UNIVERSITÀ DEGLI STUDI DI BERGAMO

Dipartimento di Ingegneria Gestionale, dell'Informazione e della Produzione

Documentazione Progetti

Corso di Informatica III - Modulo di Programmazione

Studente: **Luca Ghislotti (matr. 1052975)**

Docente: **Prof. Angelo Gargantini**

CdL Magistrale in Ingegneria Informatica

Facoltà di Ingegneria

Università degli Studi di Bergamo

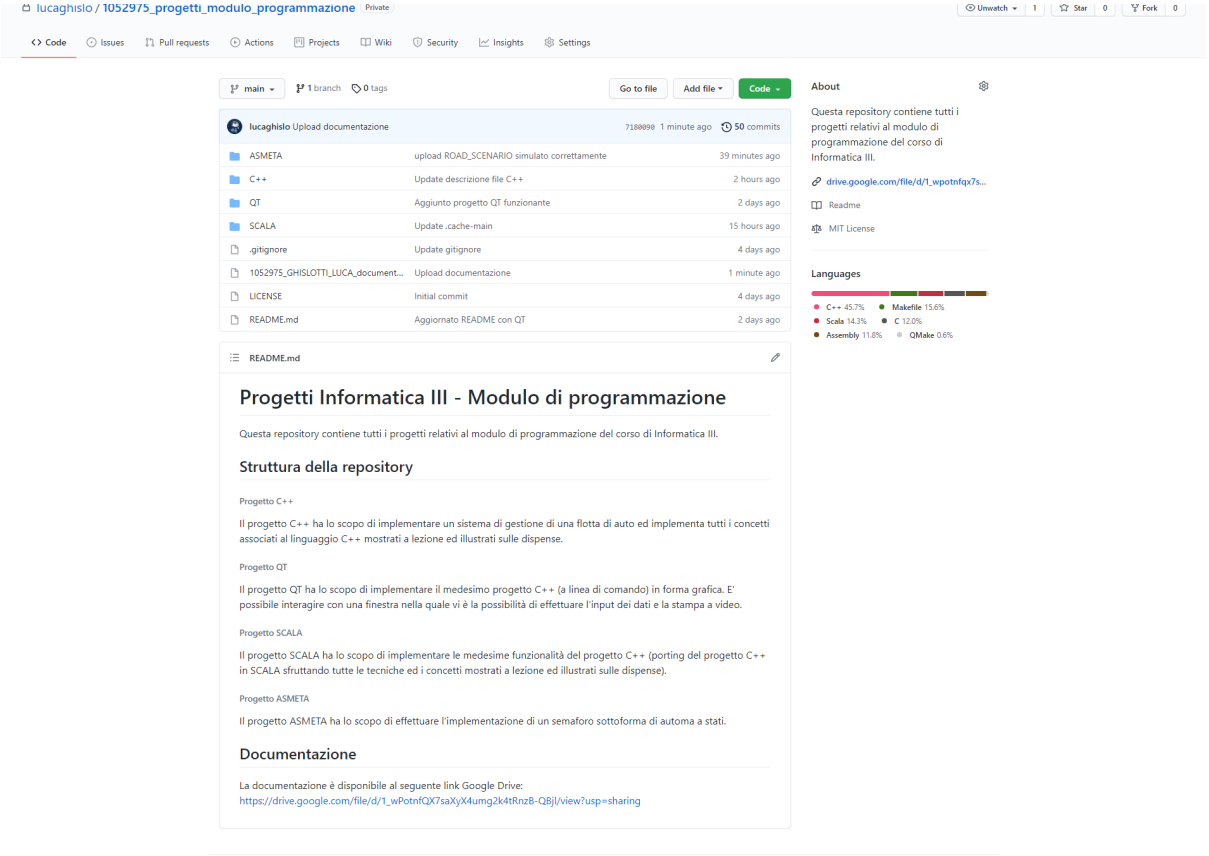
A.A. 2020/2021

Bergamo, Italia - Settembre 2021

Codice

Il codice è reso disponibile su GitHub al seguente link:

https://github.com/lucaghislo/1052975_progetti_modulo_programmazione.git



Documentazione

La documentazione in formato PDF è disponibile via Google Drive al seguente link:

https://drive.google.com/file/d/1_wPotnfQX7saXyX4umg2k4tRnzB-QBjl/view?usp=sharing

Progetto C++

Progetto C++

Il progetto prevede la gestione di una flotta di auto con varia alimentazione (diesel, benzina, ibrida ed elettrica). Il programma permette la memorizzazione di una lista di auto e l'inserimento da parte dell'utente dei singoli veicoli, con indicazione dei campi per ciascuna di esse.

Questo progetto ha lo scopo di sfruttare tutti i costrutti del linguaggio C++ che sono stati spiegati durante le lezioni ed illustrati sulle dispense; di seguito si fornisce un'indicazione dei costrutti/concetti utilizzati e la porzione di codice nei quali sono stati sfruttati, con la relativa spiegazione della loro utilità.

1. **Definizione di file .h e file .cpp (object orientation)**

Tutte le implementazioni delle classi sono state effettuate separando il file header (.h) in cui vengono rese disponibili le funzionalità associate alla classe ed il file di definizione dell'implementazione (.cpp).

2. **Overloading dei costruttori, valori di default e member initializer list**

Sono stati definiti più costruttori per la medesima classe allo scopo di garantire che un oggetto possa essere costruito diversamente sulla base dei dati a disposizione e di quelli posti a valore di default. Si sono definiti valori di default per i parametri.

```
Car();
Car(char *targaInput, int pesoInput = 1000, int potenzaInput =
                  Car::minPotenza);
```

```
Ecar::Ecar(char *targaInput, int pesoInput, int potenzaInput) :
            Car(targaInput, pesoInput, potenzaInput),
capacitaBatteria(400) {

      standards.push_back(110);
      standards.push_back(230);
      standards.push_back(380);
}
```

3. **Copy constructor**

Copy constructor definito per la classe `Car` allo scopo di garantire la costruzione di un'istanza della classe tramite copia di un'istanza già creata.

```
Car::Car(const Car &newCar) {
	strcpy(targa, newCar.targa);
	strcpy(numTelaio, newCar.numTelaio);
	Car::potenza = newCar.potenza;
	Car::peso = newCar.peso;
}
```

4. **Distruttori `virtual` ed utilizzo della `delete`**

Distruttori definiti con clausola **`virtual`** allo scopo di garantire l'invocazione del metodo distruttore delle superclassi della classe considerata.

```
virtual ~Car();
Car::~Car() {
	delete targa;
	delete numTelaio;
	cout << "cleared all cars" << endl;
}
```

5. **Definizione di funzioni, passaggio per valore e per riferimento**

Definizione di varie funzioni per ciascuna classe e passaggio di parametri per valore e per riferimento. Utilizzo del modificatore **`const`** per evitare che il metodo modifichi i membri della classe.

```
void stampaInfoAuto(Car &c) const;
void Garage::stampaInfoAuto(Car &c) const {
	c.showInfo();
}
```

6. **Funzioni `inline`**

Funzioni per le quali la chiamata viene sostituita con il corpo della funzione.

```
inline short countAuto() {
		return flotta.size();
}
```

7. **Incapsulamento e definizione di diversi livelli di visibilità**

Utilizzo dei modificatori di visibilità **`public`**, **`private`**, **`protected`**.

```
class Ecar: virtual public Car {
public:
	Ecar(vector<int> standardInput);
	Ecar(char *targaInput, int pesoInput = 1500, int potenzaInput =
50);
	virtual void showInfo();
	virtual void printResumee();
	short getCapBat();
	void setBatCap(short batCap);
	virtual string getClassName();
	virtual ~Ecar();

private:
	short capacitaBatteria;
	vector<int> standards;
};
```

8. **Funzioni `friend`**

Per garantire che la funzione dichiarata **`friend`** possa accedere ai campi privati della classe di cui è amica.

```
template<typename T> friend class TaxCalc;
```

9. **Ereditarietà, classi base e derivate, visibilità dell'ereditarietà ed ereditarietà multipla**

Definizione di classi base (classe `Car`) e classi derivate (`FFcar`, `Hcar`, `Ecar`) allo scopo di riutilizzare il codice. Riutilizzo del costruttore della classe base.

```
class Hcar: public Ecar, public FFcar {
public:
	Hcar(char *targaInput, int pesoInput, int potenzaInput, short
capInput, double urbano, double combinato, double extra, vector<int>
standardsInput);

	virtual void showInfo();
	virtual void printResumee();
	virtual string getClassName();
	virtual ~Hcar();

private:
	unsigned short capacitaCombinata;
```

```
};
```

10. Risoluzione dei problemi legati all'ereditarietà multipla: name clashing

Risoluzione del name clashing causato dalla possibilità in C++ di definire una classe derivata a partire da più classi base.

```
void Hcar::showInfo() {
	cout << "\nHYBRID ";
	Car::showInfo(); // risoluzione esplicita del name clashing
	printResumee();
}
```

11. Member duplication: problema del diamante

Utilizzo della clausola **`virtual`** sulla classe base allo scopo di risolvere il problema del diamante nel contesto dell'ereditarietà multipla.

```
class Ecar: virtual public Car { ... }
class FFcar: virtual public Car { ... }
class Hcar: public Ecar, public FFcar { ... }
```

12. Polimorfismo run-time

Utilizzo della clausola **`virtual`** per garantire implementazione del polimorfismo run-time di vari metodi definiti nelle classi.

```
class Hcar: public Ecar, public FFcar {
public:
	virtual void showInfo();
	virtual void printResumee();
	virtual string getClassName();
}
```

13. Polimorfimo compile time: template di funzioni e template di classi

Definizione di **`template`** di funzioni e di classi sfruttando parametri generici. Si riporta l'esempio della classe `TaxCalc` definita come template di classe, la quale contiene la funzione `calcoloBollo()` definita a sua volta come template di funzione.

```
template<typename T> class TaxCalc {
```

```
	static const int fattoreCorrettivo = 1.37;
	T cavalliFiscali;

public:
	TaxCalc(T const &cf) : cavalliFiscali(cf) {}

	template<typename S> T calcoloBollo(S c) {
		return c->potenza * cavalliFiscali * fattoreCorrettivo;
	}
};
```

14. Sottotipazione (evitato lo slicing)

Nel contesto della definizione di sottotipi, si è evitato il manifestarsi del fenomeno dello slicing, per il quale l'assegnamento di un'istanza più specializzata ad una meno specializzata comporta la perdita dei campi presenti nella prima e non nella seconda.

15. Classi astratte

Classe `Car` ha almeno un metodo *pure* **`virtual`**; si riportano due metodi di questa forma. Questi metodi sono implementati nelle sole classi derivate.

```
class Car {
public:
	virtual void printResumee() = 0;
	virtual string getClassName() = 0;
}
```

16. Gestione delle eccezioni

Si sono gestite le eccezioni, soprattutto per quanto riguarda l'inserimento da parte dell'utente e per evitare *bufferOverflow* durante la copia di stringhe tramite **`strcpy()`** e *raw pointers*. Si è scelto di fare il throw di un codice di errore, intercettato nel blocco *try-catch* dove il metodo viene chiamato.

```
void Car::setTarga(string newTarga) {
	if (newTarga.length() > 9)
		throw 403;
	else
		strcpy(targa, newTarga.data());
}
```

Intercettazione del codice di errore e visualizzazione del messaggio di errore.

```
try {
	ec1->setTarga("EL-101-LEEEEEEEEE");
} catch (int x) {
	cout << "Formato targa errato! Cannot update" << endl;
}
```

17. STL containers: contenitori associativi e non associativi

Utilizzo di contenitori non associativi, quali il **`vector<>`** per la memorizzazione, all'interno della classe Garage, delle auto nella flotta inserite da parte dell'utente. Si è utilizzato anche il contenitore associativo **`map<>`** per memorizzare la coppia "descrizione, consumo" nella classe `FFcar`. Sono state ovviamente utilizzare le funzioni STL associate ai contenitori per l'inserimento, l'estrazione e la modifica degli elementi nel contenitore.

```
vector<unique_ptr<Car>> flotta;
```

```
map<string, double> consumi;
```

18. Utilizzo degli iteratori

Per lo scorrimento dei contenitori STL definiti precedentemente si è utilizzato lo strumento *iterator*. Da notare come il contenitore STL non associativo flotta contenga riferimenti a smart pointers **`unique_ptr`**: in questo caso, l'iteratore viene utilizzato in congiunzione con la dereferenziazione dello smart pointer per l'ottenimento del relativo raw pointer.

```
void FFcar::printResumee() {
	for (map<string, double>::iterator it = consumi.begin();
			it != consumi.end(); ++it)
		cout << it->first << ":  " << it->second << " L/100km\n";
}
```

19. Utilizzo del tipo `bool`

C++ aggiunge, rispetto a C, il tipo booleano di default (senza ulteriori import). Per sfruttare questo tipo di dato, si è deciso di implementare una funzione che ritorni un

valore logico associato al fatto che l'auto passata come parametro sia elettrica oppure no.

```
bool isElectric(Car *c);
```

```
bool Garage::isElectric(Car *c) {
	if (c->getClassName() == "E" || c->getClassName() == "H")
		return true;
	else
		return false;
}
```

20. Utilizzo degli smart pointers

Oltre all'utilizzo dei classici puntatori a spazi di memoria allocati sullo heap tramite primitiva **`malloc`**, si sono utilizzati anche *smart pointers*, allo scopo di dimostrarne l'efficacia e l'utilità oltre che la maggior sicurezza indotta dalla gestione automatica dell'allocazione e deallocazione dello spazio di memoria. Si è scelto di definire un **`vector`** di smart pointers **`unique_ptr`** per i quali è stato necessario l'utilizzo della primitiva **`move()`**, causato dal fatto che il puntatore, essendo unico, deve essere manualmente spostato allo scopo di non perderne il riferimento.

```
unique_ptr<Car> veicoloFF(new FFcar(targa_ptr, peso, potenza, fuelCap,
urbano, combi, extra));
flotta.push_back(move(veicoloFF));
```

```
unique_ptr<Car> veicoloE(new Ecar(targa_ptr, peso, potenza));
flotta.push_back(move(veicoloE));
```

Definizione del **`vector`** di *smart pointers*:

```
vector<unique_ptr<Car>> flotta;
```

Stampa del contenuto del **`vector`**, contenente smart pointers **`unique_ptr`**:

```
void Garage::stampaFlotta() {
	cout << "***AUTO NELLA FLOTTA***" << endl;
	for (auto const &i : flotta) {
```

```
            i.get()->showInfo();
      }
}
```

21. Utilizzo di design pattern

Si è scelto di implementare la classe `Garage` sfruttando il design pattern *Singleton* allo scopo di garantire l'univoca istanziazione di questa classe, evitando la creazione da parte dell'utente di molteplici copie.

```
class Garage {
public:
      static Garage& getInstance() {
            static Garage instance;
            return instance;
      }
private:
      Garage();
      Garage(Garage const&);
}
```

Questo design pattern garantisce la creazione dell'unica istanza della classe `Garage` passando attraverso il metodo **`getIstance()`** senza utilizzare direttamente il costruttore della classe, mantenuto **`private`**.

Progetto QT

Progetto QT

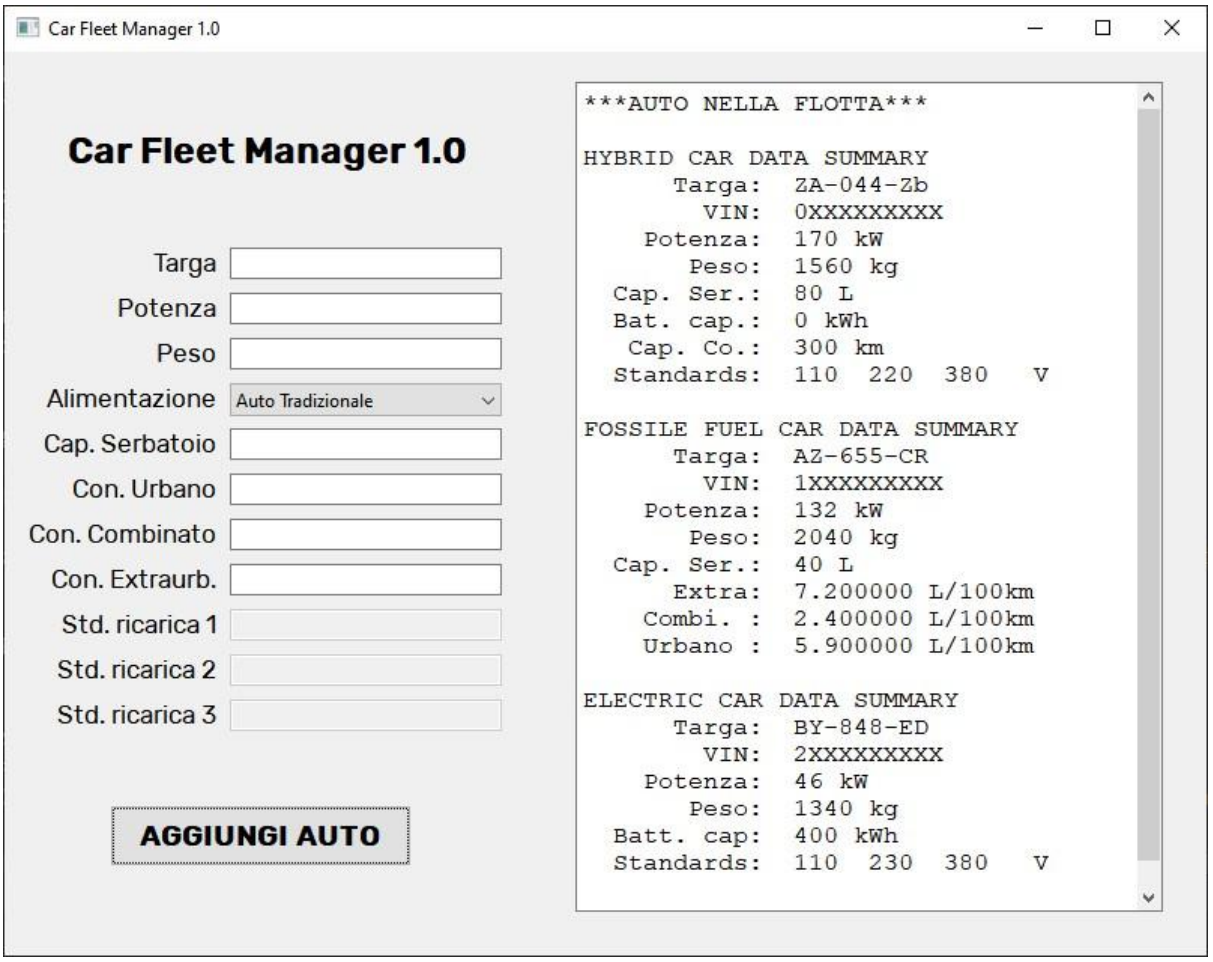
Il progetto QT è stato implementato sfruttando l'applicativo QT CREATOR, di fatto utilizzando il codice C++ realizzato per il precedente progetto. E' stata infatti modificata la classe Garage per poter accettare in input i dati raccolti tramite interfaccia grafica, diversamente da quanto effettuato nella precedente implementazione, puramente basata su interfaccia a linea di comando.

Il progetto QT si basa su tutti i file sviluppati nel precedente progetto, con l'aggiunta dell'interfaccia `carfleetmanager.h` e relativa classe `carfleetmanager.cpp`, oltre che il file per la definizione dell'interfaccia grafica `carfleetmanager.ui`.

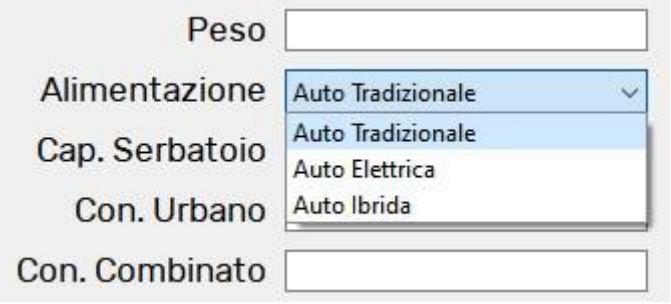
Di seguito si riporta la definizione dell'interfaccia `carfleetmanager.h`:

```
#ifndef CARFLEETMANAGER_H
#define CARFLEETMANAGER_H

#include <QWidget>

QT_BEGIN_NAMESPACE
namespace Ui { class CarFleetManager; }
QT_END_NAMESPACE

class CarFleetManager : public QWidget {
    Q_OBJECT

public:
    CarFleetManager(QWidget *parent = nullptr);
    ~CarFleetManager();

private slots:
    void on_pulsanteSubmit_clicked();
    void on_tipoAuto_currentIndexChanged(int index);

private:
    Ui::CarFleetManager *ui;
};

#endif // CARFLEETMANAGER_H
```

Si riporta anche una screenshot dell'applicativo:



Da notare come i campi siano attivati e disattivati dinamicamente in funzione del tipo di auto selezionata tramite selettore *QComboBox* (ad esempio, per l'auto tradizionale non sono definiti i campi che descrivono gli standard di ricarica).



Al momento dell'aggiunta di una nuova auto, i campi vengono resettati e svuotati, effettuando il refresh della componente di visualizzazione con l'elenco delle auto presenti nella flotta del garage.

Progetto SCALA

Progetto SCALA

Il progetto SCALA implementa il porting del progetto C++; in particolare, si è deciso di modificarne le specifiche ove necessario allo scopo di mettere in evidenza particolari costrutti del linguaggio SCALA che sono meglio evidenziabili tramite implementazioni ad hoc, o che comunque non erano messe in evidenza nella corrispettiva versione in linguaggio C++. Si è infatti voluto porre particolare enfasi sull'aspetto funzionale di Scala, oltre che sui costrutti **`fold`**, **`reduce`**, **`filter`** e **`map`** per l'analisi di strutture dati.

Questo progetto ha lo scopo di sfruttare tutti i costrutti del linguaggio SCALA che sono stati spiegati durante le lezioni ed illustrati sulle dispense; di seguito si fornisce un'indicazione dei costrutti/concetti utilizzati e la porzione di codice nei quali sono stati sfruttati, con la relativa spiegazione della loro utilità.

1. **Utilizzo di `var` e `val`**

In questo progetto sono stati utilizzate sia variabili **`var`** che **`val`**: le prime possono essere ridefinite dopo la dichiarazione, le seconde no. Sono stati utilizzati anche modificatori di visibilità delle variabili come **`private`**.

```
private var numTelaio: String = "XXXXXXXXXX"
```

```
private val minPotenza = 30;
```

2. **Costruttore, passaggio di parametri e valori di default dei parametri**

Definizione di classi tramite notazione che permette di definire il costruttore direttamente alla dichiarazione della classe. Si sono definiti i parametri sia nell'intestazione della classe che nel suo corpo. Sono stati indicati, per tutti o alcuni parametri, valori di default, rendendo i parametri opzionali e, nel caso in cui non vengano specificati l'istanziazione della classe, sono posti al valore di default.

```
class Ecar(targa: String, potenza: Int, peso: Int, var capacitaBatteria:
Int = 400, inputStandards: ArrayBuffer[Int]) extends Car(targa, potenza,
peso) { ... }
```

3. Overload del costruttore

Scala permette l'overload del costruttore tramite clausola this. Il costruttore secondario richiama il costruttore primario.

```
def this(targa: String, potenza: Int, peso: Int) {
    this(targa, potenza, peso, 400, ArrayBuffer(110, 230, 380))
}
```

4. Sottotitolazione ed ereditarietà

Le classi `FFcar` ed `Ecar` sono sottoclassi della superclasse astratta `Car`, mentre la classe `Hcar` è a sua volta sottoclasse della classe `Ecar`, con l'aggiunta di campi (ciò è causato dall'ereditarietà singola di Scala, come per Java, al contrario dell'ereditarietà multipla di C++). E' stata utilizzata la clausola extends per ereditare la classe ed implementare il `trait`.

```
class FFcar(targa: String, potenza: Int = 100, peso: Int = 1500, var
capacitaSerbatoio: Int = 50, urbano: Double = 4.5, combi: Double = 5.6,
extra: Double = 7.3) extends Car(targa, potenza, peso) { ... }
```

5. Classe astratta

La classe `Car` è stata definita **`abstract`**, pertanto non è istanziabile.

```
abstract class Car(private var targa: String, private var potenza: Int,
private var peso: Int) extends VINbuilder { ... }
```

6. Classe con oggetto companion (Singleton)

La classe `VINbuilder` ha associato un oggetto companion che permette di costruire un campo **`static`** (non essendo nativamente definito in Scala) comune a tutte le istanze della classe.

```
object VINBuilder {
  var prefix = 0
  def increasePrefix = {
    prefix += 1;
}
```

```
}

class VINbuilder { ... }
```

7. **Scala trait (interfaccia)**

L'oggetto singleton `FleetStats` implementa l'interfaccia `FleetStatsTrait` che ne specifica i metodi.

```
trait FleetStatsTrait {
  def pesoTotale(flotta: ListBuffer[Car]): Int
  def maxPotenza(flotta: ListBuffer[Car]): Int
  def consumoMedio(veicolo: FFcar): Double
  def autoSuperbollo(flotta: ListBuffer[Car]): ListBuffer[Car]
  def MapReduce(flotta: ListBuffer[Car]): Int
}
```

8. **Funzioni annidate**

Scala permette la definizione di funzioni annidate, cioè definite internamente ad altre funzioni.

```
override def showInfo {
    def printResumee {
      consumi.foreach {
        case (denominazione, valore) => println(s"$denominazione:
$valore L/100km")
    }
}
```

9. **Type inference ed indicazione esplicita del tipo**

In Scala è possibile definire variabili senza indicazione esplicita del tipo, che viene automaticamente dedotto durante l''assegnamento del valore.

```
val fuelCap = scala.io.StdIn.readInt()
val urbano = scala.io.StdIn.readDouble()
val combi = scala.io.StdIn.readDouble()
val extra = scala.io.StdIn.readDouble()
```

Il tipo può essere anche esplicitamente indicato.

```
val fuelCap: Int = scala.io.StdIn.readInt()
val urbano: Double = scala.io.StdIn.readDouble()
val combi: Double = scala.io.StdIn.readDouble()
val extra: Double = scala.io.StdIn.readDouble()
val s1: Int = scala.io.StdIn.readInt()
val s2: Int = scala.io.StdIn.readInt()
val s3: Int = scala.io.StdIn.readInt()
```

10. Override dei metodi

Le sottoclassi `Ecar`, `Hcar` ed `FFcar` effettuano l'override del metodo **`showInfo()`** della classe astratta `Car`. In Scala è necessario utilizzare la parola chiave **`override`** per effettuare la ridefinizione del metodo ereditato dalla superclasse.

```
override def showInfo {
   print("ELECTRIC ")
   super.showInfo()
   println("Batt. cap:  " + this.getCapBat + " KWh")
   this.printResumee(getArrayBufferContent)
   println("")
}
```

```
override def showInfo {
    def printResumee {
      consumi.foreach {
        case (denominazione, valore) => println(s"$denominazione:
$valore L/100km")
      }
    }

    print("FOSSILE FUEL ")
    super.showInfo()
    println("Cap. Ser.:  " + getCapSerb + " L")
    printResumee
    println("\nCon. Med.:  " + FleetStats.consumoMedio(this) + "
L/100km")
    println("")
}
```

```
override def showInfo {
    def printResumee {
      consumi.foreach {
```

```
        case (denominazione, valore) => println(s"$denominazione:
$valore L/100km")
      }
    }

    print("\nPETROL AND ")
    super.showInfo()
    println("Cap. Ser.:  " + getCapSerb + " L")
    printResumee
    println("")
  }
```

11. Passaggio di parametri call-by-name e call-by-value

Il passaggio dei parametri alle funzioni è stato effettuato sia per valore che per nome. Il passaggio dei parametri per nome avviene tramite notazione **=>**.

```
def printResumee(getArrayBufferContent: Int => String) {
    print("Standards:  ")
    print(getArrayBufferContent(2))
    print("V")
}
```

12. Funzioni di ordine superiore al primo (HOF, High Order Functions)

Nel progetto sono state implementate High Order Functions, così definite poiché prevedono come parametro in input una funzione.

```
def printResumee(getArrayBufferContent: Int => String) {
    print("Standards:  ")
    print(getArrayBufferContent(2))
    print("V")
  }
```

Nella funzione passata come parametro è stato messo in evidenza anche un'altro aspetto, associato al return delle funzioni. In particolare, il return viene effettuato senza necessariamente indicare la clausola **`return`**. Si mette in evidenza come il valore ritornato sia rappresentato dall'ultima espressione indicata nel corpo della funzione.

```
def getArrayBufferContent(numSpaces: Int = 1): String = {
    var output: String = "";

    for (i <- standards)
```

```
      output = output + i + " " * numSpaces
    output + "Piero non viene ritornato"
    output
  }
```

13. Collezioni `mutable`

In questo progetto sono state utilizzate collezioni per la maggior parte **`mutable`** (differiscono da quelle **`immutable`**, come **`List`**, per il fatto di essere modificabili dopo la loro creazione).

E' stato utilizzato l'**`ArrayBuffer`** per la memorizzazione degli standard di ricarica dell'auto elettrica nella classe `Ecar`:

```
class Ecar(targa: String, potenza: Int, peso: Int, var capacitaBatteria:
Int = 400, inputStandards: ArrayBuffer[Int]) extends Car(targa, potenza,
peso) {
    var standards = ArrayBuffer.empty[Int]
    standards = inputStandards
}
```

E' stato utilizzato il **`ListBuffer`** per la memorizzazione delle auto (tipo `Car` e relative sottoclassi `Ecar`, `FFcar`, `Hcar`) come flotta nella classe `Garage`:

```
val flotta = ListBuffer.empty[Car]
flotta += new FFcar(targa, potenza, peso, fuelCap, urbano, combi, extra)
flotta += new Ecar(targa, potenza, peso)
flotta += new Hcar(targa = targa, potenza = potenza, peso = peso,
capacitaSerbatoio = fuelCap, inputStandards = ArrayBuffer(s1, s2, s3),
urbano = urbano, combi = combi, extra = extra)
```

E' stata utilizzata la **`Map`** per la memorizzazione della coppia `<denominazione, valore>` dei consumi dell'auto tradizionale `FFcar`:

```
val consumi = Map(
    "   Urbano" -> urbano,
    "Combinato" -> combi,
    "    Extra" -> extra
)
```

14. Utilizzo del `foreach` per effettuare iterazione sulle collezioni

E' stato utilizzato il costrutto **`foreach`** per effettuare lo scorrimento delle collezioni.

Scorrimento della collezione **`Map`** con estrazione della coppia per effettuarne il printout a console:

```
override def showInfo {
    def printResumee {
      consumi.foreach {
        case (denominazione, valore) => println(s"$denominazione:
$valore L/100km")
    }
}
```

```
def stampaFlotta {
    println("***AUTO NELLA FLOTTA***")
    flotta.foreach {
      case(car) => car.showInfo()
    }
}
```

15. Folding: `foldLeft` e `foldRight`

I due costrutti sono stati utilizzati per effettuare calcoli su collezioni mutable con elementi non primitivi, ovvero definiti a partire da tipi associati a classi user-defined (classe `Car`). Si è pertanto reso necessario l'accesso al singolo elemento per l'implementazione delle operazioni di folding. La differenza nei due approcci risiede nell'ordine dello scorrimento della collezione, rispettivamente da sinistra verso destra e da destra verso sinistra. La parte di programmazione funzionale orientata all'analisi di data structures è stata implementata nell'oggetto (*singleton*) **`FleetStats`** che implementa il **`trait FleetStatsTrait`.**

La funzione **`pesoTotale()`** prende come parametro in input il **`ListBuffer flotta`** e restituisce il peso totale delle auto in esso presenti:

```
def pesoTotale(flotta: ListBuffer[Car]): Int = {
     flotta.foldLeft(0)((pesoTotale, veicolo) => pesoTotale +
veicolo.getPeso)
}
```

La funzione **`maxPotenza()`** prende come parametro in input il **`ListBuffer flotta`** e restituisce il massimo valore di potenza tra tutte le auto in esso presenti:

```
def maxPotenza(flotta: ListBuffer[Car]): Int = {
    flotta.foldRight(flotta.head.getPotenza)((veicolo, temp) => temp max
veicolo.getPotenza)
}
```

16. Utilizzo della `reduce`

Il costrutto reduce, concettualmente simile al fold, è stato utilizzato per effettuare il calcolo del consumo di carburante medio di una FFcar. Il paradigma reduce, al contrario del fold, non necessita dell'indicazione del primo valore da cui far partire l'analisi della sequenza. Da notare l'utilizzo innestato di più funzioni.

```
def consumoMedio(veicolo: FFcar): Double = {
    def getListFromMap(veicolo: FFcar): MutableList[Double] = {
      var listaConsumi = MutableList[Double]()

      veicolo.getConsumi.foreach {
        case (denominazione, valore) => listaConsumi += valore
      }

      listaConsumi
    }

    val lista = getListFromMap(veicolo)
    lista.reduce((a, b) => a + b) / lista.length
}
```

17. Utilizzo del `filter`

Il costrutto **`filter`** è stato utilizzato per effettuare il filtraggio delle sole auto per le quali è necessario pagare il superbollo, ovvero quelle auto con potenza > 184 kW. Il metodo restituisce un **`ListBuffer[Car]`** contenente le auto che soddisfando il criterio di filtraggio definito tramite **`filter`**. Da notare l'uso dell'operatore **_** (underscore) per rappresentare il generico elemento della lista. Si è ovviamente reso necessario dereferenziare il singolo elemento della collezione accedendo al suo peso tramite *metodo getter* **`getPotenza`**.

```
def autoSuperbollo(flotta: ListBuffer[Car]): ListBuffer[Car] = {
    flotta.filter(_.getPotenza > 184)
}
```

18. Utilizzo combinato di `map` e `reduce`

L'utilizzo combinato dei costrutti map e reduce ha reso possibile la determinazione della massima potenza tra tutte le auto presenti nella flotta implementando due operazioni:

- **`map`**: mapping della potenza in kW su potenza in CV, ottenendo una collezione del tutto identica a quella in input ma con valore di potenza convertito da chilowatt in cavalli.
- **`reduce`**: riduzione ad un unico valore di potenza, ovvero quello dell'auto con la potenza massima. Da notare come non sia stato effettivamente utilizzato il costrutto **`reduce`**, ma per una buona motivazione. La **`reduce`** permette di ottenere in output lo stesso tipo di dato del singolo elemento contenuto nella collezione; lavorando con una collezione di `Car` e dovendo restituire un valore numerico **`Int`** (la potenza massima) è stato obbligato l'utilizzo di una **`fold`**, in questo caso una **`foldRight`** (che assume il medesimo ruolo della reduce).

Si è in particolare definita la funzione **`map_kwToCV()`** per effettuare la conversione tra kW e CV data in input alla **`map`**.

```
def MapReduce(flotta: ListBuffer[Car]): Int = {
    def map_kwToCV(veicolo: Car): Car = {
      veicolo.setPotenza((veicolo.getPotenza * 1.36).toInt)
      veicolo
    }

    def reduce_maxPower(flottaCV: ListBuffer[Car]): Int = {
           flottaCV.foldRight(flottaCV.head.getPotenza)((veicolo, temp)
=> temp max veicolo.getPotenza)
    }

    reduce_maxPower(flotta.map(map_kwToCV))
}
```

Progetto ASMETA

Progetto ASMETA

Il progetto ASMETA prevede l'implementazione di una Abstract State Machine (ASM) in grado di simulare un semaforo.

Il semaforo è rappresentato da 3 colori:

- VERDE: permette il passaggio delle auto
- ARANCIONE: avvisa gli automobilisti dell'imminente arrivo del ROSSO
- ROSSO: impedisce il passaggio delle automobili

La durata associata a ciascuno dei 3 colori è stata definita sulla base del fatto che il colore VERDE ha generalmente una durata superiore rispetto al colore ROSSO ed il colore ARANCIONE presenta la durata minima, in particolare:

- VERDE: durata 20 secondi
- ARANCIONE: durata 5 secondi
- ROSSO: durata 10 secondi

All'avvio, il semaforo assume uno qualsiasi dei 3 colori; il colore assunto all'inizializzazione del semaforo non è infatti rilevante per il corretto funzionamento dello stesso. Ciò che conta davvero è l'avvicendamento dei colori durante il funzionamento, che deve avvenire secondo il seguente schema:

... → VERDE → ARANCIONE → ROSSO → VERDE → ...

Si riporta ora una breve descrizione delle componenti del file di specifica del semaforo, denominato **`Stoplight.asm`** e dello scenario *Avalla* con relativa simulazione.

1. **Dichiarazione di domini e funzioni**

Si riporta la dichiarazione dei domini e delle funzioni utilizzati nell'implementazione del semaforo.

```
signature:
      enum domain Colore = { VERDE | ARANCIONE | ROSSO }
      domain Seconds subsetof Integer

      dynamic controlled stoplightColor: Colore
      dynamic controlled time: Seconds
```

```
    dynamic controlled action : String

    derived f_colorDuration: Colore -> Seconds
    derived f_nextColor: Colore -> Colore
```

2. Definizione delle funzioni, delle regole, macro-regole e regola main

Si riporta la definizione delle funzioni, delle regole, della macro-regole e della regola main, principale per la ASM.

```
definitions:
    function f_colorDuration($colore in Colore) =
        if ($colore = VERDE) then 20
        else if ($colore = ARANCIONE) then 5
        else 10 endif endif

    function f_nextColor($coloreAttuale in Colore) =
        if ($coloreAttuale = VERDE) then ARANCIONE
        else if ($coloreAttuale = ARANCIONE) then ROSSO
        else VERDE endif endif

    rule r_cambioColore =
        par
            time := f_colorDuration( f_nextColor(stoplightColor) )
            stoplightColor := f_nextColor(stoplightColor)
            action := "change_next_color"
        endpar

    macro rule r_decrementaUnSecondo =
        if (time > 0) then
            par
                time := time - 1
                action := "decremento_unsecondo"
            endpar
        else r_cambioColore[]
        endif

    macro rule r_stoplightInitialize =
        par
        if (time = -1) then
            choose $colore in Colore with true do
            time := f_colorDuration($colore)
        endif
```

```
                r_decrementaUnSecondo[]
            endpar

    main rule r_Main = r_stoplightInitialize[]
```

3. Inizializzazione e valori di inizializzazione

All'avvio del semaforo, esso viene inizializzato scegliendo un qualsiasi colore tra i 3 disponibili. In particolare, il semaforo non attivo è identificabile notando che il tempo è impostato a -1.

```
default init initialize:
    function time = -1
```

4. Scenario `ROAD_SCENARIO.avalla` e output simulazione

Si riporta ora una breve descrizione dello scenario *Avalla* e di una porzione del risultato della sua simulazione.

```
//// starting scenario
scenario ROAD_SCENARIO
load Stoplight.asm
check time = -1;
check stoplightColor = undef;
step
check time = 20;
check stoplightColor = VERDE;
step
check time = 19;
check stoplightColor = VERDE;
step
check time = 18;
check stoplightColor = VERDE;
step
check time = 17;
check stoplightColor = VERDE;
step
check time = 16;
check stoplightColor = VERDE;
...
```

La simulazione ha esito positivo ed il suo output a console è il seguente:

```
path
C:/Users/ghisl/Documents/GitHub/1052975_progetti_modulo_programmazione/A
SMETA/specifiche/ROAD_SCENARIO.avalla
parsing file
C:\Users\ghisl\Documents\GitHub\1052975_progetti_modulo_programmazione\A
SMETA\specifiche\Stoplight.asm
file successfully parsed for asm Stoplight
parsing file
C:\Users\ghisl\AppData\Local\Temp\__tempAsmetaV1542155503383304380.asm
file successfully parsed for asm __tempAsmetaV1542155503383304380

** Simulation **

<Run>
<Transition>
check succeeded: time = -1
check succeeded: stoplightColor = undef
<UpdateSet - 0>
action=change_next_color
result=1
step__=1
stoplightColor=VERDE
time=20
</UpdateSet>
<State 1 (controlled)>
action=change_next_color
result=1
step__=1
stoplightColor=VERDE
time=20
</State 1 (controlled)>
clear monitored vars
</Transition>
<Transition>
check succeeded: time = 20
check succeeded: stoplightColor = VERDE
<UpdateSet - 1>
action=decremento_unsecondo
result=1
step__=2
time=19
</UpdateSet>
<State 2 (controlled)>
action=decremento_unsecondo
```

```
result=1
step__=2
stoplightColor=VERDE
time=19
</State 2 (controlled)>
clear monitored vars
</Transition>
<Transition>
check succeeded: time = 19
check succeeded: stoplightColor = VERDE
<UpdateSet - 2>
action=decremento_unsecondo
result=1
step__=3
time=18
</UpdateSet>
<State 3 (controlled)>
action=decremento_unsecondo
result=1
step__=3
stoplightColor=VERDE
time=18
</State 3 (controlled)>
clear monitored vars
</Transition>
<Transition>
check succeeded: time = 18
check succeeded: stoplightColor = VERDE
<UpdateSet - 3>
action=decremento_unsecondo
result=1
step__=4
time=17
</UpdateSet>
<State 4 (controlled)>
action=decremento_unsecondo
result=1
step__=4
stoplightColor=VERDE
time=17
</State 4 (controlled)>
clear monitored vars
</Transition>
<Transition>
check succeeded: time = 17
check succeeded: stoplightColor = VERDE
```

```
<UpdateSet - 4>
action=decremento_unsecondo
result=1
step__=5
time=16
```

5. Esempio di animazione AsmetaA

Si fornisce una screenshot del risultato dell'animazione del modello ASM su 10 step.