# RocketSensors

Edgar Carlos Djossa 2089654

Août 2024

## Contents

# 1 Introduzione

Il mio progetto consiste in un interfaccia grafica che permette di simulare diversi sensori all'interno di razzo a partire dati raccolti. L'interfaccia rende la possibilità di creare, ricercare,modificare ed eliminare i sensori in un amibiente virtuale.

I sensori sono di tre categorie principali: sensori ambientali (EnvSensor) che includono i sensori di Temperatura e di Pressione, sensori di livello (LevelSensor) che includono sensori di Carburante e infine i sensori di posizione (PositionSensor)

i sensori amibientali sensori raccolgono i dati sia all'interno che all'esterno del razzo, permetendo di visualizzare informazioni sulle condizioni ambientali. In particolare i sensori de Temperatura simulano le misurazioni delle variazioni termiche asssicurando che questi variazioni rimangono entro limiti sia minimi che massimi accetabili. Mentre i sensori di Pressione raccolgono i variazioni della pressione interna oppure esterna.

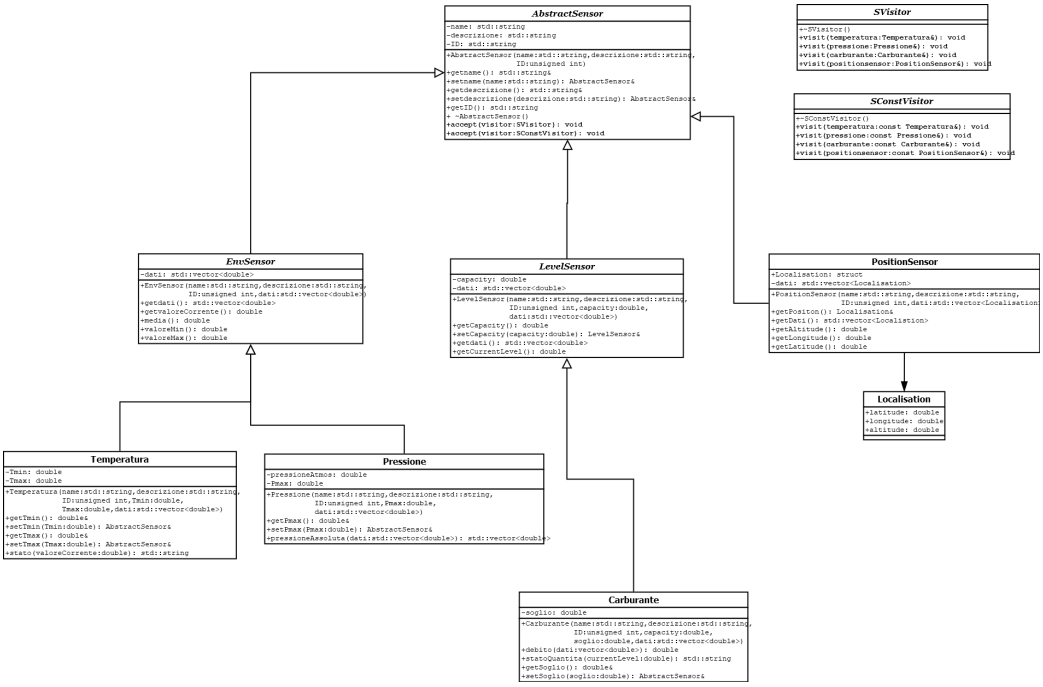
I sensori di livello, sensori di Carburante simulano l'evoluzione della quantità di carburante disponibile, permenttendo di gestire le riserve grazie alla presenza di un soglio di allerta in caso di insufficenza.

Infine, i sensori di posizione simulano grazie ai dati raccolti (longitudine, latitudine e altitudine ) la determinazione della posizione del razzo nello spazio fornendo informazioni necessarie per la navigazione.

Oltre le funzionalità elencati prima, l'utente ha la possibilità di caricare e salvare i sensori in un file Json, aggiungere un nuovo sensore oppure eliminare uno da questo file. Al carico du file json si può visualizzare la lista di tutti sensori presenti, ricercarli per nome oppure per ID. Su ogni sensore si può visualizzare vari informazione, modificare i parametri, avviare una simulazione e puoi cancellare il sensore.

# 2 Descrizione del modello

Il modello implementato in questo progetto si basa su una struttura ad oggetti, cioè le classi rappresentano i vari sensori con le loro propriétà riportate nel diagramma in seguito. Ci sono anche alcune classi di servizio per convertire i prodotti nel formato JSON (e viceversa) e salvarli su file. A tale scopo vengono utilizzati gli strumenti offerti da Qt, in particolare il QjsonObject.




Il modello si compone di diversi classi principali:

- **AbstractSensor**: Il modello parte propio di questa classe base astratta da cui derivano tutte le altre classi di sensori. Definisce le informazioni comun a tutti i sensori ovvero identificatore ID univoco, nome, descrizione, per ciascuno dei quali sono implementati metodi getter e setter. Abbiamo anche due metodi accept per accettare la visita dei visitor(const e non const).

- **EnvSensor**: Questa classe anche astratta estende AbstracSensor e rapprensenta i sensori ambientali. Possiede come attributo il vettore

di dati e implementa vari metodi come getDati, getValoreCorrente che restituisce l'ultimo dato raccolto e media che restituisce la media dei dati raccolti. Due classi concrete derivano da EnvSensor:

  - **Temperatura**: rapprensenta un sensore di temperatura. Oltre alle proprietà ereditate, ha come attributi Tmin e Tmax, che definiscono i valori minimi e massimi di soglia d'accettabilità per la temperatura rilevata. Questi valori permettono di determinare lo stato della temperatura (basso , normale oppure elevato). I metodi getter e setter opportuni sono aggiunti.
  - **Pressione**: rappresenta un sensore di pressione. Ha come attributi Pmax che rappresenta il valore massimo di soglia d'accettabilità e anche pressioneAtmos che rappresenta il valore de la pressione atmosferica. Essendo variabile per altitudine si è scelto di non renderlo statico. I dati raccolti sono dei valori relativi quindi si è agguinto un metodo pressioneAssoluta che restituisce i dati come valori assoluti. I metodi getter e setter opportuni sono aggiunti.

- **LevelSensor**: Anche questa classe estende AbstractSensor e modella i sensori di livello. Oltre il vettore di dati, ha come attributo capacity che definisce la capacità del serbatoio a cui è collegato il sensore. Deriva la classe concreta:

  - **Carburante**: rappresenta un sensore di livello del carburante. Include l'attributo soglio che rapprensenta il livello critico del carburante. ha come metodi statoQuantità e debito che calcola il debito...

- **PositionSensor**: questa classe concreta deriva direttamente da AbstractSensor e rappresenta i sensori di posizione. Include una struct Localisation, comprendendo tre attributi Altitudine, Longitudine e Latitudine. Quest'ultimo permette poi di salvare i dati in un vettore di Localisation. I metodi getter e setter sono aggiunti.

Il modello è progettato per essere facilmente estendibile, permenttendo l'aggiunto di nuovi tipi di sensori senza sforzo. Possiamo pensare a dei sensori di Umidità, di qualità dell'aria come EnvSensor e sensori di livello di ossigeno per LevelSensor.

La gestione delle operazione di creazione, di modifica, di ricerca e di eliminazione sono implementate all'interno delle classi dell'interfaccia grafica per assicurarsi una chiara separazione tra la logica del modello e l'interfaccia. La finestra principale "MainWindow" è responsabile di elementi comuni come la 'MenuBar', la 'ToolBar' e la 'StatusBar', che offrono un accesso semplice e immediato alle principali funzionalità dell'applicazione. Mentre le altre funzionalità vengono gestiti da Widget figli come "searchWidget",che si occupa della ricerca dei sensori, "sensorsPanel" che permette di visualizzare l'elenco dei sensori disponibili. Per quanto riguarda la creazione e la modifica dei sensori, queste sono gestite dal widget 'EditWidget', che è stato progettato per essere flessibile Quando l'utente salva un nuovo sensore o modifica uno esistente, il salvataggio viene gestito tramite una finestra di dialogo "QDialog", che consente all'utente di confermare le modifiche. Una volta creato o selezionato un sensore, è possibile avviare la simulazione dei dati tramite "SensorDetails". Se l'utente decide di simulare i dati, viene emesso un segnale che lancia il 'ChartPanel', una componente che utilizza 'QtCharts' per rappresentare graficamente i dati raccolti.

Per la creazione dei grafici nell'applicazione, ho utilizzato il framework QtCharts, impiegando diverse classi per visualizzare i dati in modo efficace. Per i sensori ambientali, ho utilizzato QLineSeries per tracciare le variazioni nel tempo. Per il sensore di livello, ho implementato QPieSeries e QPieSlice per rappresentare visivamente i volumi utilizzati e rimanenti. Per il sensore di posizione, ho adottato QScatterSeries per mostrare longitudine e latitudine e un ulteriore QLineSeries per visualizzare le variazioni di altitudine nel tempo.

# 3 Polimorfismo

L'utilizzo principale del polimorfismo riguarda il design pattern Visitor nella gerarchia AbstractSensor. ho usato le Visitor nella generazione dei grafici, per la costruzione dei widgets per mostrare le informazioni su ogni sensore, la visualizzazione dei valori correnti direttamente nel panello dei sensori, per la conversione in Json.

- **Generazione dei grafici**: Il "SensorDetails" chiama il costruttore di ChartPanel che a sua volta invoca un visitor chiamato 'ChartVisitor', che estende 'SVisitor' e seleziona i metodi appropriati per ciascun tipo di sensore (EnvSensor, LevelSensor, PositionSensor). Quindi abbiamo diversi tipi di grafici per ciascuno sensore: un grafico a linea per i sensori

ambientali, un grafico a torta per i sensori di livello e due grafici per i sensori di posizione, una a dispersione per l'evoluzione della latitudine e della longitudine e un altro a linea per la variazione dell'altitudine.

- **visualizzazione dei valori correnti**: Per poter visualizzare l'ultimo valore (oppure il valore corrente) raccolto dal sensore senza bisogno di cliccare su, ho implementato un visitor "CurrentValueVisitor" che alla costruzione di un widget di sensore crea un visitor per che chiama i metodi get delle classi di sensori che ritornano questi valori completandoli con l'unità di misura corrispondente.

- **Informazioni su sensore**: Ho usato anche un visitor SensorDetailsVisitor per poter generare un widget che contegono informazioni su ogni tipo di sensore. Questo visitor permette di personalizzare la visualizzazione delle informazioni in base al tipo specifico di sensore,

Ho implementato degli ulteriori visitor per la gestione del polimorfismo per la persistenza dei dati file json (JsonVisitor derivata da SConstvisitor), e per l'agguinto delle icone e dei tipi dei sensori nel sensorWidget(TypeAndIconVisitor).

L'uso del polimorfismo con il pattern Visitor rende il codice più modulare e facile da mantenere. Inoltre, permette di aggiungere nuove funzionalità e di personalizzare l'interfaccia utente in modo semplice, senza dover cambiare il design già esistente.

# 4 Persistenza dei dati

Per la persistenza dei dati viene utilizzato il formato JSON, un unico file che contengono tutti i sensori di una sessione sotto forma di vettore di oggetti. I sensori sono gestiti tramite l'agguinto di un attributo "type". I dati casuali per la simulazione dei sensori sono anch'essi inclusi. Ho collegato 4 file Json che contengono varie sensori, in particolare il file Dati1.json che offre una vasta gamma di sensori con dati rappresentativi di diverse situazioni.

# 5 Funzionalità implementate

Ecco Le funzionalità implementate. Sono, per semplicità, suddivise in due categorie: funzionali ed estetiche. Le prime comprendono:

- gestione di quattro tipologie di sensori
- conversione e salvataggio in formato JSON
- funzionalità di ricerca
- funzionalità di filtro secondo il tipo di sensore

Le funzionalità grafiche:

- visualizzazione dei sensori sotto forma di griglia a tre colonne
- visualizzazione del sensore a finestra piena con il pulsante back di ritorno
- visualizzazione del valore recente raccolto dal sensore nel widget di sensore
- visualizzazione dei sogli critici sui grafici
- barra dei menù
- Utilizzo di icone nella toolbar
- Scrociatoia da testiera: Open: Ctrl + O; Save: Ctrl + S; Save As: Ctrl + Shift + S; Add Sensor: Ctrl + N; Close: Ctrl + Q ; Search: Ctrl + Enter
- utilizzo di una toolbar attivabile/disattivabile nella view
- implementazione di una opzione di fullscreen nella view
- Utilizzo di icone nei pulsanti
- gestione del ridimensionamento
- Utilizzo di icone per i sensori
- implementazione di una StatusBar
- utilizzo di un foglio di stile QSS per applicare vari effetti grafici

| Attività | Ore Previste | Ore Effetive |
|---|---|---|
| Studio e progettazione | 10 | 10 |
| Sviluppo del codice del modello | 15 | 25 |
| Studio del framework Qt | 15 | 20 |
| Sviluppo del codice della GUI | 15 | 40 |
| Test e debug | 10 | 20 |
| Stesura della relazione | 5 | 5 |
| **totale** | 75 | 120 |

# 6 Rendicontazione ore

La mia previsione iniziale sui tempi era stata abbondante rispetto alle 50 ore dichiarate, poiché è la prima volta che faccio un progetto di programmazione nella mia vita. Non avevo un'idea chiara di come quantificare le ore necessarie. Ho perso tempo nello studio di Qt, che alla fine ho imparato più facilmente con la pratica. Inizialmente, ho sottovalutato il lavoro necessario, sviluppando un modello con 7 classi di sensori concreti, il che ha aumentato significativamente il carico di lavoro. Ho quindi deciso di ridurre a 4 classi durante lo sviluppo dell'interfaccia grafica, motivo per cui il progetto mi ha richiesto circa 40 ore. Per quanto riguarda la fase di Test e Debug mi sono confronto a due segmentation fault, che alla fine erano dei problemi legati al mio sistema Windows.

# 7 Conclusione

Con questo progetto, ho imparato molto sia sul piano tecnico che su quello organizzativo. Lavorare con Qt per creare interfacce grafiche mi ha permesso di migliorare notevolmente le mie competenze in C++, imparando meglio il linguaggio attraverso la pratica. Ho migliorato le mie conoscenze su git/github.