



City Sharing

PROGETTO LIBERTY DI PROGRAMMAZIONE AD OGGETTI

ANNO ACCADEMICO 2019/2020

Relazione scritta da

Masevski Martin 1123062

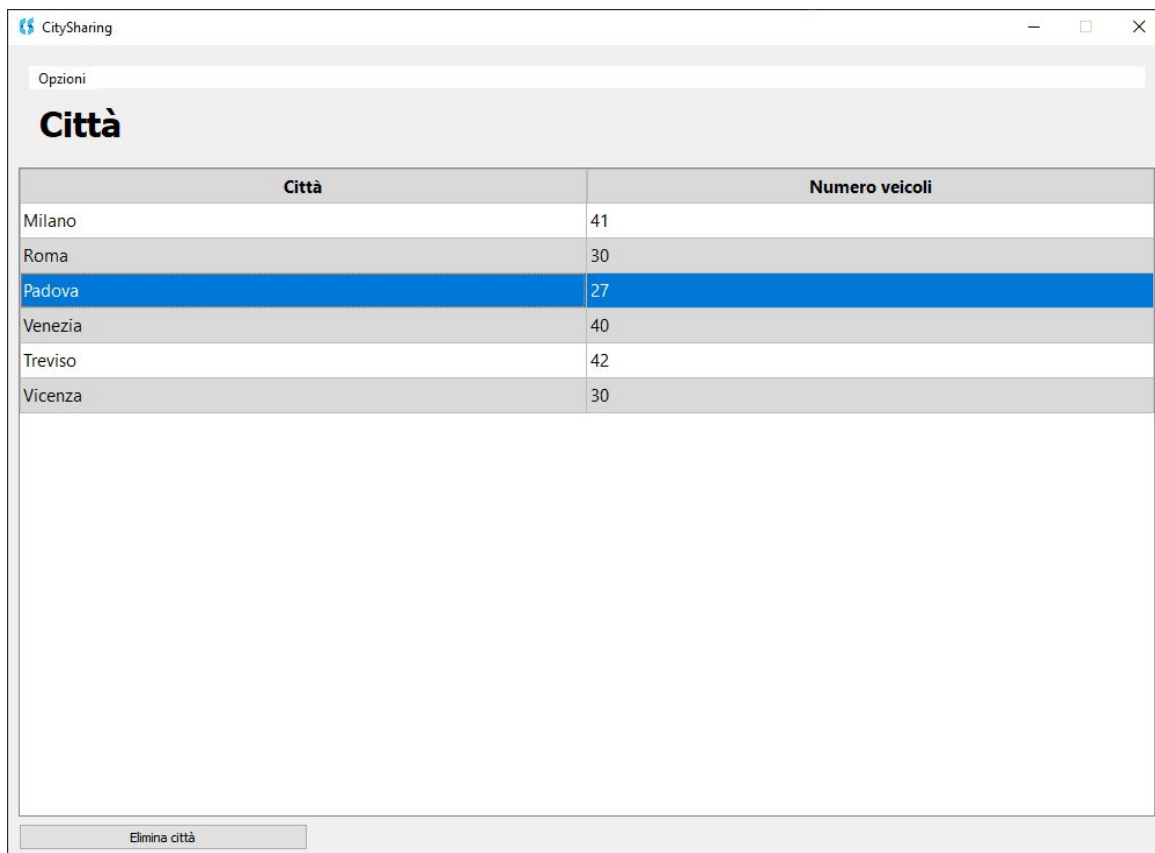
Membri Gruppo:

Brescanzin Lorenzo 1220015

Masevski Martin 1123062

Indice

1. City Sharing	3
1.1 Introduzione	3
1.2 Cosa permette di fare	3
1.3 Funzionalità	4
2. La Gerarchia	5
3. Descrizione dell'uso di chiamate polimorfe	5
4. Contenitore	6
5. Descrizione di eventuali file usati per input/output	6
6. Manuale utente della GUI	6
7. Sviluppo progetto	6
7.1 Strumenti esterni	6
7.2 Istruzioni per compilazione ed esecuzione	7
7.3 Progetto sviluppato e testato sui seguenti sistemi	7
8. Ore effettivamente richieste	8
8.1 Suddivisione lavoro di gruppo	8
8.2 Suddivisione delle ore	8



1. City Sharing

1.1 Introduzione

City Sharing è un gestionale di veicoli per la mobilità urbana. È stato sviluppato per l'azienda che offre il servizio, e quindi non prende in considerazione la parte utente e di come questo si interfaccia per usare i veicoli, ma bensì offre all'azienda degli strumenti per gestire le varie flotte. Il codice è stato sviluppato in modo da permettere l'estendibilità con la possibilità di aggiungere nuovi tipi di motore e/o nuovi tipi di veicoli. Per aggiungere la logica che gestisce l'utente non sarà necessaria la riscrittura del modello attuale, ma basterà aggiungere la propria implementazione a parte e al massimo scrivere nuovi metodi nel nostro modello.

1.2 Cosa permette di fare

Al primo avvio ci troviamo di fronte ad una tabella che raccoglie tutte le città in cui l'azienda offre il servizio e possiamo vedere subito quanti veicoli sono presenti in ogni città. Abbiamo la possibilità di fare doppio click su una qualsiasi delle righe della tabella per aprire la città corrispondente ed ottenere informazioni più dettagliate dei veicoli presenti e del loro stato.








In questa seconda schermata visualizziamo una tabella con tutti i veicoli che appartengono alla flotta della città scelta. Possiamo vedere informazioni generali come: la tipologia di veicolo, la sua targa, lo stato in cui si trova, se è stata richiesta assistenza da parte di un possibile utente (simulato solo attraverso il DataBase), se è in riserva e ha poca autonomia, il fattore green che classifica i veicoli per il loro livello di inquinamento (con scala da -10 a +10 secondo una formula inventata da noi) e il fattore di utilizzo che classifica i veicoli a secondo del loro uso (con scala da 1 a 10 con formula inventata da noi). Questa schermata permette di ottenere delle informazioni molto generali a colpo d'occhio, che possono essere utili ad un eventuale operatore che controlla le flotte. La terza colonna (Stato) permette anche l'ordinamento dei veicoli che vengono raggruppati in base al loro stato.

CitySharing

Opzioni

Flotta di Padova

Back

Tipologia	Targa	Stato	Serve assistenza	In riserva	Autonomia	Fattore green	Fattore utilizzo
	JO500EP	 Prenotato	 Problema	 No	8.50 km	-10	6/10
	IM296GG	 Prenotato	 Tutto ok	 No	2019.00 km	-1	10/10
	938MOO	 Prenotato	 Tutto ok	 No	65.00 km	6	10/10
	GM900FU	 Prenotato	 Tutto ok	 No	1974.71 km	-2	4/10
	BNS7A	 Libero	 Tutto ok	 No	52.38 km	7	5/10
	KI146VZ	 Libero	 Problema	 No	7.71 km	-10	4/10
	GK72459	 Libero	 Tutto ok	 No	16.36 km	0	8/10
	9V4MV	 Libero	 Tutto ok	 No	79.08 km	7	4/10
	DK42290	 Libero	 Tutto ok	 No	7.73 km	-2	5/10

1.3 Funzionalità

Questa è la terza schermata e ci permette di avere una visuale molto dettagliata su un determinato veicolo. Inoltre ci permette di interagire con alcune funzionalità come il cambio città (per trasferire il veicolo da una flotta all'altra) oppure la rimozione diretta del veicolo. Nel nostro caso abbiamo optato per una rimozione completa, ma si potrebbe anche implementare una rimozione "soft" che sposta il veicolo in una flotta di veicolo rimossi, così da non perdere un eventuale storico che potrebbe essere collegato al veicolo stesso.

Oltre a queste funzioni, l'operatore che usa il nostro programma potrebbe anche decidere di marcare il veicolo con un apposito checkbox e metterlo in uno stato di manutenzione se il veicolo non è occupato o prenotato, altrimenti il checkbox è disabilitato.

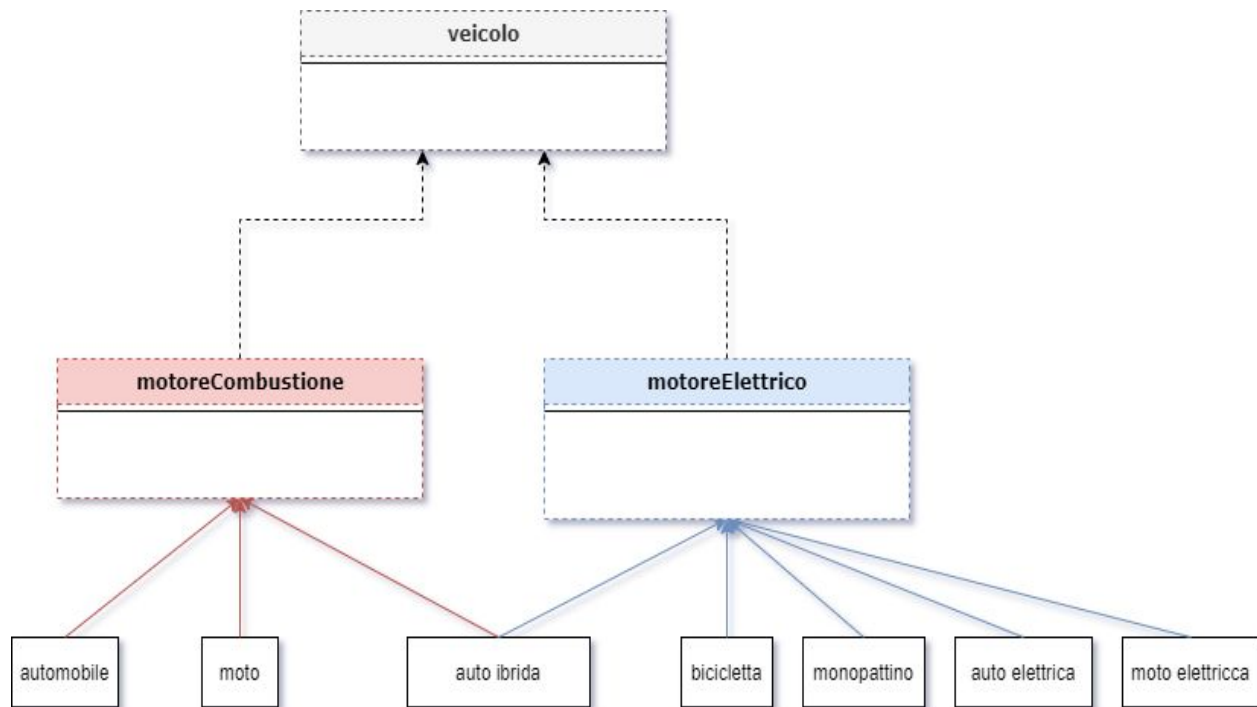
Un'altra funzionalità che offre il nostro programma è presente nella prima schermata e ci da la possibilità di eliminare completamente una città con tutti i suoi veicoli.

Ovviamente è sempre possibile andare avanti e indietro tra le schermate con l'apposito bottone "back". Ci siamo immaginati uno scenario di utilizzo in cui un operatore doveva gestire più flotte contemporaneamente e quindi abbiamo deciso di bloccare il resize della finestra per poter avere tutte le scritte e i bottoni nella stessa posizione per ogni istanza del programma aperta, così da evitare errori dovuti a pressioni accidentali.

The screenshot shows a web application window titled "CitySharing". The main content area is titled "Opzioni" and "Veicolo BNS7A". It displays various vehicle details in two columns. On the right, there are three buttons: "Back", "Cambia città", and "Rimuovi dalla flotta". There is also a checkbox for "imposta in manutenzione".

Veicolo BNS7A	
Posizione: 0.6626,-7.9781	Stato: ● libero
Chilometraggio: 9 km	Carica batteria: 100%
Consumo al km: 9	Capacità massima batteria: 471.44Ah
Numero posti: 1	Velocità di carica supportata: Veloce
Ingombro: 1 su 5	In carica: 🔌
Numero usi: 25	Velocità colonnina: 🚲 Low Public
Tempo in servizio: 244510 min	Tempo di carica rimanente: 2 min
Numero guasti: 1	
Fattore green: 7	
Fattore utilizzo: 5 su 10	
Autonomia: 52.38 km	
In riserva: 🔌 No	

2. La Gerarchia



La gerarchia prevede una classe base astratta **veicolo** che definisce le caratteristiche base che tutti i nostri veicoli hanno, definisce alcune funzioni virtuali pure che dovranno essere implementate dalle classi derivate.

Abbiamo 2 classi astratte derivate direttamente da **veicolo** che implementano le sue funzioni virtuali, rappresentano le 2 tipologie di motore che i nostri veicoli possono avere e aggiungono delle proprietà tipiche del loro motore. (rimangono astratte per via del distruttore virtuale puro)

Infine abbiamo 7 classi concrete che derivano in base al tipo di motore con il quale sono equipaggiate, da notare che le auto ibride derivano da entrambi i motori (chiusura a diamante).

3. Descrizione dell'uso di chiamate polimorfe

I metodi comuni a tutte le classi concrete sono quelli dichiarati nella classe base e sono:

`virtual double consumoKm() const;`

`virtual double autonomia() const;`

`virtual short int fattoreGreen() const;`

`virtual void checkRiserva();`

Entrambi i tipi di motore definiscono questi metodi con la propria implementazione e tra le classi concrete solo **AutoIbrida** fa l'override per poter gestire la presenza di 2 motori diversi.

In **AutoIbrida** per `consumoKm()` e `autonomia()` si richiamano implicitamente i 2 metodi presenti nelle classi motore da cui deriva e si fa la loro somma, per `fattoreGreen()` cambia leggermente perchè la somma parte da +4 (inventato da noi) e poi controlla se si superano i limiti -10 o +10, altrimenti restituisce la somma. Infine per `checkRiserva()` ritorna un valore booleano se e solo se almeno uno dei due tipi di motore è in riserva. (se `autonomia` è minore del 30%)

Abbiamo scelto di implementare i metodi polimorfi direttamente dentro alle 2 classi motore per risparmiare tempo, ma ovviamente ogni classe concreta può avere la sua variante.

4. Contenitore

Abbiamo scelto di definire il nostro contenitore **Array** imitando il comportamento di `std::vector` perchè ci serviva l'accesso in tempo costante ai suoi elementi. Lo abbiamo templatizzato per poterlo sfruttare, in futuro, anche al di fuori degli oggetti della nostra gerarchia. È dotato di tutti i metodi necessari e sfrutta anche gli iteratori.

5. Descrizione di eventuali file usati per input/output

Avendo un gruppo composto da 2 membri non avevamo il vincolo obbligatorio di implementare un sistema di salvataggio e caricamento dei dati. Nonostante questo abbiamo deciso di scrivere un sistema molto semplice per popolare il nostro programma, per evitare di dover scrivere tutto direttamente nel codice. Abbiamo usato un file di testo (**Citta.txt**) per caricare le città e un altro file di testo (**Veicolo.txt**) per caricare i veicoli. I dati presenti nel nostro Database sono stati generati casualmente, cercando sempre di restare in un range di valori realistici, per simulare un possibile scenario di utilizzo. Ho testato personalmente il programma con un Database di 100.000 veicoli e funziona perfettamente senza rallentamenti, solo l'apertura della tabella delle flotte ci mette circa 5 secondi ad essere renderizzata.

Il programma funziona anche senza la presenza del Database, ma ovviamente bisogna popolarlo con le apposite funzioni “**aggiungi città**” e “**aggiungi veicolo**” presenti sotto il menu “**opzioni**”.

Non è stato implementato nessun salvataggio dei dati.

6. Manuale utente della GUI

La GUI è molto autoesplicativa e semplice. Gli unici dubbi che potrebbero nascere riguardano le varie icone utilizzate per rappresentare alcune proprietà e stati dei veicoli, ma per ovviare a questo abbiamo inserito sempre un testo.

Per passare da una pagina all'altra bisogna usare il doppio click sulle righe della tabella e il tasto back per tornare in dietro.

Per eliminare una città bisogna selezionarla con un click dalla tabella in prima pagina e poi usare il pulsante in basso “**Elimina città**”.

Per aggiungere una nuova città o un nuovo veicolo basta usare la menu bar sotto la voce “**opzioni**”. (NB quando si aggiunge una nuova città è richiesta l'aggiunta immediata di un veicolo dentro ad essa, per nostra scelta, ovviamente possono anche esistere città vuote se si eliminano tutti i veicoli)

L'inserimento dei dati durante l'aggiunta di una nuova città o di un nuovo veicolo sono gestiti da alcune espressioni regolari che limitano ciò che l'operatore può scrivere, non sono molto rigide e permettono una certa flessibilità perché il target di utenti che andrà ad usare il programma è abbastanza esperto e sa cosa scrivere.

7. Sviluppo progetto

7.1 Strumenti esterni

Per la creazione di questo progetto abbiamo usato molti strumenti esterni che ci hanno facilitato lo sviluppo e la progettazione.

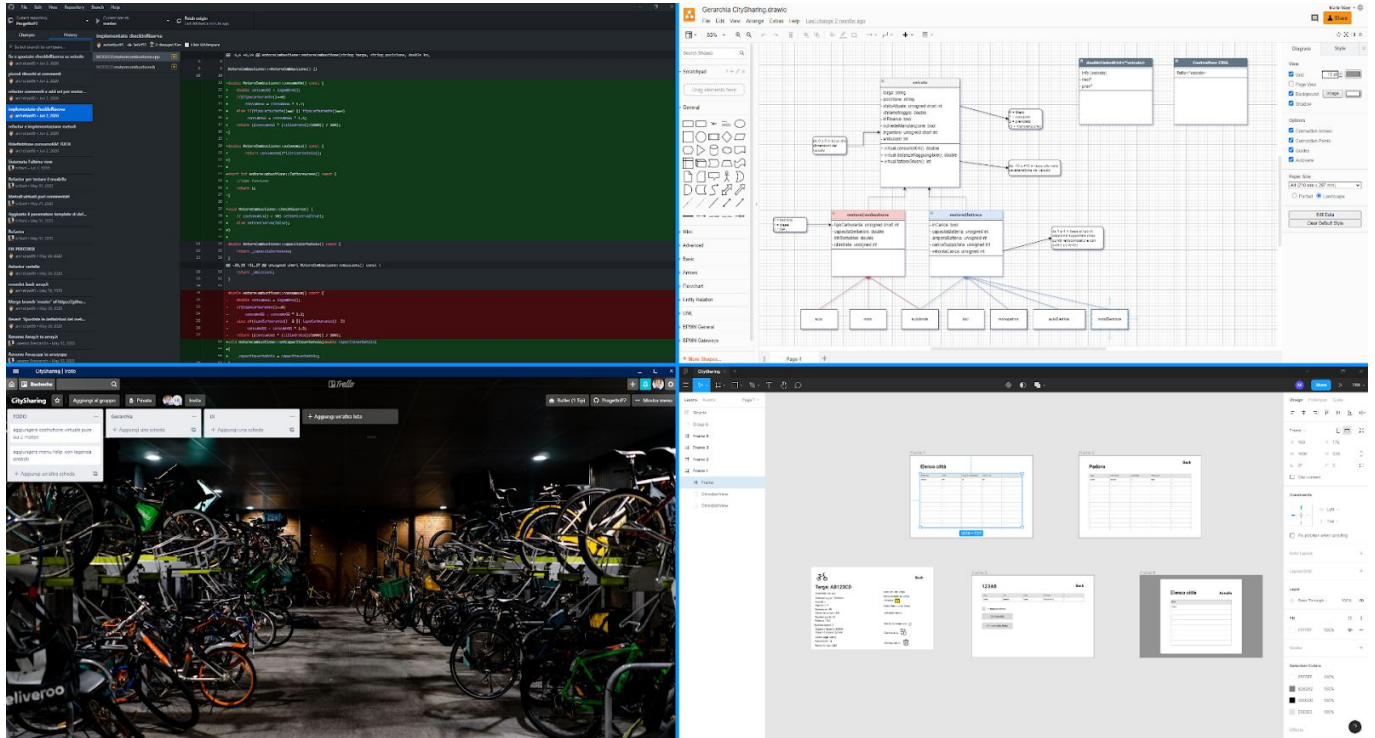
Primo tra tutti è **GitHub** che ci ha permesso di condividere, collaborare ed avere maggiore controllo sul codice scritto e sulle modifiche apportate ad ogni commit.

Abbiamo usato **Trello** per dividerci i compiti e tenere traccia di ciò che era stato fatto e ciò che doveva ancora essere implementato.

Per la progettazione della gerarchia abbiamo sfruttato **diagrams.net**, una web app che ci ha permesso di collaborare in tempo reale e disegnare la nostra gerarchia.

Per la progettazione della GUI ci siamo affidati a **figma**, visto che entrambi conoscevamo questo strumento che ci ha permesso di abbozzare una UI molto semplice, ma che ci è servita per semplificare o correggere subito degli errori che altrimenti avremmo incontrato solo durante la scrittura del codice.

Molti altri strumenti: **zoom** (causa covid), **virtualbox** (vm Linux), **paint.net** (foto editing), **google drive** (condivisione file).



[GitHub Desktop, Diagrams, Trello e Figma]

7.2 Istruzioni per compilazione ed esecuzione

Come richiesto dalle specifiche, il progetto compila in ambiente Linux con i seguenti comandi:

qmake ProgettoP2.pro
make

Per avviare il programma usare **./ProgettoP2** all'interno della cartella ProgettoP2.

Nel file .pro per far sì che compilasse abbiamo dovuto aggiungere **QT += core gui** e **QT += widgets**

7.3 Progetto sviluppato e testato sui seguenti sistemi

Sistema Operativo	Versione compilatore	Versione libreria Qt
Ubuntu 18.04 (virtual machine)	GCC 7.4.0	Qt 5.9.5
Windows 10 20.04	MinGW 5.3.0 32 bit	Qt 5.9.5
MacOS 10.15.6	Apple clang 11.0.3	Qt 5.9.5

8. Ore effettivamente richieste

8.1 Suddivisione lavoro di gruppo

Lorenzo Brescanzin	Progettazione gerarchia Scrittura view, controller Gestione GUI Bug fixing GUI e funzioni
Martin Masevski	Progettazione gerarchia Organizzazione strumenti di lavoro Scrittura model, contenitore Popolamento Database Test GUI e funzioni

8.2 Suddivisione delle ore

Personalmente ho impiegato circa 55 ore di lavoro, superando di qualche ora in più il limite delle 50 ore perché mi sono riguardato alcuni video del tutor e del professore, che farebbero sempre parte dello studio di Qt, ma ho deciso di contarle separatamente.

Lavoro	Ore impiegate
Analisi dei requisiti	3 ore
Progettazione della gerarchia	4 ore
Studio delle librerie Qt	15 ore
Video tutorato	10 ore
Codifica gerarchia e contenitore	13 ore
Codifica GUI	4 ore
Stesura relazione	2 ore
Debugging + Testing	4 ore