oggetto	Relazione progetto Programmazione a Oggetti
studente	Martinelli Davide, mat. 2034341
titolo	EveryWeather

Introduzione

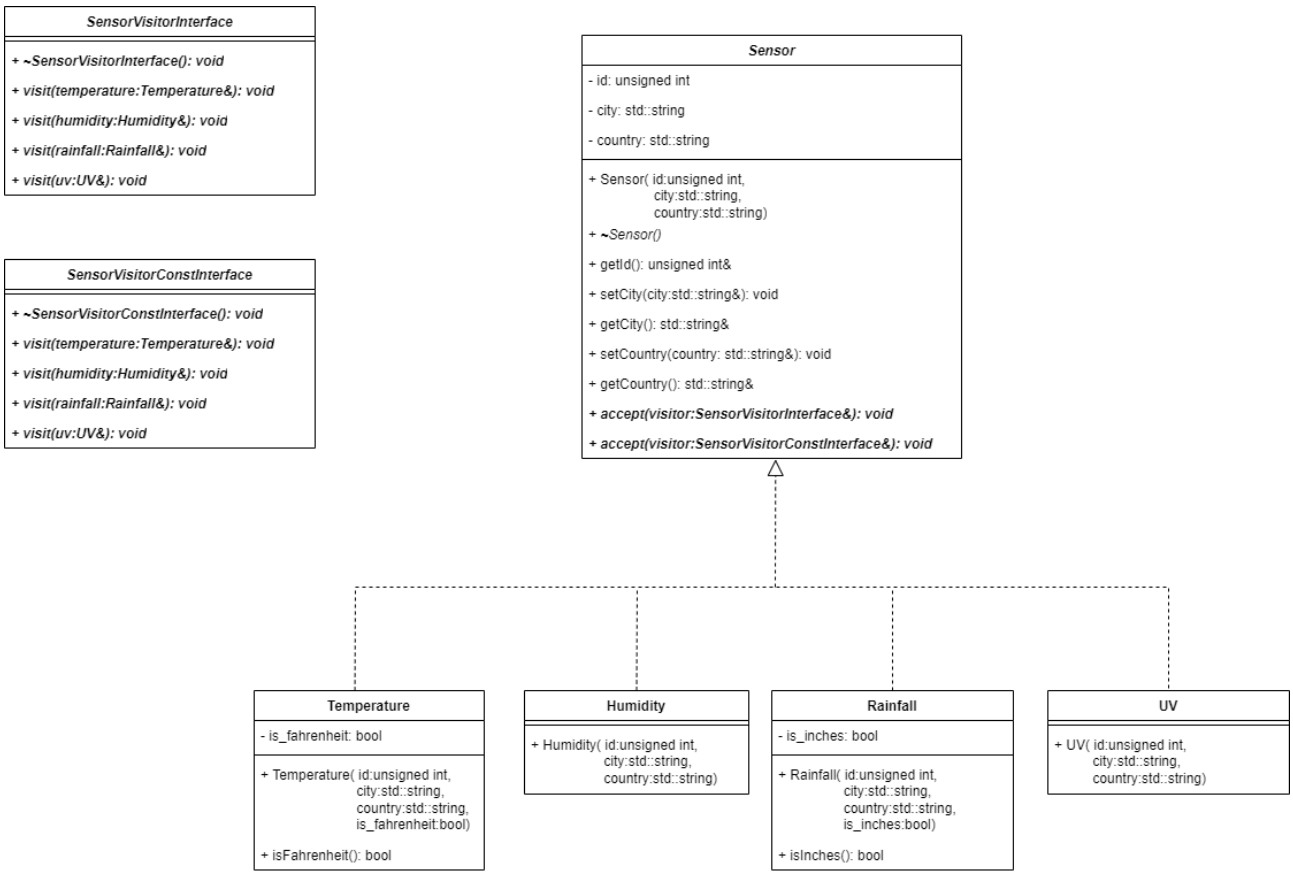
EveryWeather è un software che permette di visualizzare e gestire (creare, modificare ed eliminare) un *dataset* di sensori relativi al meteo delle città tramite un'interfaccia utente intuitiva ed esteticamente gradevole. I sensori possono essere di 4 tipologie in base a ciò che misurano, ovvero la temperatura, l'umidità, la quantità di pioggia o la quantità di raggi ultravioletti. Ciascuna tipologia di sensore ha una sua rappresentazione grafica, in forma di *widget*, dotata di un'icona significativa che ne permetta il riconoscimento immediato.

Per ciascun sensore è possibile simularne la raccolta dati, che sarà contestualmente visualizzabile sotto forma di grafico. I dati raccolti consistono nel valore medio che ha avuto quella tipologia di misurazione nel corso degli anni per ciascun mese dell'anno. L'ispirazione per questo programma proviene, infatti, dai siti web che offrono una panoramica del meteo nel corso dell'anno in un certo luogo, nati per aiutare i vacanzieri a decidere il periodo in cui programmare il loro viaggio e/o l'abbigliamento opportuno.

Vista la natura del programma, per rendere il software più fruibile da parte di un utenza internazionale è stato deciso di adottare la lingua inglese per l'interfaccia e di supportare, per i sensori che lo permettono, l'utilizzo di un'unità di misura appartenente al sistema internazionale o a quello imperiale.

Descrizione del modello

Il modello logico comprende: la gerarchia dei tipi di sensore descritta nel diagramma visibile nella pagina successiva, le classi per la simulazione della raccolta dati e anche le classi di servizio per implementare la persistenza dei dati, che in questo caso consentono la conversione dei sensori in oggetti JSON (e viceversa) e salvarli in locale su file. Per la gestione di questi ultimi sono stati utilizzati gli strumenti offerti dal framework Qt per la gestione del formato JSON. Anche la funzionalità di ricerca di un sensore è compresa, sotto forma di metodo, nella classe di servizio *JsonRepository*, che rappresenta il dataset di sensori disponibile nel programma a runtime.



Il modello si basa sulla classe astratta *Sensor* che racchiude le informazioni comuni a ciascun sensore: l'identificativo, ovviamente univoco, e la città e il paese in cui si trova tale sensore. Per questi ultimi sono implementati sia il metodo *getter* che *setter*, mentre per l'ID, dato che non è prevista la possibilità di modificarlo, è implementato solo il *getter.* Quest'ultima è una scelta di natura pratica derivata dall'implementazione, per la gestione dei sensori, di una struttura dati di tipo *std::map*, in cui le coppie chiave-valore sono formate dall'ID del sensore e un puntatore all'oggetto *Sensor.* Tutte e 4 le classi concrete derivano direttamente da *Sensor* e, come facilmente intuibile, rappresentano ciascuno un tipo diverso di sensore. Le uniche classi che aggiungono informazioni specifiche al tipo di sensore sono *Temperature* e *Rainfall*, che prevedono una variabile per stabilire se utilizzano il sistema metrico o imperiale.

Come è possibile notare, le classi del modello non implementano alcun metodo particolare: infatti per quanto riguarda la parte di persistenza dei dati (serializzazione/deserializzazione oggetto *Sensor*) si è deciso di sfruttare il *design pattern Visitor*, previsto già in partenza per il *rendering* dei diversi tipi di sensori all'interno della GUI, contribuendo così anche all'assorbimento, come conseguenza, del costo implementativo del *Visitor*, mentre per la parte di simulazione si è optato per la creazione di una classe ad-hoc per ciascun tipo di sensore che si occupa anche di istanziare il/i vettore/i contenente/i i dati simulati. Possiamo dire quindi che sostanzialmente nel programma le classi della gerarchia *Sensor* rappresentano dei *Data Transfer Object* (DTO).

Per l'implementazione del *Visitor* sono state create le classi astratte *SensorVisitorInterface* e *SensorVisitorConstInterface*, dove l'unica differenza è che la seconda consente l'accesso in sola lettura ai dati del sensore, permettendo così di sfruttare la *const correctness*. In concomitanza, alla classe astratta *Sensor* sono stati aggiunti due metodi virtuali puri *accept*, che accettano i due tipi diversi di *Visitor*.

La simulazione della raccolta dati di un sensore è molto semplice ed a puro scopo esemplificativo. Si basa sulla funzione random offerta dalla *standard library* di C e genera, per ciascun tipo ti sensore, dei valori casuali contenuti nel range di possibilità per la sua specifica tipologia di misurazione. Le classi dedicate alla simulazione contengono un solo metodo *simulate* che implementa quanto appena descritto, ma risulta di facile implementazione un'eventuale espansione delle opzioni di simulazione disponibili: sarà infatti sufficiente creare un metodo pubblico per ciascun tipo di simulazione che si vuole rendere disponibile.

Polimorfismo

L’utilizzo principale del polimorfismo riguarda il *design pattern Visitor* nella gerarchia *Sensor*. Esso viene utilizzato per la conversione da e in oggetti JSON e, in larga parte, per gestire la resa grafica degli elementi dell’interfaccia che dipendono dal tipo concreto del sensore. L’elenco esaustivo degli elementi grafici che sfruttano il *Visitor* comprende:

- Il ***widget*** che rappresenta il sensore nella **lista** presente sul lato sinistro dell’interfaccia (che può contenere tutti i sensori del *dataset* o il risultato di una ricerca effettuata)
- Il ***widget*** contenente le **informazioni esaustive** del sensore, visualizzabile in alto a destra nell’interfaccia dopo aver cliccato su un sensore della lista
- La **finestra** dedicata alla **modifica** o alla **creazione** di un sensore
- Il ***widget*** che consente la **simulazione/visualizzazione** dei dati del sensore, che costruisce un grafico (*QChart*) diverso per ciascuna tipologia di sensore

Persistenza dei dati

Per la persistenza dei dati viene utilizzato un unico file in formato JSON per ciascun *dataset* di sensori, contenente un vettore di oggetti. Gli oggetti sono composti sostanzialmente da semplici associazioni chiave-valore, e la serializzazione delle sottoclassi viene gestita aggiungendo un attributo chiamato “type”. Un esempio della struttura di un file è visibile consultando il file JSON chiamato “data.json” fornito assieme al codice, dove è possibile trovare almeno un sensore per ogni tipo.

Funzionalità implementate

Le funzionalità implementate possono essere suddivise per semplicità in funzionali ed estetiche, nel seguente modo:

funzionali:

- gestione di 4 tipi di sensore
- conversione (input) e salvataggio (output) in formato JSON
- ricerca nel *dataset* per ID o nome della città, quest’ultima *case insensitive*
- controllo della presenza di modifiche non salvate prima di uscire dal programma o di aprire o creare un nuovo file di sensori
- memoria dell’ultima ricerca fatta dall’utente, utilizzata per ripristinare la visualizzazione dopo l’eliminazione o la modifica di un sensore

estetiche:

- barra dei menù in alto, dotata di icone
- ogni tipologia di sensore ha una propria visualizzazione nella GUI
- ridimensionamento automatico del grafico contestuale al ridimensionamento della finestra
- animazione sui grafici
- scorciatoie da tastiera (visibili anche nelle voci del menù)
- cambio del colore di sfondo e visualizzazione dei pulsanti “modifica” ed “elimina” nei *widget* dei sensori al passaggio del mouse sopra di essi
- utilizzo di icone nei pulsanti di ricerca, modifica o eliminazione
- visualizzazione di un tag esplicativo al passaggio del mouse su un bottone che contiene solo un’icona
- modifica del cursore del mouse al passaggio su un bottone

- supporto sia al tema chiaro che quello scuro su tutta l’interfaccia, scelto in automatico in base tema dell’OS*
- stile preferenziale dell’interfaccia impostato su *Material*, per garantire un layout consistente tra diversi sistemi operativi

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

*disponibile solo su sistemi su cui è installato Qt 6.5 o versioni successive, per sbloccare la funzionalità leggere i commenti presenti nel file main.cpp

Rendicontazione ore

Attività	Ore Previste	Ore Effettive
Studio e progettazione	10	8
Sviluppo del codice del modello	10	12
Studio del framework Qt	10	8
Sviluppo del codice della GUI	10	14
Test e debug	5	10
Stesura della relazione	5	5
totale	50	**57**

Il monte ore è stato leggermente superato principalmente per due motivi: l’aggiunta del supporto al tema scuro, che è stata decisa durante lo sviluppo della GUI e non era prevista in fase di progettazione, e la lentezza nel risolvere alcuni bug dovuta alla scarsa dimestichezza con il framework Qt. Alcune delle ore conteggiate come debug potrebbero essere, a discrezione, considerate come ore di studio della documentazione di Qt.