Relazione Progetto Programmazione ad Oggetti

Titolo: Winery Control System

Anno Accademico 2023/2024

Gruppo: Vallotto Caterina 2076434, Matteazzi Giovanni Battista 2082852

1 - Introduzione

“Winery Control System” è un’applicazione che permette di gestire diversi sensori che hanno lo scopo di monitorare molteplici elementi sia all’interno delle stanze di una cantina, che all’interno delle singole botti. È infatti possibile creare, modificare, effettuare ricerche ed eliminare le varie tipologie di sensori di seguito elencate:

- sensori per la luce
- sensori per la temperatura
- sensori per l’umidità
- sensori per i filtri
- sensori per il volume

L’applicazione è inoltre in grado di simulare una raccolta dati che varia in base al tipo di sensore selezionato, generando così un grafico specifico con caratteristiche strutturali differenti per ciascuna delle cinque tipologie di sensori. Il grafico viene creato seguendo determinati parametri impostati dall'utente al momento della creazione del sensore, con la possibilità di modificarli successivamente e di generare una nuova raccolta dati che rispetti i nuovi valori limite.

In accordo con quanto richiesto, l'applicazione consente di memorizzare qualsiasi configurazione dei sensori che potrebbe risultare utile per un uso futuro. Inoltre, è possibile caricare una configurazione salvata in precedenza. Il processo sopra indicato avviene convertendo la configurazione in formato JSON e, quando necessario, riconvertendo quest’ultima. In tale modo, le configurazioni dei sensori possono essere facilmente salvate, condivise e riutilizzate, garantendo una gestione personalizzata del sistema di monitoraggio della cantina.

2 - Descrizione del modello

Per la realizzazione dell’applicazione sopra descritta è stato utilizzato il pattern Model-View-Controller, implementato come segue:

- Model

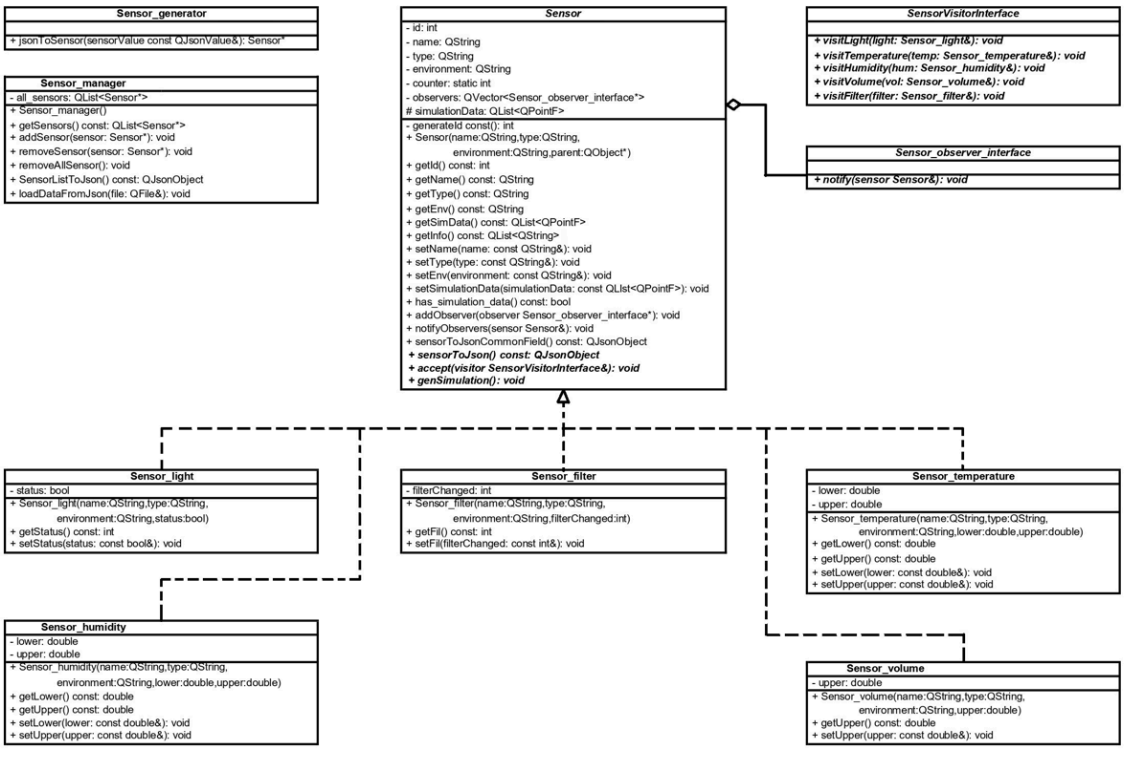

Figura 1 - schema Model

Il fulcro della parte di Model è la classe astratta `Sensor` da cui derivano cinque classi concrete (`Sensor_light, Sensor_humidity, Sensor_temperature, Sensor_volume, Sensor_filter`). La classe astratta `Sensor` contiene i campi comuni a tutti e cinque i sensori, ovvero id, name, type, environment e observers; ogni singola classe concreta è caratterizzata da uno/due singoli attributi specifici. All'interno di `Sensor` sono quindi implementati i metodi getter e setter per gli attributi comuni, invece per quanto riguarda quelli specifici ogni classe concreta ha i relativi metodi getter e setter.
Si noti che per generare una simulazione è stato creato il metodo virtuale puro che viene sovrascritto da ogni sensore concreto in base alle sue esigenze, in particolare:

- `Sensor_light`: genera dei dati che possono assumere solo valore 0 o 1 (spento o acceso)
- `Sensor_temperature`: genera dei dati compresi tra lower e upper
- `Sensor_humidity`: genera dei dati compresi tra lower e upper
- `Sensor_filter`: genera dei dati che rappresentano il numero di filtri cambiati nel tempo, quindi i valori saranno solamente crescenti
- `Sensor_volume`: genera dei dati che rappresentano la diminuzione del volume nel tempo, quindi i valori saranno solamente decrescenti; quando il volume arriva a 0, i dati assegnati saranno uguali a 0.

Dentro la classe base astratta `Sensor` sono inoltre stati inseriti i metodi relativi alla persistenza dei dati, in particolare `jsonToSensorCommonField` che si occupa di convertire i campi comuni a tutti i sensori dal formato json all'effettivo sensore, e un altro metodo virtuale puro (che verrà quindi sovrascritto da ogni sensore concreto) che tratta invece la conversione dei campi specifici e ritorna un `QJsonObject` che rappresenta quindi un sensore in formato json. Per quanto riguarda invece la trasformazione da sensore ad un formato json è stata implementata la classe `Sensor_generator` che ha un unico metodo pubblico `jsonToSensor`: in base alla tipologia di sensore presente nel `QJsonValue` passato come argomento, va a creare un sensore del tipo corretto utilizzando le caratteristiche `QJsonObject` ottenuto dal `QJsonValue`.

Sono anche presenti i metodi necessari per implementare il pattern Visitor e Observer: infatti sono implementate due classi virtuali pure `SensorVisitorInterface` e `Sensor_observer_interface` che hanno il compito di soddisfare il vincolo di polimorfismo non banale. Per rendere quindi utilizzabili questi pattern in `Sensor` sono presenti dei metodi specifici per il loro funzionamento, ovvero:

- `addObserver`: aggiunge un osservatore alla lista di osservatori
- `notifyObserver`: notifica tutti i componenti della lista degli osservatori che un evento è accaduto, nel nostro caso ogni qualvolta venga modificato un attributo comune o specifico del sensore
- `accept`: sovrascritta da ogni sensore concreto in modo tale che il visitor vada effettivamente a 'visitare' il sensore di tipo corretto, ovvero eseguire la funzione `visit`*`Type`* adeguata (ogni `visit`*`Type`* verrà sovrascritta da ogni classe derivata da `SensorVisitorInterface` - vedi sezione 'View')

Infine è presente la classe `Sensor_manager` che ha lo scopo di rappresentare un lista di puntatori a `Sensor*`. Oltre ai metodi getter e aggiunta di un sensore alla lista di sensori, sono presenti i metodi relativi alla rimozione di un sensore dalla lista e di eliminazione completa di tutta la lista.
Per quanto riguarda invece il salvataggio/caricamento di un'intera lista di sensori, sono stati implementati:

- `sensorListToJson`: per ogni sensore presente all'interno della lista, crea un `QJsonArray` in cui inserisce ogni sensore 'trasformato' in formato json, chiamando la funzione `sensorToJson()` su ogni elemento della lista
- `loadDataFromJson`: il `QJsonObject` ottenuto dal file caricato viene associato ad un `QJsonArray`, andando poi a trasformare ogni suo elemento in un sensore attraverso la funzione `jsonToSensor()`

- View

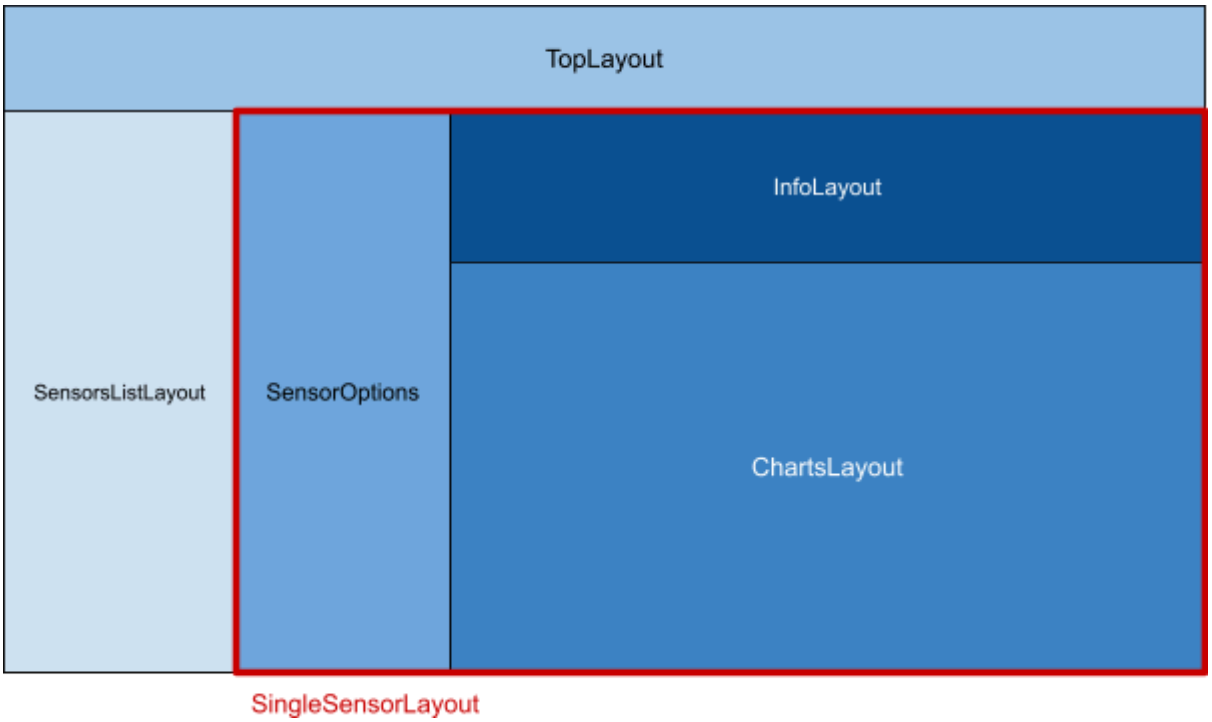

Figura 2 - modello grafico della suddivisione della schermata dell'applicazione con indicate le classi implementate

Per quanto riguarda la realizzazione della parte grafica sono state implementate svariate classi, ognuna con il compito specifico di rappresentare una delle sezioni della figura 2 (NB: in figura non sono state rappresentate graficamente le finestre di creazione, modifica e conferma eliminazione di un sensore). Ogni classe ha al suo interno gli attributi di cui necessita, gli slot, i signal e i metodi necessari per il funzionamento dell'applicazione ed eventuali metodi pubblici `setUp`*`Name`*`(Sensor*)` grazie ai quali ogni classe grafica imposta le caratteristiche necessarie in base al tipo concreto di sensore; questo è dovuto dal fatto che ogni tipologia di sensore concreto necessita di caratteristiche differenti che vengono quindi 'reimpostate' ogni volta che viene chiamato il metodo `setUp`*`Name`*.
All'interno della View sono inoltre presenti le classi e i metodi necessari per l'implementazione del pattern Visitor e Observer:

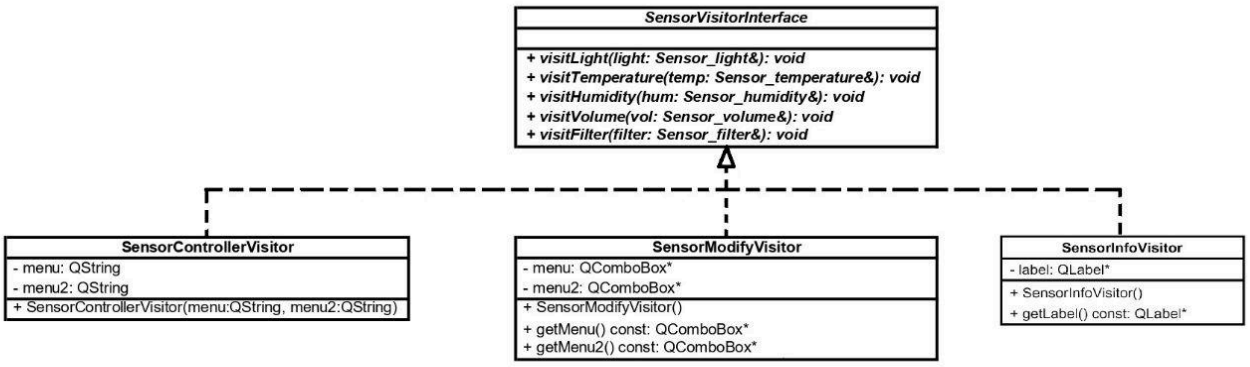

Figura 3 - schema pattern Visitor

Classi per il Visitor:

- `SensorModifyVisitor`: all'interno di questa classe vengono sovrascritti i 5 metodi virtuali puri della classe `SensorVisitorInterface` in base alle caratteristiche di cui la classe `ModifySensorWindow` necessita. All'interno di quest'ultima viene poi creata un'istanza di `SensorModifyVisitor` sulla quale il sensore invoca il metodo `s→accept(*visitor)` che è

stato definito per ogni tipologia specifica di sensore; all'interno di accept verrà a sua volta chiamata la `visitType` corretta.
esempio:
- all'interno di `ModifySensorWindow`: `puntatore a sensore→accept(*visitor)`, dove visitor è un'istanza di `SensorModifyVisitor` (derivata di `SensorVisitorInterface`)
- all'interno del tipo concreto del sensore (per esempio `Sensor_light`) nel metodo `accept(SensorVisitorInterface& visitor)`: `visitor.visitLight(*this)`
- la chiamata del metodo precedente porta quindi all'invocazione di `visitLight` della classe `SensorModifyVisitor`, che andrà a impostare le caratteristiche necessarie per modificare un sensore di tipo Light

- `SensorInfoVisitor`: funziona allo stesso modo di `SensorModifyVisitor` ma applicato al caso della classe `InfoLayout`
- `SensorControllerVisitor`: vedi sezione 'Controller'

Metodi per l'Observer: in questo caso non è necessario creare nuove classi, ma le classi della view `InfoLayout` e `SensorsListLayout` diventano classi derivate di `Sensor_observer_interface`, andando perciò a sovrascrivere il metodo virtuale puro `notify` in base alle loro esigenze:
- `SensorsListLayout`: la classe viene aggiunta alla lista degli osservatori del sensore, in questo modo quando una delle caratteristiche del sensore viene modificata, il sensore notifica tutti i suoi osservatori invocando il metodo `notify` sovrascritto da ognuno di essi. In tal modo se il nome del sensore viene cambiato, la classe `SensorsListLayout` viene avvisata e agisce di conseguenza
- `InfoLayout`: funziona allo stesso modo di `SensorsListLayout` ma all'interno della `notify` è possibile andare ad aggiornare anche tutte le altre caratteristiche del sensore e non solo il nome

All'interno della View è stato inoltre necessario creare la classe `CustomButton` che deriva da `QPushButton`, alla quale è stato aggiunto il campo privato `QPointer<Sensor> sensor`; in tal modo ogni `CustomButton` che viene creato all'interno della classe `SensorsListLayout` ha un sensore 'associato', necessario per visualizzare le informazioni del singolo sensore quando il `CustomButton` viene premuto. Anche all'interno di altre classi sono stati creati dei `CustomButton`, in particolare all'interno di `SensorOptions` per essere in grado di avviare un nuova simulazione e generare quindi nuovi dati per il 'giusto' sensore; all'interno di `ModifySensorWindow` così da poter salvare le modifiche del sensore inserite da utente.

- Controller
Il Controller ha il compito di gestire tutta l'esecuzione dell'applicazione, al suo interno sono infatti create tutte le istanze di ogni classe della View. Per adempiere al suo compito contiene inoltre tutti i signal, slot e metodi necessari per far interagire la View e la Model ogni qualvolta risulti necessario, in particolare i seguenti metodi gestiscono la creazione, modifica, eliminazione di un sensore:
- `createSensor`: crea un sensore in base al tipo selezionato da utente, lo aggiunge all'istanza di `Sensor_manager`, aggiunge un `CustumButton` alla classe `SensorsListLayout` (che viene quindi visualizzato sull'interfaccia grafica)
- `modifySensor`: modifica le informazioni del sensore; per questo metodo è stato necessario implementare la classe `SensorControllerVisitor` per gestire la modifica delle informazioni specifiche di ogni sensore (il funzionamento del pattern Visitor è descritto sopra)
- `deleteSensor`: si occupa dell'eliminazione del sensore, rimuovendolo dalla lista di sensori (sm oggetto della classe `Sensor_manager`), 'pulisce' la memoria e chiama il metodo `refresh`

- `refresh`: pulisce il layout dove sono presenti tutti i bottoni creati (che corrispondono ai sensori presenti all'interno della lista di sensori) eliminandoli tutti. Viene poi scorsa tutta la lista di sensori e, per ogni sensore al suo interno, viene ricreato un nuovo `CustomButton`.

Per il salvataggio/l'upload di una configurazione di sensori sono presenti i metodi

- `openSaveWindow`: slot che ha lo scopo di aprire la schermata di salvataggio e invocare la `func_save` con il nome selezionato
- `openLoadWindow`: slot che ha lo scopo di aprire la schermata di load e 'pulire' l'interfaccia grafica da eventuali sensori ancora presenti ed infine invoca la `func_load`
- `func_save`: avvia il processo di salvataggio del file chiamando la funzione `sensorListToJson()`
- `func_load`: avvia il processo di caricamento del file chiamando la funzione `LoadDataFromJson()`

3 - Polimorfismo

L'utilizzo del polimorfismo in maniera non banale si articola in due pattern: Visitor e Observer

- Pattern Visitor

Il pattern Visitor viene utilizzato in questi tre contesti:

- `InfoLayout`: permette di creare widget (in particolare `QLabel`) in base al tipo di sensore, garantendo quindi la possibilità di visualizzare correttamente le informazioni specifiche di ogni sensore concreto
- `ModifySensorWindow`: come evidenziato in precedenza, ogni sensore concreto ha caratteristiche specifiche differenti; applicando quindi questo pattern nel contesto della finestra di modifica di un sensore otteniamo l'aggiornamento della finestra creando di volta in volta i widget (`QComboBox`) necessari per permettere la modifica degli attributi non comuni
- `Controller`: una volta che l'utente ha determinato i campi da modificare, le informazioni vengono trasmesse al `Controller` che dovrà procede alla modifica vera e propria delle informazioni del sensore; per i motivi descritti precedentemente è stato perciò necessario applicare il Visitor anche in questo contesto, in modo tale da rendere possibile la modifica degli attributi specifici di ogni sensore

- Pattern Observer

Il pattern Observer è invece utilizzato nei seguenti contesti:

- `InfoLayout`: ogni volta che un attributo di un sensore viene modificato la classe InfoLayout (che ha il ruolo di osservatore) viene notificata e procede alla sostituzione delle `QLabel` con i nuovi dati modificati
- `SensorsListLayout`: anche in questo caso la classe viene notificata ad ogni cambiamento di informazioni del sensore, però ciò che la riguarda è solo il cambiamento del nome del sensore che viene visualizzato sui `CustomButton`

L'applicazione di questo pattern permette di non dover eliminare e ricreare l'intera interfaccia ogni volta, ma mantenere la stessa istanza di `InfoLayout` e `SensorsListLayout`, modificando solo i campi al loro interno.

4 - Persistenza dei dati

Per il vincolo di persistenza dei dati è stato utilizzato, come indicato, il formato json. Data una particolare configurazione di sensore, essa può essere salvata trasformando la lista di sensori in un unico vettore di oggetti rappresentati da associazioni chiave-valore. Un file .json può inoltre essere importato all'interno dell'applicazione passando attraverso il processo di conversione da oggetti di tipo json a istanze di sensori. Un esempio è già fornito assieme al codice: "fileConfigurazione.json" contenente un sensore per tipologia.

In particolare per il salvataggio:

- *View*:
  - Viene premuto il bottone 'Save' in `TopLayout`
  - Viene inviato un segnale `showSaveWindowSignal()`
- *Controller*:
  - Il segnale inviato viene ricevuto dalla classe `Controller` ed esegue il private slot `openSaveWindow()` aprendo quindi la finestra di dialogo per il salvataggio e, una volta selezionato il path e il nome del file viene chiamata la funzione `func_save(nomeFile)`
  - la funzione `func_save(nomeFile)` chiama a sua volta la funzione `sensorsListToJson()` della classe `Sensor_manager` salvando il risultato in un `QJsonDocument`
- *Model*:
  - `sensorsListToJson()` chiama la `sensorToJson()` per ogni sensore presente nella lista di sensori; ovviamente viene chiamata la `sensorToJson()` 'corretta' in base al tipo concreto del sensore
  - La funzione `sensorToJson()` ritorna un `QJsonObject` che conterrà il sensore trasformato in formato json

Mentre per quanto riguarda il caricamento:

- *View*:
  - Viene premuto il bottone 'Upload' in `TopLayout`
  - Viene inviato un segnale `showLoadWindowSignal()`
- *Controller*:
  - Il segnale inviato viene ricevuto dalla classe `Controller` ed esegue il private slot `openLoadWindow()` aprendo quindi la finestra di dialogo per il caricamento e, una volta selezionato il file .json desiderato, viene chiamata la funzione `func_load(nomeFile)`. Si noti che viene anche 'pulita' l'itnerfaccia grafica dai sensori presenti prima di un upload
  - La funzione `func_load(nomeFile)` chiama a sua volta la funzione `loadDataFromJson()` della classe `Sensor_manager`
- *Model*:
  - `loadDataFromJson()` chiama la `Sensor_generator::jsonToSensor()` per ogni sensore presente all'interno del file di configurazione scelto
  - La funzione `Sensor_generator::jsonToSensor()` andrà quindi a creare i sensori presenti del file .json in base al loro tipo

5 - Funzionalità implementate

Le funzioni integrate nell'applicazione comprendono:

• Azioni riguardanti i sensori:
  – Configurazione di sensori personalizzabili dall'utente di 5 tipologie diverse
  – Eliminazione dei sensori non più necessari o obsoleti
  – Modifica delle caratteristiche dei sensori per adattarli a diverse esigenze
  – Possibilità di eseguire svariate raccolte dati e visualizzarle in un grafico

• Salvataggio/caricamento di configurazioni di sensori

• Azioni interfaccia grafica
  – Ricerca efficace dei sensori per nome attraverso una barra di ricerca

– Tasto ‘Save’ che viene attivato solo se ci sono dei sensori creati, altrimenti è disattivato
– Necessità di inserire obbligatoriamente tutti i campi al momento della creazione di un sensore
– Scomparsa dell’etichetta “Create a sensor or upload a simulation” quando viene creato un sensore o caricata una configurazione di sensori. Comparsa dell’etichetta quando tutti i sensori vengono eliminati
– Scomparsa dell’etichetta “Select a single sensor” quando viene selezionato un sensore dalla liste di sensori presenti. Comparsa dell’etichetta quando il sensore selezionato viene eliminato
– Appena l’utente sceglie il tipo di sensore che vuole creare, la finestra si modifica con le informazioni necessarie per creare un sensore del tipo selezionato
– Aggiornamento delle informazioni del sensore nella View quando un sensore viene modificato
– Chiusura delle finestre di creazione, modifica ed eliminazione con apposito tasto in caso non si volesse più eseguire tale operazione
– Quando vengono mostrate le finestre di creazione, modifica o eliminazione, l’esecuzione si blocca finché la finestra non viene chiusa/non viene effettuata l’operazione per cui è predisposta. Quindi non è possibile creare, modificare o eliminare un sensore allo stesso momento.

6 - Rendicontazione ore

Nella seguente tabella sono riportate le ore previste per la realizzazione dell’applicazione:

Attività	Giovanni Battista Matteazzi	Caterina Vallotto
Studio del framework Qt	7	7
Studio e progettazione	7	7
Sviluppo codice Model	8	8
Sviluppo codice View	8	8
Sviluppo codice Controller	9	9
Debugging e fix	3	3
Test	4	4
Stesura relazione	4	4
Totale	**50**	**50**

Come si nota invece nella seconda tabella riportata, il monte ore è stato superato a causa del maggior tempo necessario per lo studio e lo scouting del framework Qt, fase durante la quale entrambi i componenti del gruppo hanno avuto evidenti difficoltà. In aggiunta, non essendo presente alcuna tipologia di esperienza pratica pregressa, anche la comprensione dei paradigmi e dei design pattern utilizzati ha richiesto più tempo del previsto.
Essendo stato inoltre il primo progetto ad essere affrontato (in coppia, ma anche in generale), l’organizzazione è stata più complessa di quanto atteso portando quindi a molti rallentamenti, soprattutto per la parte di Controller; in quest’ultima infatti si sono si sono verificati numerosi bug e problemi che sono stati risolti solo con l’utilizzo del debugger.

Attività	Giovanni Battista Matteazzi	Caterina Vallotto
Studio del framework Qt	15	15
Studio e progettazione	6	6
Sviluppo codice Model	10	6
Sviluppo codice View	7	11
Sviluppo codice Controller	13	13
Debugging e fix	6	6
Test	3	3
Stesura relazione	5	5
Totale	**65**	**65**

7 - Suddivisione delle attività progettuali

Elenco della suddivisione delle attività progettuali:

- Giovanni Battista Matteazzi:
    - Model: importazione ed esportazione dei dati in formato json
    - View: classi per la visualizzazione dell'insieme dei sensori
    - Controller: metodi necessari per la creazione e l'eliminazione dei sensori

- Caterina Vallotto:
    - Model: struttura dei sensori e classi per il polimorfismo
    - View: classi per la visualizzazione del singolo sensore, classi per il pattern Visitor
    - Controller: metodi necessari per la modifica, classe per il pattern Visitor

Si noti che comunque una buona parte del lavoro è stato svolto in collaborazione, soprattutto per risolvere i problemi incontrati durante l'implementazione dell'applicazione. Per quanto riguarda invece la parte svolta singolarmente si è cercato di mantenere il monte ore similare per avere un'equa distribuzione del carico di lavoro.