# AeroSimulator

**Stefani Riccardo, mat. 2068225**

**Relazione progetto Programmazione a Oggetti**

## Indice

# 1 Introduzione

Il mio progetto consiste in un'interfaccia di visualizzazione di dati raccolti da sensori. Questi sensori fanno parte di un software di un centro di collaudo di aeroplani, che ha l'obiettivo di testarli prima di metterli in produzione, facendo simulazioni della durata di una settimana. In particolare, viene collaudato il sistema di accelerazione dell'aereo (il sensore di accelerazione è infatti il sensore più importante): viene data istruzione al pilota di accelerare sempre in modo costante, e viene dunque visualizzata una linea blu del grafico dell'accelerazione se il sistema di accelerazione ha risposto bene, e una linea rossa se il sistema di accelerazione ha invece prodotto dei dati di accelerazione troppo incostanti (vi sono delle soglie che indicano un comportamento anomalo). Oltre al sensore di accelerazione, sono presenti dei sensori di distanza percorsa (in volo), di tempo di volo (cronometro) e di velocità. Ogni sensore di velocità possiede un sensore di distanza percorsa e un cronometro come sensori annidati, mentre ogni sensore di accelerazione possiede un cronometro e un sensore di velocità come sensori annidati, tramite i quali calcolare i propri dati. Questi sensori possono essere creati oppure caricati da un file Json, e possono essere selezionati da una pannello laterale dove sono elencati verticalmente. Per i sensori che possiedono dei sensori annidati, per ciascuno dei sottosensori del pacchetto viene visualizzato un grafico, in un'interfaccia che visualizza fino a 4 grafici. Un menù a tendina consentirà di scegliere la fonte di inserimento dei dati del grafico della distanza percorsa e del tempo trascorso nell'ambito della settimana di simulazione, che può essere casuale oppure può derivare da un qualche tipo di distribuzione (normale, uniforme ed esponenziale). I dati degli altri due grafici (di velocità ed accelerazione) verranno poi calcolati in relazione ai primi. La status bar in basso fornisce alcuni consigli se non si hanno idee, raccomandazioni per eventuali errori, e racconta quello che si sta facendo: è lei infatti a dare il responso finale, cioè se l'aereo è abilitato a falcare i nostri cieli o meno. Nei quattro loculi dei grafici, sono presenti rispettivamente il sensore della distanza, il cronometro, il sensore della velocità, rappresentati con tre diagrammi a barre, e il sensore dell'accelerazione, che è rappresentato da un grafico a linea. Sulle ascisse ci sono i giorni della settimana di simulazione di volo, mentre i valori rilevati dal sensore sono sulle ordinate, seppure i numeri sono nascosti per ragioni di spazio. Di default è visualizzato il primo sensore del dataset immesso tramite Json, oppure, è visualizzato il nuovo sensore aggiunto manualmente, prima che avvenga la prima simulazione. La simulazione parte settando il tipo di generazione dei dati dal menù a tendina, e poi premendo "Simula" o Ctrl+T.

# 2 Descrizione del modello

Il modello logico si articola in due parti: la gestione dei sensori, che comprende tutte le operazioni CRUD (create, read, update, e delete, + la ricerca) indicate dai requisiti, e la generazione dei dati da rappresentare nei grafici. La prima comprende sia le classi che descrivono i prodotti, riportate nel diagramma in

1, sia 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.

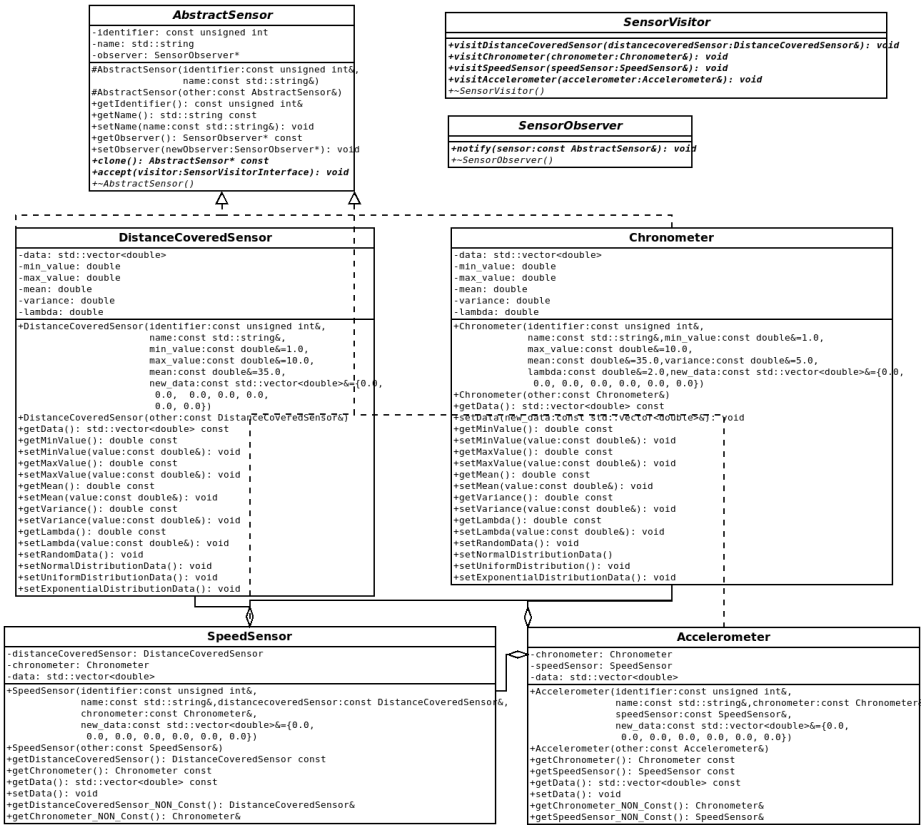


Figura 1: Diagramma delle classi del modello

Il modello parte da una classe astratta AbstractSensor che rappresenta le informazioni comuni a tutti i sensori, ovvero identificatore univoco e nome (e un puntatore a SensorObserver, di cui parlo successivamente), per i quali sono implementati metodi getter e setter. La classe concreta DistanceCoveredSensor rappresenta la classe del sensore di distanza percorsa dell'aereo in volo, mentre la classe concreta Chronometer misura il tempo di volo. Entranmbe le classi possiedono il loro vettore di dati, che rappresenta l'insieme dei valori discreti registrati dal sensori per singolo giorno nel corso di una settimana di test. Possiedono inoltre degli attributi che servono per la generazione dei dati, dei quali parlerò a seguire. Altre due classi concrete derivate da AbstractSensor sono SpeedSensor e Accelerometer, le quali, oltre al naturale vettore dei dati, contengono per composizione le classi precedenti: più di preciso, SpeedSensor contiene un DistanceCoveredSensor e Chronometer come campi dati, mentre

invece Accelerometer possiede un Chronometer e uno SpeedSensor. La chiara utilità di questa composizione è lo sfruttamento delle formule matematiche per il calcolo delle rispettive grandezze, senza avere campi dati e metodi appositi per la generazione di dati. Nonostante l'occasione, ho scelto un'implementazione semplificata che non prevede di sfruttare il pattern Composite: ho fatto sì che i campi di generazione dati coincidano tra i sensori dei dati, la qual cosa consente di avere un unico tipo di attrubuto visualizzabile nel widget sopra ai grafici e di non dover creare finestre di dialogo per la creazione dei sottosensori, quindi ho semplificato il progetto. A causa della necessità di modificare anche i sottosensori annidati, in SpeedSensor e in Accelerometer ho inserito non solo i normali getter, ma anche dei getter non costanti, che ritornano un riferimento al sottosensore, per poterlo così modificare utilizzando i suoi setter.
Le operazioni CRUD + Search le ho implementate nelle classi dell'interfaccia grafica corrispondenti al posizionamento del pulsante che comanda tali operazioni. L'operazione di aggiunta sfrutta una finestra di dialogo ridefinita che consente l'inserimento dei campi dati, ma consente anche di inserire solo il tipo, il nome e l'identificatore del sensore, poichè gli altri attributi possono essere creati tramite argomenti di default, e questo consente di velocizzare assai la creazione di un sensore. La modifica di un sensore è un meccanismo molto simile, dove la finestra di dialogo ridefinita già parte mostrando i valori precedenti nei campi di testo modificabili. Sono implementati per entrambe le finestre di dialogo dei criteri di controllo dell'input, specifici per ogni campo di testo. L'eliminazione è una operazione più semplice, poichè trattasi solamente di una erase dal vettore di sensori, arricchita da una finestra di conferma prima di compiere l'operazione irreversibile. Per la ricerca ho invece ideato un algoritmo particolare: per ognuna delle 3 etichette rappresentate nel pannello dei sensori, vado a calcolare la sottostringa comune più lunga con la sottostringa di ricerca, considero quella massima come la pertinenza, e inserisco il sensore assieme alla sua pertinenza dentro una QMultiMap (che ho preferito a QMap perchè la QMultipleMap consente anche di inserire più sensori con una stessa chiave di pertinenza), che poi viene ordinata per chiave crescente. Poi, attraverso un ciclo di scorrimento dall'ultimo elemento al primo elemento della mappa multipla (dal risultato più pertinenete a quello meno pertinente), vado a ricavare (tramite funzione di supporto) il vettore di sensori contenuto in una stessa chiave (è possibile ottenere in output anche un vettore composto da un sensore unico), e con un altro ciclo for scorro tal vettore, e dunque aggiorno i widget del pannello di sensori inserendovi i risultati della ricerca.
Per quanto riguarda la generazione dei dati, per essa ho sfruttato ulteriori 2 classi astratte, visibili anch'esse in  1, che implementano 2 design patterns, il Visitor e l'Observer. Il visitor concreto specifico per i grafici (derivato dalla classe astratta SensorVisitor) consente di differenziare la generazione in base al tipo del sensore, mentre invece l'Observer consente ai grafici (derivati dalla classe SensorObserver) di osservare che i dati dei sensori siano stati generati; questo flusso di Visitor e Observer lo approfondirò meglio nel prossimo punto.
Per quanto riguarda la generazione dei dati, per le classi SpeedSensor e Accelerometer si tratta semplicemente di una divisione matematica tra ogni dato dei

sottosensori, mentre invece per le classi DistanceCoveredSensor e Chronometer sono state implementate 4 possibilità: dati casuali, distribuzione normale, distribuzione uniforme e distribuzione esponenziale. Si tratta in sostanza di uno sfruttamento delle normali funzioni di distribuzione dati fornite dal namespace standard "std", e dalla libreria "random". Le implementazioni delle distribuzioni casuale e uniforme sfruttano solo i campi dati minvalue e maxvalue per un approccio random nell'intervallo [minvalue, maxvalue], quella della esponenziale sfrutta appieno il generatore standard dopo averlo inizializzato con lambda, mentre le implementazione della distribuzione normale, che utilizza la media, la varianza e la lambda, ha avuto bisogno di una sistemazione poichè la std::normaldistribution¡double¿ restituiva valori troppo elevati, e allora ho scalato i valori in basso usando minvalue e maxvalue.

# 3 Polimorfismo

L'utilizzo principale del polimorfismo riguarda il coordinamento tra i design pattern Visitor e Observer nelle gerarchie AbstractSensor e AbstractChart. Infatti, la generazione dei grafici funziona nel seguendo modo: il widget contenitore ChartWidget chiama il costruttore di uno dei suoi grafici, per esempio Histogram, ed esso va a chiamare lo specifico metodo di creazione dei dati del sensore basandosi sul valore del menù a tendina. Al termine di ogni metodo di sorteggio dei dati, il sensore effettua una notifica al suo observer collegato, che è rappresentato dal grafico stesso! L'implementazione della notifica nei grafici, poi, va a creare un visitor apposito per la gerarchia AbstractChart, e viene chiamata l'accettazione di esso da parte del sensore, la quale chiama il metodo di visita specifico per quel tipo di sensore. Infine, questo metodo di visita va finalmente a chiamare il metodo di aggiornamento dei dati dell'apposito grafico, che in realtà va a creare il grafico per intero. Questo comportamento fa sì che la creazione dei grafici riesca ad essere personalizzazione per ciascun sensore, e rende la gerarchia del modello logico completamente indipendente dall'interfaccia grafica! Per quanto riguarda la personalizzazione che ho scelto, i grafici prevedono due possibili visualizzazioni:

- Un grafico a barre per DistanceCoveredSensor, Chronometer e SpeedSensor, perchè i loro dati sono dei valori discreti e non pensati per una continuità l'uno con l'altro, poichè il pilota dell'aereo ha libertà di scelta su di essi, il suo unico ordine è di accelerare sempre in modo costante
- Un grafico a linea per Accelerometer, perchè i suoi dati sono il primario oggetto di test del centro di collaudo, e dunque si pone il focus sulla loro consistenza. Sono visualizzate due soglie come linee tratteggiate, una superiore ed una inferiore, entro le quali si ha l'obiettivo di mantenere i dati di accelerazione, per ottenere un aereo abilitato a volare. Quando l'accelerazione sfora i limiti, la linea del grafico da blu diventa rossa, e il messaggio sulla status bar segnala che occorrerà appunto effettuare ulteriori test per poter abilitare l'aeroplano difettoso.

Ho implementato degli ulteriori visitor per la gestione del polimorfismo per la persistenza dei dati file json, e per la creazione delle etichette nel widget sopra il grafico e nel pannello dei sensori.

Segnalo inoltre la creazione di due classi astratte, AbstractChart e AbstractDialogueWindow, che hanno l'obiettivo di offrire un'interfaccia per eventuali futuri metodi virtuali puri in comune tra i grafici e/o tra le finestre di dialogo.

# 4 Persistenza dei dati

Per la persistenza dei dati viene utilizzato il formato JSON, un unico file per centro di collaudo, contenente un vettore di oggetti sensori. Gli oggetti sono perlopiù semplici associazioni chiave-valore, e la serializzazione delle sottoclassi viene gestita aggiungendo un attributo "type". La gestione del JSON va a creare un ulteriore livello di collezione dei dati, separato dal vector presente nella MainWindow: infatti, nella classe JsonRepository viene creata una std::map, la quale effettua la copia profonda (clonazione) dei sensori del vector, e poi inserisce i suoi sensori nel file Json. Sarà poi essa stessa a prelevare i dati dal Json, ed effettuerà la clonazione per copiare i dati nel vector di MainWindow. Questa implementazione apre ad una possibile futura implementazione di un pulsante "Ripristina" che potrà permettere di cancellare le modifiche effettuate sui sensori di un determinato per ripristinare i valori che erano salvati nel file all'apertura. Un esempio della struttura dei file è dato dai JSON forniti assieme al codice, che sono 4 documenti rappresentanti varie combinazioni di possibili sensori per centro di collaudo. In particolare, Documento3.json contiene sensore per ciascuna tipologia, in modo da illustrare brevemente le diverse strutture.

# 5 Funzionalità implementate

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

- Gestione di quattro tipologie di articoli
- Conversione e salvataggio in formato JSON

Le funzionalità grafiche:

- Utilizzo di icone nella toolbar
- Status bar in fondo alla finestra
- Scorciatoie da tastiera, che ho implementato cercando di mantenere la lettera iniziale inglese della funzionalità (es.: Ctrl+A per "Add", Ctrl+O per "Open" o Ctrl+S per "Save"). Alcune scorciatoie non sono collegate nominalmente, però sono logiche, come ad esempio "Esc" per uscire da una finestra di dialogo ed Enter per applicare le sue modifiche

- Controllo della presenza di modifiche non salvate prima di uscire
- Ogni tipologia di sensore ha un proprio layout del widget che rappresenta i grafici (a parte DistanceCoveredSensor e Chronometer che hanno lo stesso), allo scopo di utilizzare lo stesso spazio in modo efficiente e polimorfo per il diverso numero di grafici che possiede ciascun tipo di sensore
- Sono disponibili due diverse visualizzazioni del grafico a seconda del sensore
- Possibilità di navigare attraverso due diverse schermate nel pannello dei sensori: la schermata principale e la schermata dei risultati di ricerca (dalla quale si può tornare alla schermata principale tramite un apposito pulsante)
- Utilizzo di icone nei pulsanti
- Utilizzo di immagini e di widget selezionabili nella visualizzazione dei sensori
- Evidenziazione dello sfondo di un widget di un sensore non appena esso viene selezionato

Le funzionalità elencate sono intese in aggiunta a quanto richiesto dalle specifiche del progetto.

# 6 Rendicontazione ore

| **Attività** | **Ore previste** | **Ore effettive** |
|---|---|---|
| Studio e progettazione | 20 | 30 |
| Sviluppo del codice del modello | 20 | 26 |
| Studio del framework Qt | 20 | 14 |
| Sviluppo del codice della GUI | 20 | 54 |
| Test e debug | 10 | 41 |
| Stesura della relazione | 5 | 8 |
| **Totale** | 95 | **173** |

La mia previsione iniziale era stata abbondante rispetto alle 50 ore dichiarate, poichè sapevo di non aver mai fatto un progetto in vita mia e quindi non avevo idea di cosa significasse di preciso fare questo e far quello. La progettazione mi ha impiegato più tempo del previsto perchè ho progettato inizialmente delle cose che non erano implementabili, come ad esempio una gerarchia di classi apposita per il dataset, che avevo fatto minuziosiamente, per poi non essere in grado di gestire il rapporto tra i sensori e il loro dataset, e quindi dover cancellare tutto, il che ha aumentato anche le ore di sviluppo del modello logico. Per quanto riguarda lo studio di Qt, ho fatto quanto potevo durante le vacanze di Natale, ma

ad un certo punto mi sono stancato di fare cose troppo teoriche e astratte e ho avuto voglia di iniziare a scrivere il codice del mio progetto. Successivamente mi sono assai pentito di questa scelta, non ero pronto a fronteggiare le lungaggine della gestione di una UI, dunque è stato particolarmente pesante programmare la GUI. Anche perchè in realtà mi ero concentrato sullo studiare qualche feature grafica troppo particolare dimenticando le basi da cui si deve sempre partire invece. E' anche questo uno dei motivi per cui la finestrella è fissa by design, perchè implementare l'allargamento e il restrigimento mi avrebbero allungato ancora di più il tempo di sviluppo, e adesso purtroppo i troppi widget composti e annidati rendono laboriosa una eventuale futura implementazione dell'allargamento e restringimento. Per quanto riguarda la sezione di test e debug, beh, lì mi si è aperto un mondo: sono cascato 5 volte in dipendenza circolare, ho avuto difficoltà a separare le classi tra di loro, soprattutto quando si trattava di includere gli header file giusti (tattiche come includere gli header file nel .cpp e non nel .h, oppure fissare delle dichiarazioni incomplete delle classi, non mi venivano naturali). Avevo anche creato nella MainWindow il vettore dei sensori come statico, e questo impediva di creare un widget senza prima aver creato la MainWindow, e viceversa. Inoltre, avendo scelto di pulire bene la memoria ridefinendo i distruttori e pulendo lo Heap ad ogni sovrascrizione dei widget, mi sono ritrovato a dover gestire i segmentation faults dovuti all'accesso a puntatori nulli, e ho dovuto impiegare delle ore a imparare a usare bene Valgrind e GDB per capire il motivo dei segmentations faults, per poi settare tutti i puntatori a nullptr nel costruttore e imparare qual è l'ordine corretto per effettuare i delete. Infine, la stesura della relazione è influenzata dal fatto che ho creato il diagramma del modello in Ida e ho imparato qualche nozione di base su LaTeX.

# 7 Altre info

Il mio progetto non funziona in modo ottimale su Ubuntu con Wayland come gestore delle finestre. La macchina virtuale per fortuna non ha wayland come gestore delle finestre, ma sul mio Ubuntu normale ho dovuto postare "Ubuntu on Xorg" perchè funzionasse. In pratica, nella finestra di dialogo di inserimento di un nuovo sensore, quando veniva cambiato il valore del menù a tendina del tipo, poi non potevi più scrivere negli altri campi di testo, e sul terminale veniva scritto "wayland: request not supported". Quindi, faccia attenzione se lei sul suo normale pc possiede wayland come gestore delle finestre. Sulla macchina virtuale, come suddetto, non dovrebbe avere problemi, nonostante appaia l'avviso qt.qpa.plugin: Could not find the Qt platform plugin "wayland" in "", all'apertura dell'eseguibile. Inoltre, ci sono alcuni commenti nel codice a riguardo di feature che avevo pensato e che alla fine non sono state introdotte, come ad esempio un tasto dedicato alla ricerca, che poi è stato scartato a favore di una ricerca in tempo reale ad ogni carattere digitato, lasciando però delle righe di codice commentate e anche l'icona che è rimasta nella cartella Assets.