Progetto di programmazione ad oggetti 2022/2023

Alessandro Bustreo Matricola: 2042383

March 28, 2023

Contents

1 Introduzione

1.1 Abstract

RTM	(*R*ailway *T*ransport *M*anager) è un gestionale che consente di gestire i mezzi ferroviari, dove è permessa: la creazione, la modifica, l'eliminazione e la visualizzazione dei vari tipi di convogli. I mezzi ferroviari sono suddivisi in due categorie principali: trainabili e da trazione. La categoria dei trainabili comprende carri merci e vagoni passeggeri, mentre quella da trazione comprende le locomotive elettriche.

Durante tutto lo sviluppo del progetto si è cercato di astrarre il più possibile le varie implementazioni delle classi, nell'ottica di rendere il progetto facile da modificare e da mantenere. Nel pratico si è fatto utilizzo: del polimorfismo, della programmazione generica (template), dei funtori e delle lambda expression.

1.2 Methods

Per sviluppare l'intero progetto è stato usato *Visual Studio Code* come editor, durante lo sviluppo si è cercato di seguire il più possibile le *GCC Coding Conventions* per la scrittura del codice. Per facilitare il lavoro di indentare correttamente e mantenere ordinati i file si è usato *clang-format* con coding style *GNU*. Si è utilizzato molto *gdb* per fare il debug di alcune porzioni specifiche di codice, e per analizzare i *core dump* e capire la natura dei crash più ostici che si sono presentati durante le fasi di debug. Oltre a questo si è fatto grande utilizzo dei *sanitizer* [1] per verificare: undefine beahviour, memory leak e accessi/operazioni illegali in memoria. Per la consegna sono state disabilitate tutte le impostazioni di debug, ed è stata impostata la configurazione di *release*.

Ambiente di sviluppo

- **OS:** Fedora Linux 37 (Workstation Edition);
- **Compilatore:** g++ version 12.2.1 20221121 (Red Hat 12.2.1-4);
- **Qt:** Qt version 6.4.1, QMake version 3.1;
- **Standard C++:** 17.
- **GDB:** Fedora Linux 12.1-6.fc37;
- **VM:** per i test è stata usata la macchina virtuale fornita dal corso.

[1] I sanitezer utilizzati sono i seguenti: AddressSanitizer, LeakSanitizer e UBSan.

2 Implementazione

Il progetto è formato da due parti distinte: il modello logico e l'interfaccia grafica.

2.1 Modello logico e gerarchia

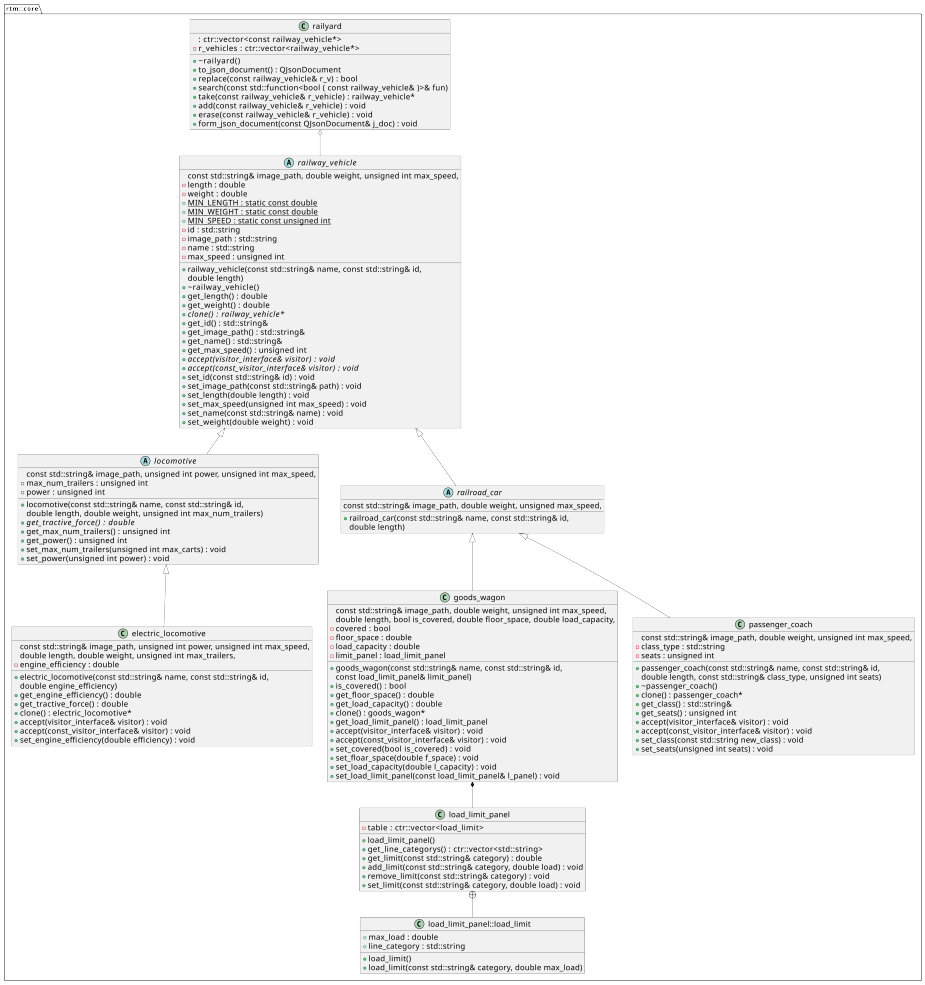

Figure 1: Gerarchia del modello logico.

Il modello logico è incapsulato all'intero del namespace `rtm::core`. La gerarchia è costruita su un'unica base astratta `railway_vehicle`, dalla quale derivano i due marco-gruppi: i vagoni trainabili e i mezzi da trazione, rispettivamente `railroad_vehicle` e `locomotive`, anche queste ultime due classi sono astratte e inoltre sono raggruppate in un unico file `vehicle_types.h`[2]. Da `railroad_vehicle` e `locomotive` derivano le classi concrete. Le varie classi

[2]Il codice delle due classi è così breve che si è deciso di unirle in un unico file. Il loro principale scopo è quello di permette di raggruppare i vari mezzi.

della gerarchia implementano dei campi pubblici, statici e constati con i valori minimi acconsenti per alcune variabili. Alcune classi devono rispettare delle invariati, ad esempio, un certo valore non deve essere sotto una certa soglia altrimenti l'oggetto non è valido, per garantire queste invarianti è stato introdotto un sistema di eccezioni per i metodi *setter*, così da controllare che i valori inseriti siano legali. Per le eccezioni è stata utilizzata la classe `std::invalid_agrument` presente nella libreria `stdexcept`, ogni eccezione sollevata riporta con se una stringa con le informazione dettagliate dell'errore. Appartengono al modello logico anche le classi: `load_limit_table`, `railyard` e `converter`, che però non fanno parte della gerarchia. Tutti i mezzi che sono destinati al trasporto merce sono in relazione *has a* con `load_limit_table`, che permette di rappresentare le tabelle internazionali per il limite massimo di carico per asse. `converter` serve a convertire gli oggetti della gerarchia in `QJsonObject` per il salvataggio dei dati su file e viceversa per lettura, maggiore dettagli alla paragrafo 3. Mentre `railyard` è il contenitore che tiene tutti i tipi di mezzi ferroviari, inoltre mantiene l'invariante di non possedere due mezzi con lo stesso id. L'id di ogni mezzo viene usato come identificatore univoco dell'oggetto, una spiegazione più approfondita della interfaccia di `railyard` viene data nel paragrafo 2.4.

2.2 Interfaccia grafica

La finestra principale: `main_window` che è derivata da `QMainWindow` mostra: il menu, la toolbar e un unico QWidget. L'idea principale dietro alla GUI è che: `main_window` gestisce il minimo dispensabile, e per questo mostra soltanto un `QWidget` che dovrà occuparsi di gestire gli elementi da mostrare. In questo modo è possibile sviluppare le varie interfacce grafiche come dei moduli separati e caricarle/mostrarle nella finestra principale quando necessario. Se bisogna mostrare una certa vista la `main_window` prepara tutti le risorse necessarie a quella vista, elimina il QWidget precedente, e in fine crea la vista richiesta e le passa tutte le risorse necessarie, da questo punto in poi il "controllo" sarà completamente della vista che gestirà come più ritiene opportuno i vari elementi da visualizzare. Ad esempio: se voglio chiamare il widget per modificare un mezzo ferroviario (`railway_vehicle_editor`) questo è quello che succede:

- `main_window` riceve l'oggetto da modificare o crea l'oggetto;
- elimina la vista attutale;
- crea l'editor, connette tutti i segnali e gli passa il veicolo da modificare;
- mostra l'editor.

Da cui in poi la `main_window` non farà più niente altro fino a quando l'editor non emetterà un segnale di `close(bool)`, ricevuto quel segnale la finestra principale distrugge l'editor e reimposta la vista di default. Questo permette in futuro di creare altri widget senza apportare pesanti modifiche alla finestra principale.

2.3 Polimorfismo

Il polimorfismo è stato utilizzato in tutto il progetto. Sia nel modello logico e sia nella GUI viene usato molto il *design pattern* ***visitor***. `railway_vehicle`

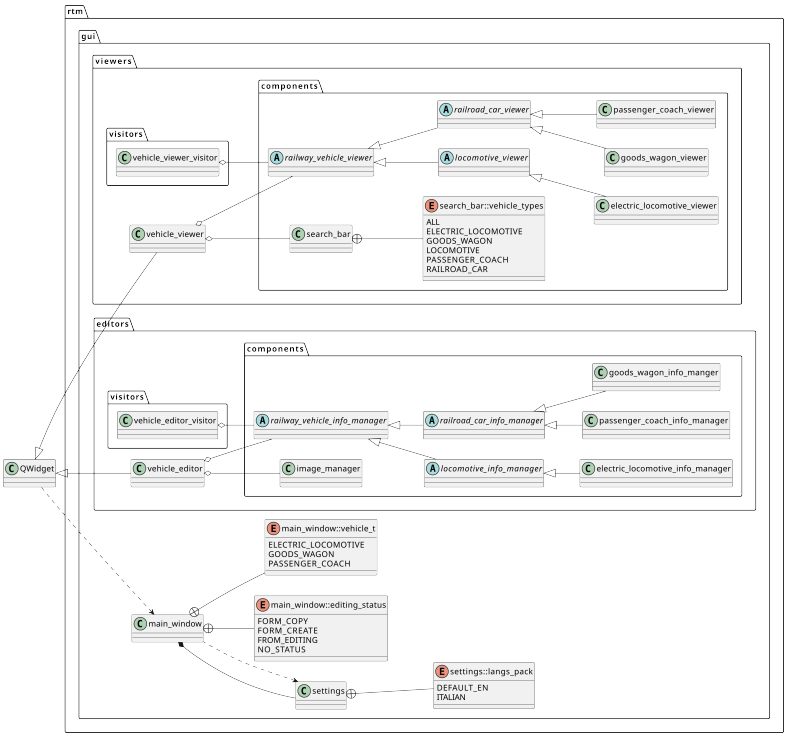

Figure 2: Struttura della gui.

possiede un metodo virtuale puro che permette di accettare un generico visitor, più precisamente: `visitor_interface` e un `const_visitor_interface` che sono una base astratta per tutti i visitor del progetto. Da il visitor generico derivano i visitor: per il converter, per l'editor e per il viewer. Oltre al visitor l'utilizzo del polimorfismo viene fatto anche in `locomotive`, il quale implementa il metodo puro virtuale `get_tractive_force()` che permette di calcolare la forza di trazione della locomotiva, ovviamente in base al tipo di locomotiva ci sarà una formula e delle variabili diverse per calcolare tale forza. Inoltre per tutti gli oggetti della gerarchia implementano il metodo virtuale `clone()` che permette di clonare l'oggetto. Nell'interfaccia grafica il polimorfismo è stato utilizzato per le varie *componenti*[3] dell'editor, le componenti sono in relazione *is a* tra loro e ognuna delle quali si occupa di gestire delle informazioni specifiche, alla base c'è un metodo virtuale che permette gestire la validità dei dati inseriti, ovviamente tale metodo può cambiare per ogni tipo di mezzo e dunque è stato segnato *virtual*, allo stesso modo c'è un metodo virtuale per il salvataggio dei dati.

[3]È il nome che ho dato al namespace di tutti i QWidget che rappresentano un sezione dell'interfaccia grafica dell'editor. In generale ho utilizzato questo nome per rifermi a tali widget. Anche il viewer ha le sue componenti.

2.4 Contenitore - *Vector & railyard*

Come contenitore ho deciso di utilizzare il vettore[4]. `Vector` si trova dentro al namespace `ctr` (container), ed è completamente separato dal qualsiasi altra parte del progetto in modo da permettere la massima portabilità sul altri progetti. `Vector` è templetizzato in modo da renderlo utilizzabile per qualsiasi necessità, nella implementazione del vettore ho cercato di seguire il più possibile il comportamento di `std::vector`, mantenendolo però solo con i metodi essenziali (insert, erase, push_back e altri). Il vettore mette a disposizione anche `vector::iterator` e `vector::const_iterator`, inoltre si è reso disponibile un convertitore indotto da iterator a const_iterator. Ho deciso di implementare il const_iterator utilizzando **derivazione privata** per le seguente ragioni:

- const_iterator **è un** iterator tecnicamente, ma la relazione di sub-typing non ha senso in questo contesto;
- manutenibilità del codice, const_iterator richiama le funzioni di iterator, se è necessario fare delle modifiche al comportamento/implementazione degli iteratori del vettore, basta modificare soltanto iterator, perché const_iterator chiamerà le funzioni di iterator. Dunque c'è un solo punto del codice da modificare;
- evita la scrittura di 2 volte dello stesso codice.

Un aspetto negativo di questa scelta è la perdita di efficienza, infatti chiamare un funzione di const_iterator comporta a sua volta una chiamata a iterator. Inoltre se istanzio un const_iterator il compilatore istanzierà anche la parte di iterator dato che sono dipendenti. Ovviamente nel caso d'uso del progetto ciò è irrilevante.

`Railyard` è il contenitore specializzato per i veicoli ferroviari, utilizza `vector` per gestire i vari mezzi, la sua interfaccia fornisce dei metodi per: aggiungere, eliminare, estrarre e cercare oggetti. Il metodo di ricerca accetta: o un funtore, o un lamda expression come argomento, in questo modo permette al programmatore di poter scrivere le proprie funzioni di ricerca senza dover per forza creare nuovi metodi all'interno della classe[5]. La classe, inoltre, mette a disposizione dei funtori per le ricerche di base: ricerca per id[6], ricerca per nome, ricerca per tipo e uno che permette di ottenere tutti i mezzi.

3 Gestione dei dati

Il programma gestisce due tipi di dati: le impostazioni del programma stesso e i progetti. I progetti sono gestisti da `converter` che permette la traduzione da QJsonObject a oggetti della gerarchia e viceversa, mentre `settings` (che è la classe che tiene tutte le informazioni riguardo le impostazioni), ha i suoi

[4]Questo è stato deciso in base a quanto visto a lezione, siccome il programma non fa quasi mai inserimenti "nel mezzo", ma sempre delle push_back, ho deciso che il vector era la struttura adatta al progetto.

[5]Questa scelta si è rivelata molto utile per il widget che permette di cercare i vari mezzi dentro il programma.

[6]Di funtori per la ricerca per id c'è ne sono due: uno che cerca proprio l'id identico alla stringa passata al funtore, un altro che cerca tutti gli id che assomigliano alla stringa passata al funtore.

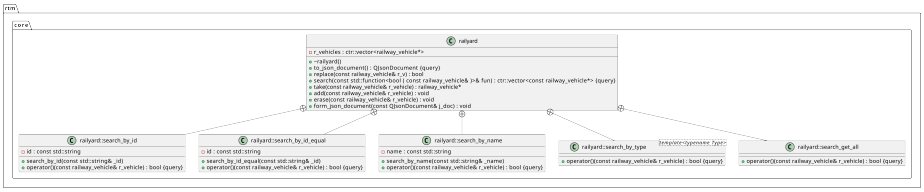

Figure 3: Struttura di `railyard`.

metodi per la scrittura e la lettura dei QJsonObject. I dati dei progetti vengono salvati in formato JSON e si possono salvare in qualunque posizione che l'utente desidera, mentre i dati di `settings` vengono salvati nella cartella dove ci sono sono tutte le configurazioni[7]. `converter` implementa anche un campo pubblico enum per tornare i vari tipi di errori che si riscontrano durante la lettura dei dati. Mentre `setting` ritorna una semplice boolean per notificare che la lettura dei dati (oppure la scrittura) è avvenuta con successo.

4 Funzionalità dell'applicativo

Di seguito le funzionalità principali di *RTM* per quanto riguarda l'esperienza dell'utente finale.

Funzionalità di base:

- creazione, eliminazione e visualizzazione di oggetti appartenenti alla gerarchia;
- salvataggio dei dati in un formato leggibile (JSON);
- semplice gestione degli errori (formati non validi o azioni illegali).

Funzionalità extra:

- toolbar della finestra principale dinamica in base al contesto;
- ricerca dei mezzi per attribuiti (id, nome e tipologia di mezzo);
- memorizzazione dei file aperti di recente;
- memorizzazione e gestione delle impostazioni del programma;
- possibilità di cambiare lingua: tra Italiano e Inglese (che è di default);
- nell'editor, controllo dei dati inseriti dall'utente con disabilitazione del pulsante di salvataggio e segnalazione dell'errore in caso di dati non validi;
- shortcut per le azioni principali: nuovo progetto, salva progetto e apri progetto;
- menu con tutti i progetti recentemente aperti, col la possibilità di pulire la cronologia.

[7] Su GNU/Linux solitamente è sotto /home/*username*/.config/, la scelta di dove salvare i dati viene fatta dal metodo statico `QStandardPaths::writableLocation (QStandardPaths::ConfigLocation)`.

5 Tempi di sviluppo e progettazione

Descrizione attività	**Ore** (h)
Creazione, raccolta informazioni per la creazione del progetto	5,00
Setup struttura del progetto	0,50
Sviluppo del container vector	9,17
Sviluppo della parte core (modello logico)	14,00
Imparare ad usare Qt e lettura della documentazione	6,53
Sviluppo della GUI	22,15
Debug, revisione del codice, creazione degli esempi	6,80
Implementazione del supporto per la traduzione, traduzione del programma	4,00
Totale ore	68,15

La separazione delle ore per attività non è netta, ma indicativa. Le ore di sviluppo previste dalla specifica del progetto sono state superate, questo per vari motivi:

- sviluppare le funzionalità extra ha richiesto molto tempo, ad esempio per la `search_bar` ho utilizzato la libreria `functional` per creare una funzione di ricerca che viene costruita a run-time in base alle impostazioni scelte dall'utente[8], lo studio della libreria `functional` e i vari test per caprine il suo funzionamento hanno richiesto tempo;
- la funzionalità di traduzione è stata aggiunta come extra a progetto finito, a richiesto molto tempo;

6 Note extra

- Nel cartella docs c'è ulteriore UML sulla GUI in caso fosse necessario una panoramica più dettagliata del progetto;
- i file di esempio che (volutamente) non funzionano sono 3: uno perché ha una sintassi JSON errata, gli altri due con dei dati non validi per il programma. Gli errori vengono mostrati sai attraverso la GUI sia in console con maggiore dettagli;
- il progetto sembra avere troppe cartelle e namespace, questo perché nell'idea originale doveva implementare un sistema per unire i vari vagoni e ottenere così i treni. Alla fine per motivi di tempi ho deciso di non implementare quella parte del progetto, ma ho comunque lasciato la struttura originale nella prospettiva che in futuro quelle funzionalità possano essere aggiunte;
- la toolbar è disabilitata finché non si apre un progetto.
- i punti di base per la sufficienza del progetto sono stati sviluppati in circa 50 ore.

[8] Si veda *src/views/viewer/componets/search_ bar.(h/cpp)*, il codice è commentano con la spiegazione del suo funzionamento.