
Relazione progetto Programmazione a Oggetti

Titolo: *Eng-u-ine*

Membri del gruppo e numero di matricola:

Piccoli Marco, 2045039

Pivetta Federico, 2009693



Indice generale

Introduzione.....	3
Descrizione del modello.....	4
Polimorfismo.....	6
Persistenza dei dati.....	7
Funzionalità implementate.....	8
Rendicontazione ore.....	9
Suddivisione delle attività progettuali.....	10

Introduzione

Il progetto di Programmazione a Oggetti dell'anno accademico 2023/2024 tratta del seguente tema: i sensori.

Per questo, abbiamo deciso che il nostro lavoro sarà incentrato sulla creazione di *Eng-u-ine*.

Eng-u-ine è una applicazione che permette, ai normali utenti, la visualizzazione di sensori che fanno riferimento allo stato di ogni singolo componente all'interno di una automobile visualizzando, per ognuno, un grafico che mostra l'andamento di tale stato negli ultimi 12 mesi.

L'utente all'interno di *Eng-u-ine* può personalizzare a suo piacimento i sensori inserendone di nuovi con all'interno dei campi preimpostati, ma anche modificare quelli già esistenti oppure eliminarli.

L'utente in fase di creazione dei sensori può scegliere su una lista di 5 sensori:

- sensore *Batteria*: è un sensore che monitora il tempo di ricarica della batteria nel tempo;
- sensore *Consumo*: è un sensore che monitora i consumi del veicolo nel tempo;
- sensore *Gas*: è un sensore che monitora i gas prodotti dai tubi di scarico per rendere consapevole l'utente dell'impatto ambientale;
- sensore *Motore*: è un sensore che monitora la composizione e la corretta accensione del motore;
- sensore *Pneumatico*: è un sensore che rileva l'usura dei pneumatici con il passare dei mesi/anni

Descrizione del modello

Il modello logico di questo progetto si divide in due parti: la prima parte comprende la gestione dei vari sensori partendo da una classe astratta *Sensor* e le sue sottoclassi come si nota in Figura 1, mentre la seconda comprende l'interfaccia che andrà ad implementare le funzionalità come la ricerca, la creazione, la modifica e l'eliminazione di tali *Sensor*.

Alla classe astratta *Sensor* e alle sue sottoclassi vengono aggiunte delle classi di servizio per convertire, in formato JSON (e viceversa), e salvare su file i risultati ottenuti dai diversi sensori. Per tale scopo si è deciso di utilizzare gli strumenti offerti da Qt, in particolare il *QjsonObject*.

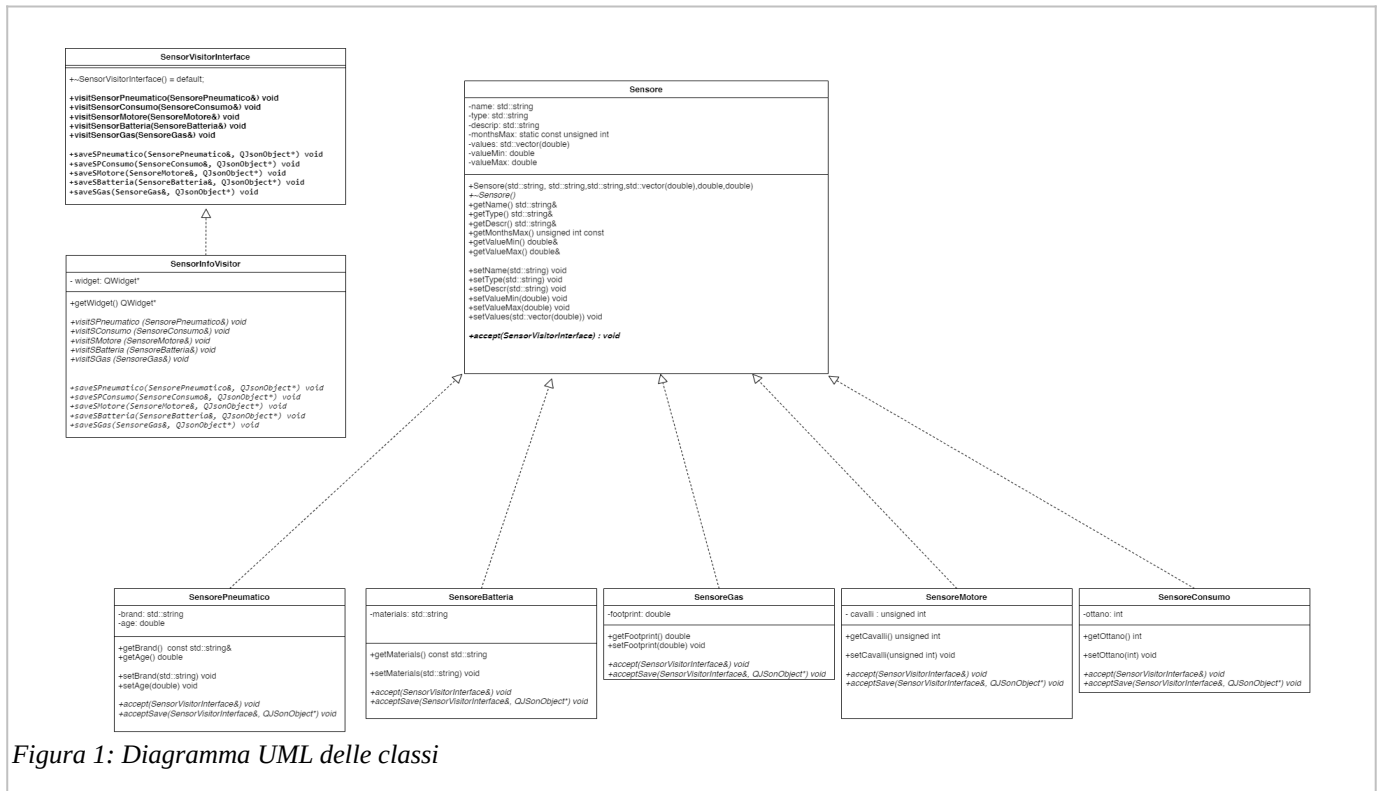


Figura 1: Diagramma UML delle classi

Tutta la struttura gerarchica parte dalla classe astratta *Sensor* che, al suo interno, contiene tutte le informazioni comuni ai sensori. Tali informazioni sono:

- nome
- tipologia di sensore
- descrizione
- un vettore contenente i valori del sensore
- il valore minimo e il valore massimo di tale vettore

Per ognuno di questi campi vengono implementati i metodi *getter* e *setter*.

Da *Sensor* vengono definite cinque classi concrete che rappresentano, come descritto nell'Introduzione, i sensori veri e propri specificando, per ognuno di essi, dei campi e delle classi aggiuntive per differenziarne l'utilizzo.

I sensori si dividono in cinque categorie:

- *Motore*: viene aggiunto il campo *cavalli* che indica il numero di CV che il motore può produrre a pieno regime;
- *Gas*: viene aggiunto il campo *footprint* che indica l'impatto ambientale che ha il gas prodotto dall'automobile;
- *Consumo*: viene aggiunto il campo *ottano*, che indica il numero di ottani che ha l'idrocarburo consumato (es. diesel = 25, benzina = 84 etc...);
- *Batteria*: viene aggiunto il campo *materials*, che indica con che materiale è composta la batteria della macchina (es. litio, nichel etc...);
- *Pneumatico*: vengono aggiunti i campi *brand* e *age* che indicano, rispettivamente, la marca e gli anni del / degli pneumatici montati.

Per ognuno dei campi aggiunti vengono implementati i metodi *getter* e *setter*.

Poiché ogni sensore specifico all'interno del modello si comporta come un **Data Transfer Object** (DTO) e non dispongono di nessuna funzione rilevante, abbiamo scelto di utilizzare il design pattern dei *Visitor* per consentirne l'arricchimento in maniera dinamica.

Per questo scopo abbiamo realizzato la classe *SensorVisitorInterface* che si comporta da interfaccia per tutti i *Visitor* a cui vengono aggiunti tutti i metodi virtuali puri; tali metodi vengono implementati dalla classe *SensorInfoVisitor*. Quest'ultima serve per consentire sia la modifica sia la ricezione delle singole informazioni all'interno di ciascun sensore. Di conseguenza, in *Sensor* vengono aggiunti i metodi virtuali puri *accept* e *acceptModify* che viene implementato da ogni singolo sensore per poter accettare la richiesta di un *SensorInfoVisitor*.

Polimorfismo

L'utilizzo principale del polimorfismo, all'interno del nostro progetto, riguarda il *design pattern Visitor* nella gerarchia *Sensor*. Quest'ultimo viene utilizzato in primo luogo per poter visualizzare nella sezione Info presente in ogni *SensorPanel* non soltanto i campi ereditati dalla classe base astratta *Sensore* come ad esempio: nome, tipo, descrizione, ecc. ma anche i campi dati che appartengono unicamente alle classi concrete, così da evitare l'utilizzo di RTTI o `dynamic_cast`. Viene anche utilizzato per salvare all'interno di un file JSON i sensori presenti nella nostra applicazione con i loro campi dati unici.

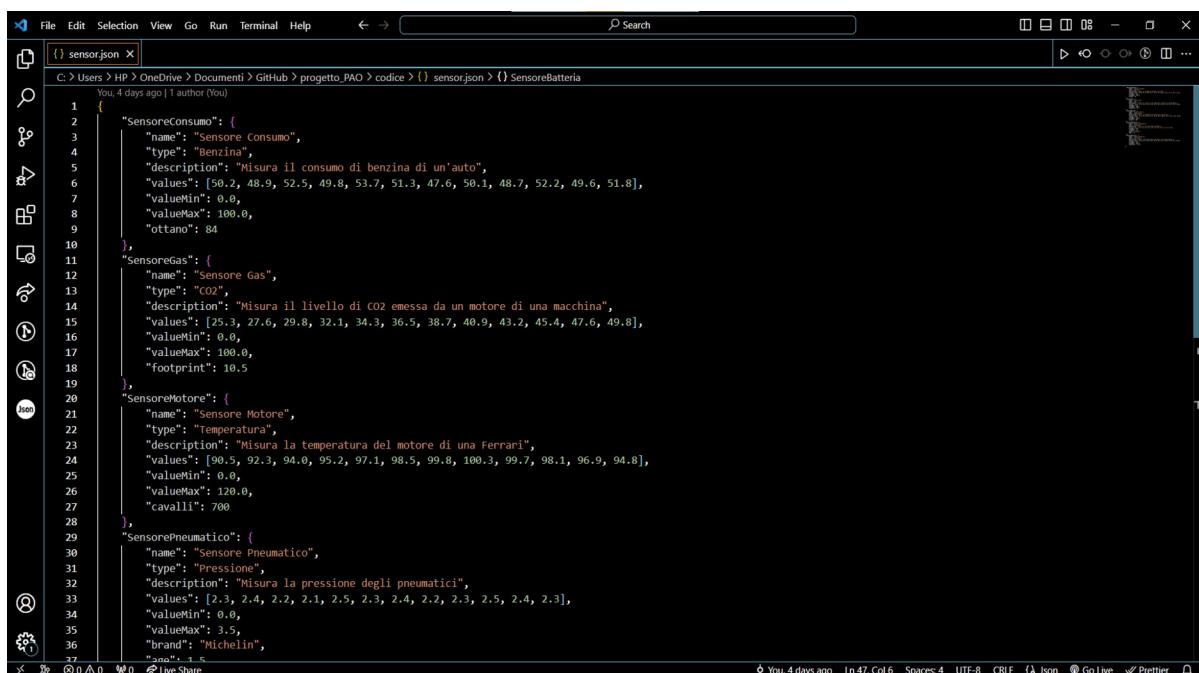
Persistenza dei dati

Per la persistenza dei dati del nostro progetto abbiamo deciso di utilizzare il formato strutturato *JSON*, questo perché utilizzando le librerie standard di Qt si possono scrivere e/o leggere file *JSON* per salvare dati senza troppe difficoltà. Nella nostra applicazione abbiamo implementato una funzione che costruisce dei sensori a partire dalle informazioni presenti all'interno di un file *.json*:

```
Sensore* createSensorFromJson(const QString& sensorName, const QJsonObject& sensorObject) {...}
```

che ritorna un puntatore polimorfo alla classe astratta *Sensore*, al quale viene assegnato il sensore costruito utilizzando i dati salvati nel file altrimenti se i dati non corrispondono a nessun sensore presente nella nostra gerarchia ritorna un puntatore nullo.

L'immagine sottostante mostra un esempio di come vengono salvati i sensori all'interno di un file *.json*.



```
1 {
2   "SensoreConsumo": {
3     "name": "Sensore Consumo",
4     "type": "Benzina",
5     "description": "Misura il consumo di benzina di un'auto",
6     "values": [50.2, 48.9, 52.5, 49.8, 53.7, 51.3, 47.6, 50.1, 48.7, 52.2, 49.6, 51.8],
7     "valueMin": 0.0,
8     "valueMax": 100.0,
9     "ottano": 84
10  },
11  "SensoreGas": {
12    "name": "Sensore Gas",
13    "type": "CO2",
14    "description": "Misura il livello di CO2 emessa da un motore di una macchina",
15    "values": [25.3, 27.6, 29.8, 32.1, 34.3, 36.5, 38.7, 40.9, 43.2, 45.4, 47.6, 49.8],
16    "valueMin": 0.0,
17    "valueMax": 100.0,
18    "footprint": 10.5
19  },
20  "SensoreMotore": {
21    "name": "Sensore Motore",
22    "type": "Temperatura",
23    "description": "Misura la temperatura del motore di una Ferrari",
24    "values": [90.5, 92.3, 94.0, 95.2, 97.1, 98.5, 99.8, 100.3, 99.7, 98.1, 96.9, 94.8],
25    "valueMin": 0.0,
26    "valueMax": 120.0,
27    "cavalli": 700
28  },
29  "SensorePneumatico": {
30    "name": "Sensore Pneumatico",
31    "type": "Pressione",
32    "description": "Misura la pressione degli pneumatici",
33    "values": [2.3, 2.4, 2.2, 2.1, 2.5, 2.3, 2.4, 2.2, 2.3, 2.5, 2.4, 2.3],
34    "valueMin": 0.0,
35    "valueMax": 3.5,
36    "brand": "Michelin",
37  }
38 }
```

Figura 2: Costruzione in *JSON* degli oggetti di tipo *Sensore*

Funzionalità implementate

Le funzionalità implementate del nostro programma sono:

- creazione, modifica ed eliminazione di sensori
- salvataggio e conversione di sensori in formato *JSON*
- funzionalità di ricerca per pertinenza
- simulazione attraverso un grafico a linee spline
- risultati della ricerca all'interno di una *ScrollArea*

Le funzionalità grafiche sono:

- menù sempre presente in alto a sinistra
- sezioni modifica e simulazione sempre interscambiabili
- utilizzo di stili grafici per campi input e pulsanti

Come caldamente consigliato durante le lezioni dal professore Zanella, prima di iniziare a lavorare al codice del nostro progetto abbiamo stilato una lista di funzionalità che volevamo implementare aiutandoci attraverso la realizzazione di alcuni sketch mediante l'uso del software *Balsamiq Wireframes* come si può notare in Figura 3 e in Figura 4.

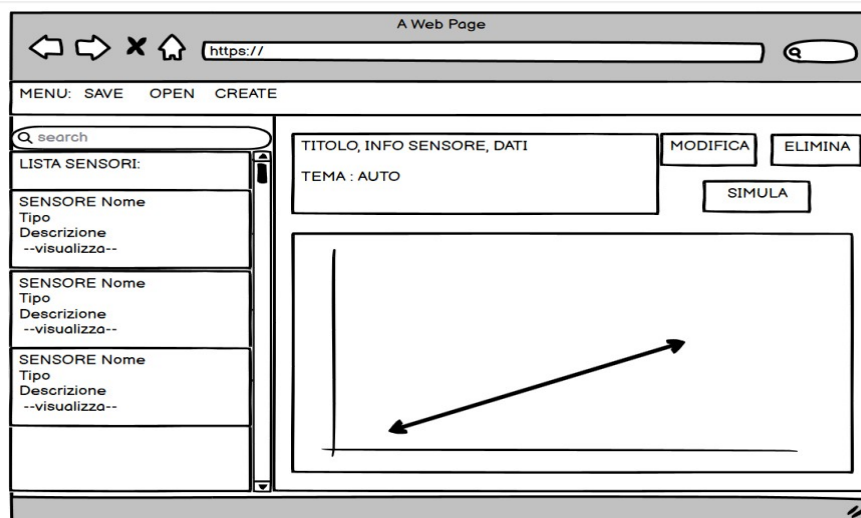


Figura 3: Markup della GUI

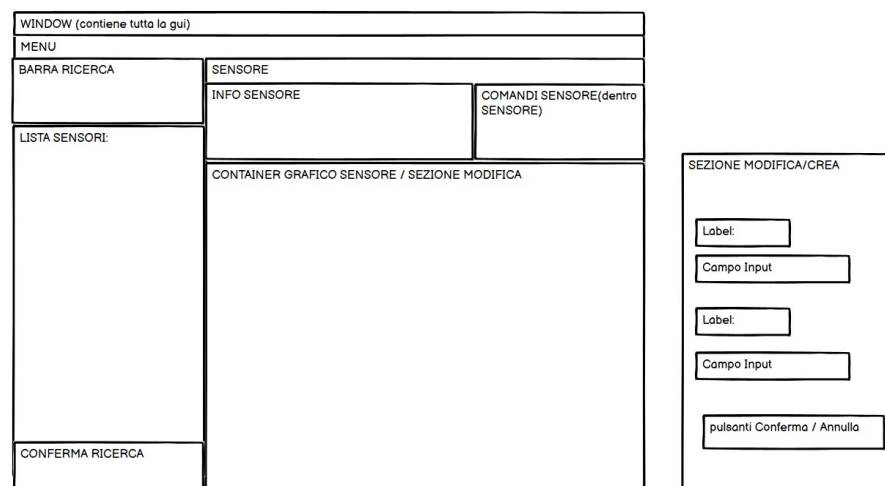


Figura 4: Markup della GUI

Rendicontazione ore

Attività	Ore Previste	Ore Effettive
Studio e progettazione	9	9
Sviluppo del codice del modello	9	10
Studio del framework Qt	10	11
Sviluppo del codice della GUI	12	14
Test e debug	6	7
Stesura della relazione	4	4
Totale	50	55

Come si può notare dalla tabella della rendicontazione delle ore abbiamo superato il monte ore previsto, questo è successo in quanto lo sviluppo del codice della GUI ci ha richiesto più tempo del previsto per essere portato a termine. Per evitare il più possibile la presenza di errori nel nostro codice abbiamo scelto di dedicare un'ora extra alla sezione Test e Debug.

Suddivisione delle attività progettuali

Come scritto nel documento di specifiche pubblicato, trattandosi il nostro di un progetto in coppia tutte le varie fasi sono state divise, per ogni singolo componente nelle seguenti parti:

- Marco Piccoli:
 - Struttura e progettazione della parte grafica della *GUI*, implementando le classi *HomePanel*, *SearchBarPanel* e *SensorPanel* come descritto in Figura 5
 - Progettazione diagramma UML
 - Progettazione di un sistema di versionamento Git condiviso tra i componenti
- Federico Pivetta:
 - Markup della GUI con il software Balsamiq
 - Struttura gerarchica della classe Sensore
 - Implementazione dei *SensorVisitor*

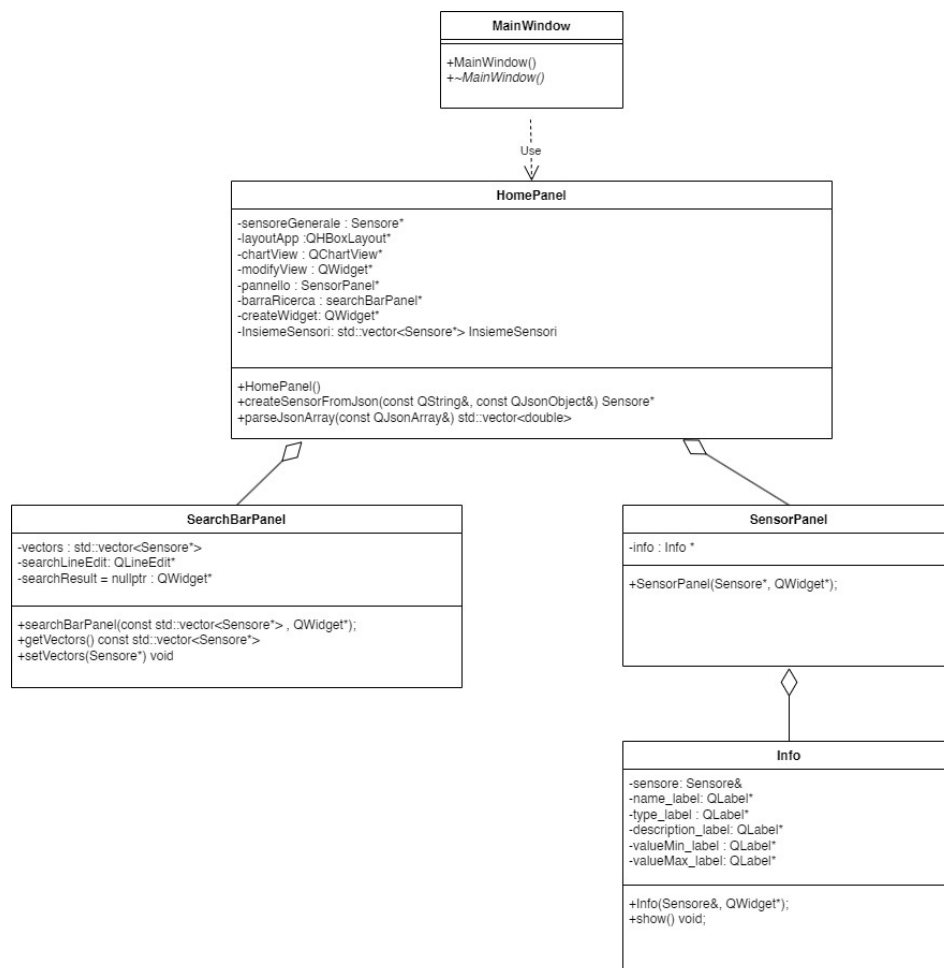


Figura 5: Struttura gerarchica della GUI