
# RELAZIONE PROGETTO PROGRAMMAZIONE AD OGGETTI

Francesco Freda – 1143643

a.a. 2018/2019

# 1. Abstract

Si vuole costruire un'applicazione per la gestione delle autovetture da inserire nel prossimo episodio della trasmissione televisiva Cars Crew. La direzione mette a disposizione un garage dove posizionare le auto, le quali differiscono per *nome*, *numero di cavalli* e altre caratteristiche legate al tipo di auto.

È soprattutto di interesse per l'azienda, capire l'ambiente dove un'automobile può circolare, di modo da poter scegliere la giusta location dove testarla all'interno dell'episodio. Quest'ultimo, metterà alla prova le diverse auto appartenenti a una delle quattro categorie scelte dalla direzione:

1- **Auto sportiva**: della quale si è interessati alla *velocità massima* espressa in km/h e il tipo di *trazione* (*anteriore, posteriore* o *integrale*).
2- **Auto da rally**: come per la auto sportiva, ma in aggiunta interessa sapere il tipo di *cambio* utilizzato (*manuale* o *sequenziale*).
3- **Fuoristrada**: oltre le caratteristiche comuni a tutti i tipi di auto, interessa conoscere *l'altezza della parte più bassa dell'autovettura, dal suolo,* espressa in cm. D'ora in avanti questa caratteristica verrà denominata con l'abbreviazione HFG* (Height From the Ground).
4- **Auto vintage**: interessa sapere *l'anno di produzione* del veicolo.

Ogni auto può essere testata in un solo possibile *ambiente* tra i cinque principali: montagna, deserto, terreno irregolare, circuito o città. La scelta dipende dal tipo e dalle qualità del veicolo:

- In *montagna,* se si tratta di un veicolo fuoristrada con HFG* strettamente minore di 25 cm.
- Nel *deserto,* se il veicolo è un fuoristrada con HFG* strettamente maggiore uguale di 25 cm e numero di cavalli strettamente maggiore di 85.
- In *circuito,* se l'autovettura è sportiva con velocità massima strettamente maggiore di 250 km/h, numero di cavalli strettamente maggiore di 80 e tipo di trazione integrale o posteriore. Se il veicolo è da rally, oltre che possedere tutte le caratteristiche appena elencate dell'auto sportiva, deve anche esser dotata di cambio manuale. Se l'auto è d'epoca, deve possedere più di 80 CV e essere prodotta dopo l'anno 1982.
- Su *terreno irregolare*, solamente se i veicoli sono da rally e non adatti per la circolazione su circuito, o fuoristrada non adatti al deserto e alla montagna.
- In *città*, solamente le auto sportive e vintage non adatte alla guida su circuito.

Si assume che il garage non abbia limiti di spazio.

# 2. Gerarchie di tipi

La gerarchia consiste di 5 classi complessive:

1. Car: *classe base polimorfa astratta* i cui oggetti rappresentano un'auto che può essere inserita nel garage e quindi candidata a partecipare nel prossimo episodio di Cars Crew. Ogni auto è caratterizzata dai campi dati nome e numero di cavalli.
2. Sport: *classe concreta derivata direttamente* da Car i cui oggetti rappresentano un'auto sportiva. Ogni oggetto Sport è caratterizzata dalla trazione che può essere posteriore, anteriore o integrale, e dalla velocità massima raggiungibile espressa in km/h.
3. OffRoad: *classe concreta derivata direttamente* da Car i cui oggetti rappresentano un veicolo fuoristrada. Ogni oggetto OffRoad è caratterizzato dall'HFG* espressa in cm.
4. Vintage: *classe concreta derivata direttamente* da Car i cui oggetti rappresentano un'auto d'epoca. Ogni

*: distanza fra il suolo e il punto inferiore del differenziale più basso.

oggetto Vintage è caratterizzato dall'anno in cui l'auto è stata prodotta

5. Rally: *classe concreta derivata direttamente* da Sport e *indirettamente* da Car, i cui oggetti rappresentano un'auto da Rally. Ogni oggetto Rally è caratterizzato dal tipo di cambio che può essere sequenziale o manuale.

Inoltre, Car contiene i due seguenti metodi virtuali puri:

- `Car* clone() const`: restituisce una copia dell'autovettura.
- `string environment() const`: restituisce l'ambiente migliore dove l'autovettura può circolare.

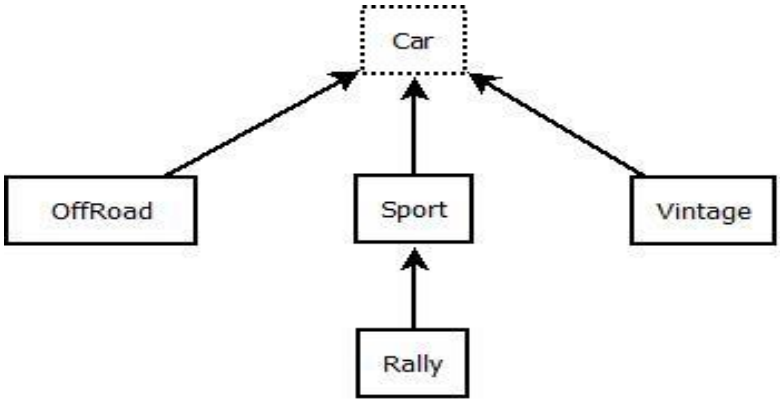


*Figura 1: Gerarchia delle classi: Freccia direzionata da classe B verso classe A indica che B è sottotipo di A.*

# 3. Progettazione

Il progetto è stato sviluppato con un approccio *MVC (Model-View-Controller)*, per gestire molteplici viste usando gli stessi dati. Anche se dispendioso come approccio, è stato scelto perché semplice da mantenere vista la meno dipendenza tra modello e vista, grazie anche all'utilizzo di una classe intermediaria *Controller* tra modello e vista. È anche più facile da utilizzare, specie quando si lavora con un progetto di gruppo, inoltre aumenta la flessibilità e il riutilizzo del programma.

# 4. Il Controller, punto d'incontro tra vista e modello

Dato un input, il controller si occupa di trasformare questo in un *comando per vista e modello*. Quando inserisco un'auto, ad esempio, la vista connette il segnale del bottone d'inserimento a un metodo slot pubblico del controller (l'input), tale slot invoca i giusti metodi del modello e della vista per inserire l'auto nel container e nella GUI list rispettivamente.

In questo caso, il controller ricava le informazioni necessarie dalla vista, per poi passarle al modello, ma è anche possibile il contrario. È ugualmente usuale ottenere le informazioni necessarie dal modello per poi passarle alla main view, che in questo caso visualizza o modifica i dati. È ciò che si manifesta quando clicco il bottone *Visualizza/Modifica* o il sottomenu *Carica* della vista.

Tutto questo fa capire che il controller avrà bisogno di due campi dati per la gestione dell'input, ossia:

- Il modello: si occuperà della *gestione della memoria interna* durante le operazioni di modifica, cancellazione ricerca e inserimento.

- la vista: si occuperà di *gestire la parte grafica* anche aprendo nuove finestre contenenti informazioni, come per esempio la finestra contenente i dettagli di un'auto (nome, cavalli, …) o la finestra di ricerca.

Il controller ha un ulteriore compito, ossia quello di *avvertire la vista della presenza di errori*, come per esempio quando si prova a cancellare un elemento dalla lista vuota. A tal fine, è stata creata una classe **Anomalia** contenete un solo campo dati di tipo char, il quale serve a *identificare il tipo di errore nei campi* quando inserisco o modifico i dati di un'auto. Tali errori vengono costruiti a livello GUI dalla vista, catturati sottoforma di eccezioni dal controller e lanciati dai metodi di container, della classe model o dalla vista.

# 5. Il Modello

Il modello è formato da queste classi:

- **Gerarchia**: il dato vero e proprio che si desidera immagazzinare, ossia le autovetture. Supporta la clonazione e distruzione polimorfa.
- **Container<T>**: la struttura dati di tipo lista doppiamente concatenata per elementi di uno stesso tipo T con le funzioni di inserimento, rimozione, ricerca, copia profonda, distruzione profonda. Presenti inoltre le implementazioni di un iteratore costante e non, più un campo dati size per contare quanti elementi sono presenti nella lista.
- **DeepPtr<T>**: classe di puntatori polimorfi al tipo T che si occupa di gestire la memoria profonda e polimorfa di puntatori a T.
- **Model**: contiene il container di deep pointer al tipo Car (classe base polimorfa astratta appartenente alla gerarchia). Si preoccupa, sotto indicazioni del controller, di inserire oggetti nel container, eliminare un elemento o più elementi, modificare un veicolo, cercare auto in base a una caratteristica.
- **XmlI**: Reader XML utilizzato nella classe model, si occupa di leggere un file .xml. per poi salvarlo nella struttura dati utilizzata da model.
- **XmlO**: Writer XML utilizzato nella classe model, ha il compito di scrivere su file .xml gli elementi presenti dentro il container di model.

# 6. Graphical User Interface (GUI)

Sono presenti quattro classi per l'implementazione grafica lato utente:

1) **insertCar** derivata da `QWidget`, contiene tutti i campi che devono essere compilati per inserire un tipo d'auto. Composta principalmente da campi dati che permettono la visualizzazione di un form da compilare (`QLabel`, `QSpinBox`, `QDoubleSpinBox`, `QLineEdit`, `QComboBox`, `QCheckBox`).
2) **CarsView** è la main view della GUI, derivata da `QWidget`, consiste in 4 layouts: `insertCar`, la `QWidgetList` contenete le auto inserite, la barra del menu utile per caricare un file .xml, salvarlo, aprire un file vuoto, ma anche per eliminare tutte le auto nella lista, e infine il filtro di ricerca per cercare o cancellare particolari tipi di auto. Nel dettaglio la classe ha come campi dati il controller, per richiamare il giusto metodo slot quando si svolge un'azione nella finestra, il nome del file, la `QWidgetList`, `CarDetail`, `ResearchingCars` e i campi che si occupano della ricerca (`QComboBox` e `QLineEdit`). Per ogni campo dati, ad eccezione del controller, è presente un metodo di get per ottenere il suo valore. Questo è di utilità per il controller, quando ad esempio deve passare il nome del file al modello (vedi metodo `loadData()`). `CarsView` usa inoltre all'interno di alcuni metodi dei `QMessageBox`, ad esempio per chiedere di salvare il file mentre si esce dal programma o per segnalare l'impossibilità di compiere un'azione (eliminare un elemento dalla lista vuota). Per il caricamento dei dati, dopo che il controller ha chiesto al modello di creare il container contenete le auto contenute nel file.xml, viene chiamato il costruttore di `CarsView` il quale usufruisce di un parametro formale di tipo bool. Questo ci dice se il modello è stato caricato da file .xml oppure se è stato scelto di aprire il

programma senza caricare dati. Nel primo caso, viene assegnata alla nuova `QWidgetList` la lista creata dal metodo `loadData()` di controller (vedi `loadPage()`).

3) **CarDetail** derivata da `QDialog`, serve a visualizzare/modificare in una nuova finestra i dettagli di una specifica autovettura. Composta principalmente da campi dati che permettono la visualizzazione di un form già compilato che si può modificare (`QLabel`, `QSpinBox`, `QDoubleSpinBox`, `QLineEdit`, `QComboBox`, `QCheckBox`)
4) **ResearchingCars** derivata da `QDialog`, visualizza in una nuova finestra la lista delle auto cercate nella finestra principale, le quali si possono anche rimuovere tutte o solo quelle selezionate. Composta da una `QListWidget`.

I metodi `value()` di `QSpinBox` e `QDoubleSpinBox` restituiscono un `int` e un `double` rispettivamente. Questi metodi sono stati utilizzati da alcuni metodi di get presenti nelle classi insertCar e CarDetail e costituiscono un'eccezione nei tipi da utilizzare nella GUI, che solitamente devono essere appartenenti alla libreria Qt per rispettare la divisione tra codice GUI e codice del modello. Si ha deciso di non rispettare totalmente questa regola in quanto si evitano ulteriori conversioni di tipo.

# 7. Ricerca di autovetture

Per la ricerca di elementi all'interno di una lista, viene richiamato il metodo `buildResearchList()` del Controller, il quale chiama uno dei metodi di ricerca del modello. I metodi di ricerca della classe Model (vedi ad es. searchCV(const int&, bool)) si differenziano solamente per il tipo di valore da cercare, ma tutti ritornano un `vector<int>` contenente le posizioni delle auto che ci interessano all'interno del container; per esempio la posizione delle auto con velocità maggiore di un certo numero. Il vettore ritornato sarà quindi utile per dire alla vista quali delle auto presenti visualizzare all'interno della lista di ricerca, di modo che il metodo possa costruire la nuova `QWidgetList` di auto che la nuova finestra conterrà, ma serve anche per dire alla classe model quali auto eliminare, come fatto dal metodo slot privato `removeAllResults()` di Controller.

# 8. Polimorfismo

Il progetto fa uso del polimorfismo tramite gli oggetti contenuti nel container di tipo DeepPtr<T>, il quale conterrà come campo dati un puntatore al tipo template T. Nel nostro caso, T è il tipo astratto Car e *chiamate polimorfe* ai metodi virtuali saranno effettuate dai metodi nelle classi:

1. Model per:
   - Modificare caratteristiche di un'autovettura.
   - Rimuovere dal garage veicoli con caratteristiche specifiche.
   - Ricercare veicoli in base a delle loro caratteristiche (es: tipo veicolo, CV, velocità…).
2. Nella classe Controller per:
   - Recuperare informazioni su di un'auto nel garage.
   - Ottenere l'ambiente dove un'autovettura può circolare.
3. Nella classe XmlO per ottenere e scrivere su file XML i dettagli di un particolare tipo di veicolo.

# 9. Load and Save Data

Lettura e scrittura avvengono su file **.xml**. La scelta di questo tipo di file ricade sulla facile interpretazione di questo, molto intuitivo per qualsiasi persona. I file .xml identificano ogni auto mediante il tag <Auto tipo=" Sport">, il quale si

riferisce ad una auto di tipo sportiva. Internamente ad esso, vengono elencate le caratteristiche tramite altri tags, ognuno dei quali contiene il valore del nome del tag come ad esempio <anno>1979 </anno>.

Le opzioni disponibili per salvataggio e caricamento di un file sono le seguenti:

- Salvare come nuovo file
- Salvare come file già esistente
- Caricare un file già creato
- Aprire un file vuoto senza nome

All'avvio e alla chiusura è possibile rispettivamente caricare un .xml o salvarlo.

# 10. Funzioni del programma

- ✓ *Inserimento* di una autovettura nel container.
- ✓ *Rimozione* di un veicolo selezionato nel container.
- ✓ *Rimozione di tutte o di specifiche* autovetture.
- ✓ *Visualizzazione* completa e dettagliata delle caratteristiche dei diversi tipi di autovetture inserite.
- ✓ *Ricerca* di veicoli in base a delle loro caratteristiche.
- ✓ *Modifica* della autovettura selezionata.
- ✓ *Scrittura* dei veicoli presenti nel garage in un file .xml.
- ✓ *Lettura* dei veicoli da inserire nel garage tramite file .xml.

# 11. Manuale d'uso

All'avvio del programma, viene chiesto di caricare un file .xml. Sono disponibili diversi esempi di .xml già costruiti.

Una volta arrivati alla schermata iniziale, in alto è presente la barra del menu contiene due voci, *File* e *Modifica*. La prima voce serve per il caricamento dei dati da file esistente, aprire un file vuoto, salvataggio e chiusura del programma. La seconda voce viene usata per rimuovere tutte le auto inserite finora.

Una volta inizializzato il programma, nella parte destra del layout è possibile *inserire* il veicolo compilando i vari campi. Nella parte sinistra invece verrà visualizzato il nome dell'auto inserita.

Selezionando con il mouse un'auto nella lista è possibile *visualizzare i dettagli, modificarla o eliminarla* premendo con il cursore sui rispettivi pulsanti sotto la lista.

Sopra la lista e sotto la barra del menu è presente lo spazio di ricerca. Compilando i due campi è possibile *ricercare* o *eliminare* veicoli che rispecchino le caratteristiche indicate.

# 12. Compilazione ed esecuzione

Per l'esecuzione del progetto, viene richiesta la compilazione del file CarsCrew.pro:

```
$ qmake CarsCrew.pro
```

```
$ make
```

Il motivo è dato dalle specifiche al suo interno della versione utilizzata C++11 e della posizione di tutti i files source e header.

Il progetto è stato realizzato con sistema operativo **Linux Mint 19.1 Cinnamon**, **Qt Creator 4.8.1** e compilatore **clang 6.0.0-lubuntu2**.

# 13. Ore di lavoro

| **Fase progettuale** | **Ore** |
|---|---|
| Analisi preliminare del problema | 4 |
| Progettazione modello e GUI | 17 |
| Apprendimento libreria Qt | 10 |
| Codifica modello | 8 |
| Codifica GUI | 25 |
| Debugging | 3 |
| Testing | 3 |
| **Totale ore** | 70 |

Le 20 ore lavorative in eccesso sono dovute:

- All'implementazione del deep pointer, inizialmente errata.
- Alla prima versione sviluppata della main view della GUI, rilevatasi molto più complessa di quanto richiesto e quindi sostituita con una ben più semplice e più adatta all'approccio MVC.
- Alla correzione di vari errori nei metodi del container, precisamente nei metodi di inserimento e rimozione degli elementi.