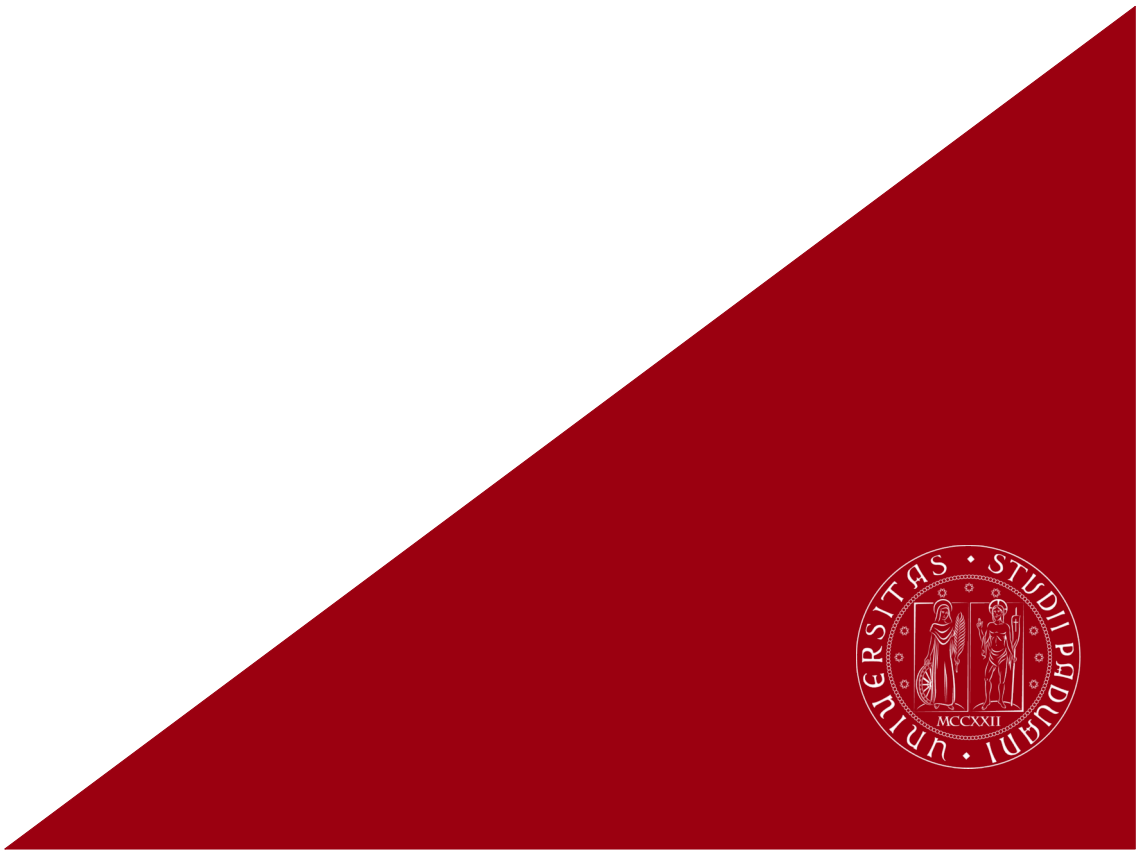
Relazione progetto Programmazione a Oggetti

Titolo: En*g-u-i*ne

Membri del gruppo e numero di matricola:

Piccoli Marco, 2045039

Pivetta Federico, 2009693



Indice generale

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 *Sensore* 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 *Sensore*.

Alla classe astratta *Sensore* 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* .

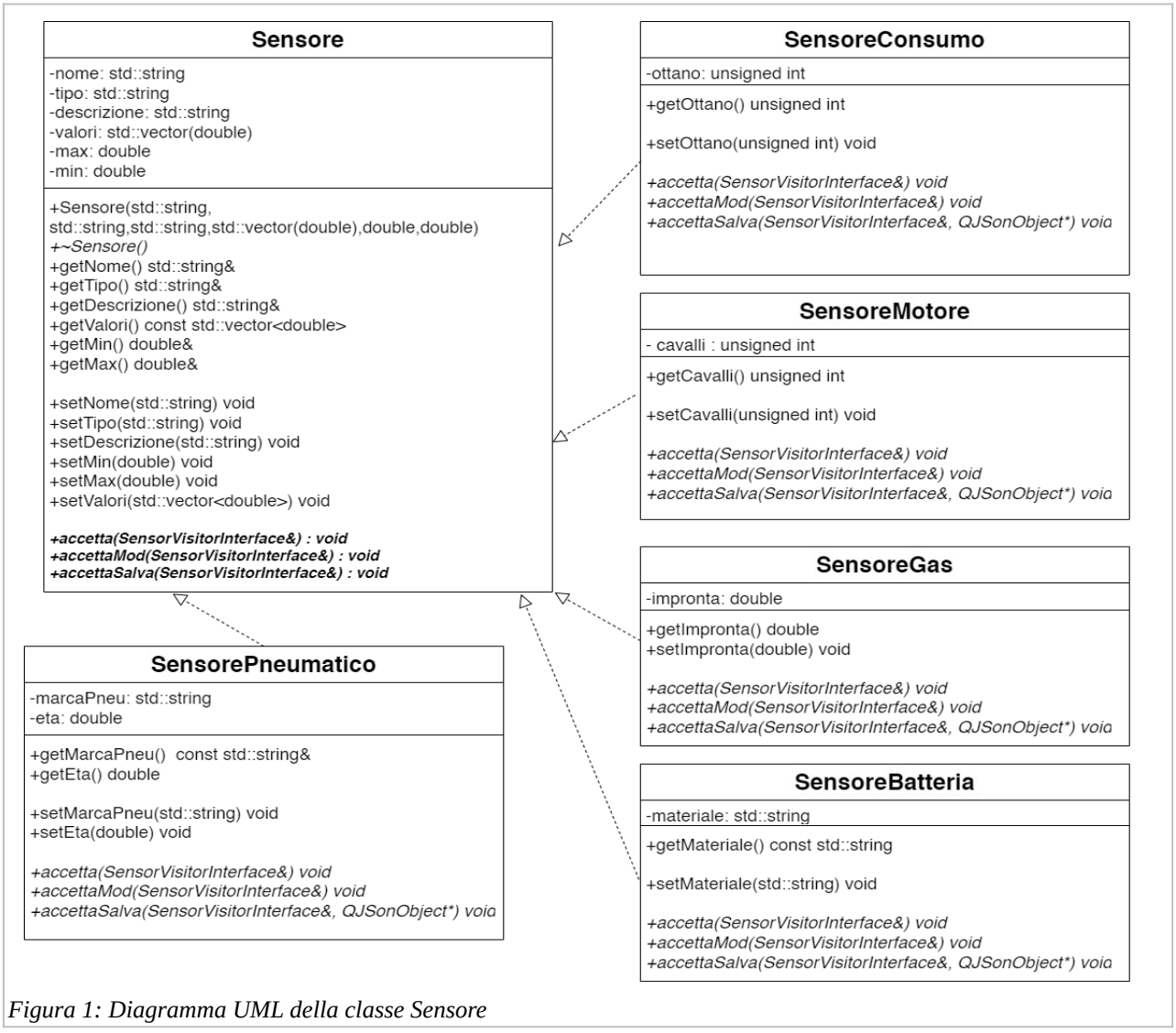


Figura 1: Diagramma UML della classe Sensore

Tutta la struttura gerarchica parte dalla classe astratta *Sensore* 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 *impronta* 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 *materiale*, che indica con che materiale è composta la batteria della macchina (es. litio, nichel etc…);
- *Pneumatico*: vengono aggiunti i campi *marcaPneu* e *eta* 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 ***D**ata **T**ransfer **O**bject* (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*, come descritto in Figura 2, 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 all'utente di visualizzare, modificare e/o salvare le informazioni dei sensori. Per questo motivo, alla classe *Sensore* vengono aggiunti tre metodi virtuali puri: *accetta*, *accettaMod* e *accettaSalva* che verranno implementati da ogni singolo sensore per poter accettare la richiesta da parte di un *SensorInfoVisitor*.

Figura 2: Diagramma UML del patter Visitor

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

che ritorna un puntatore polimorfo, appartenente alla classe astratta *Sensore*, al quale viene assegnato il sensore costruito utilizzando i dati salvati nel file. Oltre alla classe `creaSensDaJson(…)`, abbiamo implementato un'altra funzione a supporto della prima che permette la conversione dei valori, scritti nel file .*JSON*, in *std::vector*

In caso i dati non corrispondessero a nessun sensore presente nella nostra gerarchia, tale funzione ritorna un puntatore nullo.

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

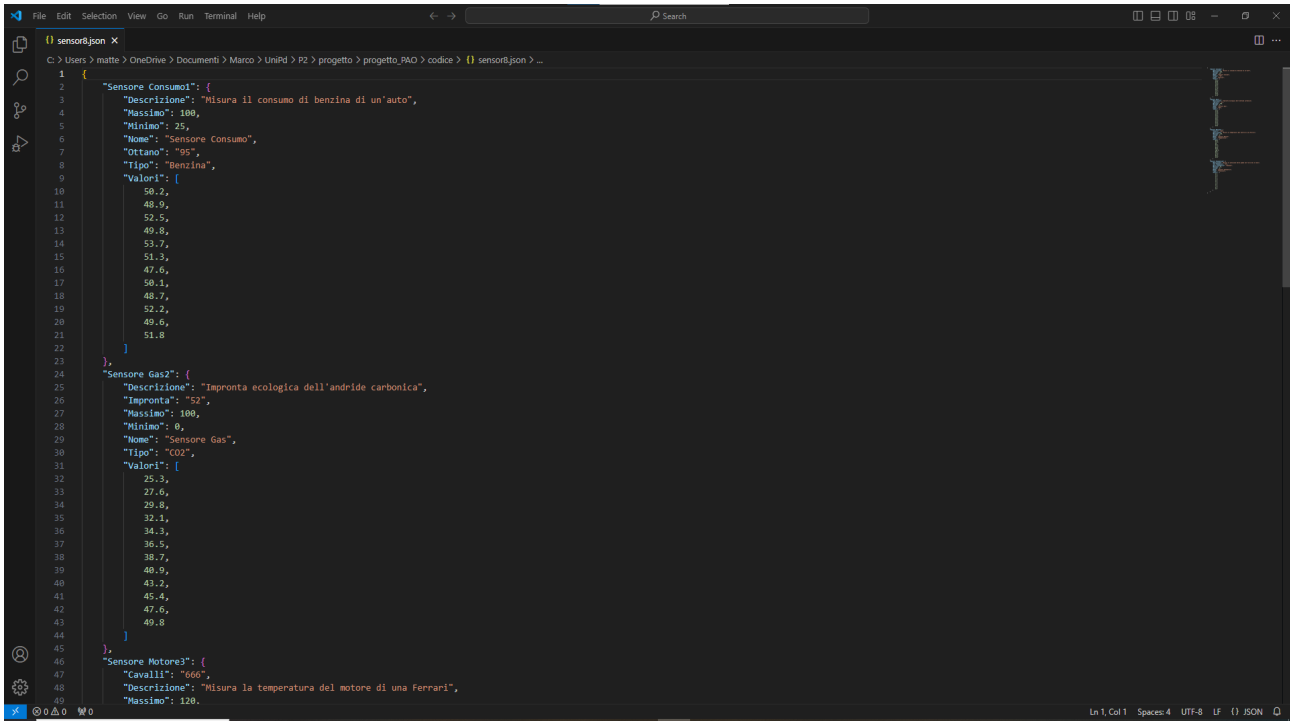


Figura 3: Costruzione in JSON degli oggetti di tipo Sensore

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 lezione 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 4 e in Figura 5.

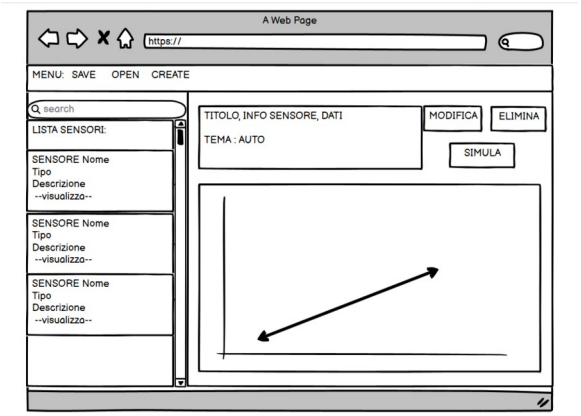


Figura 4: Markup della GUI

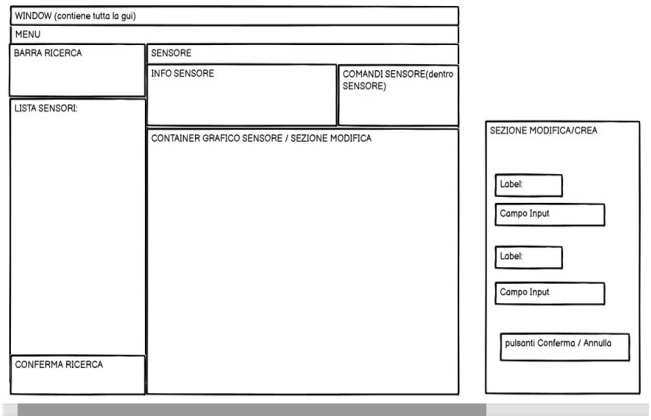


Figura 5: Markup della GUI

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	8	15
Stesura della relazione	4	7
Totale	52	66

Come si può notare dalla tabella della rendicontazione delle ore abbiamo superato il monte ore previsto, questo perché lo sviluppo del codice della *GUI* ci ha richiesto più tempo del previsto per essere portato a termine, oltre alla implementazione di nuove *feature* per migliorare l'applicazione sotto tutti gli aspetti sia grafici sia funzionali.

Inoltre, per evitare il più possibile la presenza di errori e/o *warning* all'interno del nostro codice abbiamo deciso di dedicare delle ore extra alla sezione Test e Debug, così da evitare spiacevoli inconvenienti e avere (in futuro) un codice pulito e riutilizzabile all'istante, vista la totale validità del codice, senza dover perdere ulteriore tempo.

Differenze rispetto alla consegna precedente

Rispetto alla consegna precedente abbiamo implementato e cambiato diverse funzionalità del nostro progetto, in primis concentrandoci sul risolvere il vincolo principale, come indicato nel feedback, e successivamente andando ad implementare nuovi tipi di stili e rendendo più intuitiva la *GUI*.

Come suggerito nel feedback abbiamo risolto il mantenimento della separazione netta tra modello logico e interfaccia grafica andando ad implementare il *pattern Model-View* con annessa una nuova classe *modello* (come descritta in Figura 6), quest'ultima ha il controllo totale sulla creazione, modifica ed eliminazione dei 5 tipi di sensori nonché l'apertura, il salvataggio e la conversione di tali sensori su file di tipo JSON, avendo a disposizione un vettore (*InsiemeSensori*) che contiene i puntatori agli oggetti di tipo *Sensore* con le relative informazioni.

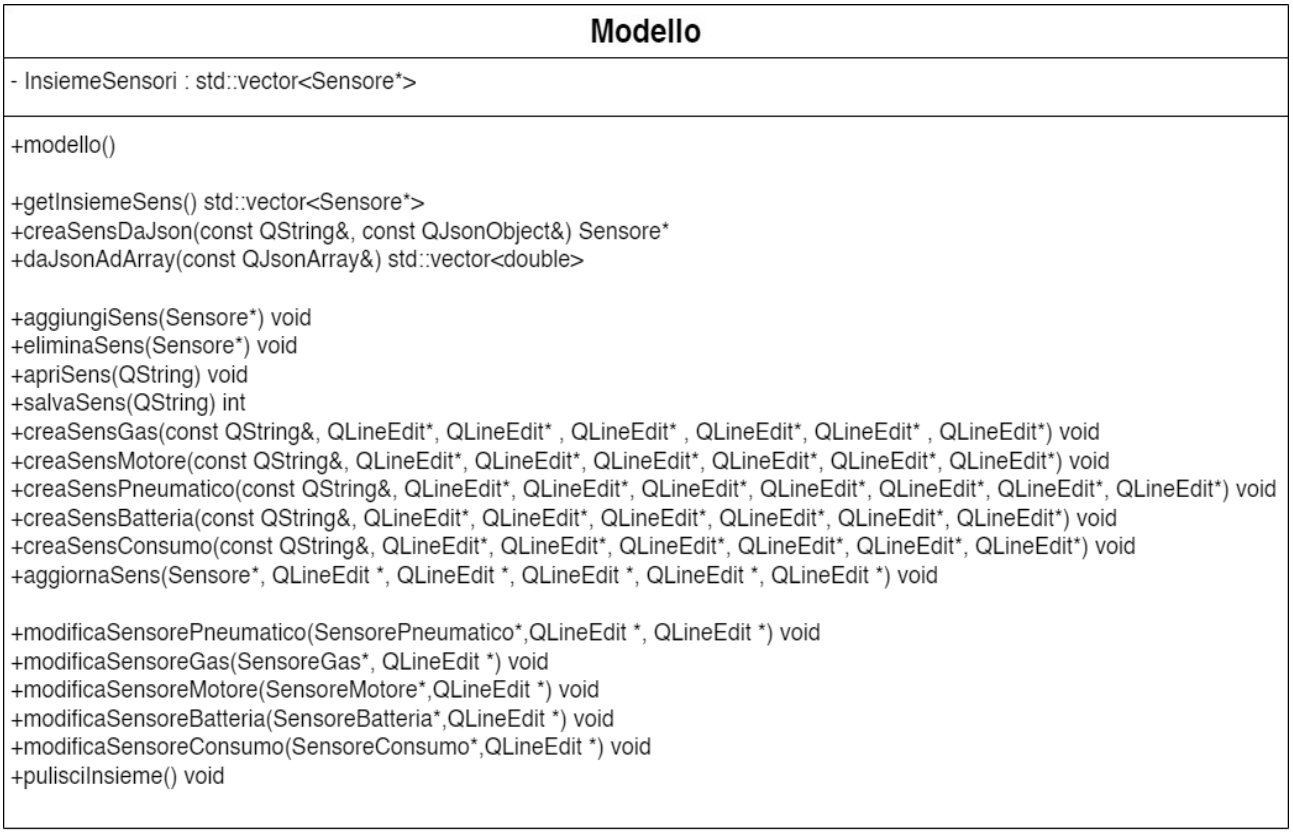


Figura 6: Schema logico della classe modello

Oltre ad aver introdotto il *pattern Model-View*, abbiamo implementato diverse funzionalità utili per l'utente per una visualizzazione a 360° dei sensori, introducendo:

- La possibilità di scegliere la più corretta, secondo l'utente, visualizzazione dei valori del sensore tramite 4 tipi di grafico (spline, linee, punti, barre)
- La possibilità di navigare tra i punti all'interno del grafico, per avere una chiara visione dei valori del sensore, avendo l'opportunità sia di muoversi sull'asse X e Y sia di ridimensionare il grafico
- La possibilità di salvare sullo stesso file *.JSON* le modifiche apportate durante l'uso dell'applicazione, ed in caso queste non vengano salvate comparirà un avviso
- La possibilità di modificare i campi specifici dei vari sensori con l'utilizzo del *pattern visitor*
- La possibilità di visualizzare tutti i sensori senza la necessità di utilizzare la barra di ricerca
- L'aggiunta di *shortcut* per i pulsanti dell'applicazione seguendo lo schema: Ctrl + iniziale parola in inglese (esempio Ctrl + S = Save , Ctrl + M = Modify, Ctrl + C = Create, etc…)

Inoltre, abbiamo aggiunto a tutti gli oggetti grafici dei fogli di stile per rendere più agevole l'utilizzo dell'applicazione, in particolare abbiamo applicato l'effetto *hover* al passaggio del mouse sui pulsanti, per migliorare ulteriormente la resa grafica del progetto.

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 7
  - 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* con annessi i suoi metodi

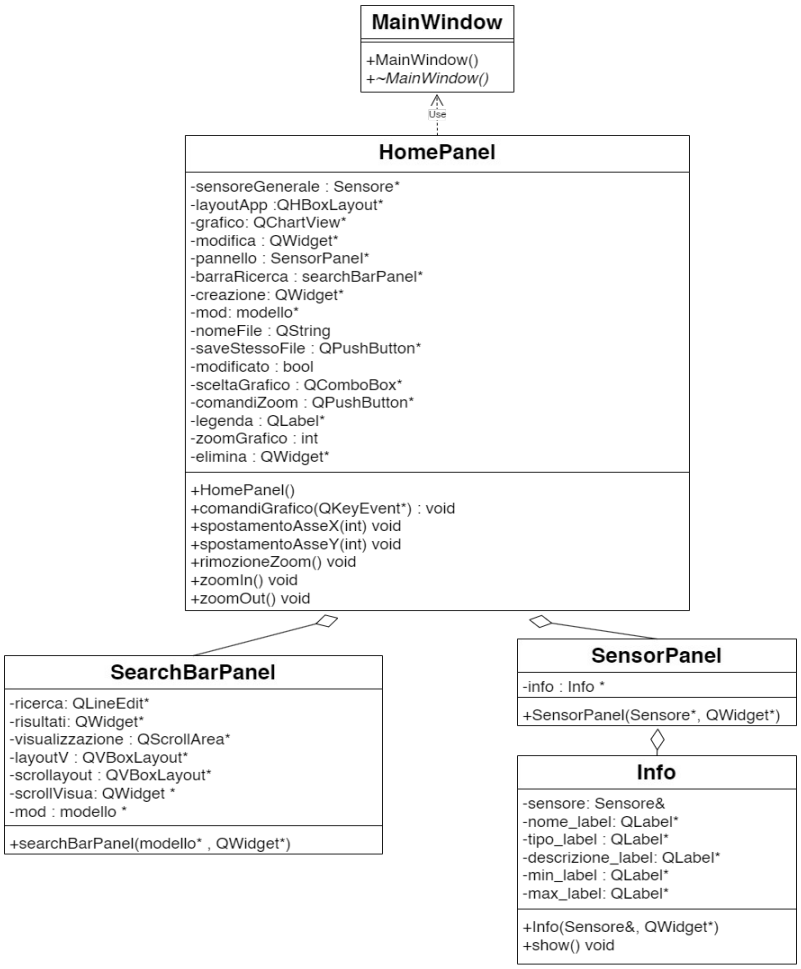


Figura 7: Struttura gerarchica della GUI