UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento di
Ingegneria Informatica

Corso di laurea in
Ingegneria Informatica

Classe n. L-8 Ingegneria dell'informazione (D.M. 270/04)

Porting dell'interfaccia utente del ventilatore polmonare MVM al pattern MVP in QT

Candidato:
Federico Baldi

Matricola n.
1046073

Relatore:
Prof. Angelo Gargantini

Anno Accademico
2019/2020

Indice

Elenco delle figure

Capitolo 1

Introduzione

Durante il mio percorso di studi in Ingegneria Informatica e lavorando come software developer per un'importante azienda, mi sono molto appassionato alla creazione e allo sviluppo di nuovi software e algoritmi. Il mio progetto di tesi si concentra sul porting dell'interfaccia utente del ventilatore polmonare MVM (Mechanical Ventilator Milano), avente software scritto in Python, al pattern MVP (Model View Presenter) in QT tramite linguaggio C++. Nel ciclo di vita del software la fase di sviluppo è spesso vista come la parte più importante. Non sono invece da sottovalutare le successive fasi di manutenzione ed ampliamento, soprattutto per software di prodotti a lunga decorrenza. È quindi necessario implementare un'architettura adeguata, che non si limiti a permetterci di sviluppare e consegnare il software al cliente, ma anche di strutturarlo in una forma che consenta facili e repentine modifiche, con lo scopo di migliorarlo e aggiornarlo continuamente. Il progetto vuole dimostrare come l'utilizzo del design pattern MVP possa fornire notevoli vantaggi dal punto di vista dello sviluppo del software ma anche della sua manutenzione nel tempo, analizzandolo applicato a questo progetto, a confronto con altri pattern ed una sua evoluzione.

Capitolo 2

Campo di applicazione: la ventilazione polmonare

L'applicazione del progetto è l'interfaccia utente di un ventilatore polmonare, uno strumento utilizzato dal personale sanitario per assistere il paziente nella respirazione. Diventa quindi indispensabile avere un'interfaccia chiara, efficiente e dinamica, per monitorare costantemente e accuratamente le condizioni del paziente.

2.1 Ventilatore polmonare

Un ventilatore polmonare, detto anche respiratore nel linguaggio comune, è una macchina ad utilizzo medico che insuffla nei polmoni una determinata miscela di gas e ne consente l'espirazione, in base ad una frequenza impostata ed un appropriato regime di pressioni, ed è utilizzato nella ventilazione artificiale per aumentare o sostituire la ventilazione spontanea di un individuo che presenta insufficienza respiratoria/ventilatoria per malattie che colpiscono il polmone o la pompa toracica. Grazie alla regolazione di alcuni parametri, il ventilatore polmonare riesce ad effettuare una ventilazione "intelligente". Gli obiettivi di un ventilatore polmonare, quindi, possono essere riassunti come segue:

Figura 2.1 – Paziente sottoposto a ventilazione polmonare. In primo piano si osserva un respiratore.

Un ventilatore polmonare, detto anche respiratore nel linguaggio comune, è una macchina ad utilizzo medico che insuffla nei polmoni una determinata miscela di gas e ne consente l'espirazione, in base ad una frequenza impostata ed un appropriato regime di pressioni, ed è utilizzato nella ventilazione artificiale per aumentare o sostituire la ventilazione spontanea di un individuo che presenta insufficienza respiratoria/ventilatoria per malattie che colpiscono il polmone o la pompa toracica. Grazie alla regolazione di alcuni parametri, il ventilatore polmonare riesce ad effettuare una ventilazione "intelligente". Gli obiettivi di un ventilatore polmonare, quindi, possono essere riassunti come segue:

- Insufflare nei polmoni quantità controllate di aria o miscele gassose.
- Arrestare l'insufflazione.
- Lasciare espirare i gas insufflati.
- Ripetere queste operazioni continuativamente e anche per lunghi periodi.

La ventilazione polmonare viene eseguita sia nei pazienti in terapia intensiva, che non possono respirare da soli per diversi motivi (anche, per esempio, a seguito di un trauma), sia nei pazienti che soffrono di particolari patologie respiratorie, quali apnee ostruttive del sonno, asma o malattie neurologiche

come la distrofia muscolare. Negli ultimi mesi, in seguito alla pandemia Covid-19, questi dispositivi sono tristemente stati oggetto di crescente richiesta e si è presentata la necessità di aumentarne la produzione e la vendita.

2.2 Interfaccia utente di un ventilatore polmonare

L'interfaccia di un ventilatore polmonare deve riportare informazioni riguardanti le condizioni respiratorie del paziente, in maniera chiara e di facile lettura. I principali parametri riportati sono:

- **Volume**: misurato in ml, è la quantità di aria che entra o esce dai polmoni.
- **Flusso** (dell'aria): misurato in litri al secondo, è la velocità dello spostamento dell'aria attraverso il ventilatore.
- **Pressione**: misurata in cmH2O, è la forza motrice della movimentazione dell'aria.

Esistono altre due grandezze derivate dai parametri principali:

- **Resistenza**: è la pervietà che l'aria incontra.
- **Compliance**: è la cedevolezza delle pareti biologiche.

Nel nostro caso, il software di partenza MVM (Mechanical Ventilator Milano) è stato scritto in Python e si presenta come nella figura 2.2.

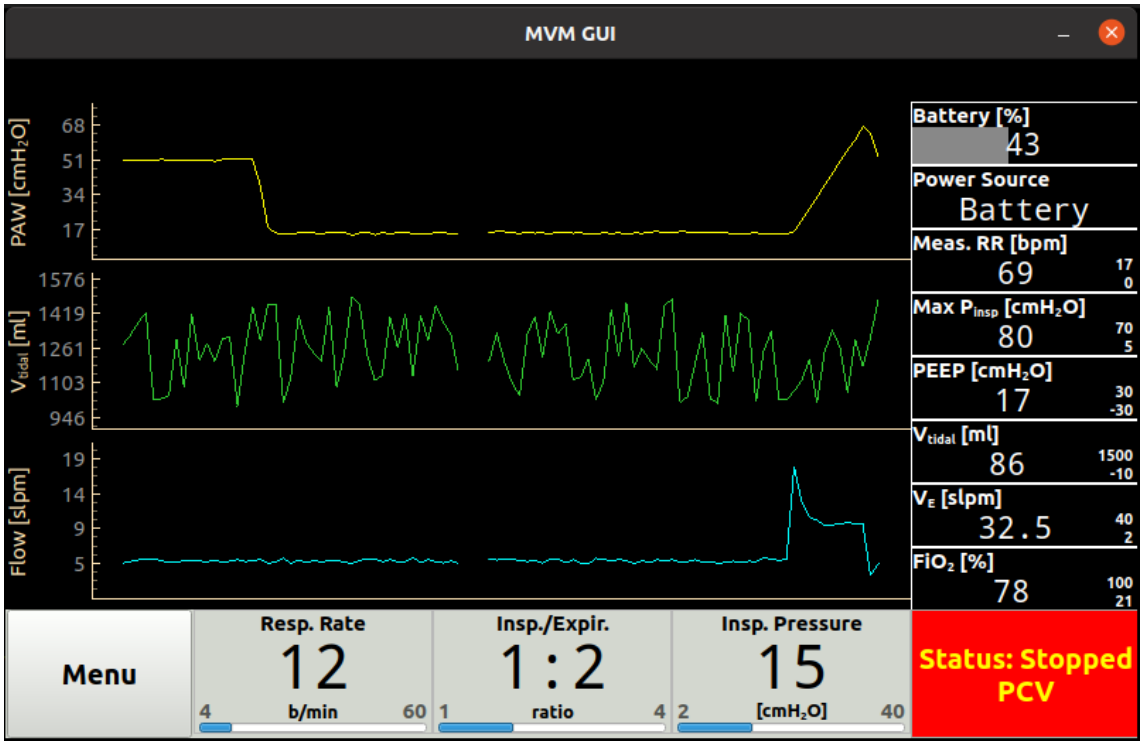

Figura 2.2 – Interfaccia utente dell'applicazione MVM scritta in Python

Come si può notare dalla figura 2.2, le tre informazioni più importanti, volume, flusso e pressione, sono riportate sottoforma di grafico, aggiornato ogni 100 ms. Infatti, è sicuramente necessario avere un tracciato nel tempo dell'andamento dei tre parametri. Sulla colonna di destra, invece, vengono riportati dati che mostrano solamente l'ultimo dato registrato.

Capitolo 3

Design Pattern

3.1 Stato dell'arte software development

Lo **sviluppo software** è il processo di ideazione, specifica, progettazione, programmazione, documentazione, test e correzione dei bug coinvolto nella creazione e gestione di applicazioni, framework o altri componenti software. Lo sviluppo del software è un processo di scrittura e mantenimento del codice sorgente, ma in un senso più ampio, include tutto ciò che è coinvolto tra la concezione del software desiderato fino alla manifestazione finale del software, a volte in un processo pianificato e strutturato. Pertanto, lo sviluppo di software può includere ricerca, nuovo sviluppo, prototipazione, modifica, riutilizzo, re-ingegnerizzazione, manutenzione o qualsiasi altra attività che si traduca in prodotti software. Per progetti di grande portata, ma ormai per qualsiasi prodotto che comprende un minimo di complessità, è necessario avere un approccio preciso e chiaro allo sviluppo del codice. Nell'ingegneria del software, un **design pattern** è una soluzione generale e riutilizzabile a un problema comune in un dato contesto nella progettazione del software. Non è un design finito che può essere trasformato direttamente in codice sorgente o macchina. Piuttosto, è una descrizione o un modello per come risolvere un problema che può essere utilizzato in molte situazioni diverse. I design pattern sono

best practice formalizzate che il programmatore può utilizzare per risolvere problemi comuni durante la progettazione di un'applicazione o di un sistema. I modelli di progettazione orientati agli oggetti in genere mostrano le relazioni e le interazioni tra classi o oggetti, senza specificare le classi o gli oggetti coinvolti nell'applicazione finale. I modelli di progettazione possono essere visti come un approccio strutturato alla programmazione tra i livelli di un paradigma di programmazione e un algoritmo concreto. Qualsiasi programmatore o team conosce e sfrutta diversi design pattern, in base alla situazione, e molte volte costruisce l'architettura e la composizione del software attorno ad uno di essi. Nel caso oggetto della mia tesi vogliamo dimostrare come il design pattern MVP (Model View Presenter) si presti perfettamente allo sviluppo di un software per sistemi embedded che necessitano di un'interfaccia grafica per l'utilizzatore.

3.2 Python vs C++

Python è un linguaggio di programmazione di più "alto livello" rispetto alla maggior parte degli altri linguaggi, orientato a oggetti, adatto, tra gli altri usi, a sviluppare applicazioni distribuite, scripting, computazione numerica e system testing. Spesso viene anche studiato tra i primi linguaggi per la sua somiglianza a uno pseudo-codice e di frequente viene usato per simulare la creazione di software grazie alla flessibilità di sperimentazione consentita da Python, che permette al programmatore di organizzare le idee durante lo sviluppo, come per esempio il back-end di un sito web tramite Flask o Django. È un linguaggio interpretato, traduce ed esegue ogni singola istruzione del programma. Legge ed esegue il codice sorgente del programma senza creare un file oggetto eseguibile. È quindi più lento rispetto alla compilazione. La compilazione traduce tutte le istruzioni di un programma in linguaggio macchina, creando un file eseguibile dal computer. La compilazione viene

eseguita da un software compilatore. Il C++, pur condividendo molti degli aspetti di un linguaggio di "alto livello", nonostante possa risultare di più difficile comprensione ed utilizzo, è un linguaggio compilato che può vantare appunto altri vantaggi. Oltre ad una maggiore velocità e minor utilizzo della memoria, è il più utilizzato in applicazioni embedded. L'occupazione certamente ridotta di un codice già compilato e quindi eseguibile rispetto a un codice da interpretare, e in questo caso necessitando di un interprete nell'applicazione, porta il C++ ad essere di gran lunga il linguaggio più utilizzato in prodotti di questa tipologia. Specialmente nel caso preso in esame, un ventilatore polmonare. Dopo quanto detto sono chiari i vantaggi concreti del C++ a discapito del Python. I costi di produzioni si ridurrebbero, di conseguenza l'hardware necessiterebbe di più basse specifiche per quanto riguarda le performance ed anche la memoria. Ciò potrebbe quindi trasformarsi in un'opportunità di miglioramento del prodotto tramite l'utilizzo o meno di altre periferiche o l'impiego di sensori di più alta qualità, che farebbero percepire al cliente acquistante il prodotto come di fascia più elevata.

3.3 MVP

Model-view-presenter (MVP), nato all'inizio del 2007, quando Microsoft introdusse applicazioni Microsoft Smart Client, è una derivazione del modello architettonico model-view-controller (MVC) e viene utilizzato per colmare le lacune di alcune delle carenze di MVC. Il suo più diffuso utilizzo è principalmente per la creazione di interfacce utente. Nel MVP, al posto del controller abbiamo il presenter che assume la funzionalità dell'"intermediario". Nel MVP, tutta la logica di presentazione è di responsabilità del presenter.

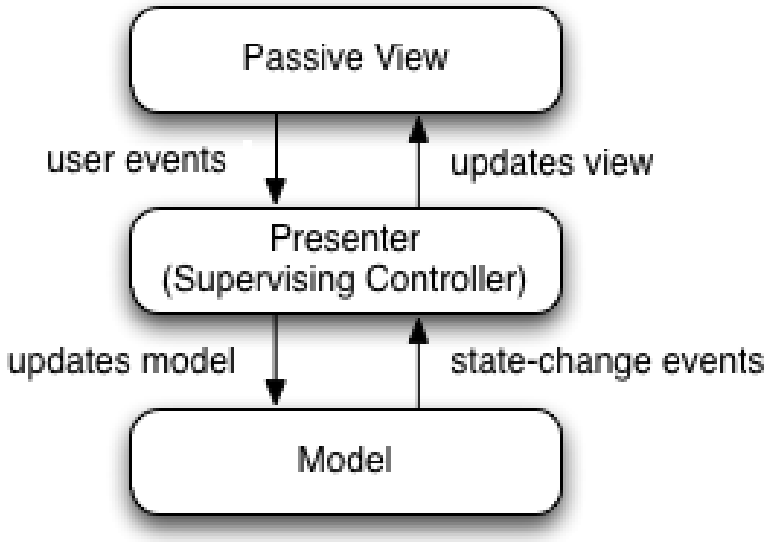

Figura 3.1 – Funzionamento di MPV design pattern

L'MVP è un design pattern tipico di un'architettura che comprende una parte di interfaccia grafica. Non è il design pattern adatto se parliamo di applicazioni o progetti i quali devono svolgere il loro compito senza la necessità di grafica o interazione con l'utente tramite di essa. È progettato per facilitare test di unità automatizzati e migliorare la separazione del codice tra i tre livelli.

Il **modello** (model) è un'interfaccia che definisce i dati da visualizzare o su cui agire in altro modo nell'interfaccia utente. Si occupa della gestione dei dati dell'applicativo e suo conseguente aggiornamento e modifica.

La **vista** (view) è un'interfaccia passiva che visualizza i dati del modello e indirizza i comandi utente (eventi) al presenter per agire su tali dati. Si limita, volutamente, a mostrare i dati e si concentra solamente sul come mostrarli, non li manipola in alcun modo.

Il **presentatore** (presenter) agisce sul modello e sulla vista. Recupera i dati dal modello e li formatta per la visualizzazione nella vista. Gestisce gli eventi che gli arrivano dalla vista e si comporta di conseguenza in base ai

parametri, eventualmente richiedendo un aggiornamento di dati al modello o un aggiornamento della vista.

Concretamente, il grado di logica consentito nella visualizzazione varia tra le diverse implementazioni. Ad un estremo, la visualizzazione è completamente passiva, inoltrando tutte le operazioni di interazione al presentatore. In questa formulazione, quando un utente interagisce con la vista (genera un evento), non fa altro che invocare un metodo del presentatore che non ha parametri e nessun valore di ritorno. Il presentatore quindi recupera i dati dalla vista tramite metodi definiti dall'interfaccia della vista. Infine, il presentatore opera sul modello e aggiorna la visualizzazione con i risultati dell'operazione. Altre versioni del model-view-presenter consentono una certa libertà rispetto a quale classe gestisce una particolare interazione, evento o comando. Questo è spesso più adatto per architetture basate sul web, dove la visualizzazione, che viene eseguita sul browser di un client, può essere il posto migliore per gestire una particolare interazione o comando per avere vantaggi di velocità, riducendo la latenza di risposta del server. Da un punto di vista della stratificazione, la classe presenter potrebbe essere considerata come appartenente al livello applicazione in un sistema di architettura multistrato, ma può anche essere vista come un livello presentatore a sé stante proprio tra il livello applicazione e il livello interfaccia utente. Avere componenti su diversi livelli, che sono indipendenti uno d'altro, i quali devono solamente conoscere le interfaccie con cui comunicano porta un notevole vantaggio. Dal punto di vista dello sviluppo infatti, più persone possono lavorare in parallelo allo stesso progetto poiché una volta concordate le interfacce tra i livelli, ognuno è libero di lavorare in autonomia sulla sua parte. Di conseguenza si può avere un team formato da diverse persone con competenze diverse e che si concentrino su un solo aspetto dell'architettura senza la necessità di conoscere molteplici linguaggi e metodologie. Il classico esempio è quello dello sviluppo delle viste, il quale può essere svolto da una persona non competente per quanto riguarda la

programmazione ma con una formazione su grafica e design.

3.4 MVP vs MVC vs MVVM

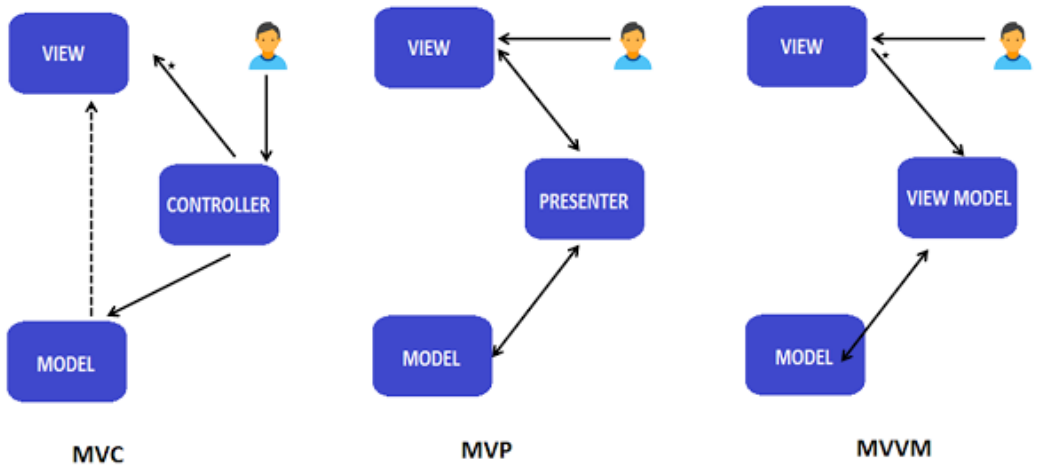

Figura 3.2 – Comparazione tra i tre design pattern MVC, MVP, MVVM.

3.4.1 MVC

Partiamo parlando del **MVC**, è un modello architettonico che è stata la prima scelta per la progettazione di applicazioni Web. MVC permette di creare applicazioni usando la logica della separazione tra livelli. Ciò riduce al minimo tutti gli sforzi necessari per estendere, testare e mantenere l'applicazione. Il processo si concentra, promuove il concetto di classe per le viste, la logica di data binding tra componenti e le business operation. Per riassumere, il modello dell'architettura MVC lavora per ridurre le dimensioni del codice rendendolo facilmente gestibile e più pulito. È molto comunemente usato. Tuttavia, si ritiene che l'MVC sia superato dai modelli MVP e MVVM molto più comuni. Rispetto al MVP di cui abbiamo parlato prima, la differenza principale è quella della sostituzione del controller con il presenter. Il controller è la parte più importante dell'architettura. È colui che prende le decisioni ed esiste tra

la vista e il modello. Il controller aggiorna la vista ogni volta che il modello cambia. Il controllore elabora i dati dopo aver ricevuto una richiesta dalla vista e prima di aggiornare qualsiasi cosa nel nostro database con il modello. È un frammento che comunica con la vista e il modello prendendo l'input dell'utente dalla vista.

Principali vantaggi:

- L'architettura MVC accelera lo sviluppo in parallelo. Consente agli sviluppatori di lavorare sui componenti contemporaneamente. Ad esempio, uno sviluppatore sarà in grado di lavorare sulla vista mentre un altro potrà lavorare contemporaneamente sul controller. Questo permette di completare il progetto più velocemente del previsto.

- Con il modello MVC, è possibile creare più viste per un solo modello. Anche qui, la business logic e i dati sono separati dalla vista e quindi la duplicazione del codice è molto limitata.

- L'architettura MVC si integra con tutti i framework JavaScript più diffusi. Ciò significa che l'architettura MVC può essere utilizzata con tutti i browser, file PDF e widget desktop.

- Si può continuare ad apportare modifiche frequenti come per i caratteri, i layout dello schermo, i colori e l'aggiunta di un nuovo supporto per dispositivi per tablet o telefoni cellulari senza influenzare il modello.

- - L'architettura MVC restituisce i dati utilizzando gli stessi componenti con qualsiasi interfaccia.

- Il modello di progettazione MVC supporta applicazioni Web o pagine Web ottimizzate per la SEO. In questo modo, gli URL SEO friendly possono essere utilizzati per generare più visite all'app o al sito.

Mentre tra gli svantaggi:

- La navigazione nel framework può essere complessa perché introduce nuovi livelli di astrazione e quindi può essere difficile da capire.

- Gli sviluppatori che coordinano e sviluppano l'architettura utilizzando MVC devono essere esperti in più tecnologie per lavorare in modo efficiente con questo framework complesso.

3.4.2 MVP

Passando invece di nuovo al **MVP**, senza ripetere il funzionamento dello stesso, voglio approfondire i vantaggi che porta oltre a quelli che sopracitati derivanti dal MVC:

- È più facile eseguire il debug di qualsiasi applicazione poiché MVP introduce tre diversi livelli di astrazioni e vi è una maggiore indipendenza tra i livelli al contrario del MVC. Nel MVP, gli sviluppatori possono eseguire test di unità durante lo sviluppo dell'applicazione poiché la business logic è completamente separata da View.

- Nel caso di MVP, si possono avere più presenter per controllare le viste. In questo modo il codice può essere riutilizzato in modo migliore. Questo è abbastanza vantaggioso anche se può comuqnue avere un presenter solo per controllare diverse visualizzazioni come nel MVC, ma qui abbiamo più flessibilità e scelta in base alla situazione.

- L'MVP mantiene la business logic e la persistent separate dalle classi Activity e Fragment. Ciò consente di avere una separazione dei livelli sicuramente migliore.

Invece, per quanto riguarda gli svantaggi:

- Viene introdotto un'ulteriore step per l'assorbimento e manipolazione dei dati.

- L'esperienza e le informazioni del programmatore non influiscono sull'uso corretto del modello.

- Troppo complesso e dispendioso per piccole o semplici applicazioni.

3.4.3 MVVM

Il rivale diretto del MVP è invece il **MVVM** È il modo più ben organizzato e più riutilizzabile per strutturare il codice. Nel pattern MVVM, troviamo un data-binding bidirezionale tra view e view-model. Il pattern MVVM organizza e struttura il codice in applicazioni manutenibili e testabili. Ciò ha i vantaggi di un test più semplice e di una modularità più elevata, riducendo anche la quantità di codice che funge solamente da collante necessario da scrivere per connettere i vari livelli. Il pattern Model-View-ViewModel (MVVM) aiuta a separare in modo pulito la business logic e la logica di presentazione di qualsiasi applicazione dalla sua interfaccia utente (UI). Questo aiuta ad affrontare una serie di problemi di sviluppo che rendono un'applicazione più facile da testare, mantenere ed evolvere. Inoltre, migliora notevolmente le opportunità di riutilizzo del codice e consente agli sviluppatori e ai progettisti dell'interfaccia utente di collaborare facilmente durante lo sviluppo delle rispettive parti di un'app. Il **View-Model** è il componente che lo distingue dalla logica di utilizzo degli altri due design pattern. È idealmente un modello per la visualizzazione dell'app. Il view-model è un componente chiave della triade poiché introduce la separazione della presentazione. È il concetto di tenere separata la gestione della vista dal modello. Invece di rendere il modello informato sulla visualizzazione deo data dell'utente, converte i dati direttamente nel formato di visualizzazione. È responsabile del coordinamento delle interazioni della vista con qualsiasi classe di modello richiesta. Le proprietà e i comandi forniti dal view-model caratterizzano le funzionalità che devono essere offerte dall'interfaccia utente. Questa logica porta ad avere

anche una view diversa da come è stata intesa fin ora. Infatti la vista si prende alcune libertà per rendere i dati più presentabili. La vista è attiva nel MVVM rispetto ad una vista passiva che non ha idea del modello ed è completamente manipolata da un controllore / presentatore. La vista nel MVVM ha comportamenti, eventi e associazioni di dati che in ultima analisi possono richiedere conoscenza e idea del modello sottostante e view-model. La vista non è responsabile del mantenimento del suo stato. Invece, si sincronizza con il view-model. A una vista sono associati anche comportamenti, come l'accettazione dell'input dell'utente.

Arrivando ai vantaggi del design pattern:

- Con MVVN è possibile raggiungere le parti più piccole o complesse del codice e apportare modifiche serenamente grazie alla separazione di diversi tipi di livelli in modo più pulito. Questo accelera il lavoro e aiuta con richieste di successivi e repentini rilasci di versione.

- La scrittura di unit test sulla logica e algoritmi di base dei livelli diventa più semplice poiché tutte le dipendenze interne ed esterne rimangono nel codice contenente la core logic dell'applicazione.

- I progettisti e gli sviluppatori possono lavorare in modo indipendente e simultaneo sui propri componenti durante il processo di sviluppo. I progettisti possono concentrarsi sulla visualizzazione, mentre gli sviluppatori possono lavorare fianco a fianco sul modello di visualizzazione e su altri componenti.

- L'interfaccia utente dell'app può essere riprogettata senza toccare il codice, ma per fare ciò la vista deve essere implementata interamente in XAML. Quindi, una nuova versione della vista dovrebbe funzionare ed incastrarsi perfettamente con il view-model esistente.

Parlando di svantaggi dobbiamo considerare:

- La comunicazione tra i vari componenti nel MVVM e il data binding può essere faticosa e impegnativa.
- La riusabilità del codice delle viste e del view-model è difficile da ottenere poiché si ha grossa dipendenza tra essi.
- L'utilizzo dei view-model e del relativo stato nelle viste nidificate e nelle interfacce utente complesse risulta difficile e problematico da implementare.
- Per i principianti è difficile da utilizzare immediatamente e con sicurezza.

3.5 Conclusioni

Sia MVP che MVVM derivano da MVC. La principale differenza tra MVC e le sue derivate è la dipendenza che ogni livello ha su gli altri livelli, nonché quanto strettamente legati sono l'uno con l'altro. In MVC, la vista si trova in cima all'architettura con il controller subito dopo. I modelli si trovano sotto il controller, quindi le visualizzazioni conoscono i controller e i controller conoscono i modelli. Qui, le viste hanno accesso diretto ai modelli. L'esposizione del modello completo alla vista potrebbe avere problemi di sicurezza e prestazioni, a seconda della complessità della nostra applicazione. In MVP, il controller viene sostituito dal presenter. I presenter rimangono allo stesso livello delle viste, ascoltano gli eventi dalla vista e dal modello e collegano le azioni tra di loro. Poiché non esiste un sistema per associare le viste ai view-models, dipende da ciascuna vista implementare un'interfaccia che consenta al presenter di interagire con la vista. MVVM, tuttavia, consente di creare diramazioni specifiche del view-model, contenenti informazioni di stato e logiche, evitando la necessità di esporre l'intero modello a una vista. Il view-model non è tenuto a fare riferimento a una vista contrariamente da quanto accade per il presenter dell'MVP. La vista può legarsi alle proprietà sul view-model,

che espone i dati contenuti nei modelli alla vista. Tuttavia, è necessario un livello di interpretazione tra view-model e vista, ciò può comportare costi di prestazioni. La complessità di questa interpretazione varia anche dalla copia dei dati alla loro manipolazione in una forma che vorremmo che la vista vedesse. MVC non presenta questo problema, perché l'intero Modello è prontamente disponibile e tale manipolazione è facile da evitare. Constatiamo che le due derivazioni del MVC sono entrambe molto valide per lo sviluppo software, possiamo sbilanciarci nel dire che l'MVP sia più consigliato per applicazioni di tipo desktop o similari. Mentre l'MVVM può portare più vantaggi nelle architetture di applicazioni web o mobile.

Capitolo 4

Progetto

4.1 Strumenti e tecnologie

L'applicazione è stata realizzata in ambiente Linux. Ho utilizzato una distribuzione 20.04 LTS Ubuntu di Linux, sfruttandola sottoforma di macchina virtuale. Per fare ciò mi sono servito del tool di Oracle per il setup e utilizzo della macchina virtuale "VM VirtualBox". Il linguaggio di programmazione è il C++, scelto, come spiegato in precedenza, per confrontarlo al Python e ai suoi vantaggi di applicazione in progetti di questo tipo. L'ambiente di sviluppo è QT, utilizzato sia per il progetto in Python che per quello C++. QT viene utilizzato per lo sviluppo di interfacce utente grafiche (GUI) e applicazioni multipiattaforma che vengono eseguite su tutte le principali piattaforme desktop e sulla maggior parte delle piattaforme mobili o integrate (embedded). La maggior parte dei programmi GUI creati con QT hanno un'interfaccia native-looking. Possono essere sviluppati anche programmi non GUI, come strumenti da riga di comando e console per server. Un esempio di tale programma non GUI che utilizza QT è il framework web Cutelyst. Qt supporta vari compilatori, incluso il compilatore GCC C++, la suite Visual Studio, PHP tramite un'estensione per PHP5. QT fornisce anche QtQuick, che include un linguaggio di scripting dichiarativo chiamato QML che consente di utilizzare

JavaScript per fornire la logica. Con QtQuick è diventato possibile lo sviluppo rapido di applicazioni per dispositivi mobili, mentre la logica può ancora essere scritta anche con codice nativo per ottenere le migliori prestazioni possibili. Altre funzionalità includono l'accesso al database SQL, parsing XML, parsing JSON, la gestione dei thread e il supporto di rete. Ho voluto utilizzare una piccola libreria GNU GPL sottoforma di due file qcustomplot.c e qcustomplot.h. Ciò mi ha permesso di replicare la grafica originale del progetto ed aggiungere altre due GUI sempre con l'utilizzo dei plot.

4.2 Architettura del codice, flusso dati

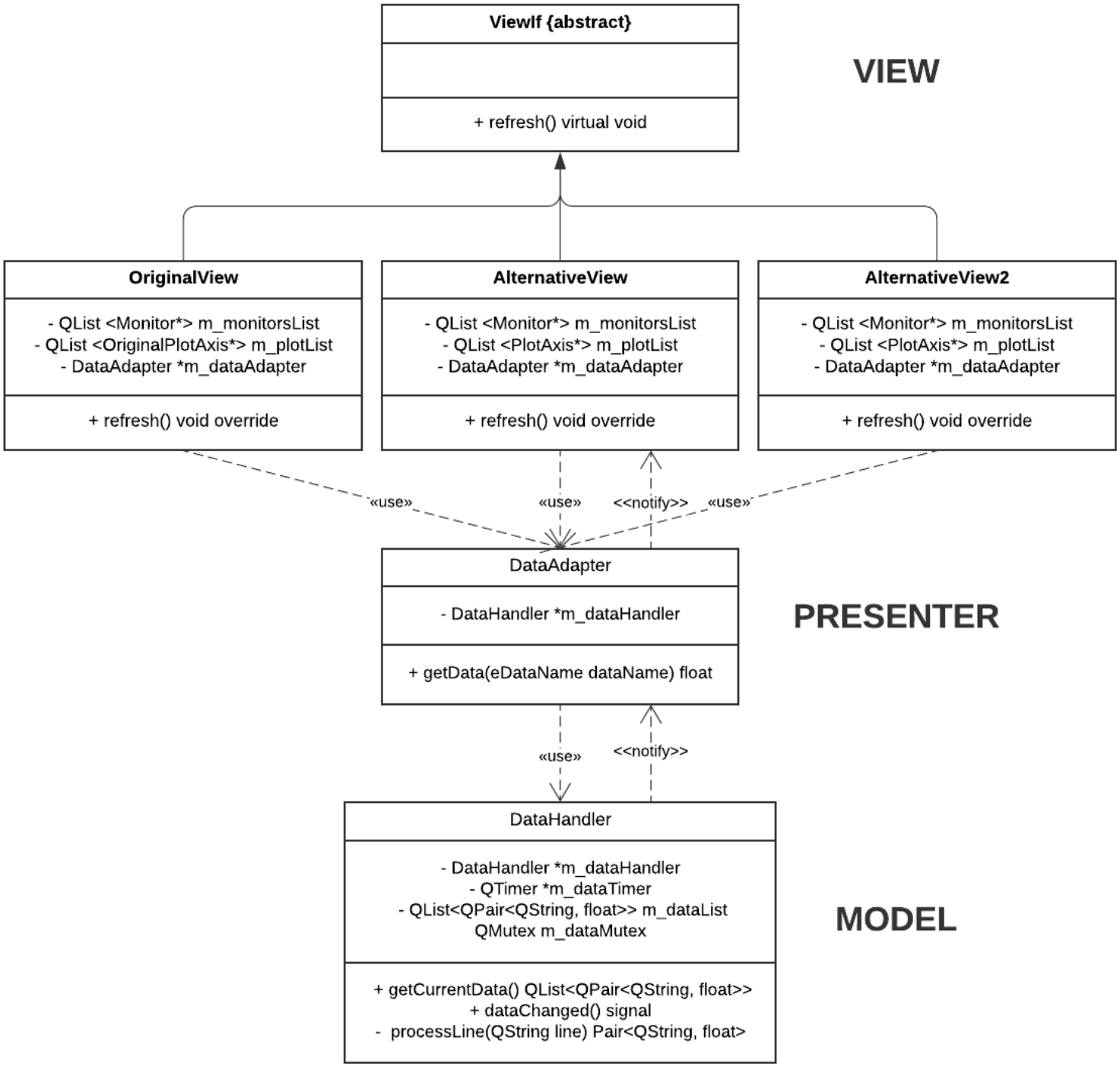

Figura 4.1 – Schema delle classi UML.

Dallo schema delle classi (figura 4.1) si può chiaramente osservare la struttura tipica del design pattern MVP, infatti troviamo i tre elementi essenziali: model,

presenter e view. Il ruolo del modello in questo caso è in carico alla classe "DataHandler", la quale in un'applicazione contenuta come questa, è sufficiente a gestire e processare i dati necessari al suo funzionamento. In particolare, si occupa di leggere i dati generati in forma grezza e processarli per poterli salvare e rendere disponibili alle interfacce di livello superiore. I dati al momento sono generati randomicamente in modo da simulare una situazione il più possibile vicina ai dati reali sul campo. Ogni qual volta vi è un aggiornamento di dati, tramite il signal "dataChanged()" viene notificato agli strati superiori il cambio di stato; il segnale verrà poi propagato dal presenter e le view che hanno interesse a ricevere aggiornamento. Questa classe è istanziata e poi passata al "DataAdapter". Il presenter appunto, è la classe "DataAdapter", la quale, essendo il componente al centro del pattern, ha interazioni sia con il modello che con le view. In particolare, una volta ricevuto un segnale di cambio stato da parte del modello lo propaga alle view ed eventualmente aggiorna il suo stato interno e compie delle azioni su eventuali altri modelli. Le view nel MVP, come detto in precedenza, hanno come obiettivo quello di occuparsi solamente della parte grafica. Infatti, una volta ricevuto il segnale di cambio stato dei dati di loro interesse, operano un refresh richiedendo al presenter direttamente i dati tramite "getData(eDataName)".

4.3 X Macro vs Java enum

Nello sviluppo del progetto ho introdotto un costrutto C++ che utilizzo spesso nella mia esperienza lavorativa. Le cosiddette X Macro, queste hanno l'obbiettivo di garantire la coerenza tra liste di dati in cui i vari elementi devono apparire nello stesso ordine (tabelle di dati). Spesso è necessario generare liste di dati collegate fra di loro. Ad esempio: un enumerativo ed un elenco di stringhe, le quali sono in relazione diretta con l'enumerativo. Se fossero implementate in modo indipendente, la manutenibilità ne soffrirebbe.

L'aggiunta di un nuovo valore nell'enumerativo comporterebbe l'aggiunta di una nuova stringa (e viceversa); la modifica andrebbe implementata in due sezioni diverse del codice. Non sarebbe automatico il fatto di avere il corretto allineamento tra valori dell'enumerativo e stringhe. La **X Macro** consente di avere tutti i dati in una sola sezione, dove l'allineamento è automatico. Infatti, i dati sono strutturati in un'unica tabella. Va usata dove vi sono dati, anche disomogenei, strutturati in forma tabellare.

```
.h
#define DATANAME_X \
  X(dnPressure , "pressure", "PAW"      , "cmH20") \
  X(dnO2       , "o2"      , "FiO2"     , "%") \
  X(dnFlow     , "flow"    , "Flow"     , "slpm") \
  X(dnBPM      , "bpm"     , "Meas. RR", "bpm") \

typedef enum
{
  #define X(enumerator, name, showName, measureUnit)
   enumerator,
    DATANAME_X
  #undef X
  dnMax
}eDataName;
static const char * m_titleNameList[];
static const char * m_measureUnitNameList[];

.c
const char * Class::m_titleNameList[] =
{
  #define X(enumerator, name, showName, measureUnit) showName,
  DATANAME_X
  #undef X
};
```

```
25 const char * Class::m_measureUnitNameList[] =
26 {
27   #define X(enumerator, name, showName, measureUnit)
      measureUnit,
28   DATANAME_X
29   #undef X
30 };
31 Class::Class()
32 {
33   String title(m_titleNameList[dataName]);
34   title.append(" [");
35   title.append(m_measureUnitNameList[dataName]);
36   title.append("]");
37   cout << title;
38 }
```

È interssante fare un confronto con gli enumerativi in **Java**. Infatti gli enumerativi in Java sono dichiarati ed utilizzati come normali classi. Estendono implicitamente java.lang.Enum e non possono estendere nessun altra classe, però possono implementare diverse interfacce. Non possiamo istanziare un'enumerativo usando l'operatore new. L'importante differenza è che possiamo anche fornire vari valori come variabili membro con le costanti dell'enumerativo.

```
1 enum DATANAME
2 {
3   dnPressure("pressure","PAW","cmH20"),
4   dnO2("o2","PAW","cmH20");
5   dnFlow("flow","Flow","slpm");
6   dnBPM("bpm","BPM","bpm");
7
8   String description, name, unit;
9   private DATANAME(String d,String n, String u)
```

```
10    {
11      description = d;
12      name = n;
13      unit = u;
14    }
15  }
16  class Main
17  {
18    public static void main(String[] args)
19    {
20      DATANAME x = DATANAME.dnPressure;
21      System.out.println(x.description);
22    }
23  }
```

Fondamentalmente, i membri dell'enumerativo sono istanze dello stesso, possiamo quindi accedere ai loro attributi sapendo che sono collegati e la coerenza tra di essi è mantenuta.

Capitolo 5

Implementazione

5.1 Gestione delle viste

La gestione delle view è stata implementata tramite una "MainWindow" che riceve in input nel suo costruttore: il presenter, la lista delle view da visualizzare e il widget parent. La lista delle view è una lista di puntatori a elementi di tipo classe astratta "ViewIf" come da figura 4.1. Ciò obbliga il chiamante a fornire una lista di oggetti view che hanno un'implementazione concreta di tale classe e quindi reimplementino la funzione "refresh()".

```
1  MainWindow::MainWindow(DataAdapter *dataAdapter,          QList<ViewIf *>
       viewList, QWidget *parent):
2  QMainWindow(parent),
3  ui(new Ui::MainWindow),
4  m_dataAdapter(dataAdapter),
5  m_viewList(viewList)
6  {
7    ui->setupUi(this);
8    for (int index = 0; index < m_viewList.size(); index++)
9    {
10     ui->ShowStackedWidget->addWidget(m_viewList.at(index));
11   }
```

```
12   ui->ShowStackedWidget->setCurrentIndex(1);
13   bool connectionOk = QObject::connect(m_dataAdapter, SIGNAL(dataChanged())
       , this, SLOT(refresh()), Qt::QueuedConnection);
14   assert(connectionOk);
15 }
```

La lista che viene passata alla "MainWindow" è costruita nella classe "Initializer" che istanza le classi specializzate delle view. L'"Inizializer" compie anche la funzione di istanziare il modello "DataHandler" in modo safe e connetterlo tramite SIGNAL/SLOT al presenter "DataAdapter".

```
1  /*!
2   * \brief Initializer::start
3   * Create a thread in a safe mode (signal-slot start) where DataHandler is
       running.
4   * Populate a QList of ViewIf objectes input to the MainWindow.
5   */
6  void Initializer::start()
7  {
8    m_dataHandler = new DataHandler();
9    m_dataHandler->moveToThread(&m_handlerThread);
10   m_handlerThread.start();
11   bool connectionOK = QObject::connect(this, SIGNAL(dataHandlerSetUp_signal
       ()), m_dataHandler, SLOT(setupDataHandler()), Qt::AutoConnection);
12   assert(connectionOK);
13
14   m_dataAdapater = new DataAdapter(m_dataHandler);
15   emit dataHandlerSetUp_signal();
16
17   m_viewList.append(new OriginalView(m_dataAdapater));
18   m_viewList.append(new AlternativeView(m_dataAdapater));
19   m_viewList.append(new AlternativeView2(m_dataAdapater));
20   m_mainWindow = new MainWindow(m_dataAdapater, m_viewList);
21   m_mainWindow->show();
```

```
22 }
```

Bisogna sicuramente menzionare che in un'applicazione più complessa e più grande, sia il ruolo del presenter che quello del modello è ricoperto da molte classi. Per comodità e pulizia ci sono quindi degli aggregatori o meglio, collettori che permettono alle view di ricevere dati e inviare input in maniera trasparente, senza la necessaria conoscenza di molteplici soggetti. Oltre a ciò, è consuetudine avere anche le classi astratte dei presenter e modelli, avendo quindi interfacce virtuali e reimplementazioni che permettono di avere un alto grado di mantenibilità e testabilità. In pratica, i presenter ricevono nel loro prototipo del costruttore puntatori ad oggetti di classi astratte dei modelli e così le view con i presenter. Avere questo livello di modularità permette di creare test di unità, creando specializzazioni di test ad-hoc di classi astratte, che permettono appunto di testare gli oggetti senza modifiche intrusive all'interno delle classi stesse.

5.2 Implementazione

Figura 5.1 – View n.1 dell'interfaccia del ventilatore polmonare in C++

L'applicazione è composta da una mainwindow che comprende un widget stack, cioè un oggetto grafico il quale è composto da una lista di widget e ne visualizza uno alla volta. Al di sotto di esso ho inserito due semplici pulsanti i quali vanno a cambiare lo stato del widget stack e vanno a cambiera la schermata/view visualizata. Nella prima view ho voluto riproporre la medesima interfaccia grafica presente nel progetto originale scritto in Python. È composta da una lista di oggetti di classe "Monitor" sulla destra, cioè di una grafica standard di visualizzazione di un dato con il suo titolo ed unità di misura. Mentre nella parte centrale vi sono i tre grafici aggiornati in tempo reale che monitorano:

1. **PAW** Mean Airway Pressure: definisce la pressione media applicata durante la ventilazione meccanica a pressione positiva e si correla

con la ventilazione alveolare, l'ossigenazione arteriosa e le prestazioni emodinamiche.

2. **Tidal Volume**: è il volume polmonare che rappresenta il volume normale di aria spostato tra la normale inspirazione ed espirazione quando non viene applicato uno sforzo supplementare. In un giovane adulto umano sano, il volume corretto è di circa 500 ml per inspirazione o 7 ml/kg.

3. **Flow**: La velocità di flusso, o velocità di flusso inspiratorio di picco, è il flusso massimo che viene erogato dal ventilatore in base al flusso di respirazione impostato per il paziente. Se la velocità di flusso di picco è troppo bassa per il paziente, possono verificarsi dispnea, asincronia paziente-ventilatore e aumento del lavoro respiratorio.

La tipologia di grafico comprende una banda nera che attraversa da sinistra a destra il plot e lo aggiorna simultaneamente. Questo tipo di grafico e comportamento è uno dei più utilizzati in campo medico e al quale dottori e infermieri sono abituati.

Passiamo ora alla seconda view:

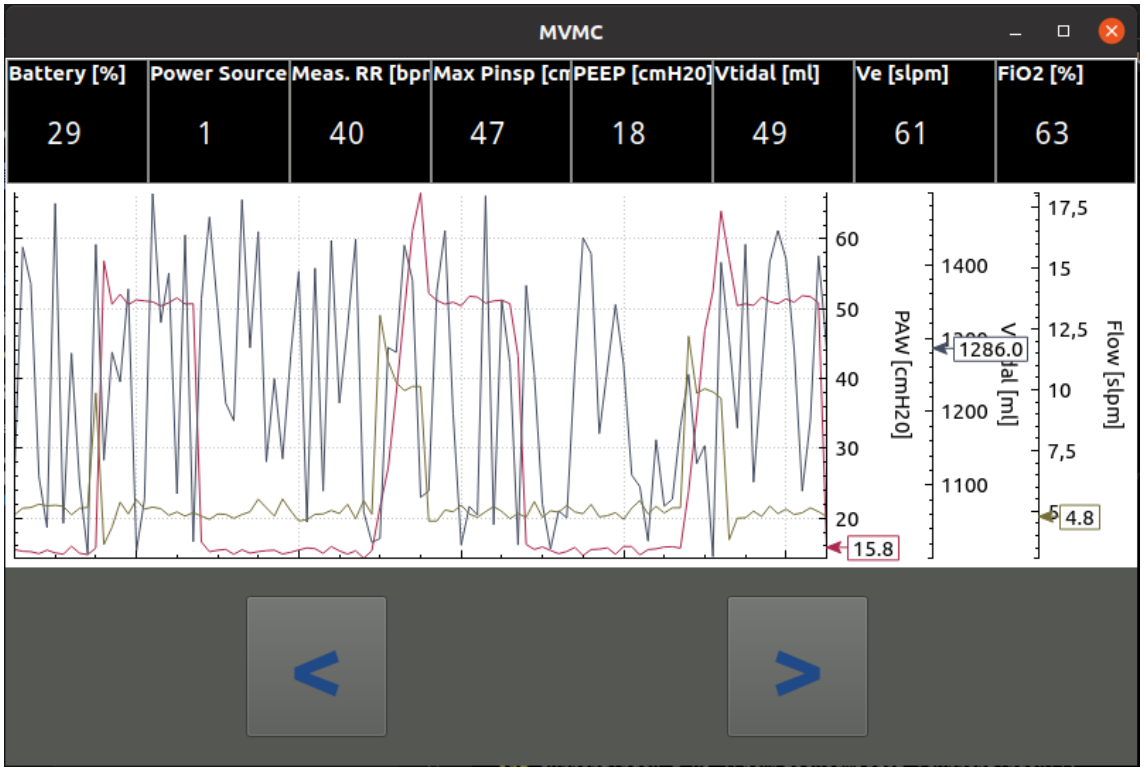

Figura 5.2 – View n.2 dell'interfaccia del ventilatore polmonare in C++

Cliccando sul pulsante con la freccia a destra si cambia lo stato dello stack widget e viene mostrata la view che si trova alla seconda posizione dello stack (figura 5.2). Qui ho voluto mostrare come, cambiando la grafica della view, posso, tramite gli stessi dati e stessa logica, avere un comportamento diverso. In particolare, ho spostato la lista di monitors sopra orientata orizontalmente. E ho lasciato più spazio alla visione dei grafici. Tramite la libreria qcustomplot, ho creato un grafico che contiene tutti e tre i parametri sopracitati e a destra inserito le loro scale relative con un pennino che segue l'andamento del grafico. Questa view è stata pensata solo a scopo didattico, infatti rende difficile una lettura chiara dei dati.

Propongo nella terza view una versione secondo me più chiara e pulita del grafico:

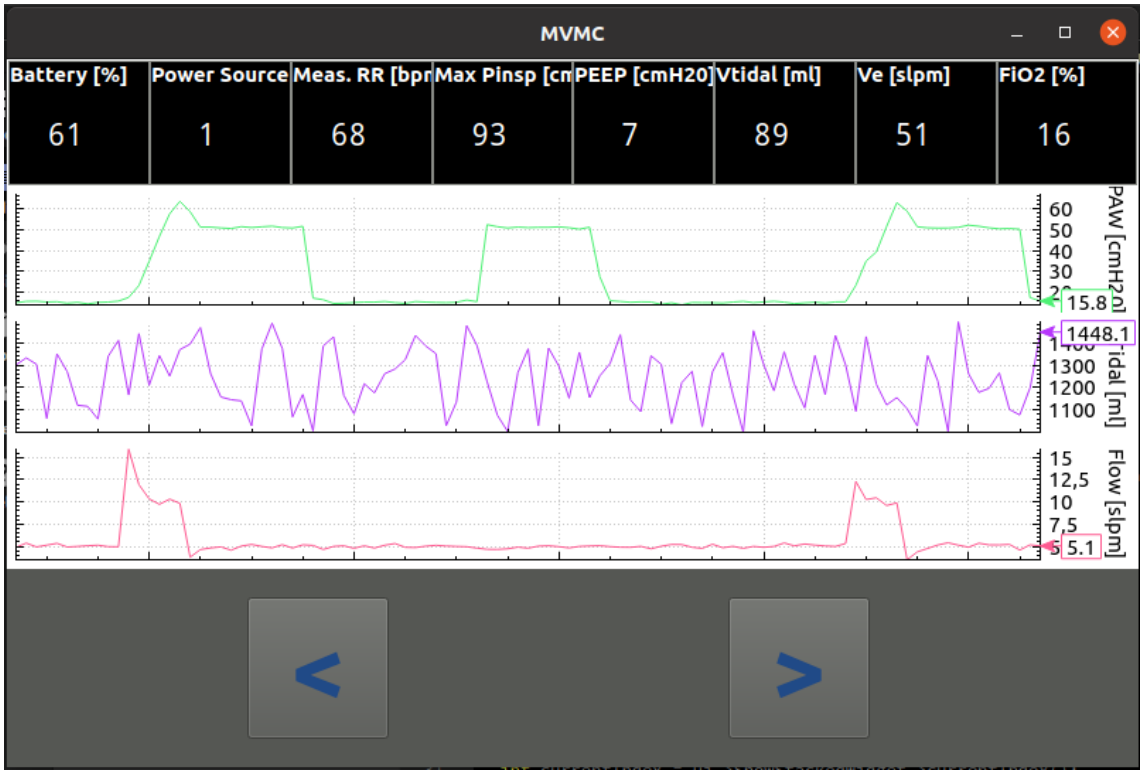

Figura 5.3 – View n.3 dell'interfaccia del ventilatore polmonare in C++

In questa view ho separato i tre grafici ma mantenendo lo stesso stile. Sicuramente risulta più chiaro che avendo un solo grafico con tre dati all'interno.

Capitolo 6

Sviluppi futuri

L'obbiettivo di questo progetto era dimostrare, anche dal punto di vista pratico, i vantaggi del design pattern. Oltre ad aver studiato in modo teorico l'MVP, abbiamo visto come si adatti facilmente ad un'applicazione di tipo embedded senza necessità di complessi costrutti o tools. Le tre view proposte mostrano il principale obbiettivo di un design pattern come questo, cioè la suddivisione pulita e corretta dei livelli dell'applicativo. La view essendo separata dal presenter richiede solamente i dati di sua pertinenza da visualizzare e non ha visione degli strati sotto di essa. Il conseguente ampliamento e sviluppo del design pattern in un'applicazione di più grande portata e complessità è rendere dinamiche le view tramite i dati. Ho avuto modo di studiare e progettare un sistema di questo tipo nella mia esperienza lavorativa.

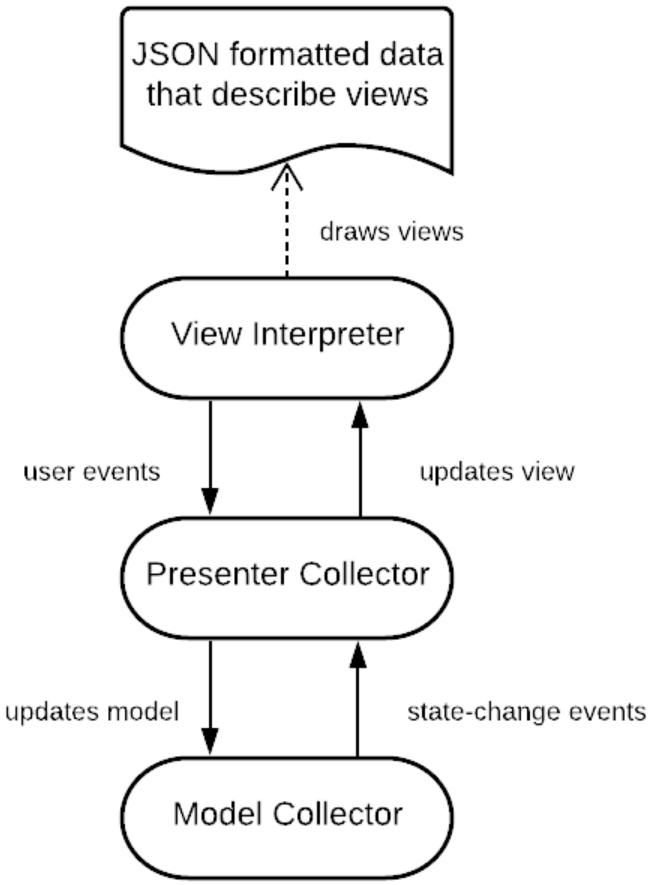

Figura 6.1 – Modello MVP con ampliamento di tipo data driven

Come precedentemente accennato, troviamo i collettori di model e presenter. Questi si occupano semplicemente di aggregare presenter e modelli per poter rendere le interfacce trasparenti a loro livelli adiacenti. La differenza principale è l'introduzione del view interpreter. In base alla UI richiamata da essere visualizzata, è l'oggetto che si occupa della lettura del file JSON che descrive i dati e la struttura della view. Secondo regole pre-impostate svolge due importanti funzioni: disegnare la grafica stessa e mettere in connessione i componenti che la costituiscono con lo strato del presenter. Un importante vantaggio di questo ampliamento, è la possibilità di sviluppare molteplici prodotti con lo stesso codice di base. E' possibile avere più applicazioni con

schermi anche di dimensioni diverse ma che avendo un file JSON contenente i dati per disegnare schermate che si adattino alle dimensioni e necessità diverse.

Bibliografia e Siti consultati

Bibliografia

[1] Larman Craig, Cabibbo Luca. *Applicare UML e i pattern. Analisi e progettazione orientata agli oggetti, Quarta Edizione*. Pearson, 2016.

Siti consultati

[S1] *Ventilatore Meccanico* https://it.wikipedia.org/wiki/Ventilatore_meccanico

[S2] *Ventilatore Polmonare: come funziona?* https://biomedicalcue.it/ventilatore-polmonare/10600/

[S3] *Model-View-Presenter* https://en.wikipedia.org/wiki/Model-view-presenter

[S4] *Comparison Between MVC Vs MVP Vs MVVM* https://www.angularminds.com/blog/article/mvc-vs-mvp-mvvm.html

[S5] *MVC vs. MVP vs. MVVM?* https://medium.com/@jerrysbusiness/mvc-vs-mvp-vs-mvvm-a48cee3036de

[S6] *QCustomPlot* https://www.qcustomplot.com/index.php/introduction

Ringraziamenti

Questo elaborato di tesi rappresenta per me l'ultimo passo verso la fine del mio percorso universitario. Desidero quindi spendere alcune parole per ringraziare le persone che ho conosciuto e coloro che mi sono stati accanto.
Voglio ringraziare il professor Gargantini per avermi dato la possibilità di sviluppare la tesi su un argomento così interessante e stimolante.
Ringrazio tutti gli amici di ogni compagnia per avermi dato la voglia e la carica per continuare questo percorso. Un grazie per tutte le giornate e serate passate insieme a ridere e scherzare.
Voglio ringraziare la mia fidanzata Greta, la quale è riuscita sempre a rimettermi sui binari quando uscivo di strada ed è sempre riuscita ad essere un punto di riferimento per me grazie alla sua instancabile forza e volontà di fare. A tal proposito voglio ringraziare la sua famiglia per avermi accolto e fin da subito trattato come uno di loro.
Ringrazio tutti i miei colleghi dell'azienda in cui lavoro: W&H Sterilization, che mi hanno insegnato tantissimo sia professionalmente che umanamente.
Infine voglio ringraziare la mia famiglia a cui devo moltissimo: i miei nonni, le mie zie, mio fratello e in particolare mia mamma che mi hanno sempre sostenuto, aiutato per qualsiasi cosa e soprattutto sempre creduto in me.