Relazione progetto CleanroomSim

Emanuele Artusi, matricola 2068223

A.A. 2023-24

Indice

1 Introduzione

CleanroomSim è un software di gestione sensori in grado di simulare la raccolta dati dei principali sensori tipici utilizzati nelle camere bianche. Una camera bianca (in inglese "cleanroom") è una stanza ad atmosfera controllata: pressione atmosferica, umidità relativa ed inquinamento particellare sono parametri che vengono costantemente monitorati. L'aria immessa è filtrata attraverso poderosi filtri HEPA. Le camere bianche trovano ampio utilizzo nell'industria di microelettronica ed in particolare nella lavorazione dei semiconduttori, ambienti nei quali è indispensabile garantire pulizia ed assenza di elettrostaticità. Le particelle in sospensione nell'aria infatti sono in grado di danneggiare irreparabilmente le micro-fotoincisioni che formano i chip: basti pensare che la particella di fumo di una sigaretta ha un diametro maggiore della distanza che c'è fra la testina magnetica di lettura/scrittura e la superficie magnetica del piatto di un classico disco rigido. Esistono varie classificazioni internazionali per le camere bianche, questo software fa riferimento allo standard maggiormente utilizzato: l'ISO 14644-1. Maggiori dettagli nella sezione 5. CleanroomSim quindi è in grado di gestire tre tipi di sensore:

- Igrometro: registra l'umidità relativa dell'aria, utile per misurare il rischio di creazione di correnti elettrostatiche o la formazione di condensa;
- Manometro: registra la pressione all'interno della camera bianca, infatti avere una pressione leggermente più alta non permette all'aria non filtrata di entrare all'interno se viene aperta la porta;
- Contatore di particelle: rileva l'effettiva presenza di particelle, anche in base al loro diametro.

Ciascun sensore può essere collegato alla rete elettrica oppure alimentato a batteria, che si consuma avviando simulazioni. Maggiori dettagli nella sezione 5.

2 Modello logico

Fare riferimento a Figura 1. La classe *AbstractSensor* è la classe base astratta per tutti i tipi di sensore e rappresenta le informazioni comuni: identificatore univoco, nome, descrizione, stato collegato alla rete elettrica e percentuale di carica della batteria e relativi getters e setters. Inoltre sono presenti dei metodi per la gestione dei vari *Observers* e *Visitors* oltre a dei metodi per la gestione delle simulazioni. Questi ultimi in particolare verranno implementati in modo diverso dalle classi dei vari sensori: *Hygrometer*, *Manometer* e *ParticleCounter*. Tutte e tre le classi di sensori concreti aggiungono specifiche statistiche di interesse, tra cui valore medio, valore massimo e minimo, deviazione media assoluta. In particolare la classe *ParticleCounter* utilizza dati definiti attraverso la classe *Particles*, che dispone di un proprio metodo per la simulazione raccolta dati, la ridefinizione di alcuni operatori oltre che la definizioni di metodi ausiliari per

il calcolo del valore massimo, minimo e la media. Il "tipo di dato" *Particles* altro non è che un array di sei numeri interi, ciascuno rappresentante in questo ordine:

1. conteggio particelle di diametro $\geq 0.1\,\mu$m;
2. conteggio particelle di diametro $\geq 0.2\,\mu$m;
3. conteggio particelle di diametro $\geq 0.3\,\mu$m;
4. conteggio particelle di diametro $\geq 0.5\,\mu$m;
5. conteggio particelle di diametro $\geq 1\,\mu$m;
6. conteggio particelle di diametro $\geq 5\,\mu$m.

La classe *Observer* presenta due metodi distinti: *notify(AbstractSensor&)*, utilizzato per le normali notifiche di modifica nella classe, e *destructor()* per la notifica delle eliminazioni dal momento che l'implementazione con il sistema di Qt ha creato problemi in fase di test (valori nulli).

3 Polimorfismo

L'utilizzo principale del polimorfismo non banale si ritrova nell'ambito della visualizzazione grafica dei sensori in due classi: *SensorViewer* e *ChartsVisitor*.

3.1 SensorViewer

Permette di costruire widget in base al sensore che interessa rappresentare poiché integra la visualizzazione di tutti i campi specifici dei vari sensori. Ad esempio, per la visualizzazione di un manometro visualizzerà il valore di soglia, ma non il massimo e la media come per un contatore di particelle.

3.2 ChartsVisitor

Si occupa della costruzione dei vari grafici per mostrare la raccolta dati di ogni sensore. Dal momento che ogni sensore raccoglie dati differenti è opportuno visualizzarli in grafici differenti e questa classe ha il compito di generare il grafico giusto per il sensore d'interesse. Ad esempio, per un igrometro un grafico a linea, per un manometro un grafico a punti e per un contatore di particelle un grafico a barre verticali in cui ogni sestupla indica una raccolta di dati *Particles*.

4 Persistenza dei dati

Per la persistenza dei dati viene utilizzato il formato JSON, un unico file per ciascuna camera bianca "simulata". I sensori ed i dati vengono salvati in un array JSON tramite associazioni chiave-valore. Si utilizza la chiave *"Type"* per discriminare in lettura il tipo del sensore.


Manometer
- data: std::vector<int>
- man: double
- threshold: int
+ accept(SensorVisitor&): void {query}
+ generateSimulation(): void
+ getData(): std::vector<int>* {query}
+ getMan(): double {query}
+ getThreshold(): int {query}
+ Manometer(unsigned int&, std::string&, std::string, bool&, unsigned int&)
+ Manometer(Manometer&)
+ ~Manometer()
+ setData(std::vector<int>&): void
+ setMan(double&): void
+ setThreshold(int&): void
+ simulate(): void
+ updateMan(): void
AbstractSensor
- batteryPercentage: unsigned int
- description: std::string
- identifier: unsigned int
- name: std::string
observersList: std::vector<SensorObserver*>
- plugged: bool
+ AbstractSensor(unsigned int&, std::string&, std::string, bool&, unsigned int&)
+ AbstractSensor(AbstractSensor&)
+ ~AbstractSensor()
+ accept(SensorVisitor&): void {query}
+ addObserver(SensorObserver*): void
+ generateSimulation(): void
+ getBatteryPercentage(): unsigned int {query}
+ getDescription(): std::string {query}
+ getIdentifier(): unsigned int {query}
+ getName(): std::string {query}
+ isPlugged(): bool {query}
+ removeObserver(SensorObserver*): void
+ setBatteryPercentage(unsigned int&): void
+ setDescription(std::string&): void
+ setIdentifier(unsigned int&): void
+ setName(std::string&): void
+ setPlugged(bool&): void
+ simulate(): void
SensorVisitor
+ SensorVisitor()
+ ~SensorVisitor()
+ visit(Hygrometer&): void
+ visit(Manometer&): void
+ visit(ParticleCounter&): void
SensorObserver
+ destructor(): void
+ notify(AbstractSensor&): void
+ SensorObserver()
+ ~SensorObserver()
ParticleCounter
- avg: Particles
- data: std::vector<Particles>
- desiredStandard: unsigned int
- max: Particles
- min: Particles
+ accept(SensorVisitor&): void {query}
+ generateSimulation(): void
+ getAvg(): Particles* {query}
+ getData(): std::vector<Particles>* {query}
+ getDesiredStandard(): unsigned int {query}
+ getMax(): Particles* {query}
+ getMin(): Particles* {query}
+ ParticleCounter(unsigned int&, std::string&, std::string, bool&, unsigned int&)
+ ParticleCounter(ParticleCounter&)
+ ~ParticleCounter()
+ setAvg(Particles&): void
+ setData(std::vector<Particles>&): void
+ setDesiredStandard(unsigned int&): void
+ setMax(Particles&): void
+ setMin(Particles&): void
+ simulate(): void
+ updateAvg(): void
+ updateMax(): void
+ updateMin(): void
+ updateStats(): void
Hygrometer
- avg: double
- data: std::vector<double>
- optimalAvg: double
+ accept(SensorVisitor&): void {query}
+ generateSimulation(): void
+ getAvg(): double {query}
+ getData(): std::vector<double>* {query}
+ getOptimalAvg(): double {query}
+ Hygrometer(unsigned int&, std::string&, std::string, bool&, unsigned int&)
+ Hygrometer(Hygrometer&)
+ ~Hygrometer()
+ setAvg(double&): void
+ setData(std::vector<double>&): void
+ setOptimalAvg(double&): void
+ simulate(): void
+ updateAvg(): void
Particles
- values: int [1..6] ([6])
+ generateMaximum(Particles&, Particles&): void
+ generateMinimum(Particles&, Particles&): void
+ generateSimulation(unsigned int&): void
+ getDotFiveumCount(): int {query}
+ getDotOneumCount(): int {query}
+ getDotThreeumCount(): int {query}
+ getDotTwoumCount(): int {query}
+ getFiveumCount(): int {query}
+ getOneumCount(): int {query}
+ operator/(int&): Particles
+ operator+(Particles&): Particles
+ operator=(Particles&): Particles&
+ operator=(int&): Particles&
+ Particles(int&, int&, int&, int&, int&, int&)
+ Particles(Particles&)
+ ~Particles()
+ setDotFiveumCount(int&): void
+ setDotOneumCount(int&): void
+ setDotThreeumCount(int&): void
+ setDotTwoumCount(int&): void
+ setFiveumCount(int&): void
+ setOneumCount(int&): void

5 Funzionalità implementate

Le funzionalità implementate vengono divise in due categorie: funzionali ed estetiche.

5.1 Funzionali

- gestione operazioni CRUD di vari sensori;
- gestione simulazione semi-interattiva di vari sensori: l'utente può modificare alcuni parametri chiave (ad esempio una soglia per un manometro o lo standard ISO per un contatore di particelle) i quali influenzano la generazione della raccolta dati;
- gestione livello batteria sensori: se il sensore è collegato alla rete elettrica, il livello batteria è sempre al massimo; altrimenti, ciascun sensore ha un proprio consumo di batteria per ciascuna simulazione avviata. **Attenzione**: se la batteria è scarica avviare una simulazione **NON** genererà una nuova raccolta dati! Per ricaricare, collegare e scollegare dalla rete elettrica;
- conversione e salvataggio in formato JSON;
- ricerca in tempo reale case-insensitive.

5.2 Estetiche

Vedere Figura 2 per avere un esempio di interfaccia grafica.

- utilizzo di icone, stili e pulsanti;
- barra degli strumenti con relative scorciatoie da tastiera (es.: Ctrl+S per salvare, Ctrl+O per aprire un file). Altre scorciatoie logiche come Enter per applicare le modifiche;
- controllo della presenza di modifiche non salvate, segnalato da un asterisco (*) nel titolo della finestra;
- possibilità di ridimensionare la finestra principale e la lista di sensori a lato;
- utilizzo di grafici differenti;
- utilizzo di icone per indicare il livello di batteria/alimentazione e per segnalare che il sensore ha rilevato dati fuori norma (oltre che segnare in grassetto il valore anomalo): per un igrometro significa che la media valori supera la media desiderata, per un manometro significa che la varianza supera di 2.5 la soglia desiderata, per un contatore di particelle significa che la raccolta dati con più particelle non rientra nello standard ISO desiderato. Vedere Tabella 1 per gli standard ISO di riferimento, fonte Wikipedia.org.

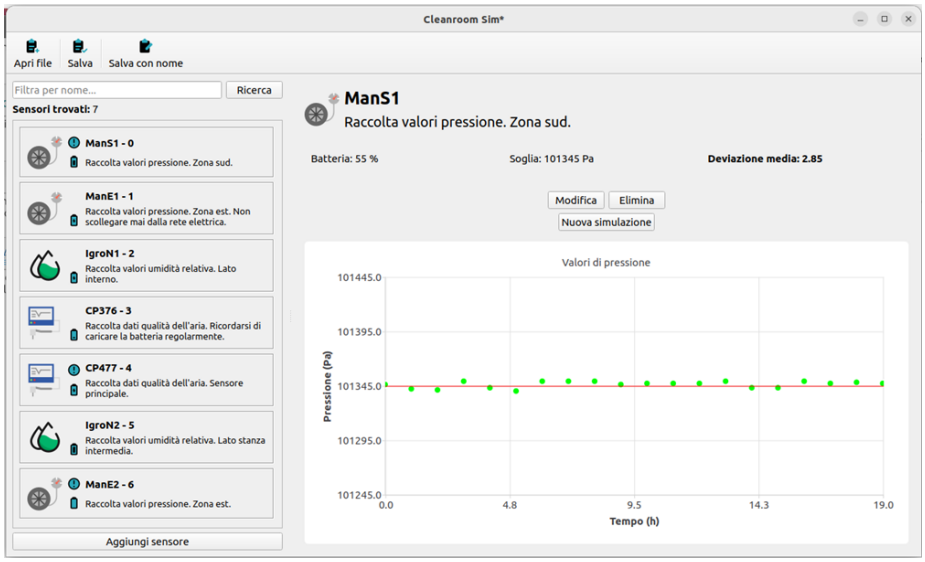

Figura 2: GUI

Classe	Numero massimo di particelle/m^3						FED STD 209E
	$\geq 0.1\,\mu$m	$\geq 0.2\,\mu$m	$\geq 0.3\,\mu$m	$\geq 0.5\,\mu$m	$\geq 1\,\mu$m	$\geq 5\,\mu$m	Corrispondente
ISO 1	10	2					
ISO 2	100	24	10	4			
ISO 3	1,000	237	102	35	8		Classe 1
ISO 4	10,000	2,370	1,020	352	83		Classe 10
ISO 5	100,000	23,700	10,200	3,520	832	29	Classe 100
ISO 6	1,000,000	237,000	102,000	35,200	8,320	293	Classe 1000
ISO 7				352,000	83,200	2,930	Classe 10,000
ISO 8				3,520,000	832,000	29,300	Classe 100,000
ISO 9				35,200,000	8,320,000	293,000	Aria nella stanza

Tabella 1: ISO 14644-1

Attività	Ore previste	Ore effettive
Studio e progettazione	10	8
Sviluppo del codice del modello	10	13
Studio del framework Qt	10	17
Sviluppo del codice della GUI	15	22
Test e debug	5	6
Stesura della relazione	2	4
Totale	52	70

Tabella 2: Rendicontazione ore.

6 Rendicontazione ore

Fare riferimento a Tabella 2. Il monte ore è stato superato perché, per quanto abbia già avuto esperienza nella modellazione di progetti in Java, che ha aiutato nelle prime fasi, non ho mai sviluppato GUI né utilizzato il framework Qt. Perciò ho dedicato più tempo del previsto per la seconda fase. Inoltre ho scritto la relazione in LaTeX ed avuto problemi con la figura del modello logico.

7 Note

Per i tipi primitivi (come *int*, *char*, ecc.), l'uso di *const* nei *setter* ha un impatto limitato, poiché il passaggio avviene generalmente per valore. Tuttavia, per coerenza e chiarezza, ho ritenuto avesse senso utilizzare *const* nei parametri di tipo puntatore o riferimento. Nella Figura 1, l'etichetta "{string}" sostituisce la keyword *const*, purtroppo una corruzione imprevista mi impedisce di correggere per tempo il file. Alcune funzionalità rimosse o non introdotte rimangono nel codice sorgente, commentate. Ad esempio un metodo in *Archive.cpp* per ottenere il percorso della cartella *Documenti* in base al sistema operativo. Durante l'esecuzione nella macchina virtuale del corso potrebbe comparire il warning:

```
Ignoring WAYLAND_DISPLAY on Gnome.

Use QT_QPA_PLATFORM=wayland to run on Wayland anyway.
```

significa che Qt ha rilevato che l'ambiente grafico GNOME sta utilizzando X11 anziché Wayland, anche se la variabile *WAYLAND_DISPLAY* è impostata. Questo avviene perché GNOME su Ubuntu spesso utilizza X11 come backend grafico per compatibilità e stabilità, soprattutto in ambienti come macchine virtuali. Forzare l'uso di Wayland potrebbe essere una soluzione ma dal momento che probabilmente è un warning che dipende dall'esecuzione in VMware non ho ritenuto necessario risolverlo.