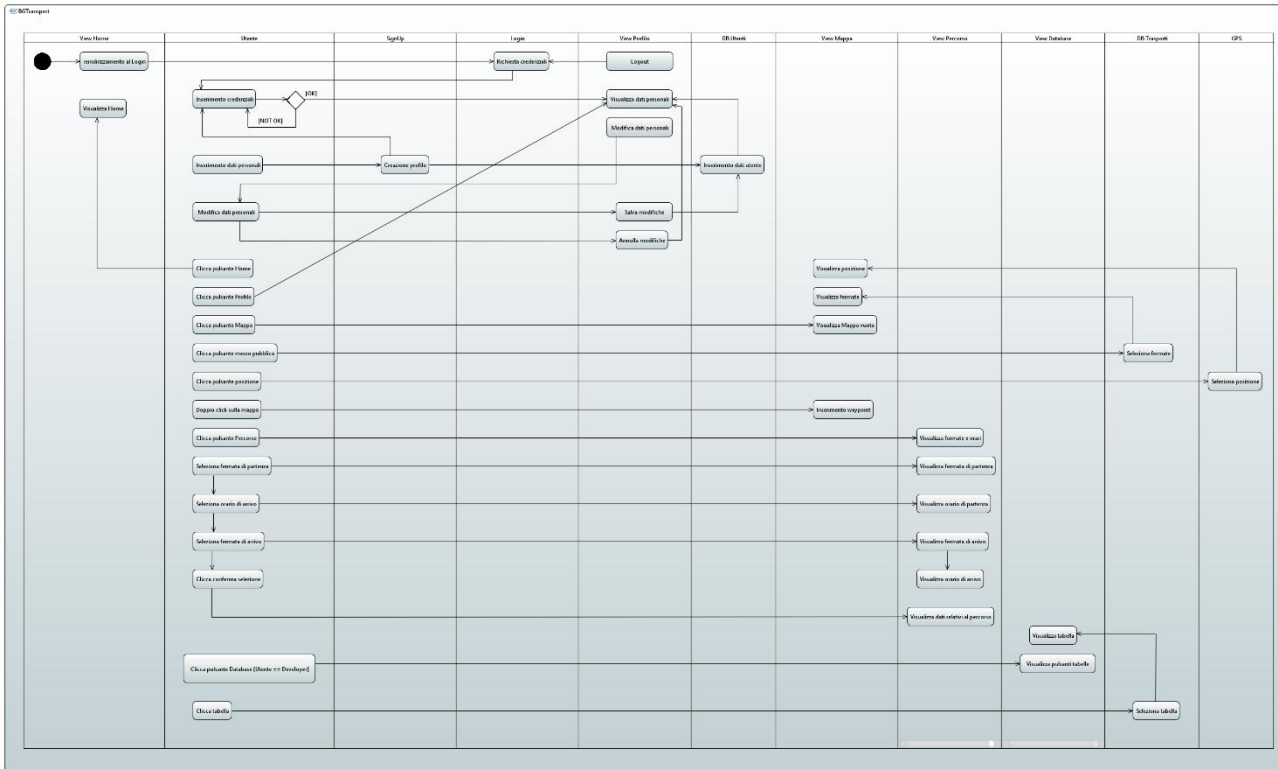
L'interfaccia dell'applicazione dovrà essere chiara e facilmente navigabile da utenti di tutte le età e competenze tecniche.

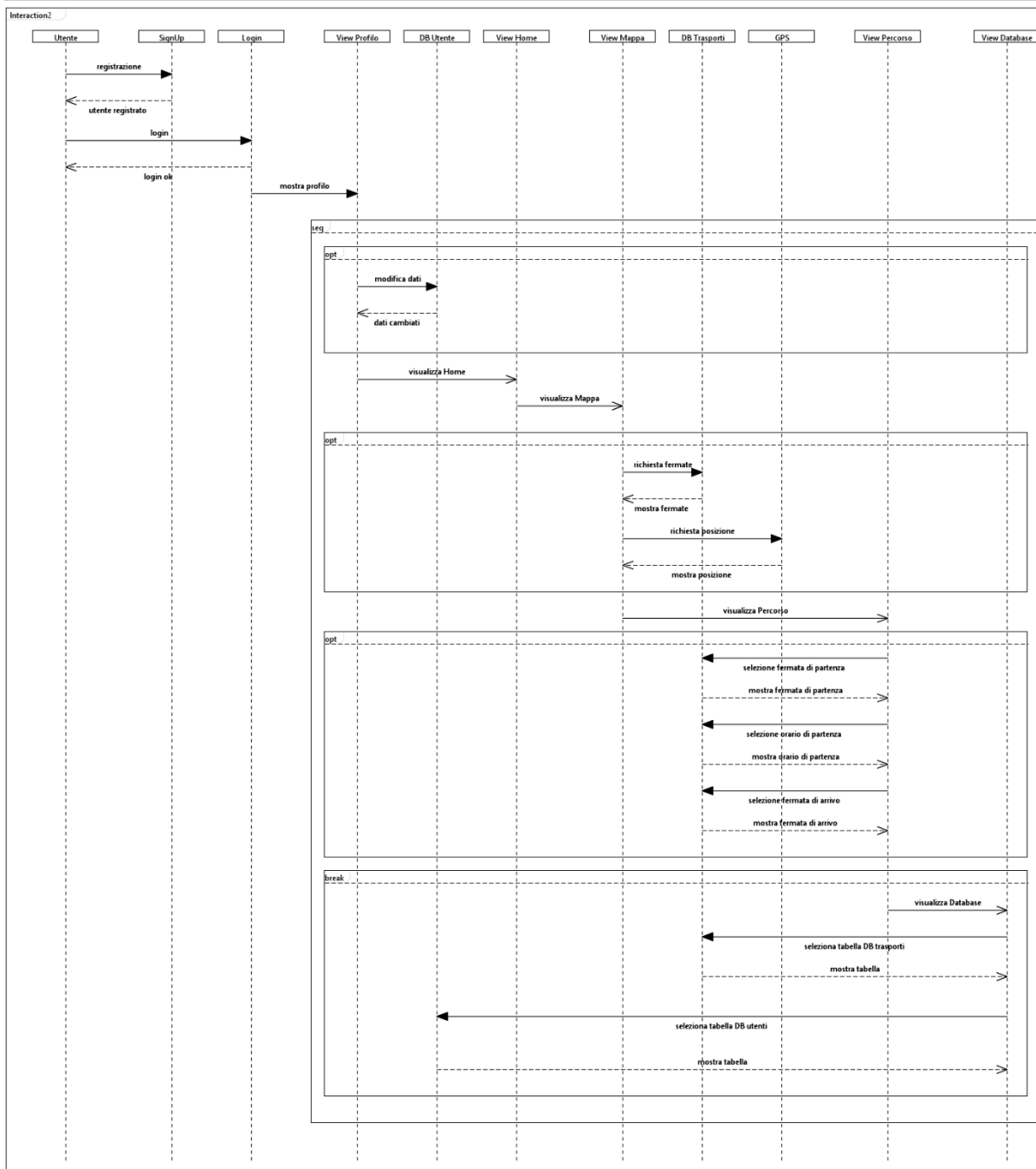
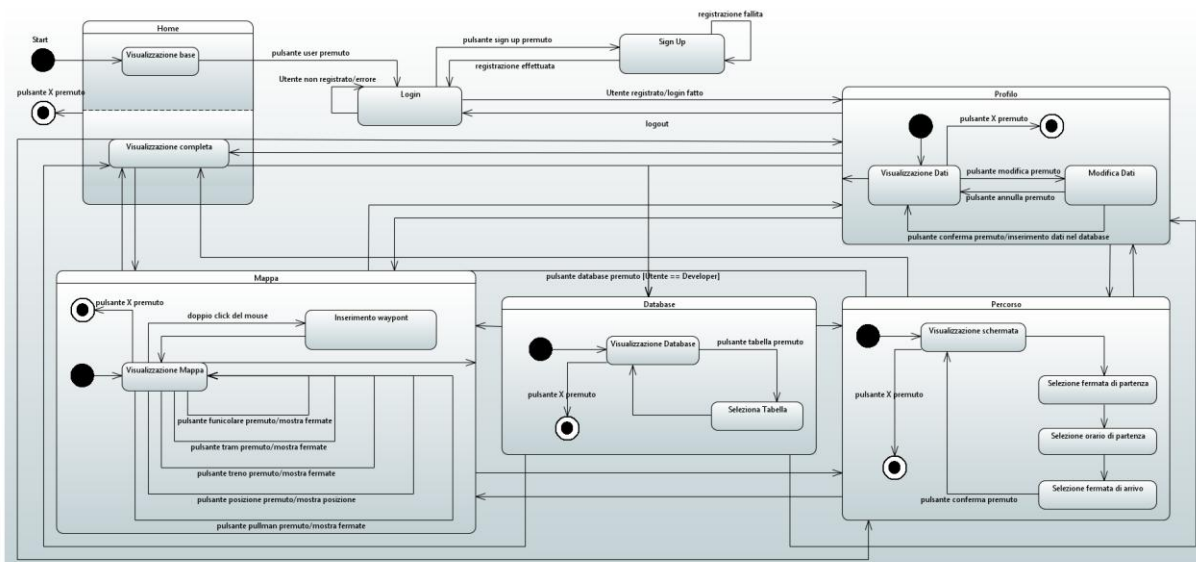
Vincoli e Limiti

- **VL-1: Connessione Internet**

L'applicazione richiede una connessione Internet attiva per caricare i dati relativi ai trasporti e la mappa in tempo reale.

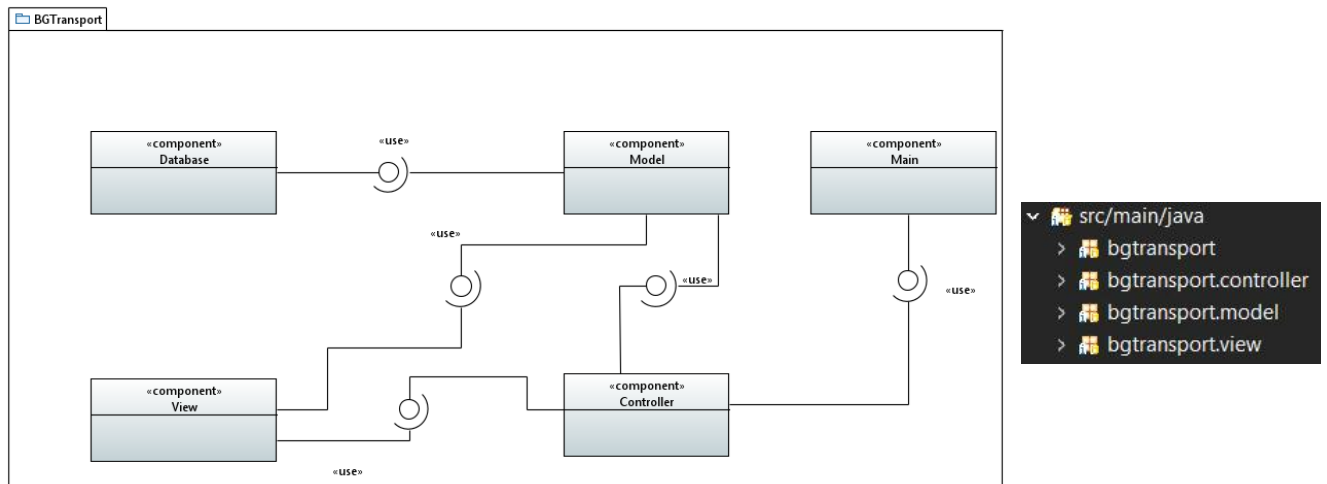
Modelling





Software Architecture

Il sistema **BGTransport** adotta un'architettura **Model-View-Controller (MVC)**, arricchita da un componente principale nel pacchetto **bgtransport** denominato **Main**. Questa scelta architetturale permette una separazione chiara delle responsabilità all'interno del sistema, favorendo una gestione più efficiente e modulare delle diverse componenti. Ogni pacchetto e classe ha compiti ben definiti, e questo approccio facilita anche la manutenzione e l'estensibilità del codice.



1. Model (bgtransport.model)

Il pacchetto **Model** è responsabile della gestione dei dati e della logica di business del sistema. Al suo interno, la classe principale è **MainModel**, che si occupa della gestione delle operazioni sui database e dell'implementazione della logica applicativa. Una delle principali funzioni di **MainModel** è l'interazione con un database **SQLite**, il quale memorizza i dati relativi ai trasporti pubblici, come le linee di autobus, le fermate, gli orari e i percorsi.

La gestione dei dati avviene tramite **JOOQ**, una libreria che fornisce un'API fluida e tipizzata per interagire con i database. Con l'utilizzo di **JOOQ**, il sistema è in grado di eseguire query SQL complesse in modo sicuro e performante. **MainModel** si occupa anche di validare i dati, assicurandosi che le informazioni estratte dal database siano corrette e coerenti con le esigenze dell'applicazione. Le operazioni comuni di gestione dei dati, come l'inserimento, l'aggiornamento e la cancellazione delle informazioni, sono centralizzate in questo pacchetto.

Inoltre, il **Model** si occupa della gestione della logica di business, come ad esempio la validazione delle ricerche da parte dell'utente, l'applicazione di filtri sui dati o la manipolazione dei dati ricevuti dal database prima che vengano visualizzati all'utente.

2. View (bgtransport.view)

Il pacchetto **View** è dedicato alla **Graphical User Interface (GUI)**, ovvero alla parte dell'applicazione che consente agli utenti di interagire con il sistema. La **View** è costituita da diverse finestre (o schermate), che vengono visualizzate in base alla sezione scelta dall'utente. Ogni finestra è progettata per offrire un'interazione chiara e intuitiva con le informazioni sui trasporti pubblici, come ad esempio la visualizzazione delle fermate, dei percorsi e degli orari.

Le finestre della **View** sono progettate per essere moduli indipendenti che possono essere caricate o nascoste a seconda delle azioni dell'utente. Ad esempio, se un utente seleziona una sezione relativa ai percorsi, verrà mostrata una finestra che mostra una mappa interattiva, i dettagli delle linee e altre informazioni pertinenti. All'interno di ciascuna finestra, l'utente può anche interagire con vari controlli, come pulsanti, campi di input, menu a tendina, ecc.

La **View** è completamente separata dalla logica che gestisce i dati. La modifica dell'interfaccia utente avviene esclusivamente attraverso il **Controller**, che invia comandi per aggiornare o modificare i contenuti visibili.

3. Controller (bgtransport.controller)

Il pacchetto **Controller** ha il compito di gestire la logica di interazione tra l'utente e il sistema. Il **MainController**, che è la classe principale di questo pacchetto, è responsabile di inizializzare l'applicazione e di coordinare l'interazione tra **Model** e **View**. Quando un utente interagisce con l'interfaccia grafica (ad esempio cliccando su un bottone o selezionando un'opzione), il **Controller** riceve l'evento e decide come rispondere.

Una delle funzioni principali del **Controller** è quella di modificare la **View** in base alle azioni dell'utente. Questo include la gestione del ridimensionamento delle finestre e dei relativi componenti, permettendo agli utenti di personalizzare l'aspetto dell'applicazione in base alle proprie preferenze. Inoltre, il **Controller** consente anche di gestire la modalità di visualizzazione del tema, offrendo due modalità principali: **chiara** e **scura**. L'utente può scegliere la modalità di visualizzazione preferita, e il **Controller** si occupa di applicare i corretti stili visivi alla **View**.

In sintesi, il **Controller** è il cuore del flusso di interazione del sistema: si occupa di tradurre le azioni dell'utente in operazioni concrete, interagendo con il **Model** per recuperare o modificare i dati, e con la **View** per riflettere i cambiamenti all'utente.

4. Main (bgtransport)

Il pacchetto **Main** contiene la classe principale **Main**, che rappresenta il punto di avvio dell'applicazione. Quando l'utente avvia il software, il sistema inizia la sua esecuzione dalla classe

Main, che si occupa di inizializzare tutti i componenti necessari per il funzionamento dell'applicazione. In particolare, **Main** crea le istanze delle classi **Model**, **View**, e **Controller**, stabilendo le connessioni tra di esse.

La classe **Main** è anche responsabile della gestione dell'avvio e del ciclo di vita dell'applicazione. Inizia il software, configura eventuali risorse necessarie (come la connessione al database), e lancia la **View** iniziale. È importante notare che la classe **Main** si occupa di fare partire il sistema, ma non contiene alcuna logica di business o di interazione diretta con l'utente. La sua funzione principale è quella di avviare l'infrastruttura dell'applicazione, mentre il **Controller** prende in carico la gestione delle operazioni quotidiane.

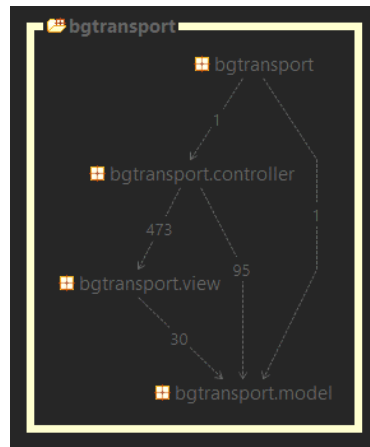
Di seguito elenchiamo le librerie esterne utilizzate nel progetto:

- **Spring Boot:**
 - spring-boot-starter (versione 3.4.0)
 - spring-boot-starter-web (versione 3.4.0)
- **Logging:**
 - logback-classic (versione 1.5.12)
 - slf4j-simple (versione 2.0.16)
 - log4j-slf4j-impl (versione 2.24.2)
- **Mappe:**
 - jaxmapviewer2 (versione 2.8)
 - openlayers (versione 6.1.0) — tramite WebJars
- **Testing:**
 - junit-jupiter-api (scope test)
 - junit-jupiter-params (scope test)
 - h2 (versione 2.3.232, scope test)
- **Database:**
 - sqlite-jdbc (versione 3.43.2.2)
- **ORM/SQL:**
 - jooq (versione 3.19.15)
 - jooq-meta (versione 3.19.15)
 - jooq-codegen (versione 3.19.15)
- **Formattazione e Parsing dei Dati:**
 - json (versione 20210307)
 - jackson-databind (versione 2.18.2)
- **UI:**
 - flatlaf (versione 2.6)
- **HTTP Client:**
 - okhttp (versione 4.11.0)

Software Design

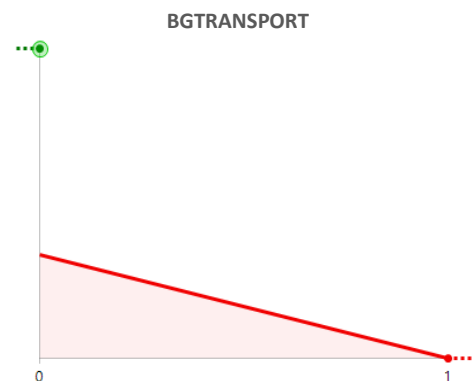
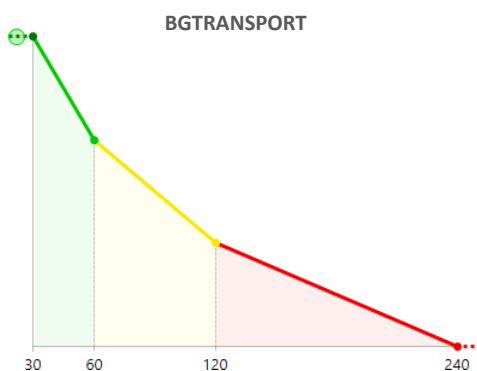
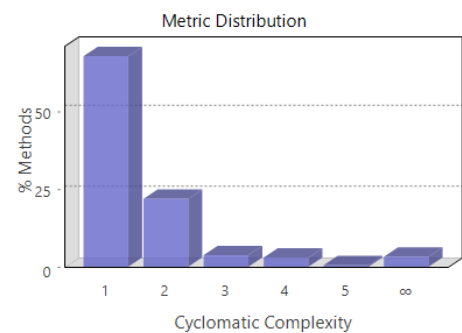
Abbiamo condotto un'analisi approfondita del software utilizzando STAN4J, uno strumento avanzato per la valutazione della qualità del codice e dell'architettura. Tale analisi si è focalizzata sull'identificazione di eventuali criticità strutturali, sul grado di coesione e accoppiamento tra i moduli, nonché sul rispetto dei principi di progettazione orientata agli oggetti.

L'analisi principale dei cicli è rappresentata dal seguente schema:

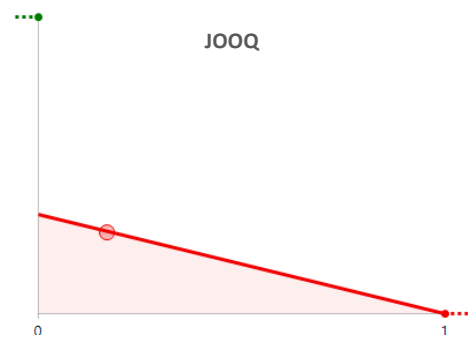


I risultati emersi hanno evidenziato alcune aree di miglioramento che possono essere affrontate per ottimizzare la manutenibilità, la leggibilità e la robustezza del software. Di seguito riportiamo i principali esiti dell'analisi:

- bgtransport:
 - Cyclomatic Complexity = 1,71
 - Fat = 5
 - Tangled = 0%



- transportation.jooq.generated:
 - Cyclomatic Complexity = 1,04
 - Fat = 2
 - Tangled = 16,67%
- user.jooq.generated:
 - Cyclomatic Complexity = 1,03
 - Fat = 2
 - Tangled = 16,67%



Design Pattern

Delegation Pattern

Nel nostro progetto, per migliorare l'architettura e la gestione del flusso di interazioni tra l'utente e l'applicazione, abbiamo adottato il **Delegation Pattern** come approccio strategico nella gestione delle **View** e dei relativi **Controller**. In particolare, abbiamo utilizzato questo pattern per garantire una separazione chiara delle responsabilità tra i vari componenti del sistema, consentendo una gestione modulare e facilmente estendibile delle funzionalità di visualizzazione.

Quando ci trovavamo a gestire una **View** in particolare, la nostra architettura prevedeva l'adozione di un **Controller** dedicato esclusivamente a quella view. Ogni controller era incaricato di orchestrare l'interazione tra la vista dell'utente e la logica sottostante, delegando la responsabilità di aggiornare la visualizzazione della view a specifici metodi del controller.

Il **Delegation Pattern** ha svolto un ruolo fondamentale nel nostro approccio. Quando una determinata interazione avveniva sulla **View** (ad esempio, l'utente compiva un'azione come l'aggiornamento di un campo o la selezione di un'opzione), la **View** non gestiva direttamente la modifica della visualizzazione o dei dati sottostanti. Invece, la **View** delegava questa responsabilità al **Controller**.

Il **Controller**, come delegato della logica di visualizzazione, riceveva l'input dalla **View** e invocava i metodi appropriati per modificare lo stato o i dati necessari, aggiornando successivamente la visualizzazione della **View** in base alle nuove informazioni.

Observer Pattern

Nel nostro progetto, abbiamo implementato l'**Observer Pattern** come soluzione per gestire gli eventi derivanti dalle interazioni dell'utente con l'interfaccia grafica, in particolare per i **pulsanti** e il **ridimensionamento della finestra**. Questo approccio ci ha permesso di separare chiaramente la logica degli eventi dalla gestione diretta degli oggetti dell'interfaccia, migliorando la modularità, la manutenibilità e la reattività del sistema.

Listener dei Pulsanti

Per la gestione delle azioni sui **pulsanti**, abbiamo utilizzato il **Listener Pattern**, una forma concreta dell'**Observer Pattern**. Ogni pulsante dell'interfaccia utente agisce come un **subject** che mantiene una lista di **observer** (listener), i quali sono notificati quando l'utente interagisce con il pulsante (ad esempio, cliccando su di esso). In questo modo, i listener sono in grado di rispondere agli eventi e di eseguire l'azione associata, come l'aggiornamento dell'interfaccia o l'esecuzione di operazioni.

In pratica, ogni pulsante è stato associato a un **listener** che, quando attivato (ad esempio da un click), esegue una serie di azioni specifiche. Grazie a questa implementazione, è stato possibile mantenere un codice chiaro e focalizzato sulla logica di presentazione, delegando la gestione degli eventi a componenti esterni in modo modulare e scalabile.

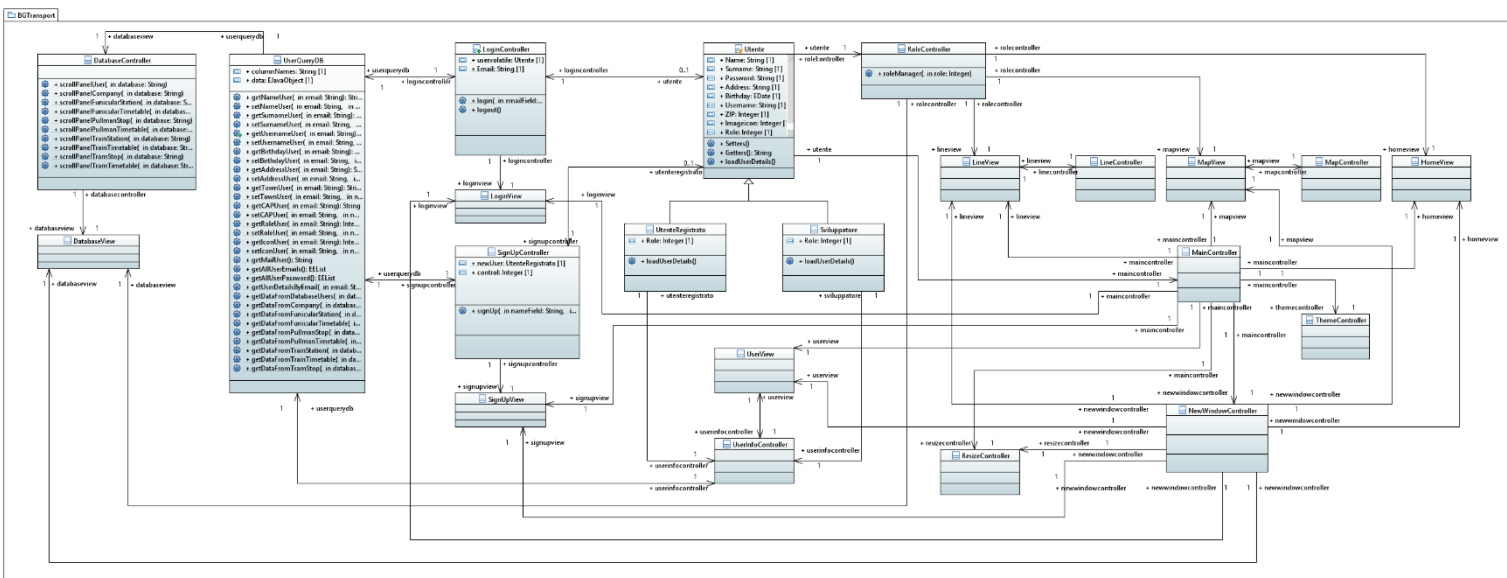
Listener del Ridimensionamento della Finestra

Analogamente, per la gestione del **ridimensionamento della finestra** quando l'utente ridimensiona la finestra, il sistema notifica automaticamente i listener registrati, che si occupano di aggiornare altri componenti dell'interfaccia o di adattare la visualizzazione in base alle nuove dimensioni.

L'uso dell'**Observer Pattern** in questo contesto ci ha consentito di separare la logica di gestione del ridimensionamento da quella di visualizzazione. Questo approccio ha migliorato l'interoperabilità tra i diversi componenti dell'interfaccia utente, garantendo una risposta dinamica e fluida alle modifiche delle dimensioni della finestra.

Class Diagram

Il diagramma delle classi presentato rappresenta l'architettura di un'applicazione sviluppata seguendo il modello **MVC** (Model-View-Controller). È importante precisare che la rappresentazione non riguarda in maniera dettagliata la componente grafica, poiché il software **Papyrus** non consente di fare riferimento diretto ai componenti di Java Swing. Per tale motivo, la struttura diagrammata include unicamente gli elementi indispensabili per descrivere il funzionamento e l'organizzazione dell'applicazione.











Software Testing

Il testing del software è stato eseguito al termine della maggior parte dello sviluppo del codice, seguendo due approcci principali:

1. Il primo approccio concerne il testing delle classi contenute nei pacchetti `bgtransport.model` e `bgtransport.controller`, realizzato tramite l'uso di JUnit. I test sono stati progettati per verificare il corretto funzionamento dei metodi associati alle operazioni con i database.
2. Il secondo approccio riguarda il testing dell'interfaccia grafica, con particolare attenzione alle classi presenti nel pacchetto `bgtransport.view`. In questo contesto, sono stati testati tutti i pulsanti e le relative sezioni ad essi collegate, verificando che ogni pulsante conducesse correttamente alla sezione prevista e che eseguisse le operazioni corrette.

In entrambi i casi, il testing ha garantito che il software fosse funzionale e conforme alle specifiche richieste.

Nella prima fase di testing, tramite l'utilizzo di EclEmma, è stata raggiunta una copertura del codice pari a circa il 75% mediante test JUnit. Tale risultato è stato influenzato dal fatto che, per la gestione dei database relativi ai mezzi di trasporto, non si è operato direttamente sul database reale, bensì su una sua simulazione. Questa scelta è stata adottata al fine di evitare modifiche involontarie ai dati reali e per ridurre il carico computazionale dovuto alla complessità del database.

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
BGTransport	 74,2 %	15.657	5.439	21.096
src/main/java	 71,5 %	13.397	5.353	18.750
bgtransport.controller	 61,3 %	3.655	2.309	5.964
bgtransport.model	 54,7 %	2.675	2.219	4.894
bgtransport.view	 89,5 %	7.053	825	7.878
bgtransport	100,0 %	14	0	14
src/test/java	 96,3 %	2.260	86	2.346
bgtransport.model	 89,8 %	680	77	757
bgtransport.controller	 99,6 %	1.569	6	1.575
bgtransport	78,6 %	11	3	14

I test JUnit hanno comunque verificato, seppur in maniera indiretta, anche la parte di interfaccia grafica (GUI) relativa all'inserimento dei dati. Tuttavia, la componente visiva della GUI è stata gestita separatamente attraverso test manuali condotti dai tester.

Software Maintenance

Per quanto riguarda l'attività di manutenzione del software, sono stati seguiti due principali criteri:

1. Il primo criterio consiste nella creazione di classi dedicate a contenere la maggior parte delle costanti. Questa scelta è stata fatta per evitare l'intasamento del codice con stringhe direttamente inserite e per facilitare agli sviluppatori l'eventuale modifica del codice in futuro, migliorando così la manutenibilità e la chiarezza del progetto.
2. Il secondo criterio prevede il ricorso ai consigli forniti da SonarLint/SonarQube, con l'obiettivo di migliorare la qualità del codice. Le operazioni di refactoring eseguite includono:
 - Modifica dei nomi delle variabili
 - Modifica dei nomi dei metodi
 - Modifica dei nomi delle classi
 - Modifica dei nomi dei pacchetti
 - Eliminazione di parti di codice non utilizzato
 - Semplificazione di parti di codice eccessivamente complesse

Inoltre, sono stati aggiunti commenti JavaDoc al fine di garantire una migliore comprensione del codice per gli sviluppatori che potrebbero doverlo mantenere o estendere in futuro.

È importante sottolineare che queste operazioni di manutenzione sono state svolte sia durante la fase di implementazione del codice, sia successivamente alla sua conclusione, con l'obiettivo di mantenere un alto standard di qualità e leggibilità del software.