oggetto	Relazione progetto Programmazione a Oggetti
gruppo	Nesler Michele, mat. 2000559
titolo	Sensor Data Manipulator

Introduzione

Sensor Data Manipulator è un programma che permette di creare, modificare, cancellare e visualizzare i dati di diversi tipi di sensori. Inoltre, è possibile effettuare una simulazione che genera 20 valori casuali ma dipendenti dai valori precedenti. I sensori attualmente esistenti sono di 3 tipi: sensore d' umidità, di polveri sottili PM 2,5 e di temperatura, ciascuno con le proprie caratteristiche tecniche differenti.

Questo programma offre anche la funzionalità di ricerca tra i sensori aperti e il loro salvataggio in un file che permette la persistenza dei dati. Ovviamente è possibile anche aprire un file salvato precedentemente.

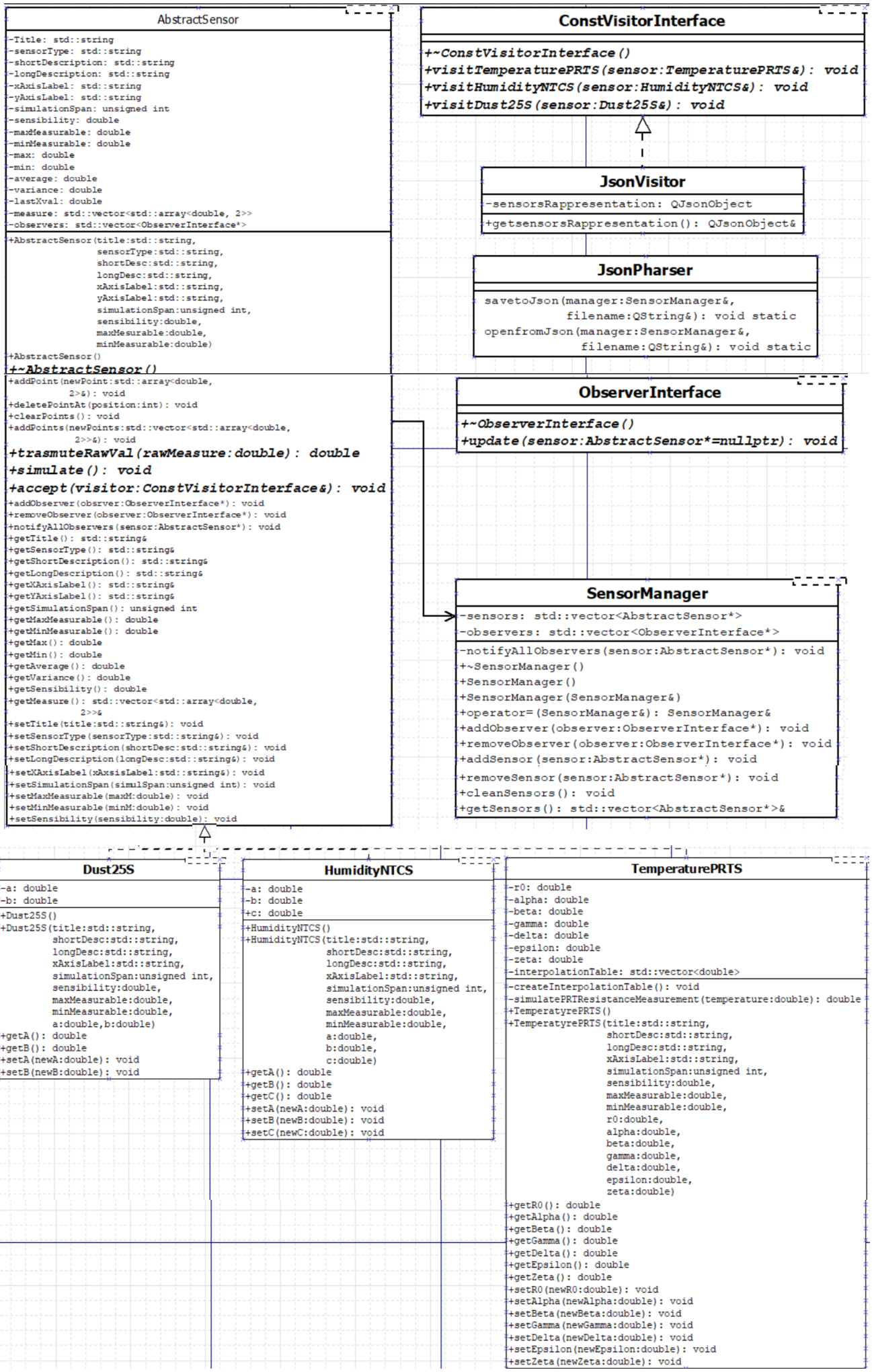
Inizialmente ho pensato che un buon pretesto per utilizzare il polimorfismo in modo non banale potesse essere un metodo virtuale (trasmuteRawVal) che traducesse i valori "grezzi" registrati dai sensori, dipendenti dalla specifica tecnologia del sensore (come, ad esempio, la misura in OHM di un resistore), nelle misure effettive dei fenomeni registrati facilmente interpretabili dalle persone (ad esempio una misura di temperatura in C°).

Descrizione del modello

SensorManager è la classe che ha all'interno un contenitore di sensori e fa da interfaccia alla vista per tutte le operazioni che riguardano il modello.

Le classi JsonPharser e JsonVisitor sono classi ausiliarie che realizzano la persistenza dei dati in formato Json.

Verso la fine della realizzazione del progetto mi sono accorto che sarebbe stato meglio utilizzare un set piuttosto che un vector contente array di coppie di valori per memorizzare le misurazioni, non ho però apportato questa modifica perché avevo già sforato le 50 ore previste. Sarebbe stato meglio perché il set garantisce l'univocità delle chiavi (asse x) e mantiene in ordine i dati, inoltre contiene coppie di valori che avrebbero sostituito gli array.


AbstractSensor
-Title: std::string
-sensorType: std::string
-shortDescription: std::string
-longDescription: std::string
-xAxisLabel: std::string
-yAxisLabel: std::string
-simulationSpan: unsigned int
-sensibility: double
-maxMeasurable: double
-minMeasurable: double
-max: double
-min: double
-average: double
-variance: double
-lastXval: double
-measure: std::vector<std::array<double, 2>>
-observers: std::vector<ObserverInterface*>
+AbstractSensor(title:std::string, sensorType:std::string, shortDesc:std::string, longDesc:std::string, xAxisLabel:std::string, yAxisLabel:std::string, simulationSpan:unsigned int, sensibility:double, maxMesurable:double, minMeasurable:double)
+AbstractSensor()
+~AbstractSensor()
+addPoint(newPoint:std::array<double, 2>&): void
+deletePointAt(position:int): void
+clearPoints(): void
+addPoints(newPoints:std::vector<std::array<double, 2>>&): void
+trasmuteRawVal(rawMeasure:double): double
+simulate(): void
+accept(visitor:ConstVisitorInterface&): void
+addObserver(obsrver:ObserverInterface*): void
+removeObserver(observer:ObserverInterface*): void
+notifyAllObservers(sensor:AbstractSensor*): void
+getTitle(): std::string&
+getSensorType(): std::string&
+getShortDescription(): std::string&
+getLongDescription(): std::string&
+getXAxisLabel(): std::string&
+getYAxisLabel(): std::string&
+getSimulationSpan(): unsigned int
+getMaxMeasurable(): double
+getMinMeasurable(): double
+getMax(): double
+getMin(): double
+getAverage(): double
+getVariance(): double
+getSensibility(): double
+getMeasure(): std::vector<std::array<double, 2>>&
+setTitle(title:std::string&): void
+setSensorType(sensorType:std::string&): void
+setShortDescription(shortDesc:std::string&): void
+setLongDescription(longDesc:std::string&): void
+setXAxisLabel(xAxsisLabel:std::string&): void
+setSimulationSpan(simulSpan:unsigned int): void
+setMaxMeasurable(maxM:double): void
+setMinMeasurable(minM:double): void
+setSensibility(sensibility:double): void
ConstVisitorInterface
+~ConstVisitorInterface()
+visitTemperaturePRTS(sensor:TemperaturePRTS&): void
+visitHumidityNTCS(sensor:HumidityNTCS&): void
+visitDust25S(sensor:Dust25S&): void
JsonVisitor
-sensorsRappresentation: QJsonObject
+getsensorsRappresentation(): QJsonObject&
JsonPharser
savetoJson(manager:SensorManager&, filename:QString&): void static
openfromJson(manager:SensorManager&, filename:QString&): void static
ObserverInterface
+~ObserverInterface()
+update(sensor:AbstractSensor*=nullptr): void
SensorManager
-sensors: std::vector<AbstractSensor*>
-observers: std::vector<ObserverInterface*>
-notifyAllObservers(sensor:AbstractSensor*): void
+~SensorManager()
+SensorManager()
+SensorManager(SensorManager&)
+operator=(SensorManager&): SensorManager&
+addObserver(observer:ObserverInterface*): void
+removeObserver(observer:ObserverInterface*): void
+addSensor(sensor:AbstractSensor*): void
+removeSensor(sensor:AbstractSensor*): void
+cleanSensors(): void
+getSensors(): std::vector<AbstractSensor*>&
Dust25S
-a: double
-b: double
+Dust25S()
+Dust25S(title:std::string, shortDesc:std::string, longDesc:std::string, xAxisLabel:std::string, simulationSpan:unsigned int, sensibility:double, maxMeasurable:double, minMeasurable:double, a:double,b:double)
+getA(): double
+getB(): double
+setA(newA:double): void
+setB(newB:double): void
HumidityNTCS
-a: double
-b: double
-c: double
+HumidityNTCS()
+HumidityNTCS(title:std::string, shortDesc:std::string, longDesc:std::string, xAxisLabel:std::string, simulationSpan:unsigned int, sensibility:double, maxMeasurable:double, minMeasurable:double, a:double, b:double, c:double)
+getA(): double
+getB(): double
+getC(): double
+setA(newA:double): void
+setB(newB:double): void
+setC(newC:double): void
TemperaturePRTS
-r0: double
-alpha: double
-beta: double
-gamma: double
-delta: double
-epsilon: double
-zeta: double
-interpolationTable: std::vector<double>
-createInterpolationTable(): void
-simulatePRTResistanceMeasurement(temperature:double): double
+TemperatyrePRTS()
+TemperatyrePRTS(title:std::string, shortDesc:std::string, longDesc:std::string, xAxisLabel:std::string, simulationSpan:unsigned int, sensibility:double, maxMeasurable:double, minMeasurable:double, r0:double, alpha:double, beta:double, gamma:double, delta:double, epsilon:double, zeta:double)
+getR0(): double
+getAlpha(): double
+getBeta(): double
+getGamma(): double
+getDelta(): double
+getEpsilon(): double
+getZeta(): double
+setR0(newR0:double): void
+setAlpha(newAlpha:double): void
+setBeta(newBeta:double): void
+setGamma(newGamma:double): void
+setDelta(newDelta:double): void
+setEpsilon(newEpsilon:double): void
+setZeta(newZeta:double): void

Polimorfismo

Il polimorfismo è utilizzato in primo luogo implementando il visitor pattern in modo da evitare di utilizzare un elseif a cascata con dynamic_cast che avrebbe reso il codice difficilmente mantenibile. Il visitor viene usato nella classe JsonPharser (nella vista) per generare la rappresentazione in formato Json del sensore in base al tipo dinamico di questo. Viene utilizzato anche dalla classe SensorDialog (nella vista) per generare il widget che costituisce la parte di interfaccia relativa alla modifica dei dati tecnici specifici del sensore e che quindi contiene campi diversi in base al tipo dinamico del sensore.Questo è reso possibile dal metodo virtuale puro accept() della classe AbstractSensor.

In secondo luogo, implementando l'observer pattern, utilizzato dalla vista per aggiornare immediatamente le informazioni dei sensori rappresentati in caso di aggiunta di un nuovo sensore al modello o di modifiche ad uno preesistente, sia nella visualizzazione a pagina intera che in quella della lista nella barra laterale.

Per finire il polimorfismo è implementato anche attraverso il metodo virtuale puro trasmuteRawVal() (descritto sopra) ma questo non viene mai invocato nel progetto in quanto è un metodo che è utile solamente utilizzando dei sensori reali che generano valori grezzi non facilmente interpretabili a prima vista.

Il metodo simulate() invece è virtuale solo per permettere estensioni future, ma è implementato unicamente nella classe base essendo questo un progetto didattico.

Persistenza dei dati

Per la persistenza dei dati viene utilizzato il formato JSON, un unico file per catalogo di sensori, contenente un vettore di oggetti. Gli oggetti sono perlopiù semplici associazioni chiave-valore, a parte un array di array rappresentati le singole misurazioni. La serializzazione delle sottoclassi viene gestita aggiungendo un attributo "type". Un esempio della struttura dei file è dato dal file JSON fornito assieme al codice, denominato "small.json" contiene un sensore per ciascuna tipologia, in modo da illustrare brevemente le diverse strutture.

```json
{
    "sensors": [
        {
            "ADust": "10",
            "BDust": "3",
            "longDesc": "dati provenienti da : https://www.pgt.comune.milano.it",
            "maxMeasurable": "50",
            "measure": [
                [34,2006],
                [31,2007],
                [30,2008],
                [25,2009]
            ],
            "minMeasurable": "0",
            "name": "Polveri PM 2,5 Milano",
            "sensibility": "1",
            "shortDesc": "Misurazioni annue dal 2006",
            "simulationSpan": "1",
            "type": "Dust25S",
            "xAxisLabel": "Anno"
        }
    ]
}
```

1 esempio di rappresentazione di un oggetto in formato JSON

Funzionalità implementate

Le funzionalità aggiuntive rispetto alle specifiche implementate sono:

- Finestre con richieste di Conferma per azioni potenzialmente pericolose come la chiusura di un file o di un sensore
- messaggi di errore nel caso di input richiesti mancanti nella finestra di dialogo dedicata all'aggiunta/modifica di sensori
- messaggio di errore (e annullamento dell'operazione) nel caso in cui si cerchi di salvare un sensore immettendo più di un valore Y per lo stesso punto sull'asse X .
- validatore che consente di inserire solamente valori validi nei campi in cui è richiesto un double nella finestra di dialogo dedicata all'aggiunta/modifica di sensori
- calcolo della media e varianza all'aggiunta di ogni valore
- ricerca parziale (non solo corrispondenza esatta) nei titoli dei sensori

Rendicontazione ore

Attività	Ore Previste	Ore Effettive
Studio e progettazione	10	20
Sviluppo del codice del modello	5	10
Studio del framework Qt	10	10
Sviluppo del codice della GUI	10	20
Test e debug	10	25
Stesura della relazione	5	5
totale	50	**90**

Il monte ore è stato superato di molto in quanto la preliminare progettazione fatta non teneva in considerazione alcuni aspetti, e quindi ho valutato che potesse essere più efficiente fermarmi e riprogettare da capo, modificando poi ciò che avevo già fatto in modo che rispecchiasse la nuova progettazione e implementando le cose che ancora non avevo implementato. In particolare, non avevo considerato il comportamento dei contenitori di oggetti osservatori alla distruzione degli oggetti osservati. Inoltre, nell'implementare il Visitor Pattern mi sono accorto che la mia comprensione iniziale aveva delle falle, malgrado durante la progettazione mi sembrasse completa.

In conclusione, devo dire che non tutto il male vien per nuocere perché è proprio a causa di queste difficoltà che ho avuto nella progettazione e nella comprensione dei design pattern che mi è cresciuta la motivazione e curiosità di seguire Ingegneria Del Software il prossimo semestre.