POLITECNICO
DI TORINO

Politecnico di Torino
corso di laurea in ingegneria informatica

RELAZIONE DI TIROCINIO

Studente: Luca Ghio

Numero di matricola: 173428

Tutore accademico: Prof. Bartolomeo Montrucchio

Tutore aziendale: Ing. Alessandro Sappia

Indice

1 Sommario

Questa relazione ha lo scopo di illustrare ciò che è stato fatto durante l'attività di tirocinio svoltasi presso Biotechware e quali sono stati i risultati.

La commercializzazione di un dispositivo medico portatile progettato interamente inhouse comporta la cooperazione e l'integrazione di varie componenti, e tra queste la componente software svolge un ruolo primario: è infatti necessario un sistema software sviluppato ad hoc per il dispositivo affinché l'utente possa interagire al meglio con esso.

La prima parte della relazione approfondirà alcune problematiche relative allo sviluppo di un software che, nonostante i limiti operativi imposti dalla portabilità del dispositivo, deve garantire una certa fluidità di utilizzo e una certa flessibilità, e analizzerà alcune possibili soluzioni a tali problematiche.

Poiché il dispositivo è progettato per essere usato in mobilità, la connessione alla rete può non essere sempre disponibile. È pertanto necessario garantire la piena funzionalità del dispositivo anche in assenza della connettività: tutte le operazioni svolte mentre il dispositivo è disconnesso devono essere memorizzate nella memoria interna, e successivamente devono essere automaticamente registrate sul server dell'azienda non appena il dispositivo viene connesso alla rete.

Le operazioni di sincronizzazione non devono mai interrompere la funzionalità del dispositivo, e devono essere portate a termine nei limiti operativi inevitabilmente imposti dalla portabilità del dispositivo: la dimensione finita della memoria di archiviazione interna, che limita il salvataggio dei dati in locale, e le ridotte prestazioni delle connessioni mobili, che limitano lo scambio di dati con il server, comportano la necessità di individuare i dati di cui l'utente ha bisogno di più per conservarli nella memoria locale pronti all'uso.

La relazione quindi proseguirà nella descrizione della struttura e del funzionamento del codice che implementa le soluzioni progettate sviluppato durante il tirocinio. Dopo una panoramica generale, la trattazione dettaglierà ogni singola classe seguendo un approccio top-down: si partirà dalle classi che si occupano della visualizzazione dei dati nell'interfaccia grafica, quindi si proseguirà con le classi che implementano i modelli della struttura model–view–controller e che gestiscono le chiamate asincrone al server

e la memorizzazione dei dati nel database locale, per arrivare alle classi di più basso livello e più prossime al server dell'azienda.

2 Introduzione

2.1 L'azienda

Il tirocinio è stato svolto presso Biotechware, una startup nata ufficialmente nel febbraio 2011 ed ospitata all'interno dell'incubatore I3P del Politecnico di Torino.

Il gruppo di lavoro è formato da laureati in ingegneria informatica ed elettronica che collaborano al progetto per la realizzazione di un dispositivo di telemedicina professionale e tecnologicamente avanzato.

2.1.1 Descrizione del dispositivo

Il dispositivo, chiamato CardioPad Pro, è un elettrocardiografo portatile dotato di uno schermo sensibile al tocco e capace di connettersi ad Internet per l'accesso ad una piattaforma cloud.

Le sue funzionalità consistono nella registrazione di elettrocardiogrammi (ECG) tramite degli elettrodi collegati al dispositivo stesso, e nel loro successivo invio ad una piattaforma cloud tramite connessioni Internet mobile.

2.2 Il tirocinio

2.2.1 Obiettivi del tirocinio

Il tirocinio consiste nella progettazione e nella programmazione di un'applicazione software che dialoga con il server remoto dell'azienda simulando le condizioni di lavoro all'interno del dispositivo.

Il codice realizzato durante il tirocinio potrà essere integrato, opportunamente riadattato, nel software principale del dispositivo come modulo di backend che opera in maniera automatica sotto l'interfaccia grafica, al fine di minimizzare i problemi derivanti dall'intermittenza della connessione alla rete e dare al dispositivo la piena operatività e fluidità in ogni situazione senza l'intervento dell'utente.

2.2.2 Strumenti utilizzati

Come linguaggio di programmazione è stato adottato il C++, potenziato con le librerie incluse nel framework Qt (versione 4), che sono molto utilizzate per lo sviluppo di applicazioni multi-piattaforma e stanno alla base dello sviluppo di software KDE.

Il codice è stato interamente scritto all'interno dell'ambiente di sviluppo integrato Qt Creator (installato su un sistema operativo GNU/Linux), è stato testato tramite Google Test, è stato tracciato nel tempo grazie a Git, ed è stato documentato con Doxygen. È stato anche messo in funzione un server allo scopo di simulare il server dell'azienda durante le sessioni di test.

Oltre alla libreria di Google Test, il progetto si è sviluppato sulla libreria esterna XML-RPC for C and C++[1]. Il file del database locale è in formato SQLite, che offre migliori prestazioni rispetto a un normale database SQL in sistemi embedded con una potenza di calcolo ridotta.

2.2.3 Motivazioni della scelta del tirocinio

Ho scelto questo tirocinio perché sono sempre stato appassionato di programmazione, e il tirocinio mi ha dato la possibilità di approfondire il campo della programmazione ad oggetti. Non avevo mai usato le librerie Qt ma ne conoscevo le potenzialità, e questo tirocinio mi ha offerto la possibilità di prendere familiarità con esse.

Un altro contributo fondamentale alla scelta del tirocinio è stata la possibilità di lavorare nell'ambiente GNU/Linux che uso abitualmente: il tirocinio mi ha abituato a lavorare con il terminale e mi ha permesso di approfondire diversi aspetti di questo sistema operativo.

2.2.4 Motivazioni della relazione

Questa relazione intende analizzare il lavoro che è stato svolto e illustrare al lettore tutte le problematiche che sono sorte e le soluzioni che sono state ideate durante l'attività di tirocinio.

Questa relazione vuole anche essere una sorta di documentazione scritta del codice stesso, che potrà essere consultata se ci sarà la necessità di integrare una parte di esso nel software principale del dispositivo e di ricordare i motivi che hanno condotto a certe soluzioni implementative. Per motivi di scorrevolezza del testo è stato scelto di non addentrarsi troppo in dettagli tecnici, per i quali esiste la documentazione tecnica allegata.

[1] D'ora in avanti la libreria verrà chiamata più semplicemente "XML-RPC".

3 Analisi del problema

3.1 Funzionalità del dispositivo

Il codice è stato sviluppato per un dispositivo medico portatile con funzionalità di sincronizzazione con il server tramite la rete Internet.

Il dispositivo dispone di molte funzionalità per offrire all'utente un'esperienza completa, ma nella trattazione che segue ci si concentrerà sulle caratteristiche che sono state di specifico interesse per l'attività di tirocinio. Si tralasceranno anche i dettagli inerenti la struttura del database del server aziendale poiché non sono di particolare interesse per la trattazione del codice sviluppato durante l'attività di tirocinio.

Immaginiamo un utilizzo tipo del dispositivo: un operatore sanitario vuole sottoporre un paziente ad un elettrocardiogramma. L'utente allora accende il dispositivo e apre la lista dei pazienti per cercare quello che deve essere eseguire l'esame. Se è fortunato compare già tra i primi della lista, altrimenti inizia a digitare il suo nome e cognome per avviare la ricerca. Una volta trovato, lo seleziona e avvia l'elettrocardiogramma. Il dispositivo è in grado di visualizzare il tracciato grafico e di salvarlo, in modo che la prossima volta che l'utente accede a quel paziente può ritrovare tutti i tracciati passati.

3.2 Requisiti di progetto

Tutta questa serie di operazioni deve avvenire in maniera fluida e senza interruzioni dovute ai collegamenti con il server. Per esempio, per la lista dei pazienti è richiesta la massima reattività possibile: molte entry della lista vengono infatti scaricate dal server sul momento, e l'utente deve accorgersi il meno possibile di questo meccanismo e deve attendere meno tempo possibile per avere ciò che sta cercando.

Inoltre, il dispositivo deve poter garantire la massima operatività possibile anche in assenza della connessione alla rete. Per esempio, il tracciato dell'elettrocardiogramma non deve andare perso se il server non è disponibile in quel momento, ma va salvato nella memoria interna e va trasferito sul server non appena il dispositivo viene collegato alla rete. Questa operazione deve essere svolta automaticamente, senza l'intervento dell'utente.

Sebbene dal dispositivo l'utente non possa eliminare un paziente o modificarne le informazioni personali, queste operazioni possono essere svolte se invece l'utente accede direttamente al database memorizzato sul server dell'azienda. Questo vuol dire

che se l'operatore elimina un paziente dal server e quel paziente è però salvato nella memoria interna del dispositivo, quest'ultimo deve essere in grado di capire che quel paziente non è più esistente e quindi di eliminarlo dalla memoria interna nel più breve tempo possibile per garantire la consistenza dei dati.

Il motore del dispositivo deve quindi avere una certa "intelligenza" nello svolgimento di una serie di operazioni di cui l'utente non è consapevole, e per questo motivo può essere pensato come una sorta di backend, in contrapposizione al frontend che è la parte visibile all'utente e che nasconde l'operato del backend.

3.3 Limiti operativi

Il dispositivo avrebbe un funzionamento perfetto se fosse caratterizzato da una potenza di calcolo e da una memoria interna paragonabili a quelle degli odierni PC, e soprattutto avesse a disposizione una connessione ad Internet ad alta velocità.

Purtroppo non è così: il software sviluppato non è sufficiente che funzioni su un comune PC, ma è destinato a lavorare in condizioni di lavoro limitate dalla portabilità del dispositivo e dall'intermittenza della connessione alla rete.

La portabilità del dispositivo comporta inevitabilmente una ridotta disponibilità di spazio interno di memorizzazione, in particolare rispetto al server. Nella memoria locale quindi non può essere salvato l'intero database del server, ma solo una piccola parte di esso; il software deve quindi essere in grado di individuare di volta in volta quali dati sono necessari per soddisfare le richieste dell'utente, evitando di salvare dei dati che invece non servono.

Il dispositivo non può accedere direttamente al database del server ma tutto deve passare tramite Internet, in particolare tramite connessioni Wi-Fi e soprattutto 3G. È quindi facile che il dispositivo venga portato anche in aree dove la connessione è scarsa e quindi la quantità di dati che si può scaricare è ridotta. Oltretutto una connessione 3G costituisce per l'utente un costo economico, che aumenta all'aumentare del tempo di connessione e della quantità di dati scaricati. La ridotta banda di connessione a disposizione va quindi usata solo per il download e l'upload dei dati utili senza sprechi.

3.4 Soluzioni adottate

Poiché il database completo che è memorizzato dal server è molto vasto e non può essere scaricato e salvato interamente nel dispositivo per limiti sia di spazio sia di ban-

da del collegamento, si rende pertanto necessario un meccanismo che sia in grado di scaricare e salvare solo i dati che realmente servono all'utente.

Il tirocinio si è concentrato principalmente sulla gestione della lista dei pazienti in modo incrementale e dinamico, con l'obiettivo di permettere la ricerca di un dato paziente anche se questo non è presente nel database locale.

La lista viene visualizzata in un riquadro dotato di una barra di scorrimento che permette di scorrere la lista stessa. La strategia adottata è stata quella di visualizzare immediatamente tutta la lista dei pazienti memorizzati nel database locale, in modo da riempire in un primo momento almeno le prime righe della tabella.

Subito dopo si verifica se nell'area visibile del riquadro ci sono delle righe vuote oppure se la lista risulta più lunga, e solo nel primo caso si passa allo scaricamento di un numero di pazienti dal server pari a quanto è strettamente necessario per far apparire la lista piena, in modo da ottimizzare l'utilizzo di banda e non esaurire troppo presto lo spazio su disco. Se l'utente intende scorrere ulteriormente la lista o cercare un determinato paziente che non è presente in essa, allora lo scaricamento di pazienti continua per soddisfare le richieste dell'utente.

Le connessioni al server devono avvenire sempre in maniera asincrona, cioè non devono mai provocare un blocco dell'interfaccia grafica, anche se la risposta del server tarda ad arrivare a causa della lentezza della connessione. La soluzione adottata per risolvere questa problematica è stata quella di eseguire la chiamata al server in un thread separato rispetto al thread principale dell'applicazione, sfruttando le funzionalità per il multithreading offerte dalle librerie Qt.

Nel corso del tirocinio si è anche cercato un sistema per evitare che il dispositivo si blocchi perché lo spazio su disco è finito, e quindi cercare un modo per farlo continuare a funzionare anche in questa evenienza. La soluzione scelta è stata quella di assegnare una priorità ad ogni paziente che è memorizzato nel database locale: quando lo spazio su disco sta per esaurirsi, il software provvede autonomamente ad eliminare il paziente a priorità minima dal database locale per far posto ad un altro paziente. Un ottimo criterio per individuare i pazienti da eliminare per primi è quello della data di ultimo accesso: un paziente a cui l'utente ha fatto accesso da molto tempo difficilmente gli servirà nell'immediato, mentre è più probabile che l'utente faccia degli accessi consecutivi nel tempo a un gruppo di pazienti ristretto. Il paziente eliminato dal database

locale ovviamente non viene eliminato dal server, e potrà essere riscaricato in locale in un secondo momento se si rende necessario.

L'affidabilità del sistema è ulteriormente migliorata da un sistema a cronometro che periodicamente controlla la consistenza dei dati, riparando eventuali errori e incoerenze tra le informazioni memorizzate nel database locale e il server remoto e mantenendo tutto il sistema completamente sincronizzato.

4 Implementazione software

4.1 Fondamenti teorici

4.1.1 Il design pattern model–view-controller

Il design pattern model–view–controller è un concetto cardine molto usato nella programmazione ad oggetti, ed è il principio fondamentale si cui si è basato lo sviluppo dell'applicazione in oggetto.

Il design pattern model–view–controller si basa sulla separazione dei compiti tra i componenti software che si occupano della gestione dei dati:

- il model fornisce i metodi per l'accesso alle informazioni;
- la view visualizza le informazioni contenute nel model all'utente;
- il controller riceve i comandi dell'utente attraverso la view e li attua modificando lo stato degli altri due componenti.

Il vantaggio principale di questo approccio consiste nella possibilità di implementare ciascun componente in modo indipendente dagli altri, in modo che se si intendono apportare dei cambiamenti a quel componente non è necessario rivedere tutto il codice ma è sufficiente prestare attenzione alle connessioni tra un componente e l'altro.

4.1.2 Il meccanismo di signal e slot

Il meccanismo di signal e slot è una delle principali funzionalità offerte dalle librerie Qt appositamente studiata per permettere la cooperazione tra le varie classi C++ all'interno di un'applicazione.

Il principio di funzionamento è semplice: quando una classe emette un certo segnale, una o più classi possono catturarlo ed eseguire uno slot. Il segnale può portare con sé dei parametri, utili se lo slot ha bisogno di elaborare dei dati prodotti dalla classe che ha lanciato il segnale.

Per determinare una coppia signal e slot è necessario effettuare una cosiddetta connessione, tipicamente nel costruttore di una delle due classi coinvolte nella connessione. La connessione avviene sempre tra due classi distinte: la classe che emette il segnale è detta "sender", la classe che cattura il segnale ed esegue lo slot è detta "receiver".

4.1.3 Il design pattern singleton

Ogni volta che si costruisce una classe si crea una nuova istanza di essa, così che nell'esecuzione del programma possano coesistere diverse istanze della stessa classe. Nell'applicazione sviluppata nel tirocinio, si è reso necessario realizzare delle classi che potessero avere un'unica istanza per tutta la durata dell'esecuzione, e che quindi non potessero coesistere due o più istanze di tali classi.

Ad esempio, la classe che si occupa di effettuare l'autenticazione prima di una chiamata al server remoto deve avere un'unica istanza, perché se si creassero due istanze di tale classe verrebbero svolte due autenticazioni al server, mentre l'autenticazione deve essere effettuata una volta sola alla connessione del dispositivo alla rete.

Questo problema è stato risolto adottando il design pattern singleton, appositamente studiato per limitare le istanze di una classe. Il "trucco" è quello di rendere privato il costruttore della classe, e di esporre al pubblico un metodo che crei un'istanza della classe solo se necessario e restituisca la unica istanza. Grazie a questo pattern, tutte le classi possono condividere la medesima istanza della classe singleton.

4.1.4 Il protocollo XML-RPC

Il dispositivo può comunicare con il server dell'azienda tramite il protocollo XML-RPC, uno standard che aiuta la comunicazione tra due software tramite Internet.

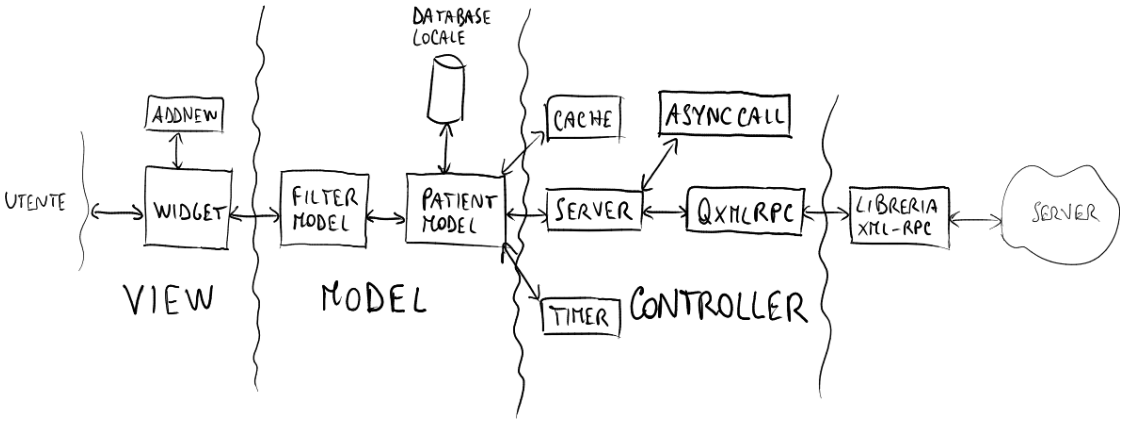
Il protocollo si basa essenzialmente sullo scambio di file codificati nel linguaggio di markup XML: sia le chiamate da parte dell'applicazione, sia le risposte del server sono sempre contenute in file di testo XML, che è facile da elaborare e grazie alla sua struttura ad albero è adattabile a qualsiasi tipo di informazione ed è estremamente flessibile in termini di espansione delle dimensioni del file.

4.2 Panoramica generale

4.2.1 Schema a blocchi

Il progetto si compone di diverse classi C++, ciascuna delle quali svolge un ruolo ben specifico e interagisce con le altre classi tramite il meccanismo di signal e slot.

Il seguente schema a blocchi illustra le classi sviluppate nel codice e le interconnessioni tra di esse:



Da sinistra verso destra si possono riconoscere le tre aree definite dal design pattern model–view–controller:

- l'area della view si compone delle classi che si occupano dell'interfaccia grafica;
- l'area del model si compone delle classi che implementano i modelli e che rappresentano le informazioni;
- l'area del controller si compone delle classi che controllano l'interazione tra la vista e i modelli svolgendo l'attività di backend.

Queste aree sono in stretta cooperazione tra loro e interagiscono con il mondo esterno:

- l'area della view interagisce con l'utente;
- l'area del model svolge operazioni di lettura dal database locale e di scrittura verso di esso;
- l'area del controller fornisce l'accesso al server dell'azienda tramite la libreria XML-RPC.

La trattazione dettagliata delle classi implementate rispetterà di seguito questa suddivisione di base.

4.2.2 View

Le classi appartenenti all'area della view corrispondono alle finestre dell'applicazione:

- la classe denominata `Widget` è la finestra principale, e contiene l'elenco dei pazienti e i pulsanti per usare l'applicazione;
- la classe denominata `AddNew` è la finestra usata per aggiungere un nuovo paziente nell'elenco.

La vista ha lo scopo di simulare le condizioni di lavoro all'interno del dispositivo e di verificare facilmente il corretto funzionamento del codice che "gira" al di sotto di essa, ma non costituisce l'obiettivo principale del tirocinio poiché andrà sostituita con l'interfaccia vera e propria già presente all'interno del dispositivo. Pertanto il lavoro sull'interfaccia ha avuto un ruolo di secondo piano e si è limitato al minimo indispensabile per presentare in maniera visibile il risultato del progetto delle classi più interne, senza "abbellimenti" visivi e senza funzioni che semplificano l'utilizzo dell'applicazione da parte dell'utente.

4.2.3 Model

Le classi appartenenti all'area del model sono responsabili della gestione delle informazioni a cui l'utente deve accedere, in particolare della loro memorizzazione nel database locale, della loro sincronizzazione con il server, e della loro selezione in base alle esigenze dell'utente.

L'idea iniziale era quella di modellizzare le varie entità esposte dal server dell'azienda, ma per mancanza di tempo ci si è limitati ad un modello semplificato dell'entità Patient, tralasciando il lavoro su entità quali Record e Contact. Grazie alla modularità della struttura progettata, in un secondo momento è facilmente possibile aggiungere un qualsiasi numero di modelli a fianco del modello Patient già esistente.

Le classi che compongono l'area del model sono pertanto due:

- la classe `PatientModel` modellizza l'entità Patient del server;
- la classe `FilterModel` si occupa del filtraggio e dell'ordinamento delle informazioni contenute nel modello affinché possano essere visualizzate nella vista;
- la classe `Patient` rappresenta un modo conveniente per memorizzare le informazioni su un certo paziente e può essere usata facilmente nelle connessioni signal e slot tra una classe e l'altra.

È stato scelto di implementare nelle classi del model, oltre alla gestione del database locale, anche la "intelligenza" nell'effettuare le chiamate al server per una questione di comodità: sembrava più difficile demandare a un'altra classe la capacità di decidere quando effettuare le chiamate al server e quali chiamate effettuare e la capacità di elaborare le risposte delle chiamate, data la stretta dipendenza tra il database locale ed il server remoto. Pertanto l'area dedicata al controller si limiterà ad effettuare chiamate

generiche di cui non è consapevole del loro significato né è in grado di elaborarne la risposta.

4.2.4 Controller

Le classi appartenenti all'area del controller forniscono l'interfaccia verso il server remoto e costituiscono il cuore del sistema di backend dell'applicazione. Il backend si contrappone al frontend per il fatto che la sua esecuzione è invisibile all'utente ed è la base su cui poggia l'interfaccia grafica.

L'area del controller è piuttosto popolata in quanto è una componente di basso livello:

- la classe `Cache` entra in gioco quando è necessario effettuare una chiamata al server ma il dispositivo è disconnesso dalla rete;
- la classe `Server` è il punto di accesso al server per tutte le classi che modellizzano le sue entità, ed effettua alcune operazioni correlate al server dell'azienda come l'autenticazione o la verifica degli errori nelle risposte;
- la classe `AsyncCall` fornisce la possibilità di effettuare delle chiamate al server in modo asincrono, ovvero l'esecuzione dell'applicazione non viene interrotta in attesa della risposta del server;
- la classe `QXmlRpc` è quella che "dialoga" direttamente con la libreria esterna XML-RPC, ed è stata progettata in modo tale da essere indipendente dal server specifico dell'azienda ed utilizzabile per qualsiasi server che supporti il protocollo XML-RPC;
- la classe `Timer` aggiunge un ulteriore grado di affidabilità verificando periodicamente la consistenza dei dati del database locale rispetto al server.

Nel tirocinio è stata sicuramente più corposa la scrittura di questa parte: un'interfaccia che si affida ad un backend solido e completo è preferibile in termini di debug rispetto a un'interfaccia solida ma che soffre di problemi legati al backend.

4.3 Descrizione dettagliata del codice

4.3.1 Configurazione del progetto

Il progetto, interamente creato all'interno di Qt Creator, è un progetto di tipo "Qt Gui Application", ossia eredita tutte le funzionalità offerte dal modulo `QtGui/QApplication` delle librerie Qt.

Il progetto è stato configurato in modo da poter essere eseguito in due diversi build environment, "Release" e "Test": il primo esegue il normale `main()` dell'applicazione, il secondo esegue il `main()` che lancia i test tramite Google Test.

Inoltre, tramite un apposito flag nel file di progetto è possibile specificare, in uno qualsiasi dei due build environment, a quale server connettersi all'esecuzione dell'applicazione; in particolare è possibile commutare tra il server normale e il server di test.

4.3.1.1 Libreria XML-RPC

L'applicazione software poggia sulla libreria esterna XML-RPC per la comunicazione con il server.

La libreria XML-RPC non è stata compilata dal codice sorgente, ma è stata installata nell'ambiente GNU/Linux di sviluppo dai repository di Canonical, selezionando il pacchetto `libxmlrpc-c++4`. Poi il riferimento alla libreria è stato aggiunto nel file di progetto tramite delle apposite opzioni di compilazione.

4.3.1.2 Libreria di Google Test

Un'altra libreria esterna utilizzata nel progetto è la libreria sviluppata da Google, il cui nome completo è Google Testing Framework, che fornisce al programmatore la possibilità di svolgere in modo rapido dei test sulla propria applicazione.

Grazie a questa libreria, durante il tirocinio sono stati realizzati dei test per garantire una maggiore affidabilità del codice scritto.

La libreria è stata compilata dal codice sorgente per l'ambiente a 64 bit su cui è stato sviluppato il codice, ed è stata integrata nel progetto tramite delle apposite opzioni di compilazione.

4.3.1.3 File del database locale

Il database locale è memorizzato in un file SQLite molto semplice: è caratterizzato infatti da una sola tabella chiamata *Patients*, che contiene le informazioni sui pazienti:

- *rowid*: ogni riga ha un identificativo univoco;
- *first_name*: il nome del paziente;
- *surname*: il cognome del paziente;
- *health_code*: il codice fiscale del paziente;
- *is_pending*: valore booleano che informa se il paziente è in attesa di essere sincronizzato sul server;

- *last_accessed*: la data di ultimo accesso al paziente, utilizzata dal sistema di gestione delle priorità.

Il database presenta un unico vincolo: se si prova ad aggiungere un paziente con un certo codice fiscale, e nel database esiste già un paziente con quel codice fiscale, allora il nuovo paziente sostituirà il paziente esistente.

4.3.1.4 File di configurazione di Doxygen

Ogni corposo progetto software necessita la redazione di una documentazione, allo scopo di mettere per iscritto determinate scelte implementative prima che siano dimenticate e di essere di aiuto alla comprensione del codice per i programmatori che in un secondo momento si apprestano a continuarne lo sviluppo.

Doxygen è un programma a riga di comando che genera una documentazione in modo automatizzato, permettendo di scrivere la documentazione direttamente nel codice secondo una sintassi apposita. Il risultato è la documentazione tecnica allegata a questa relazione.

La documentazione generata può essere personalizzata in ogni particolare tramite un file di configurazione. Nel caso in oggetto, non è stata configurata alcuna particolare opzione per la documentazione, a parte l'impostazione della proprietà `RECURSIVE` per la documentazione dei file sorgente contenuti nelle sottocartelle del progetto.

4.3.2 Main

Il file *main.cpp* contiene la funzione `main()`, che come in ogni applicazione C++ è il punto di partenza dell'esecuzione.

La funzione `main()` svolge alcune operazioni preliminari: chiama il costruttore della classe Qt `QApplication`, effettua una copia di backup del file del database locale, apre una connessione al database locale, infine mostra a video la finestra principale ed esegue l'applicazione.

4.3.3 View

4.3.3.1 Classe `Widget`

La classe `Widget` deriva dalla classe di Qt `QWidget`, e contiene il codice per il funzionamento della finestra principale dell'applicazione.

Aspetto visivo

La finestra si compone di vari elementi dell'interfaccia. Dall'alto verso il basso troviamo:

- l'oggetto `lineEdit` è una casella di testo (`QLineEdit`) per filtrare l'elenco dei pazienti secondo un pattern specificato dall'utente;
- l'oggetto `tableView` è una tabella (`QTableView`) che elenca il nome, il cognome e il codice fiscale dei pazienti nell'elenco;
- l'oggetto `buttonAddNew` è un pulsante (`QPushButton`) che richiama una finestra per aggiungere un nuovo paziente all'elenco;
- l'oggetto `buttonUpdatePriority` è un pulsante (`QPushButton`) che aggiorna la priorità del paziente selezionato, simulando l'avvio di un elettrocardiogramma (ECG) per il paziente selezionato;
- gli oggetti `buttonConnect` e `buttonDisconnect` sono due pulsanti (`QPushButton`) che simulano rispettivamente la connessione alla rete e la disconnessione da essa, operazioni che nel dispositivo reale verranno effettuate dal Network Manager.

Costruttore

Il costruttore della classe `Widget` prima di tutto inizializza le istanze delle classi singleton `Server`, `Cache` e `Timer` e ne collega i segnali e gli slot, quindi crea un'istanza della classe `FilterModel`, effettua le connessioni signal e slot e la associa come modello all'oggetto `tableView`. Infine sistema la visualizzazione dell'oggetto `ta-`

bleView e carica l'elenco dei pazienti fornendo alla classe FilterModel una stringa vuota (come se l'utente avesse svuotato la casella di testo).

Metodo showEvent()

Il metodo showEvent() viene richiamato dopo il costruttore e subito prima della visualizzazione della finestra. Visualizza una finestra di dialogo di input che chiede all'utente di digitare la dimensione massima, in termini di entry, del database locale, in modo da simulare la dimensione limitata della memoria di archiviazione interna del dispositivo. È ovviamente richiesto che durante l'intera esecuzione dell'applicazione il numero di entry nel database locale non superi il numero digitato, altrimenti sul dispositivo si verificherebbe un esaurimento dello spazio su disco.

Metodo getSelectedPatient()

Il metodo getSelectedPatient() fornisce un modo conveniente per ottenere un puntatore a un oggetto di classe Patient contenente le informazioni sul paziente attualmente selezionato nell'elenco dei pazienti.

Metodo on_buttonConnect_clicked()

Il metodo on_buttonConnect_clicked() è lo slot che viene chiamato quando l'utente fa clic sul pulsante buttonConnect per simulare la connessione del dispositivo alla rete. Notifica all'istanza della classe Server la connessione ed emette il segnale sigConnected().

Metodo on_buttonDisconnect_clicked()

Il metodo on_buttonDisconnect_clicked() è lo slot che viene chiamato quando l'utente fa clic sul pulsante buttonDisconnect per simulare la disconnessione del dispositivo dalla rete. Notifica all'istanza della classe Server la disconnessione.

Metodo on_lineEdit_textChanged()

Il metodo on_lineEdit_textChanged() è lo slot che viene chiamato quando l'utente modifica il testo contenuto nell'oggetto lineEdit per filtrare l'elenco dei pazienti. Emette due segnali apparentemente simili:

- il segnale sigSetFilterPattern() segnala il nuovo pattern alla classe FilterModel per la ricerca nel database locale;
- il segnale sigMatchFilterPattern() segnala il nuovo pattern alla classe PatientModel per la ricerca sul server remoto.

Metodo `on_buttonUpdatePriority_clicked()`

Il metodo `on_buttonUpdatePriority_clicked()` è lo slot che viene chiamato quando l'utente fa clic sul pulsante `buttonUpdatePriority` per simulare l'avvio di un ECG per il paziente correntemente selezionato. Emette il segnale `sigUpdatePatientPriority()`.

Metodo `on_buttonAddNew_clicked()`

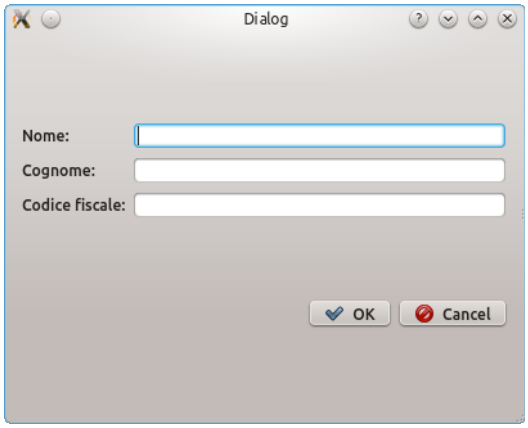
Il metodo `on_buttonAddNew_clicked()` è lo slot che viene chiamato quando l'utente fa clic sul pulsante `buttonAddNew` per aggiungere un nuovo paziente nell'elenco. Crea un'istanza della classe `AddNew` e ne chiama il metodo `show()` per visualizzare la finestra di dialogo all'utente.

4.3.3.2 Classe `AddNew`

La classe `AddNew` contiene il codice per il funzionamento della finestra di dialogo per aggiungere nuovi pazienti all'elenco.

La finestra di dialogo viene richiamata tramite l'apposito pulsante sulla finestra principale dell'applicazione, e deriva dalla classe di Qt `QDialog`, vale a dire può sfruttare tutta la serie di facilitazioni offerta dalle librerie Qt per realizzare una finestra di dialogo con la minima quantità di codice da scrivere.

Aspetto visivo



La finestra di dialogo presenta tre campi di testo in cui inserire le informazioni sul nuovo paziente e i pulsanti standard in basso per confermare o annullare l'operazione.

Metodo `on_buttonBox_accepted()`

Il metodo `on_buttonBox_accepted()` viene chiamato quando l'utente fa clic sul pulsante *OK*. Crea un nuovo puntatore a `Patient` contenente le informazioni che l'utente ha specificato nei campi, quindi emette il segnale `sigAddNewPatient()`.

4.3.4 Model

4.3.4.1 Classe `FilterModel`

La classe `FilterModel` eredita dalla classe Qt `QSortFilterProxyModel` la capacità di ordinamento del modello sorgente, e reimplementa la capacità di filtraggio in base al pattern specificato dall'utente.

Costruttore

Il costruttore della classe `FilterModel` si occupa del modello sorgente: crea una nuova istanza della classe `PatientModel`, ne effettua le connessioni signal e slot, la rende il suo modello sorgente, e imposta l'ordinamento.

Metodo `slotSetFilterPattern()`

Il metodo `slotSetFilterPattern()` è lo slot connesso al segnale `sigSetFilterPattern()` della classe `Widget`, che imposta il nuovo pattern specificato dall'utente e chiama per ogni riga della tabella il metodo `filterAcceptsRow()` invalidando il filtro corrente.

Metodo `filterAcceptsRow()`

Il metodo `filterAcceptsRow()` è una reimplementazione in overloading all'omonimo metodo della classe Qt `QSortFilterProxyModel`, e viene chiamato automaticamente per ogni riga della tabella.

Il suo compito è verificare che la riga corrente della tabella contenga il pattern corrente, e se no restituire `false` per nasconderla dalla vista. La ricerca del pattern è case insensitive (cioè senza distinzione tra maiuscole e minuscole) e viene svolta nei tre campi nome, cognome e codice fiscale.

Metodo `canFetchMore()`

Il metodo `canFetchMore()` viene chiamato dall'oggetto `tableView` ogniqualvolta effettua l'aggiornamento della propria visualizzazione. Se questo metodo restituisce `true`, allora il metodo `fetchMore()` potrebbe venire chiamato successivamente.

Metodo `fetchMore()`

Il metodo `fetchMore()` viene chiamato quando l'oggetto `tableView` ha bisogno di visualizzare degli altri pazienti nell'elenco, perché ci sono delle righe vuote nell'area visibile della tabella oppure l'utente vuole scorrere la lista oltre l'ultima riga corrente.

Emette il segnale `sigFetchMore()` per chiedere alla classe `PatientModel` di recuperare degli altri pazienti dal server a seconda delle necessità dell'oggetto `tableView`. Il recupero dei pazienti dal server avviene in modo asincrono, evitando così il blocco dell'interfaccia nell'attesa della risposta.

Questo metodo viene chiamato solo quando effettivamente l'utente vuole vedere altri pazienti oltre a quelli che sono già presenti nel database locale. Ciò permette di riempire in modo incrementale la lista e di dare la sensazione all'utente di avere tutti i pazienti nella lista quando in realtà solo una parte di essi è presente.

4.3.4.2 Classe `PatientModel`

La classe `PatientModel` eredita dalla classe Qt `QSqlTableModel` la capacità di aggiungere, rimuovere e modificare le entry del database locale.

Costruttore

Il costruttore della classe `PatientModel` effettua le connessioni signal e slot con le classi singleton `Timer` e `Cache`, quindi importa la tabella `Patients` effettuando un'operazione di select, infine imposta i titoli delle colonne della tabella.

Metodo `selectDb()`

Il metodo `selectDb()` effettua l'operazione di select sul database locale, cioè ne estrae le entry e le rende disponibili alla classe `FilterModel`. L'operazione di selezione viene effettuata sull'intero database locale.

Metodo `fieldIndex()`

Il metodo `fieldIndex()` fornisce un modo conveniente per accedere ai campi del database locale. Se si decide di modificare il nome di un campo, basta cambiare questa funzione senza dover riconsiderare tutto il codice.

Metodo `getRowIndex()`

Il metodo `getRowIndex()` recupera l'indice della riga corrispondente al paziente avente il codice fiscale specificato. La ricerca viene effettuata con distinzione di maiuscole e minuscole.

Metodo `getLastAccessedField()`

Il metodo `getLastAccessedField()` recupera la data di ultimo accesso di una particolare entry del database locale, che costituisce la priorità del paziente.

Metodo `checkExistence()`

Il metodo `checkExistence()` verifica se esiste un paziente avente il codice fiscale specificato.

Metodo `matchesFilterPattern()`

Il metodo `matchesFilterPattern()` verifica se almeno uno degli attributi del paziente specificato corrisponde al pattern del filtro corrente, e quindi se un certo paziente verrà mostrato nella vista o verrà nascosto a causa del filtro. La corrispondenza non distingue le maiuscole dalle minuscole.

Metodo `prepareAddPatient()`

Il metodo `prepareAddPatient()` svolge alcune operazioni preliminari all'aggiunta di un paziente al database, tra cui l'eliminazione del paziente a priorità più bassa (cioè il paziente che non è stato acceduto da più tempo) nel caso in cui il database locale sia pieno. Questo metodo viene chiamato generalmente prima dell'aggiunta di un nuovo paziente al database locale.

Metodo `parseFetchingResults()`

Il metodo `parseFetchingResults()` si occupa del parsing, ovvero dell'analisi, dei pazienti ricevuti nella risposta del server: verifica la presenza di tutti gli attributi (nome, cognome e codice fiscale), aggiunge i pazienti non ancora presente nel database locale, e se necessario avvia un'altra chiamata asincrona per chiedere altri pazienti.

Metodo `setIsPending()`

Il metodo `setIsPending()` permette di impostare un valore booleano nell'attributo `isPending` di una certa entry nel database locale, per contrassegnare quel paziente come in attesa di sincronizzazione o sincronizzato.

Metodo `deletePatient()`

Il metodo `deletePatient()` elimina il paziente specificato dal database locale. Questo metodo può venire chiamato per eliminare un paziente a priorità più bassa.

Metodo `updatePatient()`

Il metodo `updatePatient()` permette di aggiornare le informazioni su un certo paziente per assicurare la consistenza dei dati tra il database locale e il server remoto: il database locale può infatti contenere delle informazioni più vecchie di quelle che ci sono sul server perché l'utente ha la possibilità di modificare i pazienti sul server.

Se a causa dell'aggiornamento il paziente scompare dalla vista, effettua automaticamente una richiesta al server affinché la riga rimasta vuota venga occupata da un altro paziente.

Metodo `fetchPatients()`

Il metodo `fetchPatients()` serve per effettuare il fetch di alcuni pazienti dal server, effettuando una chiamata al server e chiedendo di ricevere un numero specificato di pazienti che corrispondono al pattern del filtro corrente.

Se alcuni dei pazienti che verranno ricevuti in risposta sono già presenti nel database locale, occorrerà eseguire un altro fetch fino a quando il numero di pazienti desiderato non sarà raggiunto (o fino a quando non saranno finiti i pazienti sul server).

Metodo `getFilterModelRowCount()`

Il metodo `getFilterModelRowCount()` contiene un algoritmo che calcola il numero di righe occupate nella vista secondo il pattern del filtro corrente. Serve alla classe `PatientModel` per sapere di quanti pazienti ha bisogno la classe `FilterModel` per riempire tutte le righe visibili nella vista.

Metodo `slotMatchFilterPattern()`

Il metodo `slotMatchFilterPattern()` è lo slot che, se è necessario, esegue il fetch dei pazienti in base al nuovo pattern del filtro specificato dall'utente.

Se la chiamata asincrona ha successo viene chiamato lo slot `slotMatchFilterPattern_Person_quickSearchSuccessful()` che effettua il parsing della risposta, altrimenti viene chiamato lo slot `slotMatchFilterPattern_Person_quickSearchFailed()`.

Metodo `slotConnected()`

Il metodo `slotConnected()` è lo slot che viene chiamato alla connessione del dispositivo alla rete, e provvede a recuperare, se necessario, dei pazienti dal server.

Se la chiamata asincrona ha successo viene chiamato lo slot `slotConnected_Person_quickSearchSuccessful()` che effettua il parsing della risposta, altrimenti viene chiamato lo slot `slotConnected_Person_quickSearchFailed()`.

Metodo `slotFetchMore()`

Il metodo `slotFetchMore()` è lo slot che viene chiamato quando la vista ha bisogno di altri pazienti dal server per riempire le righe vuote della lista.

Se la chiamata asincrona ha successo viene chiamato lo slot `slotFetchMore_Person_quickSearchSuccessful()` che effettua il parsing della risposta,

altrimenti viene chiamato lo slot `slotFetchMore_Person_quickSearchFailed()`.

Metodo `slotUpdatePatientPriority()`

Il metodo `slotUpdatePatientPriority()` è lo slot che serve per aggiornare la priorità di un certo paziente, sovrascrivendo la data e l'ora di ultimo accesso con la data e l'ora correnti. Viene chiamato quando si vuole simulare l'avvio di un ECG per quel paziente, e non effettua alcuna chiamata asincrona al server.

Metodo `slotAddNewPatient()`

Il metodo `slotAddNewPatient()` è lo slot che serve per aggiungere un nuovo paziente all'elenco. Il metodo aggiunge il paziente al database locale, quindi effettua una chiamata al server con lo scopo di applicare la modifica anche sul server remoto.

Se il dispositivo non è connesso alla rete, contrassegna il paziente con il flag `isPending` in modo che alla successiva connessione del dispositivo possa essere sincronizzato.

Se la chiamata asincrona ha successo viene chiamato lo slot `slotAddNewPatient_Person_createSuccessful()`, altrimenti viene chiamato lo slot `slotAddNewPatient_Person_createFailed()`.

Metodo `slotSyncPatients()`

Il metodo `slotSyncPatients()` è lo slot che viene chiamato quando il dispositivo è stato connesso alla rete e ci sono dei pazienti in attesa di essere sincronizzati. Questo metodo scorre tutto il database locale alla ricerca di pazienti da sincronizzare, quindi per ognuno di essi effettua una chiamata di creazione sul server.

Se la chiamata asincrona ha successo viene chiamato lo slot `slotSyncPatients_Person_createSuccessful()`, altrimenti viene chiamato lo slot `slotSyncPatients_Person_createFailed()`.

Metodo `slotCheckConsistence()`

Il metodo `slotCheckConsistence()` è lo slot che viene chiamato dalla classe `Timer` per controllare la consistenza dei dati. Il metodo verifica, un paziente per volta, se le informazioni contenute nel database locale corrispondono a quelle memorizzate sul server, e se necessario le aggiorna.

Se il paziente non esiste sul server, provvede ad eliminarlo dal database locale. Oltre a ciò, verifica se la rimozione di tale paziente comporta lo svuotamento di una riga

nella vista, e in tal caso provvede anche ad effettuare il fetch di un altro paziente per riempire quella riga.

Se la chiamata asincrona ha successo viene chiamato lo slot `slotCheckConsistence_Person_advancedSearchSuccessful()` che effettua l'eventuale aggiornamento, altrimenti viene chiamato lo slot `slotCheckConsistence_Person_advancedSearchFailed()`.

Metodi per le chiamate generiche al server

Infine ci sono alcuni metodi molto simili tra loro, ciascuno dei quali corrisponde ad uno specifico metodo del server dell'azienda: codificano i parametri in modo da rispettare i corretti nomi di parametro attesi dal server, quindi chiamano il metodo `asyncCall()` della classe `Server` per avviare la chiamata asincrona.

4.3.5 Controller

4.3.5.1 Classe `Cache`

La classe `Cache` eredita semplicemente da `QObject`, poiché necessita solo delle funzioni di base offerte da Qt (primariamente la capacità di inviare e ricevere segnali per comunicare con le altre classi).

La classe `Cache` è una classe di tipo singleton, vale a dire esiste una unica istanza di quella classe per l'intero programma.

Metodo `slotLocalDbChanged()`

Il metodo `slotLocalDbChanged()` è lo slot che viene chiamato dalla classe `PatientModel` per notificare la modifica di una entry nel database locale.

La notifica viene inviata solo quando il dispositivo non è connesso alla rete, e quindi la modifica non può essere apportata nell'immediato al server. La classe `Cache` tramite un apposito flag alla successiva connessione effettuerà la sincronizzazione delle modifiche in sospeso al server.

Metodo `slotConnected()`

Il metodo `slotConnected()` è lo slot che viene chiamato per notificare la connessione del dispositivo. Se ci sono delle modifiche in sospeso, avvia la loro sincronizzazione al server.

4.3.5.2 Classe `Server`

La classe `Server` eredita semplicemente da `QObject`, poiché necessita solo delle funzioni di base offerte da Qt (primariamente la capacità di inviare e ricevere segnali per comunicare con le altre classi).

Lo scopo della classe è quello di poter effettuare chiamate asincrone al server dell'azienda. Questo vuol dire che è compatibile solo con il server dell'azienda, in quanto effettua delle operazioni specifiche che sono legate al server stesso, quali l'autenticazione per il recupero dell'ID di sessione e la verifica di eventuali errori nella risposta.

Un attributo della classe molto importante è la variabile `m_threadQueue` di tipo `QThreadPool`, perché grazie ad essa è possibile effettuare delle chiamate al server in modo asincrono: è infatti in grado di gestire un insieme ("pool" in inglese) di thread, ovvero di porzioni del codice, appartenenti al programma stesso, che vengono eseguite in concorrenza (cioè contemporaneamente) all'esecuzione del thread principale. Il vantaggio è che possono coesistere e procedere nello stesso momento due parti diverse del medesimo programma, evitando che l'esecuzione di una parte interrompi quella dell'altra; nel caso dell'applicazione creata nel tirocinio, la parte del programma che deve essere eseguita in modo asincrono è la "attesa" della risposta da parte del server, che non deve bloccare l'interfaccia utente. Il pregio del pool di thread è che, impostando il massimo numero di thread a 1, se una chiamata asincrona giunge quando un'altra è ancora in esecuzione, allora quella chiamata attenderà (sempre in modo asincrono) il completamento della chiamata in esecuzione prima di essere inoltrata al server.

La classe `Server` è una classe di tipo singleton, vale a dire esiste una unica istanza di quella classe per l'intero programma.

Costruttore

Il costruttore della classe `Server` inizializza il pool di thread, settando il massimo numero di thread concorrenti ad 1, quindi crea una nuova istanza della classe `QXmlRpc` connettendone i segnali e gli slot.

Metodo `convertQMap()`

Il metodo `convertQMap()` viene usato internamente per convertire una mappa di parametri da un tipo generico `QMap<QString, P>` nel tipo standard `QMap<QString, QVariant>`.

Metodo `getFromMap()`

Il metodo `getFromMap()` recupera un elemento da una mappa che contiene la risposta della chiamata al server. Possiede diverse implementazioni in overloading, per coprire tutti i possibili tipi di dato.

Metodo `call_private_success()`

Il metodo `call_private_success()` viene chiamato quando la chiamata al server non presenta errori ed è stata completata con successo. Il metodo recupera i risultati della chiamata dalla mappa completa della risposta, accedendo all'elemento *results*.

Metodo `call_private_fail()`

Il metodo `call_private_fail()` viene chiamato quando la chiamata al server presenta errori quali WARNING, EXCEPTION o ERROR. Il metodo stampa le informazioni sull'errore, e nel caso di soli WARNING procede a recuperare il risultato della chiamata come se essa fosse stata completata senza errori.

Metodo `call_private_parseReply()`

Il metodo `call_private_parseReply()` determina se la chiamata è stata completata con successo o con errori: nel primo caso chiama il metodo `call_private_success()`, nel secondo caso chiama il metodo `call_private_fail()`.

Metodo `call_private()`

Il metodo `call_private()` chiama il metodo `call()` della classe `QXmlRpc`, quindi ne chiama il parsing della risposta.

Metodo `authenticate()`

Il metodo `authenticate()` effettua se necessario l'autenticazione sul server dell'azienda recuperando un ID di sessione. Questo metodo deve essere invocato prima di effettuare una qualsiasi chiamata al server, perché una chiamata al server priva di autenticazione non avrà successo.

Metodo `call()`

Il metodo `call()` permette di effettuare una chiamata al server in modo sincrono, a differenza del metodo `asyncCall()`.

Metodo `asyncCall()`

Il metodo `asyncCall()` permette di effettuare una chiamata al server in modo asincrono. Aggiunge al pool di thread un nuovo thread di classe `AsyncCall()`, che di fatto chiamerà il metodo `call()` della classe `Server` una volta che verrà eseguito.

Metodo `connect()`

Il metodo `connect()` lancia il segnale `sigConnected()` per notificare la connessione del dispositivo alla rete.

Metodo `isConnected()`

Il metodo `isConnected()` restituisce un valore booleano che indica se il dispositivo è attualmente connesso alla rete.

Metodo `isAuthenticated()`

Il metodo `isAuthenticated()` restituisce un valore booleano che indica se il dispositivo è attualmente autenticato sul server remoto. L'autenticazione viene effettuata automaticamente subito prima della prima chiamata al server.

Metodo `putIntoMap()`

Il metodo `putIntoMap()` inserisce un elemento nella mappa dei parametri da passare alla chiamata al server. Possiede diverse implementazioni in overloading, per coprire tutti i possibili tipi di dato.

4.3.5.3 Classe `AsyncCall`

La classe `AsyncCall` deriva da `QObject` e da `QRunnable`, in modo che possa essere eseguita come in un apposito thread nel pool della classe `Server`.

Il suo compito è quello di memorizzare alla sua costruzione le informazioni sulla chiamata, e di avviarla quando il pool le concede il permesso di esecuzione.

Costruttore

Il costruttore della classe `AsyncCall` memorizza in variabili interne alla classe le informazioni sulla chiamata da effettuare, e connette agli slot della classe `Server` i segnali che saranno lanciati al termine della chiamata.

Metodo `run()`

Il metodo `run()` chiama il metodo `call()` della classe `Server`, quindi attende la risposta. Siccome è in esecuzione in un thread separato, l'attesa per la risposta non blocca l'interfaccia utente. Quando arriva la risposta, emette il segnale contenente la risposta della chiamata e interrompe la sua esecuzione, lasciando il posto per un'altra eventuale chiamata asincrona in attesa.

4.3.5.4 Classe `QXmlRpc`

La classe `QXmlRpc` eredita semplicemente da `QObject`, poiché necessita solo delle funzioni di base offerte da Qt (primariamente la capacità di inviare e ricevere segnali per comunicare con le altre classi).

I compiti principali della classe `QXmlRpc` sono la conversione della mappa dei parametri dai tipi Qt ai tipi della libreria XML-RPC, la chiamata del metodo `call()` della libreria XML-RPC, e infine la conversione della mappa dei risultati della chiamata dai tipi della libreria XML-RPC ai tipi Qt.

La classe `QXmlRpc` non è consapevole se sta operando in modo sincrono o asincrono, cioè se è in esecuzione nel thread principale o in un thread secondario, quindi blocca in ogni caso il thread su cui sta girando in attesa della risposta.

Metodi `serverUrl()` e `setServerUrl()`

Siccome la classe `QXmlRpc` è stata progettata per funzionare con ogni server che supporti il protocollo XML-RPC e non solo con il server aziendale, l'URL del server deve essere fornito dalla classe `Server`, che invece è specifica del server, tramite il metodo `setServerUrl()`. Il metodo `serverUrl()` effettua l'operazione opposta, restituendo l'URL del server correntemente impostato.

Metodo `convertParams()`

Il metodo `convertParams()` avvia la conversione della mappa dei parametri dai tipi Qt ai tipi della libreria XML-RPC.

Metodo `call()`

Il metodo `call()` chiama il metodo `convertParams()` per la mappa dei parametri, quindi chiama il metodo interno `call_private()` per la chiamata vera e propria.

Metodo `call_private()`

Il metodo `call_private()` effettua la chiamata vera e propria alla libreria XML-RPC (in particolare viene chiamato il metodo `call()` della classe `clientSimple`), rileva eventuali errori sollevati dalla libreria XML-RPC, e infine avvia la conversione della mappa della risposta dai tipi della libreria XML-RPC ai tipi Qt.

Metodo `getXmlRpcValue()`

Il metodo `getXmlRpcValue()` converte il valore di un parametro della mappa dal tipo Qt al tipo della libreria XML-RPC. Nella maggior parte è richiesto un passaggio intermedio dai tipi Qt ai tipi standard del C++.

Operazioni analoghe ma specifiche per un solo tipo di dato vengono effettuate dai metodi seguenti: `getXmlRpcValue_ByteArray()` si occupa della conversione di vettori di byte, `getXmlRpcValue_List()` si occupa della conversione di liste, e `getXmlRpcValue_Map()` si occupa di conversione di mappe.

Metodo `getQtValue()`

Il metodo `getQtValue()` converte il valore di un parametro della mappa dal tipo della libreria XML-RPC al tipo Qt. Nella maggior parte è richiesto un passaggio intermedio dai tipi della libreria XML-RPC ai tipi standard del C++.

Operazioni analoghe ma specifiche per un solo tipo di dato vengono effettuate dai metodi seguenti: `getQtValue_DateTime()` si occupa della conversione di coppie data e ora, `getQtValue_ByteArray()` si occupa della conversione di vettori di byte, `getQtValue_List()` si occupa della conversione di liste, e `getQtValue_Map()` si occupa di conversione di mappe.

4.3.5.5 Classe `Timer`

La classe `Timer` eredita semplicemente da `QObject`, poiché necessita solo delle funzioni di base offerte da Qt (primariamente la capacità di inviare e ricevere segnali per comunicare con le altre classi).

Lo scopo della classe `Timer` è la verifica periodica della consistenza dei dati nel database locale rispetto al server remoto. In particolare, a intervalli di 5 minuti notifica alla classe `PatientModel` di prendere un paziente dal database locale, scaricarne le informazioni dal server, e controllare che esse siano coerenti con quanto è memorizzato nel database locale. Siccome l'utente può modificare i pazienti sul server, questa funzione è stata studiata per evitare che informazioni troppo vecchie e non più coerenti continuino a risiedere nel dispositivo troppo a lungo. Viene verificato solo un paziente per volta e ad un ritmo così basso per evitare di occupare troppo la banda della connessione.

La classe `Timer` è una classe di tipo singleton, vale a dire esiste una unica istanza di quella classe per l'intero programma.

Metodo `timerEvent()`

Il metodo `timerEvent()` viene chiamato ogniqualvolta il timer, definito internamente alla classe, scatena un evento corrispondente allo "scadere del tempo". Questo metodo lancia il segnale `sigCheckConsistence()` che sarà ricevuto dalla classe `PatientModel`.

Metodo `slotConnected()`

Il metodo `slotConnected()` avvia il timer se il dispositivo è stato connesso alla rete, o lo interrompe se il dispositivo è stato disconnesso dalla rete.

Metodi `startTimer()` e `stopTimer()`

I metodi `startTimer()` e `stopTimer()` permettono rispettivamente di avviare e di interrompere il timer.

5 Conclusioni

Credo che il tirocinio sia un'esperienza incredibilmente formativa: esso infatti catapulta lo studente direttamente nel mondo del lavoro, dove può misurare le proprie capacità e trovare il giusto orientamento per sé una volta uscito dall'università.

Il tirocinio è stato molto interessante perché mi ha dato la possibilità di imparare un linguaggio di programmazione ad oggetti, il C++, che è molto utilizzato nel mondo del lavoro.

L'applicazione sviluppata durante l'attività di tirocinio è funzionante e ha centrato in larga parte gli obiettivi prefissati, e le soluzioni implementative adottate hanno risposto in maniera soddisfacente ai requisiti di progetto, seppur con qualche sacrificio a cui si è dovuti cedere per mancanza di tempo: per esempio si sarebbero potute realizzare altre classi oltre alla classe `PatientModel` per estendere il raggio d'azione alle altre entità che il server dell'azienda espone, o si sarebbe potuto lavorare sull'implementazione di un'interfaccia touch progettata appositamente per l'input dallo schermo sensibile al tocco del dispositivo invece di limitarsi a realizzare un'applicazione per mouse e tastiera.

Spero che almeno una parte del mio codice verrà integrata, ovviamente riadattata opportunamente, nel dispositivo reale che verrà commercializzato, perché mi renderebbe felice aver dato il mio piccolo contributo ad un progetto così vasto.

6 Sitografia

- www.biotechware.com
- qt.digia.com
- xmlrpc-c.sourceforge.net
- code.google.com/p/googletest
- it.wikipedia.org