UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Dipartimento di Scienze Matematiche, Fisiche e Informatiche
Corso di Laurea in Informatica

Progetto e Sviluppo di un applicativo mobile di supporto all'infettivologo per le consulenze e il monitoraggio di pazienti

Candidato:
Giorgia Lombardi

Relatore:
Prof. Riccardo Martoglia

Correlatrice:
Prof. Federica Mandreoli

Anno Accademico 2018/2019

PAROLE CHIAVE
Android Studio
Cartella Clinica
Applicazione Mobile
SQLite

Indice

Parte II
Implementazione

Capitolo III - "Progettazione del software"

Capitolo IV - "Implementazione"

Capitolo V - “Conclusioni e Sviluppi futuri”

Indice Figure

Introduzione

L'enorme processo evolutivo che ha caratterizzato l'informatica negli ultimi anni ha inevitabilmente coinvolto nella sua crescita svariate aree ancora non esplorate dal progresso tecnologico, tra di esse : la medicina, grande produttrice di informazioni e in continuo bisogno di analisi e test da attuare su di essi.

E' da questa semplice ma forte unione che nasce il progetto sviluppato durante il tirocinio che ho svolto all'interno all'Università, avente l'obiettivo di apportare migliorie ed agevolazioni alla professione di medico.

Il progetto vede la sua nascita con la collaborazione del mio relatore e della mia correlatrice, rispettivamente il professore Riccardo Martoglia e la professoressa Federica Mandreoli, con alcuni medici del Policlinico di Modena.

Durante le ore di tirocinio ho avuto l'onore di collaborare con la dottoressa Erica Franceschini e il medico Giovanni Guaraldi, i quali hanno stabilito un punto di partenza illustrando le necessità tecnologiche che migliorerebbero determinati processi svolti da loro quotidianamente.

Il progetto consiste in una Cartella Clinica Elettronica, più specificatamente un'applicazione per tablet che possa supportare gli infettivologi all'interno del Policlinico nelle operazioni di monitoraggio e analisi delle condizioni dei pazienti.

Per lo sviluppo dell'applicazione è stato utilizzato Android Studio come IDE e SQLite come gestore dei dati utilizzati all'interno del software.

La tesi è strutturata in 5 capitoli, il cui contenuto accompagnerà passo per passo il

lettore durante le fasi di studio delle tecnologie, l'analisi dei prerequisiti, la progettazione e l'implementazione dell'applicazione.
Il primo capitolo introdurrà l'esperienza di collaborazione con i medici del Policlinico di Modena, raccontando gli incontri svolti e le idee emerse durante questi ultimi, le quali hanno determinato le linee base e la struttura dell'applicazione.
Nel secondo capitolo verrà fatta una panoramica di quelle che sono state le tecnologie che hanno permesso di realizzare CartellaClinica e i motivi per cui sono state scelte.
In seguito si percorreranno, rispettivamente nei capitoli 3 e 4, le fasi di progettazione e di implementazione dell'applicazione, caratterizzate dall'analisi dei casi d'uso e delle funzionalità di base dell'applicazione e la vera e propria realizzazione del prodotto, andando a studiare gli elementi che la compongono.
Dopodiché, nel capitolo 5, si concluderà il percorso con una veloce riflessione sul lavoro svolto e sull'utilità dell'applicazione creata, passando poi, infine, alle migliorie e gli sviluppi futuri che potranno essere realizzati.

Parte I

Il Caso di Studio

Capitolo 1
Analisi degli Obiettivi

1.1 Collaborazione con il Policlinico di Modena

La realizzazione dell'applicazione CartellaClinica nasce dalla collaborazione tra il professore Riccardo Martoglia, il mio relatore, la professoressa Federica Mandreoli, la mia correlatrice, e il professore Giovanni Guaraldi.
Giovanni Guaraldi è professore associato presso il dipartimento Chirurgico, Medico, Odontoatrico e di Scienze Morfologiche; è, inoltre, a capo della clinica metabolica di Modena (centro inter-specialistico deputato alla diagnosi e al trattamento della lipodistrofia e delle patologie HIV correlate e della fragilità in HIV).
L'interesse di Giovanni Guaraldi per questo progetto è scaturito da migliorie, realizzate dall'ingegnere Andrea Malagoli, apportate a determinate tecnologie usate frequentemente in Policlinico, volte ad ottimizzare operazioni di grande importanza nell'ambito medico. Alla luce dei grossi vantaggi portati da esse è nata l'idea e la convinzione che una forte unione tra tecnologia e la professione di medico possa portare grandissimi vantaggi nel mondo ospedaliero.
L'informatica dà infatti la possibilità di realizzare applicazioni che possano velocizzare e

facilitare azioni svolte quotidianamente all'interno dell'ospedale, realtà che è stata confermata dai progetti, precedentemente citati, di Andrea Malagoli, data scientist che collabora con il Policlinico e con Giovanni Guaraldi.
Durante il periodo di tirocinio sono stati organizzati diversi incontri, durante i quali ho avuto il piacere di conoscere Erica Franceschini, dottoressa epidemiologa che lavora al Policlinico di Modena. La partecipazione di Erica è stata essenziale per lo sviluppo del software, poiché mi ha illustrato in maniera efficiente, introducendomi nel suo ambito di lavoro, le possibili migliorie a cui avrei potuto contribuire.

1.2 Possibili progetti emersi durante gli incontri

I progetti emersi durante gli incontri con Erica Franceschini sono essenzialmente tre, la quale idea generale verrà spiegata qui sotto :

- Realizzare un'applicazione che permette di avere una **cartella clinica** dei pazienti curati dal medico interessato. Il medico può accedere al proprio account personale e visionare tutti i reparti in cui si trovano i pazienti da lui seguiti. E' possibile selezionare il paziente interessato e visionare, eliminare o aggiungere dati anagrafici, i parametri vitali, le allergie, le consulenze, gli esami bioumorali, la sua microbiologia e le terapie antibiotiche.
- Come seconda opzione è emersa l'idea di un'applicazione focalizzata sulle **infezioni**, che ci permetta di avere un'analisi chiara e istantanea della situazione giornaliera delle infezioni che possono essersi "propagate" all'interno del reparto. In questo caso lo studio sarebbe rivolto ai pazienti ricoverati nell'edificio "malattie infettive", poiché è possibile che, per piccole sviste da parte dei medici, alcune infezioni si propaghino all'interno dell'edificio.

- Come ultima possibilità Erica mi ha parlato di un progetto che si concentri sul **controllo dei farmaci da somministrare ai pazienti**. Durante l'incontro abbiamo osservato alcuni documenti, relativi a pazienti reali, nascondendo i dati personali di quest'ultimi per motivi di privacy. I documenti appena citati contengono gli antibiotici a cui il paziente è resistente e non. La logica del software sarebbe quella di ottimizzare il controllo sui farmaci che possono essere somministrati o meno al paziente interessato. Ovvero si attuerebbe un "filtraggio" di informazioni che possa velocizzare il controllo fatto dal medico, dal momento in cui dovrebbe passare da documento in documento per leggere informazioni scritte in tabelle poco leggibili.

I progetti appena elencati sono stati pensati per essere sviluppati all'interno di una macro applicazione. Quest'ultima metterebbe a disposizione un menu principale dal quale il medico possa scegliere a quale delle tre aree accedere e quindi quali dati analizzare.
Date le diverse opzioni di progettazione ho avuto la possibilità di scegliere quale dei tre software sviluppare per primo nel mio periodo di tirocinio.

1.3 Decisione del software da sviluppare

Ho deciso di sviluppare la Cartella Clinica Elettronica perché mi avrebbe permesso di lavorare sullo studio della realizzazione di un'applicazione più user-friendly possibile. Questo implica un consistente studio riguardante la gestione dei Layout e di componenti messi a disposizione da Android Studio. L'obiettivo dell'applicazione è proprio quello di rendere la lettura di grandi quantità di dati prolissi più rapida e scorrevole, comprendendo anche l'integrazione di grafici che ne facilitino l'interpretazione . Lo sviluppo di un'applicazione di questo tipo mi ha permesso inoltre di approfondire le conoscenze riguardo l'interazione del software con il database sottostante e la gestione logica di grandi quantità di dati.

1.4 Valutazione delle tecnologie da usare

Durante gli incontri svolti ho avuto il piacere di conoscere l'ingegnere, precedentemente citato, Andrea Malagoli, il quale mi ha fornito importanti informazioni riguardo le tecnologie utilizzate all'interno del Policlinico. Questo passaggio è stato essenziale per la realizzazione di un'applicazione che rispettasse gli strumenti usati quotidianamente dai medici. Tuttavia, l'analisi delle tecnologie usate verrà trattata in seguito nel Capitolo 2.

1.5 Presentazione dell'applicazione

Il software creato è nato per agevolare l'azione di monitoraggio dei pazienti da parte dei medici. Esso mette a disposizione molteplici azioni che permettono all'utente di visualizzare, aggiungere e modificare facilmente quelli che sono i dati dei pazienti da lui curati.
Uno dei principali obiettivi è stato quello di rendere l'applicazione più intuitiva e user-friendly possibile, realizzando pagine pulite, semplici, facili da leggere e da utilizzare. Questo perché l'obiettivo dell'applicazione è quello di facilitare l'analisi di questi dati ai medici, unendo tutte le informazioni più significative all'interno di un'unica app. Sono stati inseriti inoltre alcuni grafici, il quale scopo è quello di mostrare l'andamento nel tempo di specifici parametri, particolarmente rilevanti, del paziente in cura in modo da avere un'idea visiva e diretta della situazione.

Capitolo 2
Tecnologie utilizzate

2.1 Android Studio e Sistema Operativo Android

In questo capitolo si andranno a ripercorrere le fasi preliminari di studio e analisi delle tecnologie da utilizzare che hanno preceduto il momento di progettazione del software. Sono state fatte prove e valutazioni prima di individuare i mezzi ottimali per realizzare l'applicazione.

2.2 Introduzione Android Studio

Per la creazione dell'applicazione è stato utilizzato un ambiente di sviluppo integrato

per la piattaforma Android. La scelta della tecnologia deriva e dipende direttamente dal sistema operativo dei dispositivi elettronici utilizzati all'interno del Policlinico di Modena, dovendo sviluppare un'applicazione compatibile con questi ultimi.
Dopo un'analisi preliminare dei possibili ambienti di sviluppo è stato scelto Android Studio, attualmente l'IDE primario di Google per lo sviluppo nativo di applicazioni Android.
I motivi per cui è stato utilizzato Android Studio sono diversi: esso mette a disposizione una grande quantità di materiale e documentazione online, offre molti strumenti del tutto gratuiti ed è ottimo e semplice da utilizzare per lo sviluppo dell'interfaccia grafica dell'applicazione.

2.2.1 Storia e Sviluppo di Android Studio

Android Studio è basato su IntelliJ IDEA, un IDE intuitivo prodotto dalla società JetBrains, è inoltre un software molto portabile, è infatti compatibile con Windows, Max OS X e Linux. La forte specializzazione del software nello sviluppo mirato di applicazioni Android e la sua semplicità hanno fatto si che sostituisse gli Android Development Tools (ADT) di Eclipse, ambiente su cui si è formata la prima generazione di sviluppatori Android. E' importante, inoltre, mettere alla luce il fatto che il progetto di Android Studio è tuttora in continuo ampliamento, caratteristica che lo rende un software fortemente aggiornato e al passo con lo sviluppo delle tecnologie Android.

2.2.2 Recenti migliorie apportate al software

Una recente miglioria apportata nell'ultima major release di Android Studio, la quale ha avuto un grande ruolo nella decisione di utilizzare questo IDE, riguarda l'incremento della produttività e alla riduzione dei tempi di sviluppo grazie alla nascita dell'**Instant Run**. L'**Instant Run** permette di svolgere con grande rapidità le modifiche apportate al codice senza dover aspettare lenti rebuild del progetto. Si può infatti riscontrare un grande ostacolo nell'utilizzo degli emulatori di dispositivi elettronici, i quali, a volte,

sono più prestanti delle macchine utilizzate per sviluppare, rendendo l'esecuzione del codice quasi impossibile. Per sormontare questo ostacolo Android Studio ha potenziato i suoi emulatori, rendendo più efficiente e scorrevole il testing del codice e la sua esecuzione.

2.3 Android SDK

Il **Software Development Kit** (SDK) è un insieme di strumenti messi a disposizione per lo sviluppo di applicazioni Android. Grazie a questo pacchetto qualsiasi utente possiede l'accesso al codice sorgente di tutte le versioni di Android e ha la possibilità di modificarlo autonomamente.
L'SDK è composto da molti strumenti, quali programmi, piattaforme per diverse versioni di Android, emulatori, tutorial e debugger. La composizione di questi strumenti viene gestita tramite l'Android SDK Manager, un programma avviabile sia da Eclipse che da Android Studio. Il Manager dà la possibilità di usufruire degli elementi all'interno del pacchetto dell'SDK nella maniera più consona per lo sviluppo del progetto.

2.4 Architettura Android

Nella fase preparatoria alla creazione dell'applicazione è stato effettuato uno studio di quelle che sono le componenti principali dell'architettura della piattaforma Android. L'approfondimento è stato mosso dall'interesse nel capire i meccanismi che stanno alla base della grande flessibilità e portabilità che hanno portato il Sistema Operativo Android a diventare uno dei più diffusi al mondo.

Android è un sistema operativo sviluppato da Google Inc, e basato sul kernel Linux,

tuttavia non è da considerarsi interamente né un sistema unix-like né una distribuzione GNU/Linux, bensì una distribuzione embedded Linux, dal momento in cui la maggior parte delle utilità GNU è sostituita da software in Java. E' stato progettato principalmente per smartphone e tablet, per poi arrivare a interfacce utente specializzate per televisori (Android TV), automobili (Android Auto), orologi da polso (Wear OS) e occhiali (Google Glass).
Android è composto da sei componenti principali : *un sistema operativo*, *un ambiente di esecuzione* (run-time), l'*Hardware Abstraction Layer*, *librerie*, *Application Framework* e *applicazioni*.

Vengono mostrati nella seguente figura gli elementi appena citati :

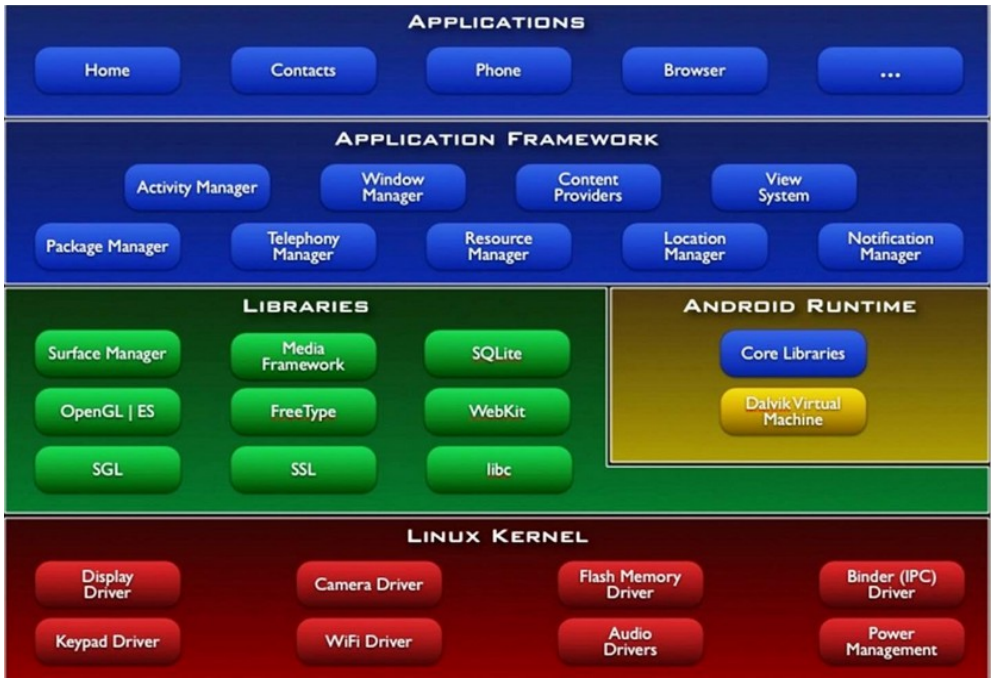

Figura 2.1: Schema rappresentante l'architettura della piattaforma Android

- **Linux Kernel :** a livello più basso troviamo il kernel Linux. Il Kernel fornisce un livello di astrazione tra l'hardware e i livelli superiori di architettura. E' qui che si trovano le componenti necessarie alla gestione energetica (display, fotocamera, audio, tastiera) e i driver che permettono di interfacciarsi con

l'hardware fornito dal dispositivo.

- **Hardware Abstraction Layer :** appena sopra il kernel Linux troviamo l'HAL (Hardware Abstraction Layer), il quale offre un'interfaccia standard che illustra le funzionalità hardware del dispositivo a livelli superiori. Al suo interno troviamo una serie di moduli, ognuno dei quali implementa un'interfaccia per uno specifico componente hardware, come possono essere i sensori, la fotocamera o il Bluetooth.
- **Android Runtime :** un'applicazione, prima di venire caricata su un dispositivo, viene compilata in un formato bytecode intermedio, denominato DEX. Il bytecode verrà successivamente tradotto in istruzioni native dall'Android Runtime, utilizzando un processo di compilazione denominato AOT (Ahead-of-Time). Il processo appena descritto è conosciuto come ELF (Executable and Linkable Format), il quale sarà già in esecuzione ogni qual volta l'applicazione verrà lanciata, determinando una maggiore velocità dell'app e prolungando la durata della batteria del dispositivo.
- **Librerie :** l'ambiente di sviluppo Android include alcune librerie denominate Android Libraries, un insieme di librerie java specifiche per Android. Tra le più importanti troviamo :
  - **android.app** – Fornisce l'accesso al modello applicativo e costituisce la base di tutte le applicazioni Android
  - **android.content** – Consente la comunicazione tra applicazioni e componenti
  - **android.database** – Dà l'accesso ai dati pubblicato da un fornitore di contenuti e include classi per la gestione di database SQLite
  - **android.graphics** – API grafica bidimensionale per il rendering di colori, punti, canvas e altri elementi grafici.
  - **android.os** – fornisce funzionalità per l'utilizzo di servizi base del sistema
  - **android.media** – mette a disposizione classi per abilitare la riproduzione di audio e video
- **Application Framework :** L'application Framework è un insieme di API scritte in Java che mettono a disposizione tutte le funzioni che il sistema operativo

offre. Qui si trovano gli strumenti per costruire le interfacce, accedere ai componenti grafici, ai diversi layout, operazioni necessarie per sviluppare le applicazioni ed accedere a tutto ciò che offrono i livelli inferiori dello stack.

- **Applicazioni di sistema :** Nel livello più elevato troviamo le applicazioni di sistema, le quali forniscono un insieme di app fondamentali, quali browser, client, email e messaggistica. Assieme alle applicazioni di terze parti installate dall'utente sono presenti anche applicazioni native della versione di Android in uso sul dispositivo.

2.5 Creazione applicazione su Android Studio

Una volta avviato l'IDE viene mostrata una finestra di benvenuto, la quale dispone della lista di progetti aperti di recente per un accesso rapido e intuitivo e un menu che elenca le possibili modalità di creazione di un nuovo progetto.
Selezionando la voce *Start a new Android Studio project* si viene reindirizzati su una nuova schermata in cui è possibile specificare il nome dell'app. Il nome scelto comporrà il **Company Domain**, nonché il nome completo del package Java e la collocazione del progetto nel file system, che costituirà la prima parte del package Java dopo la sua creazione. Successivamente si potrà scegliere il fattore di forma dell'app, come smartphone/tablet, TV, wear o Google Glass, da cui dipenderanno anche i vari **template** di Activity disponibili per lo sviluppo del progetto (empty Activity, fragments, Google Maps Activity e molti altri).

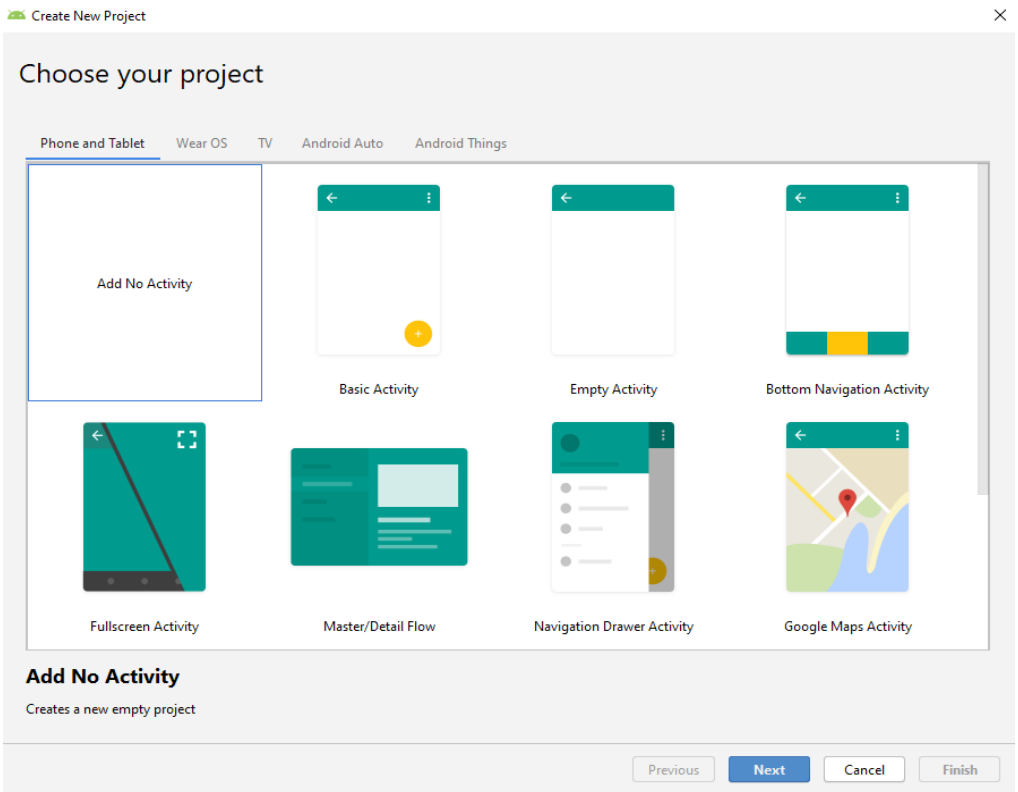

Figura 2.2: Finestra di Android Studio di selezione del template iniziale

E' facilmente intuibile, dagli strumenti messi a disposizione, che questa IDE sia stata realizzata per il mondo Android nella sua interezza.

2.6 Struttura Applicazione

Una volta configurato il progetto si può facilmente notare che la sua struttura è composta da tre elementi principali : la cartella con il codice **Java**, la cartella **res** e la cartella **manifests**, contenente il file di configurazione chiamato *AndroidManifest.xml*, inseriti nel modulo principale di default "app".

Viene mostrato uno screenshot dell'organizzazione dei file all'interno di Android Studio, è possibile distinguere i tre elementi principali appena citati :

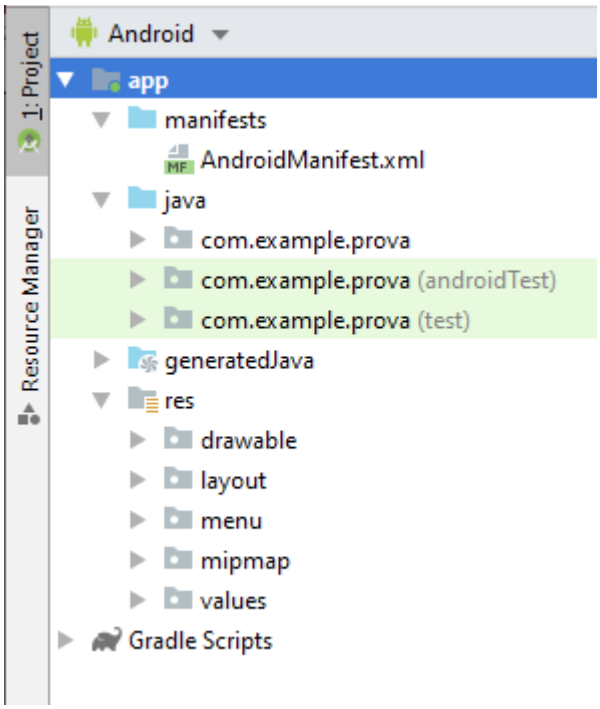

Figura 2.3 : Struttura di un progetto Android

2.6.1 La cartella Java

La cartella Java racchiude al suo interno tutti i file con estensione java che compongono l'applicazione. Tra di essi troviamo il file *MainActivity.java*, il quale rappresenta la prima Activity creata nella fase di configurazione del progetto. Il file sorgente *MainActivity.java* rispetta le proprietà del template scelto dal menu iniziale, mostrato nella figura 2.2, e sarà la prima pagina aperta all'esecuzione dell'applicazione. Modificando il codice all'interno del file è possibile regolare e gestire il comportamento dell'activity sfruttandone le sue caratteristiche.

2.6.2 La Cartella Res

La cartella Res contiene i file relativi alle interfacce grafiche delle Activity e tutte le risorse utilizzate dall'applicazione. Le risorse e i file vengono organizzati in sottocartelle, tra le più importanti troviamo **layout**, **values** e **drawable.**

- **Layout** contiene i file xml che definiscono l'architettura e il design

dell'interfaccia utente

- **Values** contiene stringhe, colori, dimensioni e altri valori che vengono definiti dal programmatore e utilizzati, come contenuto di appositi tag XML, all'interno di altre risorse o nel codice Java
- **Drawable** racchiude le immagini di diversi formati utilizzate all'interno dell'applicazione

2.6.3 La Cartella Manifests

All'interno di questa cartella si trova il file *AndroidManifest.xml*, file di configurazione che contiene informazioni essenziali sull'applicazione. Esso definisce il nome dell'applicazione, il nome del package, l'icona che rappresenta l'app sul dispositivo e il nome delle Activity. Le Activity che compongono l'applicazione devono essere tutte presenti all'interno del file manifest, dove vengono inseriti dettagli implementativi su di esse. Di grande importanza è la definizione di un elemento denominato *intent-filter* che viene associato a una sola Activity, la quale sarà il punto di partenza dell'applicazione quando verrà avviata attraverso il launcher del dispositivo. Viene riportato in seguito il codice del file AndroidManifest.xml creato dopo la configurazione di una semplice applicazione chiamata HelloWorld, composta dalla sola Activity MainActivity.java:

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3           package="com.example.helloworld">
4
5     <application
6             android:allowBackup="true"
7             android:icon="@mipmap/ic_launcher"
8             android:label="@string/app_name"
9             android:roundIcon="@mipmap/ic_launcher_round"
10            android:supportsRtl="true"
11            android:theme="@style/AppTheme">
12        <activity android:name=".MainActivity">
13            <intent-filter>
14                <action android:name="android.intent.action.MAIN"/>
15
16                <category android:name="android.intent.category.LAUNCHER"/>
17            </intent-filter>
18        </activity>
19    </application>
20
21 </manifest>
```

2.6.4 Gradle

Il progetto è contenuto all'interno di una cartella chiamata *app,* ovvero il modulo di default. *App* è soltanto uno dei contenitori che compongono l'applicazione, Android Studio divide infatti il progetto in diversi moduli, ognuno dei quali ha ruoli e finalità diverse.
Dopo il modulo di default troviamo la sezione **Gradle Scripts,** contenente i file di build che utilizza Gradle per trasformare il progetto in un'app funzionante.

Gradle è un *kit di automazione di sviluppo* utilizzato da Android Studio, esso si occupa di automatizzare una serie di attività spesso presenti nelle fasi di sviluppo di un processo. Tra di esse possono essere evidenziate le principali :

- compilazione
- collaudo
- packaging
- esecuzione di test automatizzanti
- documentazione
- pubblicazione
- distribuzione finale del software

Nella sezione **Gradle Scripts** troviamo due file chiamati *build.gradle*, il primo è dedicato all'intero progetto e il secondo soltanto al modulo di default app. Modificando il file riguardante il modulo app il programmatore può definire configurazioni personalizzate per l'implementazione dell'applicazione.
La sezione *dependencies* è molto importante per l'espansione delle funzionalità del progetto, essa permette di includere facilmente file binari o librerie.

Viene mostrato in seguito il file build.gradle(app) :

```
1 apply plugin: 'com.android.application'
2
3 apply plugin: 'kotlin-android'
4
5 apply plugin: 'kotlin-android-extensions'
6
7 android {
8     compileSdkVersion 29
9     buildToolsVersion "29.0.2"
10    defaultConfig {
11        applicationId "com.example.helloworld"
12        minSdkVersion 15
13        targetSdkVersion 29
14        versionCode 1
15        versionName "1.0"
16        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
17    }
18    buildTypes {
19        release {
20            minifyEnabled false
21            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'),
22            'proguard-rules.pro'
23        }
24    }
25 }
26
27 dependencies {
28    implementation fileTree(dir: 'libs', include: ['*.jar'])
29    implementation"org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
30    implementation 'androidx.appcompat:appcompat:1.0.2'
31    implementation 'androidx.core:core-ktx:1.0.2'
32    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
33    testImplementation 'junit:junit:4.12'
34    androidTestImplementation 'androidx.test:runner:1.2.0'
35    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
36 }
```

2.7 Il ciclo di vita di un'Activity

Durante l'utilizzo di un'applicazione l'utente ha la possibilità di navigare tra le finestre che la compongono, di passare all'utilizzo di un'altra app o semplicemente di chiuderla. Queste azioni determinano quello che viene chiamato lo *stato dell'applicazione*. Ogni Activity, ovvero la finestra contenente la parte visiva dell'applicazione, possiede un ciclo di vita illustrato nello schema sottostante :

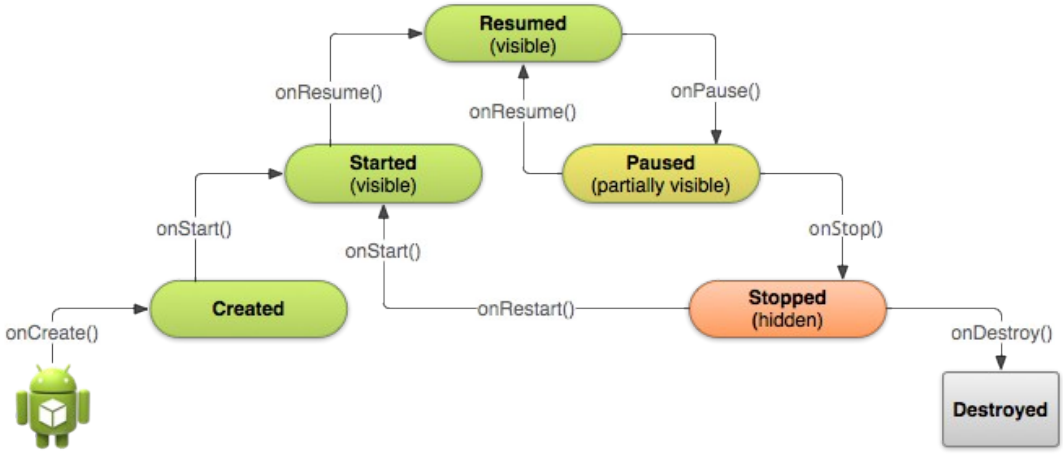

Figura 2.4 : Ciclo di vita di un'Activity

Il ciclo di vita di un'Activity è composto da 6 stati, i quali entrano in gioco in seguito all'invocazione da parte del sistema di metodi di callback, raffigurati con il proprio nome sulle frecce che portano da uno stato ad un altro.

Quando un'activity viene avviata per interagire direttamente con l'utente vengono inevitabilmente invocati tre metodi :

- **onCreate(boundle) :** è invocato quando l'activity viene avviata per la prima volta, essa viene creata e le vengono assegnate le configurazioni di base e il layout dell'interfaccia. Il Bundle savedInstanceState serve per riportare l'Activity nello stesso stato in cui si trovava la precedente istanza dell'Activity terminata.
- **OnStart() :** viene invocato quando l'activity sta per essere visualizzata, devono quindi essere attivate le funzionalità che mettono a disposizione le informazioni all'utente.
- **OnResume() :** è invocato quando l'activity inizia ad interagire con l'utente, diventa quindi destinataria di tutti i suoi input.

Quando l'utente mette in “pausa” l'applicazione, quindi passa all'utilizzo di altre attività del sistema, o semplicemente cambia finestra, Android mette a riposo l'activity,

passando per i tre seguenti metodi di callback :

- **onPause() :** invocato quando l'activity sta per essere “ibernata”, notificando la cessata interazione con essa.
- **OnStop() :** il metodo viene invocato quando l'activity non è più visibile all'utente.
- **OnDestroy() :** segnala la distruzione dell'activity, ovvero la sua chiusura.

S i può trovare l'invocazione del metodo onCreate anche all'interno del file MainActivity.java, in cui viene salvato lo stato dell'Activity e successivamente assegnato il file xml che disegna l'interfaccia grafica alla view :

```
1 package com.example.helloworld
2
3 import androidx.appcompat.app.AppCompatActivity
4 import android.os.Bundle
5
6 class MainActivity : AppCompatActivity() {
7
8     override fun onCreate(savedInstanceState: Bundle?) {
9         super.onCreate(savedInstanceState)
10        setContentView(R.layout.activity_main)
11    }
12 }
```

2.7.1 Intent e messaggi

Un'applicazione Android può essere composta da una o più Activity e, come è stato spiegato anche nel paragrafo precedente, l'utente deve avere la possibilità di navigare tra di esse. E' quindi necessario un meccanismo che implementi questa funzionalità, il quale prende il nome di **Intent**.

Gli **intent** rappresentano una forma di messaggistica gestita dal sistema operativo con cui una componente può chiedere l'esecuzione di un'azione da parte di un'altra

componente.

Essi posseggono una caratteristica molto utile: la capacità di “trasportare” dati che possono essere letti dal destinatario, i quali vengono generalmente chiamati **Extras**. Il passaggio degli Extras all'interno dell'intent è simile ad una struttura dati a mappa: con dei metodi *put* viene inserito un valore etichettato con una chiave e con i corrispondenti metodi get viene prelevato dalla componente di destinazione.

2.7.2 Implementazione

A livello di codice l'Intent, con relativo Extra caricato e successivamente prelevato, si presenta in questo modo :

```
1 Intent i=new Intent(this, SecondActivity.class);
2 i.putExtra("dati", dati_passati);
3 startActivity(i);
```

Viene quindi dichiarato l'Intent come normale oggetto java. Tra gli Extras è inserita la stringa “dati”, contrassegnata dalla chiave *dati_passati,* la quale verrà trasportata fino all'Activity di destionazione (SecondActivity). Una volta definito viene invocata la funzione StartActivity(), avente come parametro l'Intent appena creato.

```
1 Intent i=getIntent();
2 String dati=i.getStringExtra("dati_passati");
```

Nel metodo onCreate della seconda Activity si trova l'invocazione del metodo getIntent(), attraverso il quale sara' possibile, attravero il metodo getStringExtra, estrapolare la stringa definita nell'Activity precedente.

I meccanismi attivati e utilizzati dall'Intent vengono schematizzati attraverso pochi passaggi nella seguente figura :

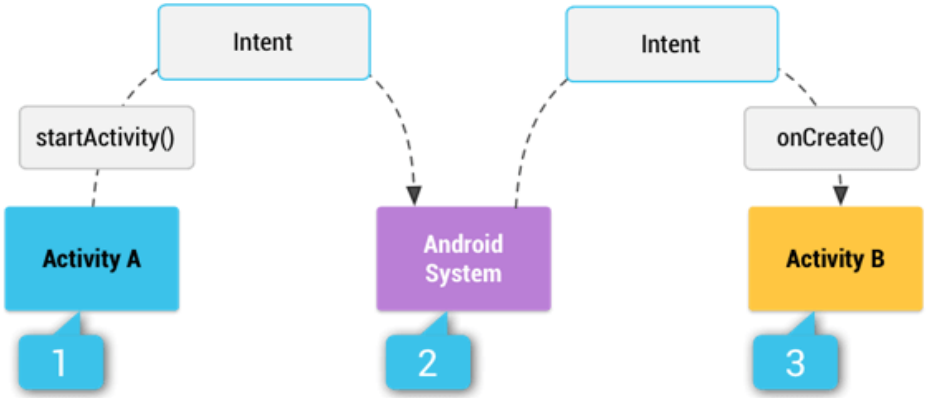

Figura 2.5 : Funzionamento di un Intent

2.8 Tecnologia Database SQLite

Per quanto riguarda la gestione logica e il salvataggio dei dati di CartellaClinica si è deciso di utilizzare il **database SQLite**. I motivi per cui è stata fatta questa scelta sono molteplici, data la perfetta propensione del software ad essere utilizzato nell'ambiente degli applicativi mobili.

SQLite è infatti leggerissimo, occupa poco spazio in memoria e sul disco, caratteristica ottimale per gestire una quantità di risorse limitate e ottimizzarne l'utilizzo. Per poter

integrare questo database alla propria applicazione Android non è necessario installare niente, poiché la libreria SQLite è già presente all'interno del sistema operativo.

2.9 Introduzione a SQLite

Figura 2.6 : Logo tecnologia SQLite

SQLite è una libreria software che implementa un DBMS SQL di tipo ACID (termine che deriva dall'acronimo **A**tomicità, **C**oerenza, **I**solamento e **D**urabilità), incorporabile all'interno di applicazioni. E' stata creata da D. Richard Hipp, il quale lo ha rilasciato nel pubblico dominio, rendendolo utilizzabile senza alcuna restrizione.
Contrariamente a molti altri sistemi di gestione di basi di dati, SQLite non è un motore di basi di dati client-server, ma è incorporato nel programma stesso. SQLite permette di creare una base di dati completa, quindi composta da tabelle, query, form, report e trigger, in un unico file.
Essendo un software compatto e leggero può essere trasportato o copiato in altri sistemi e continuare a funzionare in maniera regolare. E' un progetto in continua espansione, i quali punti di forza l'hanno portato ad essere il motore di database più diffuso al mondo.

2.10 SQLite in Android Studio

La creazione e la gestione del database avvengono direttamente all'interno dell'IDE, dove è possibile creare il database attraverso degli script SQL, aggiornarlo grazie alla classe SQLiteHelper e sottoporgli query di diversa natura.
E' necessario quindi dedicarsi a tre fasi essenziali per avere a disposizione un database con il quale interagire.

1. Creare la struttura del database

2. Creare una classe Java che estenda SQLiteOpenHelper
3. Creare una classe per l'interazione con il database

2.10.1 Classe DBhelper

In primo luogo si crea la classe Java che estende *SQLiteOpenHelper*, una classe astratta che implementa il pattern per la creazione, l'aggiornamento e la gestione della base di dati.
La sottoclasse di **SQLiteOpenHelper** deve implementare i metodi *onCreate()* e *onUpgrade()*, chiamati nel momento in cui non sia già presente il database, per crearlo, e quando la versione del database corrente è più alta della precedente, per aggiornarlo. Sarà poi possibile lavorare su oggetti **SQLiteDataBase** attraverso le operazioni di *Insert*, *query* componibili, *Update* e *Delete*. Qui sotto uno schema rappresentante la relazione tra gli elementi appena descritti.

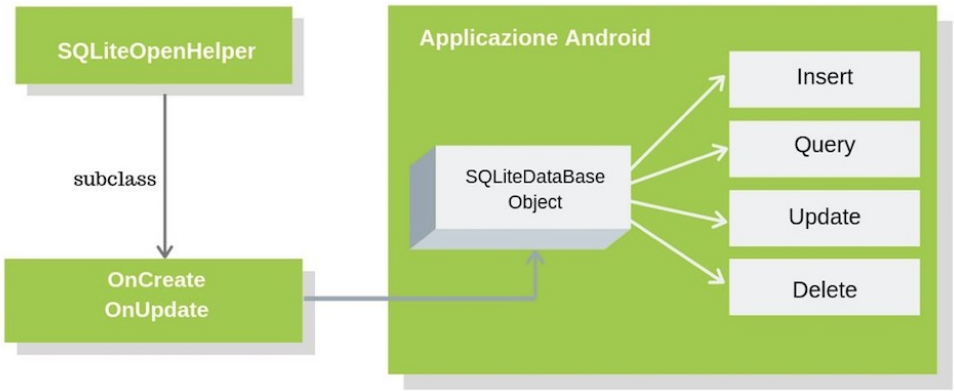

Figura 2.7 : Meccanismo della classe SQLiteOpenHelper e dei suoi metodi

La struttura interna del database nasce da uno script SQL scritto dal programmatore. Ne vengono definite dunque le entità, gli attributi, le chiavi primarie e le tabelle, rappresentanti i dati dell'applicazione. Il prodotto finale è una stringa che detta i comandi necessari per la creazione del database in linguaggio SQL.

Per una gestione più pulita e ordinata del db è conveniente creare una classe, chiamata generalmente DatabaseString, che si dedichi interamente alla definizione delle *stringhe del database.* All'interno vengono create e inizializzate le costanti che rappresentano le tabelle e le colonne del database, le quali verranno chiamate in altri file per effettuare operazioni di diverso genere.

Viene mostrato in seguito un esempio della classe Java appena descritta, avente lo script per la creazione del database e l'utilizzo delle costanti definite all'interno del file DatabaseStrings.

```
1 public class DBhelper extends SQLiteOpenHelper {
2
3 public static final String DATABASE_NAME="NAME_DATABASE";
4
5 public DBhelper(Context context) {
6      super(context, DATABASE_NAME, null, 1);
7 }
8
9 @Override
10 public void onCreate(SQLiteDatabase db){
11      String q="CREATE TABLE
12      "+DatabaseStrings.TBL_NAME+
13      " ( _id INTEGER PRIMARY KEY
14      AUTOINCREMENT," + DatabaseStrings.FIELD_SUBJECT+" TEXT,"+
15      DatabaseStrings.FIELD_TEXT+" TEXT," +
16      DatabaseStrings.FIELD_DATE+" TEXT)";
17      db.execSQL(q);
18 }
19
20 @Override
21 public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion){ }
22
23 }
```

Nello script viene notificata l'intenzione di creare il database attraverso il comando “CREATE_TABLE”, e successivamente vengono assegnati i ruoli di “PRIMARY KEY” e “TEXT” rispettivamente all'ID e al resto degli attributi della tabella. E' quindi molto semplice e intuitivo gettare le basi dello scheletro del database.

2.10.2 Classe DBManager

Per quanto riguarda l'interazione con il database viene creata una classe destinata a contenere i metodi per attuare le operazioni sui dati e la loro selezione. Oltre che essere veloce, questa organizzazione del codice, è fortemente intuitiva, rendendo la navigazione del programmatore tra i file del database molto rapida e diretta.
Nel corpo delle funzioni di questa classe vengono implementate tutte le operazioni necessarie per salvare una nuova scadenza all'interno del database, cancellarne una in base all'id o recuperare l'intero contenuto attraverso una query.
Per accedere al database e lavorare su un oggetto SQLiteDatabase vengono utilizzati due metodi in particolare : *getReadableDatabase()* e *getWriteableDatabase()*, i quali restituiscono un riferimento al database di "sola lettura" o di "scrittura", dando la possibilità di prelevarne o modificarne i dati.
Una volta recuperato l'oggetto desiderato vengono applicate le quattro operazioni CRUD, fondamentali per la persistenza del database : *Create*, *Read*, *Update*, *Delete*.
Di grande importanza anche la classe **Cursor**, impiegata per puntare ad un set di risultati delle query. Esso punta ad un record preciso del set di risultati e può essere spostato attraverso i metodi *moveToNext*, *moveToFirst*, *moveToLast* fino al ritrovamento del dato cercato.
In seguito un esempio di classe DBManager che implementa il salvataggio del database e la struttura di un metodo che utilizza la classe Cursor per la selezione di record all'interno del database :

```
public class DbManager{
     private DBhelper dbhelper;
     public DbManager(Context ctx){
          dbhelper=new DBhelper(ctx);
}
public void save(String sub, String txt, String date)
{
     SQLiteDatabase db=dbhelper.getWritableDatabase();
     ContentValues cv=new ContentValues();
     cv.put(DatabaseStrings.FIELD_SUBJECT, sub);
     cv.put(DatabaseStrings.FIELD_TEXT, txt);
     cv.put(DatabaseStrings.FIELD_DATE, date);
try {
db.insert(DatabaseStrings.TBL_NAME, null,cv);
}
catch (SQLiteException sqle){
// Gestione delle eccezioni
  }
}

public Cursor query() {
     Cursor crs=null;
     try {
        SQLiteDatabase db=dbhelper.getReadableDatabase();
        crs=db.query(DatabaseStrings.TBL_NAME, null, null, null,
        null, null, null, null);
      }
     catch(SQLiteException sqle) {
        return null;
     }

    return crs;
 }
}
```

2.10.3 DB Browser for SQLite

DB Browser for SQLite è un software open source per la gestione di database SQLite. Sono disponibili versioni per Mac e Windows, il software nasce come strumento molto versatile che permetta di eseguire operazioni o la semplice visualizzazione del database senza dover ricorrere a comandi SQL.

DB Browser for SQLite ha una grafica lineare e comprensibile, sin dalla prima pagina è possibile visualizzare le tabelle del database caricato e il contenuto dei suoi record. E' stato utilizzato questo strumento per monitorare velocemente il risultato delle operazioni effettuate sul database, controllando il valore dei dati e la loro posizione all'interno degli archivi. E' inoltre possibile effettuare operazioni di modifica, rimozione e creazione di tabelle e campi e componimento di query SQL.

Nonostante sia necessario installare software aggiuntivo il programma è essenziale per testare correttamente il funzionamento del database.

Viene mostrato lo screenshot della prima pagina dell'applicazione

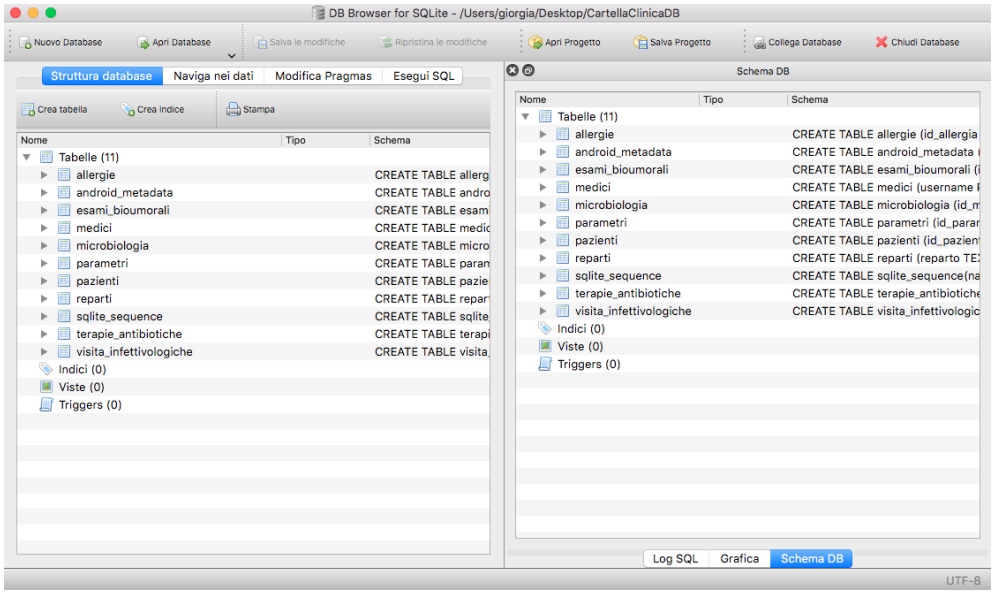

Figura 2.8 : Pagina principale del software DB Browser for SQLite

Per recuperare il file del database su Android Studio si ha a disposizione la sezione Tools, in cui è possibile accedere al File System. Una volta individuato il progetto, nella cartella *database,* sarà presente il file di estensione .db. E' possibile salvare il file e aprirlo con DB Browser for SQLite per esaminarne la struttura e il contenuto.

2.11 MPAndroid Chart

All'interno di CartellaClinica sono stati inseriti dei grafici rappresentanti l'andamento di determinati valori del paziente.

MPAndroidChart
created by Philipp Jahoda

Figura 2.9 : Logo MpAndroid Chart

Per realizzarli è stata utilizzata MPAndroid Chart, una libreria nativa, gratuita, affidabile e facile da usare.
E' ben documentata ed è possibile trovare molti esempi e tutorial utili per imparare a disegnare grafici online. Integrarla nel proprio progetto è semplice, è necessario aggiungere la seguente riga di codice nella sezione *dependencies* del file gradle.app e ri-sincronizzare il progetto:

```
dependencies {
    implementation 'com.github.PhilJay:MPAndroidChart:v3.1.0'
}
```

MPAndroid Chart mette a disposizione una vasta scelta di grafici, come :

- *LineChart*
- *BarChart,*
- *PieChart,*
- *ScatterChart,*
- *CandleStickChart,*
- *BubbleChart*
- *RadarChart*

Essi sono fortemente personalizzabili, è possibile modificarne la grandezza, il colore, i valori inseribili, le modalità di interazione, le legende, i limiti e le animazioni.
Durante lo sviluppo di CartellaClinica sono stati studiati in maniera più approfondita i LineChart, dovendo mostrare i dati solo sotto forma di Grafico Lineare per facilitarne l'interpretazione.

Dovendo realizzare numerosi grafici all'interno dell'applicazione è stata scelta una

tecnologia che permettesse di crearli in maniera rapida e senza occupare molta memoria. MPAndroid Chart è ottima poiché per creare un grafico sono necessari pochi passaggi: inizialmente si posiziona il grafico all'interno del layout relativo all'Activity in cui ci si trova.

```
1 <com.github.mikephil.charting.charts.LineChart
2         android:id="@+id/chart"
3         android:layout_width="match_parent"
4         android:layout_height="match_parent" />
```

Poi grafico viene recuperato all'interno della classe Java dell'Activity,

```
LineChart chart = (LineChart) findViewById(R.id.chart);
```

Dopodiché vengono creati gli oggetti di tipo Entry, i quali rappresentano i valori all'interno del grafico. Essi vengono inseriti uno ad uno all'interno della lista entries.

```
YourData[] dataObjects = ...;
List<Entry> entries = new ArrayList<Entry>();

for (int i = 0 ; i < entries.size() ; i++) {

    //conversione dei dati in oggetti Entry
    entries.add(new Entry(data.getValueX(), data.getValueY()));

}
```

Infine viene creato un oggetto di tipo LineDataSet, al quale viene aggiunta la lista di entries appena creata e riempita di dati. Vengono definite le operazioni per personalizzare la grafica e il comportamento del chart e infine viene aggiunto l'oggetto LineDataSet a un oggetto LineData, il quale contiene tutti i dati rappresentati da un'istanza del grafico e mette a disposizione ulteriori stili da applicare.

```
1   LineDataSet dataSet = new LineDataSet(entries, "Label");
2
3   dataSet.setColor(...);
4   dataSet.setValueTextColor(...); //viene personalizzato il grafico
5
6   LineData lineData = new LineData(dataSet);
7   chart.setData(lineData);
8   chart.invalidate(); //il grafico viene aggiornato
```

Il risultato sarà orientativamente questo :

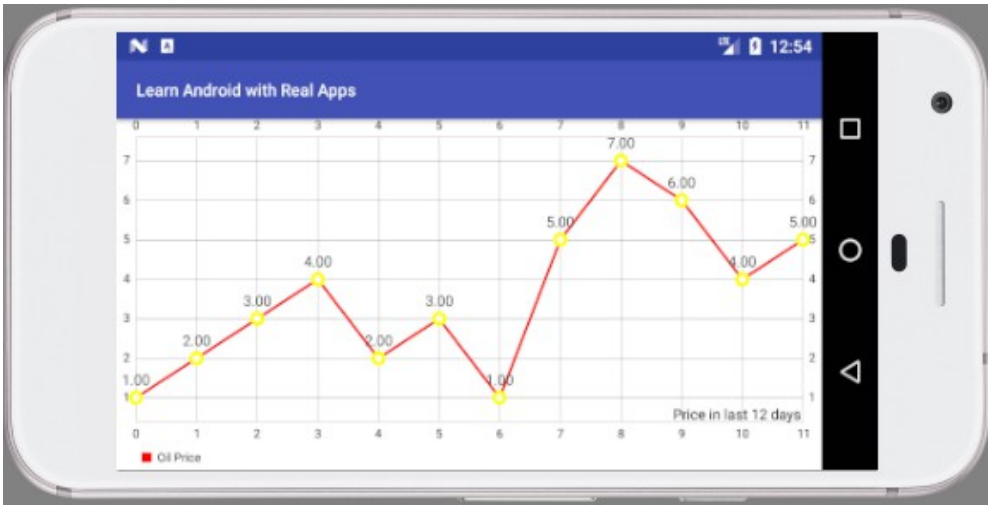

Figura 2.10 : Esempio di grafico lineare composto con MPAndroid Chart

Il grafico è molto semplice e chiaro, ha una struttura che facilita la lettura degli assi e dei valori, i quali sono anche cliccabili e/o ingrandibili.
La libreria MPAndroid Chart è stato un componente fondamentale per la realizzazione di un software completo e dall'immediata lettura e comprensione.

2.12 NFC

Figura 2.11 : Unione delle tecnologie NFC e Android

All'inizio di questo percorso è stato fatto un brainstorming su quelle che sarebbero potute diventare le funzionalità di CartellaClinica. Oltre che un applicativo mobile capace di mostrare e analizzare i dati di pazienti ricoverati si era valutato di integrare al

software la tecnologia NFC, per caricare dati all'interno del database grazie all'avvicinamento del medico al letto del paziente interessato. Questa possibilità andrebbe a eliminare la necessità di inserire i valori e i dati dei pazienti a mano, funzione del progetto che in futuro verrà sicuramente rimossa e rimpiazzata da un metodo di inserimento e/o selezione di informazioni automatico.

2.12.1 Cos'è NFC

NFC è l'acronimo di **N**ear **F**ield **C**ommunication. E' una tecnologia che permette a due device di connettersi tra loro attraverso un sistema wireless a corto raggio. Attualmente questa tecnologia è parecchio comune nell'ambito dei pagamenti elettronici, ma potendo stabilire uno scambio di dati bidirezionale può essere sfruttata facilmente anche per altre necessità.

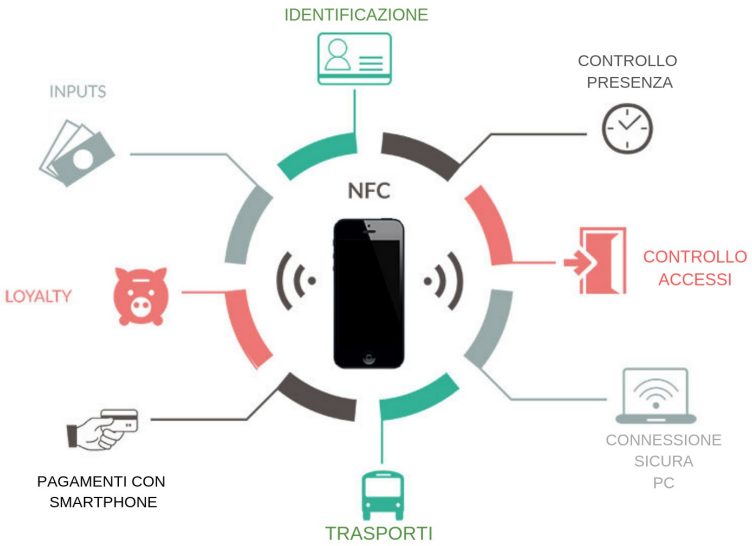

Figura 2.12 : Situazioni in cui viene sfruttata la tecnologia NFC

Ciò che rende molto accattivante questa tecnologia è che i target sono di dimensioni molto ridotte, piuttosto economici e integrabili in oggetti piccoli come etichette, portachiavi o carte. Inoltre, i tag NFC, non hanno bisogno di alimentazione, poiché sono device passivi: vengono alimentati mediante il campo elettromagnetico indotto dallo smartphone o dal tablet una volta vicino al tag.

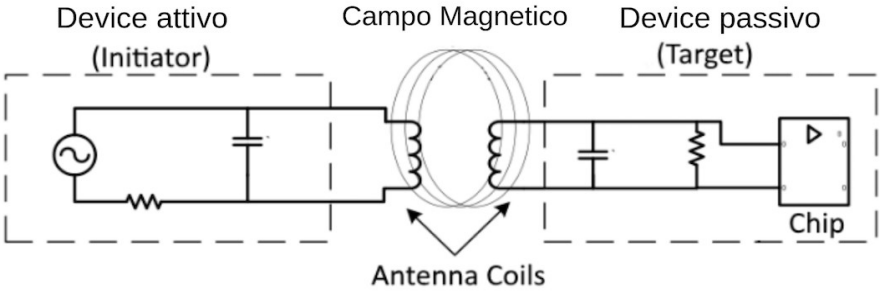

Figura 2.13 : Campo Magnetico indotto dall'avvicinamento dei device

2.12.2 Come avviene il dialogo

Possiamo individuare e denominare i due device interessati come :

- l'**initatior,** solitamente uno smartphone o un tablet, quindi un device evoluto, dotato delle API necessarie per l'interazione in NFC;
- i l **target,** un elemento molto semplice su cui vengono scritti o letti dati, denominato anche tag.

Quando questi due elementi si avvicinano (dai 4 ai 10 cm di distanza) la comunicazione tra i due può avere luogo.
Il dialogo tra i dispositivi può essere di diverso tipo e avere quindi una differente finalità, ovvero :

- *Peer Mode :* l'initatior e il target scambiano dati dialogando sullo stesso livello, ovvero con ruoli equivalenti;
- *Reader/Writer Mode :* Lo smartphone, o comunque il device evoluto, legge e/o scrive dati da/su un target passivo.
- *Card emulation mode :* Il dispositivo agisce emulando il comportamento di una card in modo da poter essere letto da ulteriori dispositivi.

2.12.3 Utilizzo NFC all'interno di CartellaClinica

Come è stato già accennato nel primo paragrafo, l'integrazione dell'NFC ottimizzerebbe parecchio l'utilizzo dell'applicazione. L'idea è quella di riuscire a caricare all'interno dell'app i dati utili nel momento in cui il medico si avvicina al letto dotato di NFC del paziente. Ogni target dovrebbe contenere il codice univoco che identifica il paziente ricoverato, e trasmetterlo direttamente al device, il quale sfrutterebbe il codice per selezionare i dati specifici del paziente e caricarli nell'interfaccia grafica.
Sono stati fatti degli studi su come implementare questi meccanismi ma non è stato ancora possibile realizzarli a causa del tempo limitato a disposizione per il tirocinio.
I dati arriverebbero direttamente dalla dorsale del Policlinico di Modena, operazione che richiede ulteriori accorgimenti e approfondimenti per interagire in maniera corretta con le tecnologie usate.
Tuttavia, sono state comunque fatte ricerche su come verrà integrata la tecnologia NFC all'interno dell'applicazione Android.

2.12.4 NFC in Android

Dalla versione 2.3 è stato introdotto il supporto per NFC su Android, arrivando fino alla nascita di Android Beam nella versione Ice Cream Sandwich, che ha definitivamente messo a disposizione la Peer Mode.

Affinché possano essere utilizzate le classi android.nfc e android.nfc.tech, è necessario rispettare tre prerequisiti, i quali vanno specificati all'interno dell'Android Manifest :

1. la versione minima dell'SDK deve essere la 10, inserendo per tanto :

```
1  <uses-sdk android:minSdkVersion="10"/>
```

2. deve essere presente il lettore NFC nel dispositivo, richiedendolo come apposita feature :

```
1  <uses-feature android:name="android.hardware.nfc" android:required="true"/>
```

3. e deve essere richiesta la permission necessaria con :

```
1  <uses-permission android:name="android.permission.NFC"/>
```

2.12.5 Classe NfcAdapter

La classe NfcAdapter è colei che permette al programmatore di accedere al "mondo" NFC. Una volta ricavato un suo riferimento si ha la conferma della sua validità, ovvero la disponibilità della tecnologia NFC, se esso non è nullo e se restituisce true all'invocazione del metodo *adapter.isEnabled()*.

2.12.6 Come avviene la comunicazione

Il meccanismo di comunicazione avviene, come in tante altre attività del sistema Android, mediante Intent. L'obiettivo di questi Intent è quello di essere intercettati da un'Activity e di essere classificati in base alle loro caratteristiche, le quali permettono di riconoscere il tipo di tag individuato. Vengono distinte in tre modi :

- *NfcAdapter.ACTION_NDEF_DISCOVERED :* è stato intercettato un tag che implementa lo standard NDEF;
- *NfcAdapter.ACTION_TECH_DISCOVERED :* viene notificato questo genere di tag quando, il tag individuato, implementa una tecnologia che non rientra nello standard NDEF o anche nel caso in cui il tag è di tipo NDEF ma non c'è alcuna applicazione installata in grado di gestirla
- *NfcAdapter.ACTION_TAG_DISCOVERED :* significa che non c'è alcuna applicazione sul dispositivo in grado di gestire gli standard cui il tag si riferisce.

Infine, affinché l'applicazione possa essere destinataria di uno di questi Intent è necessario che siano specificati all'interno del Manifest uno o più *IntentFilter* necessari :

```
1  <activity
2          android:name=".....
3          android:label=".....">
4          <intent-filter>
5                  <action android:name="android.intent.action.MAIN"/>
6                  <category android:name="android.intent.category.LAUNCHER"/>
7          </intent-filter>
8          <intent-filter>
9                  <action android:name="android.nfc.action.NDEF_DISCOVERED"/>
10                 <category android:name="android.intent.category.DEFAULT"/>
11                 <data android:mimeType="text/plain"/>
12         </intent-filter>
13 </activity>
```

In seguito viene mostrato uno schema che riassume il processo di intercettazione dell'Intent in base alle diverse casistiche spiegate in questo paragrafo :

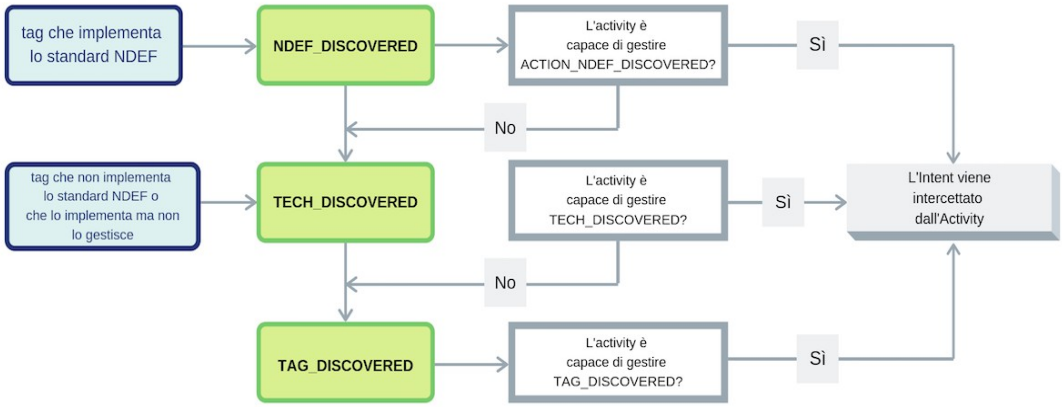

Figura 2.14 : Processo di intercettazione di un Intent NFC

Parte II

Implementazione

Capitolo 3
Progettazione del Software

3.1 Requisiti funzionali del sistema

Un requisito è una scrittura più o meno formale di una caratteristica del sistema. La stesura dei requisiti è preliminare allo sviluppo di un sistema software ed è una delle fasi più critiche della nascita dell'applicazione, poiché influenza in modo diretto la qualità e l'efficienza del prodotto finale.
Per quanto riguarda la progettazione logica si è utilizzato, come precedentemente spiegato, un database SQLite. Tuttavia, le caratteristiche del database verranno spiegate in seguito, dopo una iniziale introduzione di quella che è stata la progettazione più generale, ma di grande importanza, dei requisiti del software.
La valutazione di questi ultimi è stata fatta in collaborazione con la dottoressa Erica Franceschini, la quale ha apportato un grande contributo alla stesura dei comportamenti che il sistema avrebbe dovuto implementare per produrre un software di qualità.
Seguire le linee guida e le idee provenienti da una persona che utilizzerà l'applicazione è stato indispensabile per realizzare un software conforme alle esigenze di quest'ultima.

3.2 Funzionalità e operazioni di base

Come prima fase sono state analizzate le operazioni principali dell'applicazione, ovvero i meccanismi di base che determinano lo scopo e le funzionalità da lei offerte.
In seguito a un confronto con Erica sono state individuate le seguenti azioni centrali le quali costituiscono il fulcro dell'applicazione :

- **Registrazione** : Il medico ha la possibilità di registrarsi all'interno dell'applicazione, scegliendo il proprio nickname, univoco all'interno del database, la propria password e i reparti in cui lavora al momento della registrazione, i quali potranno essere modificati una volta loggato.

- **Login** : Un meccanismo di login è stato implementato nella prima finestra dell'app, dove l'utente può accedere, attraverso le sue credenziali determinate al momento della registrazione, al suo account.

- **Visualizzazione dei reparti** : Una volta effettuato l'accesso il medico ha a disposizione una pagina contenente la lista dei reparti in cui lavora al momento, all'interno dei quali si troveranno i pazienti ricoverati.

- **Modifica dei reparti e dei pazienti** : Il medico può inserire un nuovo reparto nella sua lista personale e ha inoltre la possibilità di inserire un nuovo paziente, il quale verrà posizionato all'interno del reparto in cui è stato ricoverato.

- **Visualizzazione del profilo del paziente** : Una volta inseriti i pazienti il medico potrà cliccarli per accedere all'area dell'applicazione, la quale può essere definita come il vero e proprio fulcro di essa : il profilo del paziente.

- **Aggiunta/Eliminazione dei parametri del paziente** : All'interno del profilo del paziente l'utente potrà analizzare diversi dati, in una prima facciata vedrà i dati anagrafici, un grafico sulla temperatura corporea e la lista delle allergie del

paziente. Successivamente abbiamo una sezione dedicata ai parametri (es. pressione, glicemia, frequenza respiratoria etc.) i quali sono inseribili e cancellabili dall'utente loggato. Le stesse funzionalità sono state implementate per le aree riguardanti le allergie, le consulenze, gli esami bioumorali, la microbiologia, le terapie antibiotiche e le co-patologie

- **Visualizzazione dei grafici degli Esami Bioumorali** : E' stata inoltre sviluppata un'area dedicata ai grafici rappresentanti l'andamento degli esami bioumorali del paziente. Vi si accede attraverso un bottone situato all'interno della sezione degli esami bioumorali, l'utente potrà visualizzare velocemente lo sviluppo dei valori nel tempo in maniera rapida e intuitiva. E' stato disegnato un grafico per ogni valore dell'esame fatto. (es. globuli bianchi, emoglobina, albumina, urea, eGRF etc.).

3.3 Casi d'uso

Il caso d'uso è una tecnica utilizzata per illustrare in maniera esaustiva e non ambigua i vari comportamenti che può avere il software durante la sua esecuzione. Non è altro, quindi, che una sequenza di azioni, con varianti, che producono un risultato osservabile da un attore.

Il documento dei casi d'uso descrive gli scenari elementari che si realizzano durante l'utilizzo del sistema da parte degli attori che si interfacciano con esso (in questo caso esseri umani). I casi d'uso devono essere elementari, quindi non scomponibili in casi d'uso minori che assumano ancora senso compiuto per gli attori. E' importante che ogni caso d'uso abbia un nome, un attore principale e eventualmente un attore secondario, deve inoltre avere un obiettivo, ovvero il motivo per il quale l'attore si trova ad interagire con esso.

L'attore rappresenta un'entità che non fa parte del sistema, ed è colui che esegue i casi d'uso.

Nel nostro caso l'attore del sistema è l'utente, più specificatamente il medico, e i casi d'uso sono : Registrazione, Login, Logout, Visualizza reparti, Visualizza Pazienti,

Visualizza profilo del Paziente, Modifica pazienti, Modifica reparti e Modifica parametri del paziente. Sono stati racchiusi i casi d'uso che riguardano le diverse informazioni dei pazienti (es. Allergie, Esami Bio Umorali, Visite Infettivologiche etc.) all'interno del caso d'uso “Profilo del Paziente”, poiché tutti possono essere solamente aggiunti o rimossi e quindi possono essere riassunti con un unico caso d'uso.

3.4 Diagramma dei Casi d'Uso

In seguito viene riportato uno schema del caso d'uso del software, il quale riassume le azioni che si possono compiere all'interno dell'applicazione, le loro conseguenze e le condizioni che escludono situazioni prive di senso :

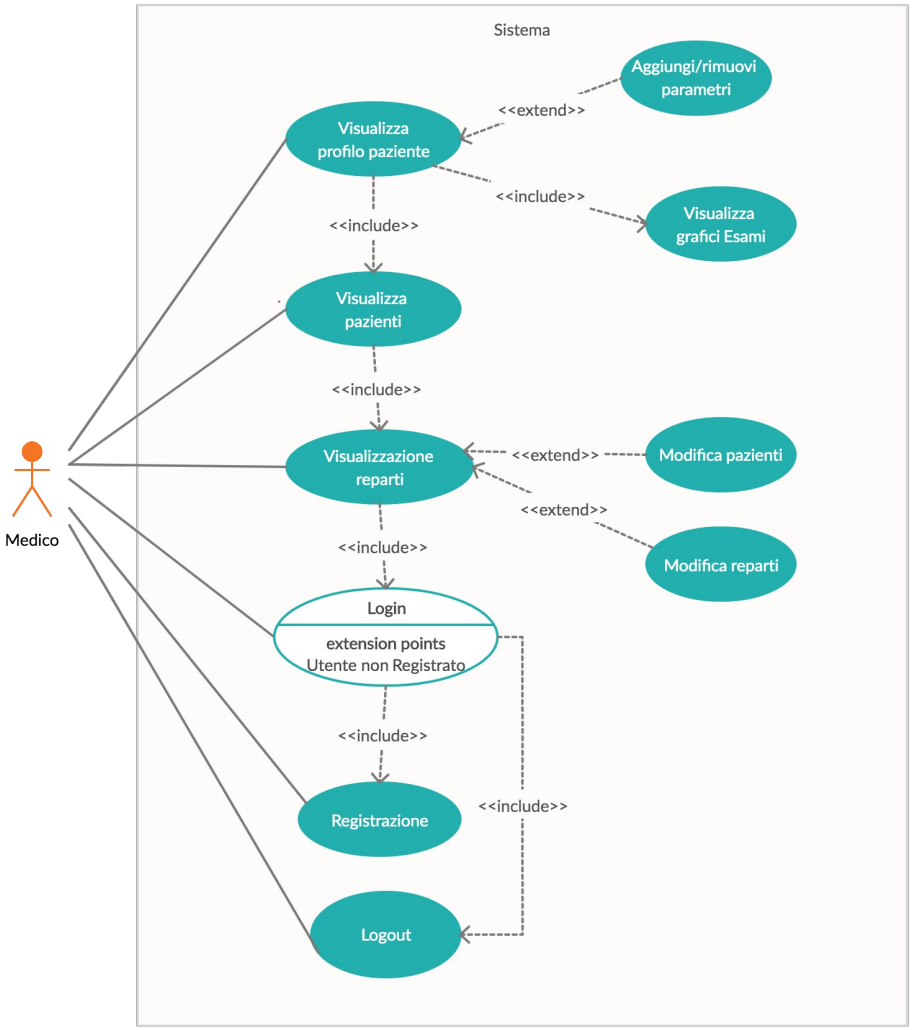

Figura 3.1 : Schema dei casi d'uso dell'applicazione

3.5 Diagramma delle attività

Un diagramma delle attività in UML è un diagramma che descrive un processo, organizza più entità in un insieme di azioni secondo un determinato flusso.
Esso modella un'attività relativa ad un qualsiasi elemento di modellazione, come :

- classi
- casi d'uso
- interfacce
- componenti
- operazioni di classe

Si utilizzano per modellare, come nel nostro caso, il flusso di un caso d'uso o modellare il funzionamento di un'operazione.

3.5.1 Componenti di un diagramma delle attività

Figura 3.2 : Componenti per costruire un diagramma delle attività

Gli elementi più semplici che costituiscono un diagramma delle attività sono, come mostrati in figura :

- Nodi azione : Specificano unità atomiche, non interrompibili e istantanee di comportamento
- Nodi di inizio e di fine : sono nodi speciali che indicano l'inizio e la fine del flusso
- Nodi oggetto : sono oggetti particolarmente importanti usati come input e output di azioni

- Nodi controllo : descrivono il flusso delle attività

Il diagramma delle attività viene utilizzato per rappresentare le transizioni del flusso di un caso d'uso, descrivono quindi il comportamento dinamico di un sistema. Questa rappresentazione delle attività è comoda in quanto consente di rappresentare sinteticamente il flusso principale e i flussi alternativi.

In seguito, nella figura 3.3, viene mostrato il diagramma delle attività dell'applicazione

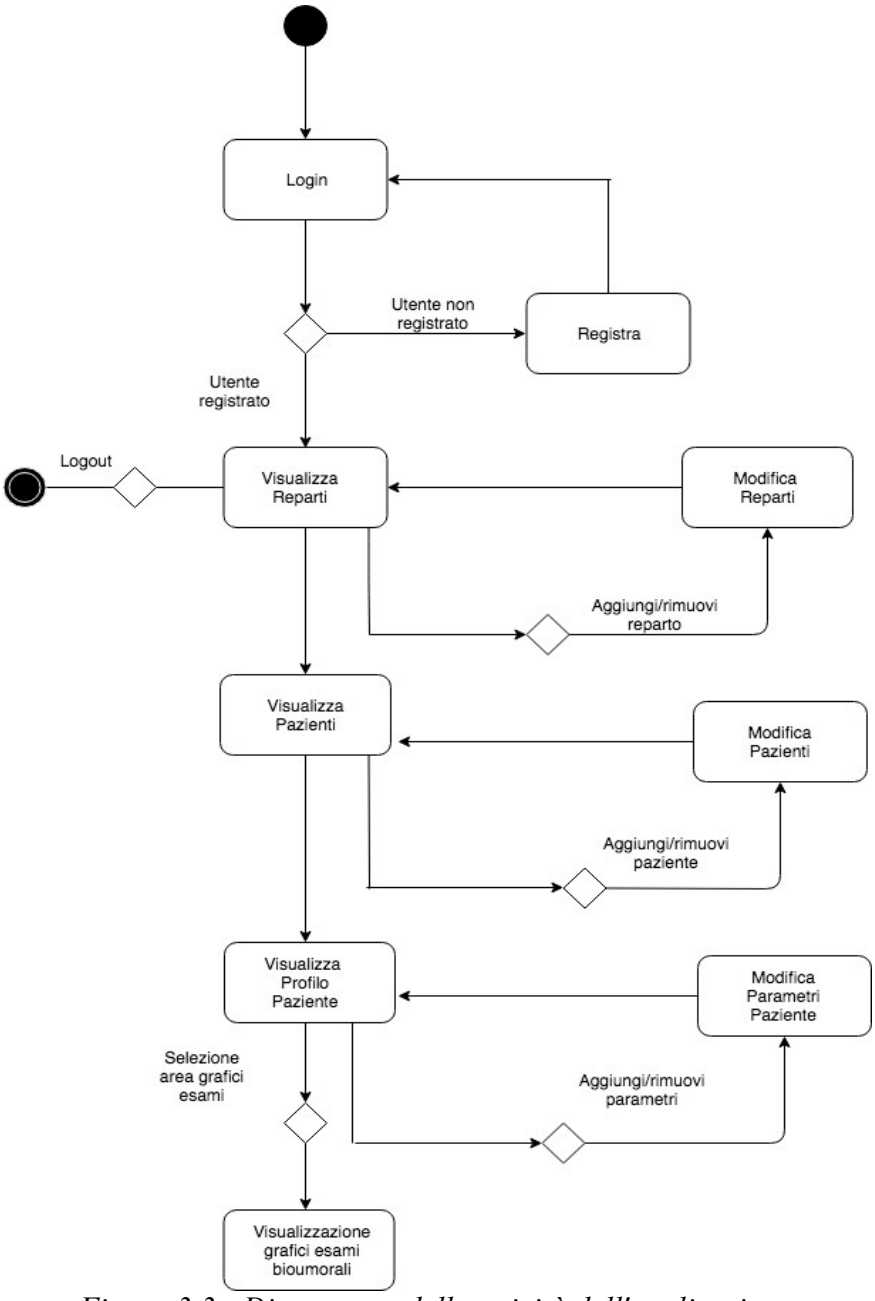

Figura 3.3 : Diagramma delle attività dell'applicazione

3.6 Progettazione del Database

In questa fase si definisce la struttura del Database dell'applicazione, come già spiegato in precedenza è stato utilizzato un Database di tipo SQLite, il quale verrà prima progettato in maniera astratta e poi creato attraverso alcuni script SQL.

3.7 Progettazione concettuale del Database

Per definire in maniera astratta le entità e le relazioni che vi sono tra di esse è essenziale disegnare un modello ER (modello entity-relationship) che rappresenti graficamente i dati utilizzati all'interno dell'applicazione.
Il modello E-R è la tecnica che definisce la progettazione concettuale della base di dati, preparativa alla fase avente come protagonista il modello relazionale e la fase di progettazione fisica, in cui si prendono infine in considerazione i software e hardware applicativi.

3.8 Modello ER di CartellaClinica

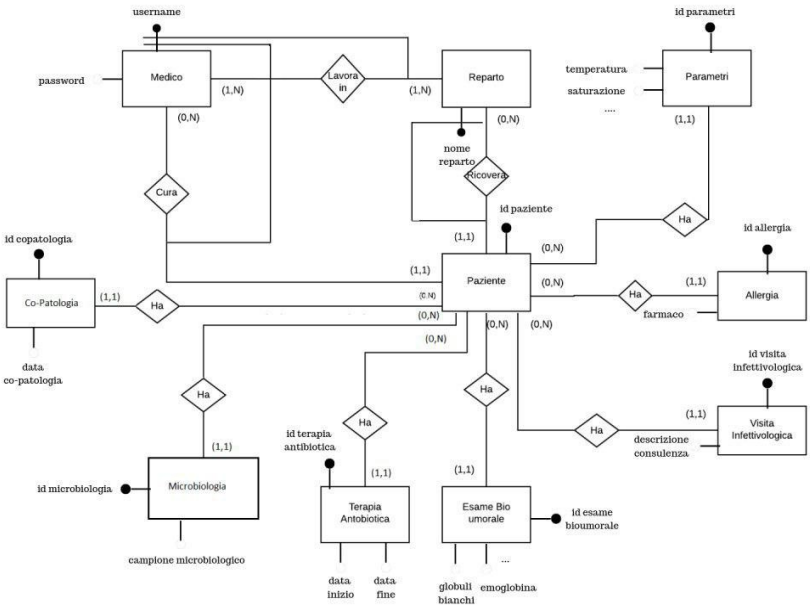

Figura 3.4 : Schema ER dell'applicazione

Come si può notare nella figura 3.4 lo schema ER è composto da 10 entità, i quali attributi sono stati mostrati solo in parte per una lettura facilitata, a causa del loro numero elevato. Questo accorgimento vale soprattutto per le entità legate direttamente al paziente quali : Parametri, Esami Bio-Umorali, Terapia Antibiotica, Microbiologia etc. poiché devono racchiudere numerosi valori medici difficilmente riportabili nello schema.

L'entità Medico, Reparto e Paziente rappresentano i “mattoncini” fondamentali da cui dipende il buon funzionamento del database. Tra di esse intercorrono, infatti, associazioni importanti che verranno approfondite in seguito.

3.8.1 Entità Medico

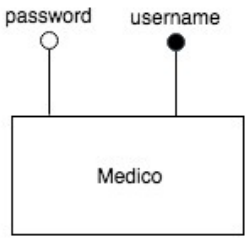

Figura 3.5 : Entità Medico

L'entità Medico serve per tenere traccia di tutti gli utilizzatori del sistema, nonché i medici registrati all'interno dell'applicazione. Ogni medico possiede uno username univoco all'interno del sistema e una password da lui generata, insieme compongono le credenziali che utilizzerà per accedere all'interno della sua area personale.

	username	password
	Filtro	Filtro
1	prova	123

Figura 3.6 : Tabella dell'entità Medico

Gli attributi di Medico sono :

- **username :** identificatore univoco dell'utente all'interno del sistema (primary key)
- **password :** parola chiave utilizzata per accedere all'interno del proprio account. Per quanto riguarda questo attributo non sono state imposte particolari condizioni (es. presenza di lettere maiuscole, numeri o caratteri speciali).

3.8.2 Entità Paziente

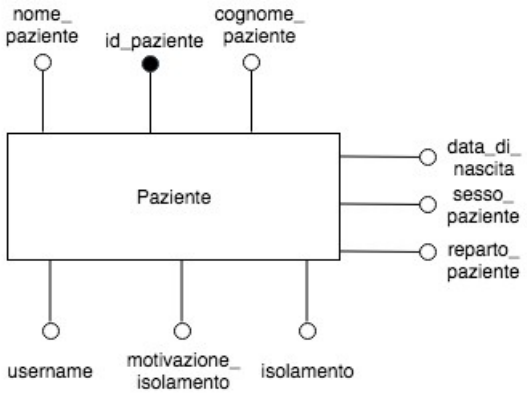

Figura 3.7 : Entità Paziente

L'entità Paziente rappresenta il paziente registrato all'interno del sistema. Un paziente non può esistere in maniera indipendente, sarà sempre il medico ad inserirlo all'interno dell'applicazione, specificandone i dati anagrafici (nome, cognome, data di nascita, sesso, reparto, isolamento da contatto). Un utente può essere inserito e in seguito rimosso dal sistema, ed è inoltre possibile modificare il suo stato riguardante l'isolamento da contatto : una volta uscito dall'isolamento il medico può variare l'attributo anche all'interno dell'applicazione.
Ogni utente ha il proprio id per essere individuato univocamente all'interno del database, chiaramente è stato necessario usufruire di un codice piuttosto che il loro nome, poiché è molto facile che esistano omonimi.

	id_paziente	nome_paziente	cognome_paziente	data_di_nascita	sesso_paziente	reparto_paziente	isolamento	ıtivazione_isolamer	username
	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro
1	1	mario	rossi	12/12/1996	uomo	Anestesia e Rian...	True	è isolato	prova

Figura 3.8 : Tabella dell'Entità Paziente

Gli attributi di Paziente :

- **id_paziente :** codice univoco che identifica un paziente e primary key dell'attributo.
- **nome_paziente :** nome proprio del paziente registrato
- **cognome_paziente :** cognome del paziente registrato
- **data_di_nascita :** data di nascita del paziente
- **sesso_paziente :** sesso del paziente registrato
- **reparto_paziente :** attributo di grande importanza, viene memorizzato nel record di ogni paziente il reparto nel quale è stato inserito, innanzitutto per posizionarlo graficamente nel luogo giusto all'interno dell'applicazione e anche per mostrarlo come semplice label all'interno del suo profilo.
- **isolamento :** attributo binario, determinato dal medico, che spiega se il paziente si trova in isolamento da contatto (True) o no (False).
- **motivazione_isolamento :** attributo che possiede un valore solo se la variabile "isolamento" è uguale a True, non è altro una descrizione aggiuntiva per specificare il motivo per il quale il paziente si trova in isolamento da contatto.
- **username :** l'attributo username non è riferito allo username del paziente, bensì a quello del medico che ha aggiunto il paziente all'interno del sistema. Questa variabile, in realtà, in questa versione del progetto non viene utilizzata particolarmente, poiché ogni medico può vedere tutti i pazienti ricoverati (anche se non inseriti da lui) all'interno dei reparti in cui lavora. La variabile era stata inserita, inizialmente, nel caso si preferisse mostrare soltanto i pazienti aggiunti dal medico loggato, ma valutando la cosa con Erica è stato deciso di scartare questa opzione. Sebbene quanto appena spiegato, si è preferito lasciare la

variabile all'interno del database, facilitando una possibile modifica delle funzionalità futura.

3.8.3 Entità Reparto

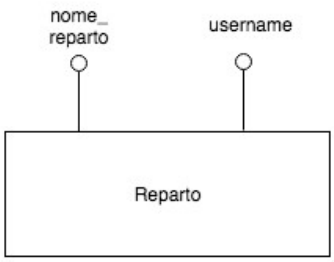

Figura 3.9 : Entità Reparto

L'entità Reparto viene usata per tenere traccia dei reparti in cui i diversi medici lavorano. Questa tabella, e i suoi rispettivi record, sono stati creati per agevolare le operazioni di inserimento, selezione, visualizzazione ed eliminazione dei reparti nei vari account dei medici. Viene infatti creato un nuovo record ogni volta in cui un medico aggiunge nella sua lista di reparti interessati un nuovo elemento.
Più nello specifico la riga all'interno della tabella avrà : all'interno dell'attributo username lo username univoco del medico e, in nome_reparto, il nome del reparto da lui appena aggiunto, chiaramente non già presente all'interno della sua lista. Non sono presenti infatti valori univoci e non replicabili, poiché lo stesso reparto può essere aggiunto da diversi medici, e lo stesso medico può aggiungere reparti già inseriti da altri medici all'interno del loro account.
Una tabella interamente dedicata a questi dati facilita enormemente le operazioni sopra citate, senza dover ricorrere a query troppo complicate o analisi incrociate di dati all'interno di diverse tabelle.

	reparto	username
	Filtro	Filtro
1	Anestesia e Rian...	prova
2	Anestesia e Rian...	prova

Figura 3.10 : Tabella dell'entità Reparto

Gli attributi di Reparto :

- **nome_reparto** : contiene il nome del reparto interessato. La scelta dei reparti da aggiungere è limitata, il medico non può comporre a mano il nome del reparto, ma è obbligato a scegliere tra una lista già esistente contenente i reparti del Policlinico di Modena. La stessa cosa avviene nel momento della rimozione di uno di essi, la lista si limita a mostrare i reparti precedentemente aggiunti dal medico e non l'intera lista, in modo da guidare l'utente verso operazioni che non portino il sistema in errore.
- **username** : username del medico che ha aggiunto il reparto nella sua lista personale.

3.8.4 Entità Parametri

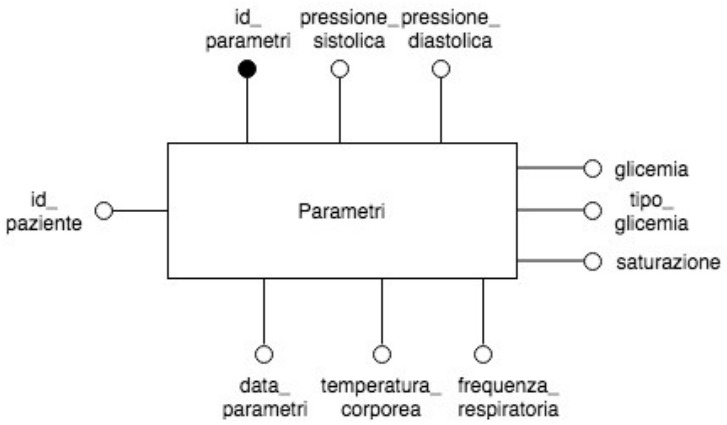

Figura 3.11 : Entità Parametri

La spiegazione dell'Entità parametri verrà utilizzata come descrizione generica di tutte le entità rimanenti (es. Co-Patologie, Esami Bioumorali, Visita Infettivologica, Allergia

etc.). Non è necessario descrivere entità per entità, poiché lo scopo e le operazioni compiute su di esse sono le medesime per tutte quante, variano soltanto gli attributi inseriti in base all'ambito descritto dall'entità.

In questo caso si può notare come prima cosa l'id dei parametri, essenziale per individuarli all'interno del database per le operazioni di visualizzazione ed eliminazione.

Le Entità in questione sono strettamente legate al paziente a cui sono state aggiunte o associate. Più specificatamente, il medico, una volta trovatasi all'interno del profilo di un determinato paziente, ha la possibilità di aggiungere questi parametri manualmente. L'aggiunta dei parametri viene in blocco, ossia, il medico deve inserire la pressione sistolica, quella diastolica, la glicemia, il suo tipo, la frequenza respiratoria etc. tutti insieme, i quali appartengono alla stessa classe Parametri. Essi verranno memorizzati il blocco e faranno parte dello stesso esame, compiuto in una determinata data e in una determinata ora. Questo vale anche per le entità Allergia (con gli attributi id_allergia, nome_allergia, descrizione_allergia, data_allergia, id_paziente) o per esempio Terapie Antibiotiche (con gli attiributi id_terapia, nome_terapia, inizio_terapia, fine_terapia, id_paziente). Esse vengono aggiunte ai dati del paziente, e vengono associate a quest'ultimo grazie al campo "id_paziente", il quale rende univoca la coppia Entità(Parametri, Allergia, Esami Bioumorali etc.)-Paziente.

	id_parametri	pressione_sistolica	pressione_diastolica	tipo_glicemia	valore_glicemia	valore_saturazione	frequenza_respirator	temperatura_corpore	data_parametri	id_paziente
	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro
1	3	69	4	Colazione	2	5	5	35	Lunedì 19 Agost...	1
2	4	12	2.5	Colazione	6.3	3	23	37.3	Lunedì 19 Agost...	1
3	5	2.0	6.3	Colazione	6	3	36	38.8	Lunedì 19 Agost...	1

Figura 3.12 : Tabella dell'Entità Parametri

3.9 Diagrammi UML

Figura 3.13 : Logo UML

Per concludere il percorso riguardante la progettazione dell'applicazione verranno mostrati i diagrammi UML delle classi dell'applicazione con l'obiettivo di rappresentare visivamente l'intero sistema. Utilizzando un'immagine è possibile individuare e capire al volo alcuni difetti o errori presenti nel software.

I diagrammi che verranno mostrati in seguito sono *diagrammi delle classi* di CartellaClinica, i quali metteranno in mostra i loro attributi, i loro metodi e soprattutto le loro relazioni.

Più specificatamente, all'interno dei diagrammi le classi sono composte da tre campi :

- Il nome della classe, nel riquadro iniziale
- Gli attributi della classe, subito sotto il nome
- I metodi della classe, per ultimi

3.9.1 Diagramma delle classi di CartellaClinica

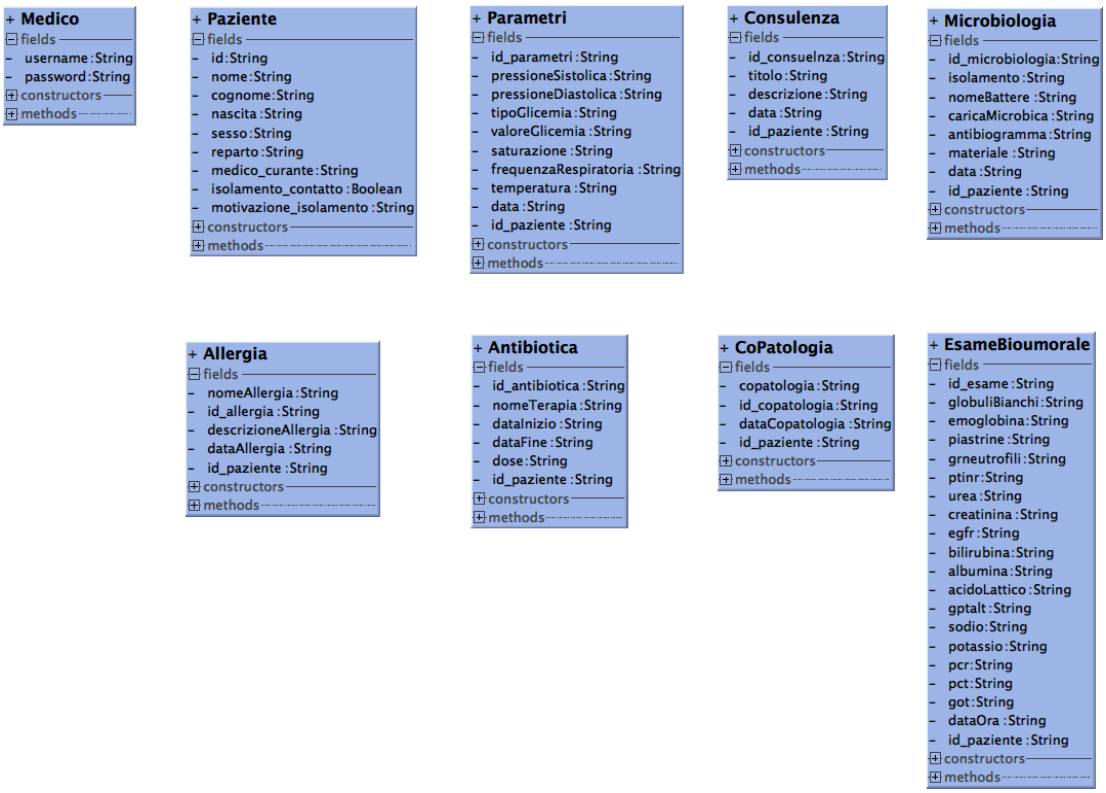

Figura 3.14 : Diagramma delle Classi di CartellaClinica

Le classi rappresentate in questo Diagramma sono le classi che costituiscono la struttura

dati del software. Si possono infatti leggere tutti gli attributi riportati sotto la categoria "Fields", essi corrispondono agli attributi assegnati alle entità del database.
Ogni classe rappresenta un oggetto a sé stante all'interno del programma, il quale ha dati, metodi e costruttori diversi.
Le classi di questo tipo sono state utilizzate per facilitare l'inserimento di dati nel database e la manipolazione di essi.

3.9.2 Diagramma delle Classi del Database

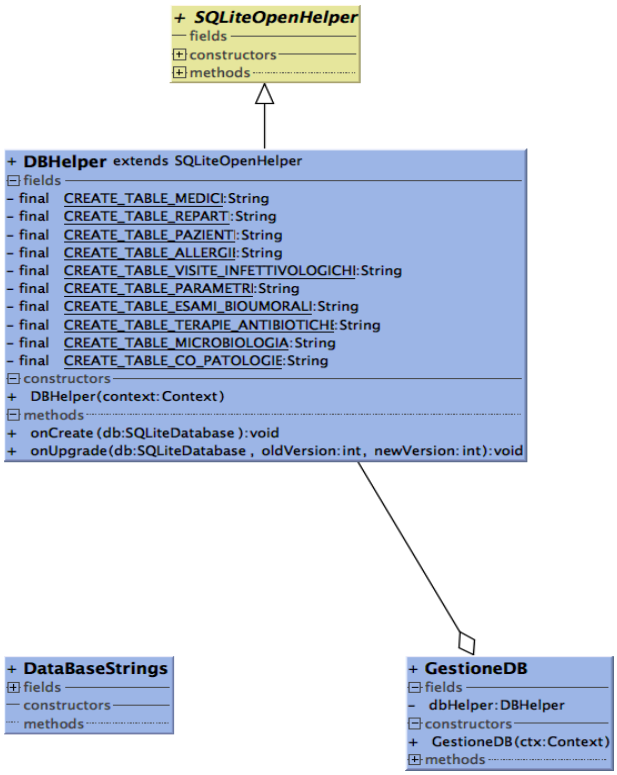

Figura 3.15 : Diagramma UML delle classi del Database

Vengono mostrate le tre classi create per l'utilizzo del database le loro relazioni.
DBHelper dipende direttamente dalla classe SQLiteOpenHelper, dal momento in cui alla sua creazione viene definito esplicitamente che la classe "estende SQLiteOpenHelper" per poter implementare le funzioni necessarie alla creazione, aggiornamento e eliminazione del database.
All'interno della classe GestioneDB è presente un riferimento alla classe DBHelper per

applicarne modifiche e query varie, motivo per cui viene disegnato un collegamento tra di esse.

3.9.3 Diagramma delle Classi riguardanti la gestione dei Reparti

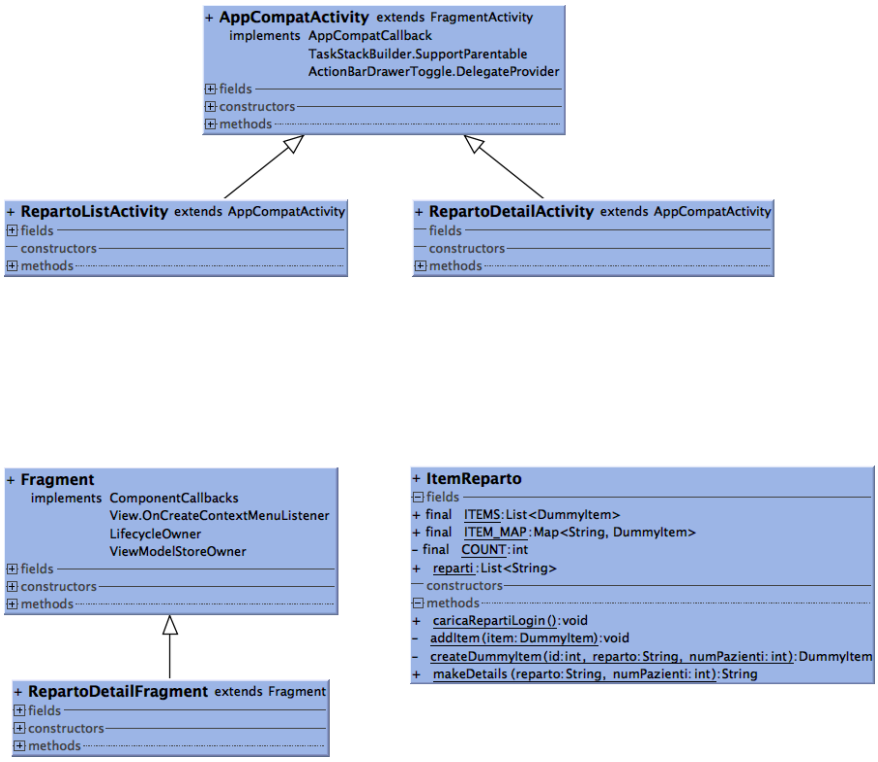

Figura 3.16 : Diagramma delle classi dei Reparti

Viene riportato come esempio anche il diagramma UML che schematizza le classi dedicate alla costruzione e gestione del Master/Detail Flow. Questo Layout di Android Studio verrà approfondito nel capitolo seguente riguardante l'implementazione del software. Intanto vengono riportate le classi che ne implementano le diverse parti, le loro dipendenze e caratteristiche.

La classe ItemReparto descrive l'oggetto che verrà caricato all'interno di questo layout.

3.9.4 Diagramma delle Classi delle Activities

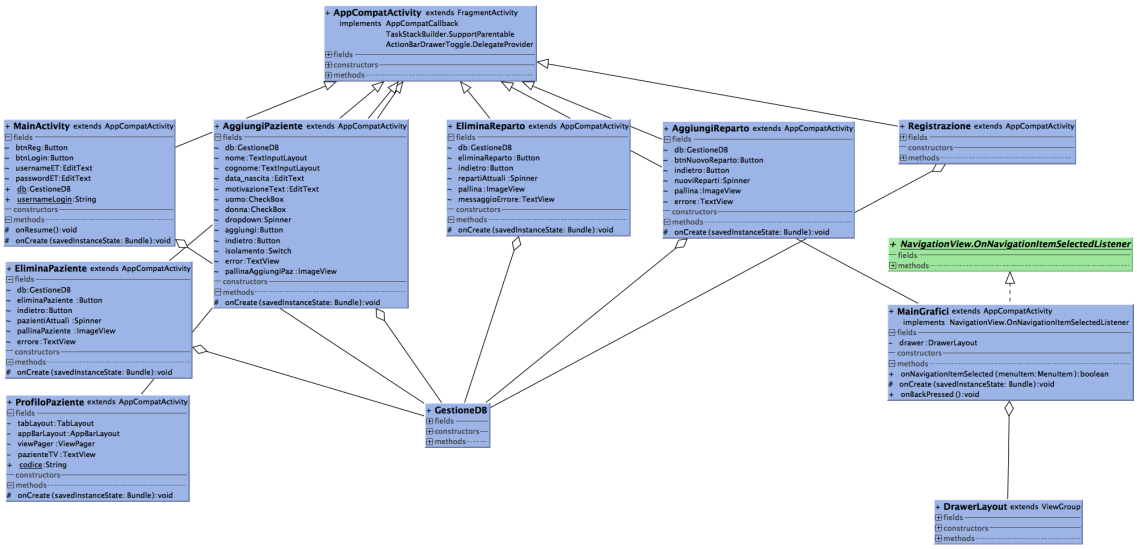

Figura 3.17 : Diagramma UML delle Activities

Sono state schematizzate tutte le Activities che compongono l'applicazione. Esse derivano tutte dalla Superclass AppCompatActivity, un tipo specifico di Activity che supporta le librerie necessarie per l'implementazione delle action bar e delle loro caratteristiche.

E' presente anche la classe GestioneDB, siccome all'interno di quasi tutte le activity è presente un riferimento ad essa per fare query sul database.

A destra troviamo infine la definizione della classe MainGrafici, la quale implementa l'evento OnNavigationItemSelectedListener direttamente legato alla componente NavigationDrawer che compone l'interfaccia. L'Activity riguardante i grafici è infatti stata implementata utilizzando il Layout NavigationDrawer, avente come componente integrata un DrawerLayout.

Capitolo 4
Implementazione

4.1 Introduzione all'implementazione

In questo capitolo si andranno a ripercorrere le fasi che hanno caratterizzato la creazione dell'applicazione. Inizialmente si studieranno i file dedicati alla realizzazione e interazione con il database, dopodiché si navigherà tra le diverse Activity, le loro componenti, la loro interfaccia e i dettagli implementativi che ne hanno contraddistinto le funzionalità.
Durante la descrizione e l'analisi delle finestre dell'applicazione verranno approfonditi i componenti usati all'interno di esse, ripercorrendone gli stadi di studio e creazione con alcune nozioni di teoria. Essi verranno infine, dopo essere stati spiegati, inseriti nel contesto di CartellaClinica, aggiungendo le accortezze e le modifiche apportate per rispettare le caratteristiche dell'applicazione.

Le Activities di Cartella Clinica verranno esaminate a fondo attraverso alcune fasi , come già introdotto nel paragrafo 3.2, “Funzionalità e operazioni di base”, che vengono individuate come 4 macro sezioni del progetto:

- Fase di Login e Registrazione
- Visualizzazione dei Reparti e dei Pazienti (con possibili modifiche)
- Visualizzazione del Profilo del Paziente (con possibili modifiche ai valori)
- Visualizzazione dei Grafici degli Esami Bio Umorali

E' stato inoltre creato un semplice schema che potesse aiutare la lettura del seguente capitolo, dal momento in cui contiene molti paragrafi e argomenti di diverso genere. Durante la lettura il lettore verrà accompagnato attraverso il flusso del processo della creazione dell'applicazione seguendo, uno ad uno, gli step evidenziati nello schema.

Figura 4.1 : Schema generale riassuntivo del capitolo

Per l'implementazione è stato usato, come già spiegato, Android Studio Versione 3.3.2, e per l'esecuzione e il testing del codice è stato usato un tablet Asus ZenPad 10 (Z300CL), 10.1" HD, con sistema operativo Android 6.0.1.

4.2 Implementazione del DataBase

Figura 4.2 : Schema generale con focus sull'implementazione del Database

Per creare il database, riprendendo ciò che è stato spiegato nel capitolo 2 riguardante le tecnologie usate, è necessario scrivere specifici script sotto forma di stringhe, i quali, essendo in linguaggi SQL, verranno tradotti per generare la struttura del database.

Iniziando ad analizzare il codice del progetto più nel dettaglio, possiamo vedere i file che racchiudono le operazioni più importanti per la creazione, modifica e aggiornamento del database.
La gestione del database viene spartita in tre diversi file : *DBHelper.java*, per la sua creazione, eliminazione e aggiornamento, *GestioneDB.java*, per la stesura delle query e *DataBaseStrings.java* per la definizione delle costanti rappresentanti gli elementi del database.

4.2.1 File DBHelper di CartellaClinica

All'inizio del file DBHelper vengono specificati tutti gli script riguardanti ogni tabella all'interno del database, per passare in seguito alla loro creazione. Gli script vengono

scritti all'interno di stringhe e successivamente passati alla funzione *onCreate*, dove verranno tradotti, grazie al metodo *execSQL()*, eseguito su ognuna di esse, in comandi SQL.

- **execSQL(String sql) :** esegue una singola istruzione SQL che non sia una SELECT o qualunque altra istruzione SQL che ritorna dati.

```
1 public class DBHelper extends SQLiteOpenHelper {
2
3     private static final String CREATE_TABLE_MEDICI = "create table "+
4             DataBaseStrings.TABLE_MEDICI + " (" +
5             DataBaseStrings.USERNAME_MEDICO + " PRIMARY KEY, " +
6             DataBaseStrings.PASSWORD_MEDICO + " not null)";
7
8     private static final String CREATE_TABLE_REPARTI = "create table "+
9             DataBaseStrings.TABLE_REPARTI + " (" +
10            DataBaseStrings.NOME_REPARTO + " TEXT, " +
11            DataBaseStrings.USERNAME_MEDICO + " TEXT)";
12
13    private static final String CREATE_TABLE_PAZIENTI = "create table "+
14             DataBaseStrings.TABLE_PAZIENTI + " (" +
15            "id_paziente INTEGER PRIMARY KEY AUTOINCREMENT, " +
16            DataBaseStrings.NOME_PAZIENTE + " TEXT, " +
17            DataBaseStrings.COGNOME_PAZIENTE + " TEXT, " +
18            DataBaseStrings.DATA_DI_NASCITA + " TEXT, " +
19            DataBaseStrings.SESSO + " TEXT, " +
20            DataBaseStrings.REPARTO + " TEXT, " +
21            DataBaseStrings.ISOLAMENTO_DA_CONTATTO + " TEXT, " +
22            DataBaseStrings.MOTIVAZIONE_ISOLAMENTO + " TEXT, " +
23            DataBaseStrings.USERNAME_MEDICO + " TEXT )";
24
25    private static final String CREATE_TABLE_ALLERGIE = "create table "+
26            DataBaseStrings.TABLE_ALLERGIE + " (" +
27            "id_allergia INTEGER PRIMARY KEY AUTOINCREMENT, " +
28            DataBaseStrings.NOME_ALLERGIA + " TEXT, " +
29            DataBaseStrings.DESCRIZIONE_ALLERGIA + " TEXT, " +
30            DataBaseStrings.DATA_ALLERGIA + " TEXT, " +
31            DataBaseStrings.ID_PAZIENTE + " TEXT )";
```

Il codice riportato è quello utilizzato per creare le Entità, precedentemente descritte e analizzate : Medico, Reparto, Paziente e Allergia.

Sempre all'interno dello stesso file viene fatto l'Override dei metodi *OnCreate* e *OnUpdate*, utilizzati rispettivamente per la creazione e l'aggiornamento della versione del database.

```
1  @Override
2     public void onCreate(SQLiteDatabase db) {
3         db.execSQL(CREATE_TABLE_REPARTI);
4         db.execSQL(CREATE_TABLE_MEDICI);
5         db.execSQL(CREATE_TABLE_PAZIENTI);
6         db.execSQL(CREATE_TABLE_ALLERGIE);
7         db.execSQL(CREATE_TABLE_VISITE_INFETTIVOLOGICHE);
8         db.execSQL(CREATE_TABLE_PARAMETRI);
9         db.execSQL(CREATE_TABLE_ESAMI_BIOUMORALI);
10        db.execSQL(CREATE_TABLE_TERAPIE_ANTIBIOTICHE);
11        db.execSQL(CREATE_TABLE_MICROBIOLOGIA);
12        db.execSQL(CREATE_TABLE_CO_PATOLOGIE);
13
14    }
15
16    @Override
17    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
18        String query_medici = "DROP TABLE IF EXISTS " + DataBaseStrings.TABLE_MEDICI;
19        String query_reparti = "DROP TABLE IF EXISTS " +
20        DataBaseStrings.TABLE_REPARTI;
21        String query_pazienti = "DROP TABLE IF EXISTS " +
22        DataBaseStrings.TABLE_PAZIENTI;
23        String query_allergie = "DROP TABLE IF EXISTS " +
24        DataBaseStrings.TABLE_ALLERGIE;
25        String query_visite_infettivologiche = "DROP TABLE IF EXISTS " +
26        DataBaseStrings.TABLE_VISITE_INFETTIVOLOGICHE;
27        String query_parametri = "DROP TABLE IF EXISTS " +
28        DataBaseStrings.TABLE_PARAMETRI;
29        String query_esami_bioumorali = "DROP TABLE IF EXISTS " +
30        DataBaseStrings.TABLE_ESAMI_BIOUMORALI;
31        String query_terapie_antibiotiche = "DROP TABLE IF EXISTS " +
32        DataBaseStrings.TABLE_TERAPIE_ANTIBIOTICHE;
33        String query_microbiologia = "DROP TABLE IF EXISTS " +
34        DataBaseStrings.TABLE_MICROBIOLOGIA;
35        String query_co_patologie = "DROP TABLE IF EXISTS " +
36        DataBaseStrings.TABLE_CO_PATOLOGIE;
37
38        db.execSQL(query_medici);
39        db.execSQL(query_reparti);
40        db.execSQL(query_pazienti);
41        db.execSQL(query_allergie);
42        db.execSQL(query_visite_infettivologiche);
43        db.execSQL(query_parametri);
44        db.execSQL(query_esami_bioumorali);
45        db.execSQL(query_terapie_antibiotiche);
46        db.execSQL(query_microbiologia);
47        db.execSQL(query_co_patologie);
48
49        this.onCreate(db);
50    }
```

Sono riportati nella porzione di codice i due metodi appena citati, implementati per eseguire le operazioni sugli script definiti a inizio file.

4.2.2 File DataBaseStrings di CartellaClinica

Viene data una veloce occhiata a come si presenta il file DataBaseStrings, importantissimo per avere un'organizzazione pratica e mentale delle variabili da utilizzare per la gestione dei singoli dati del Database.

Vengono racchiuse al suo interno tutte le variabili e costanti che rappresentano gli elementi del database :

```
1 public class DataBaseStrings {
2
3     public static final int DATABASE_VERSION = 5;
4     public static final String DATABASE_NAME = "CartellaClinica";
5
6
7     public static final String TABLE_MEDICI = "medici";
8     public static final String USERNAME_MEDICO = "username";
9     public static final String PASSWORD_MEDICO = "password";
10
11    public static final String TABLE_REPARTI = "reparti";
12    public static final String NOME_REPARTO = "reparto";
13
14    public static final String TABLE_PAZIENTI = "pazienti";
15    public static final String ID_PAZIENTE = "id_paziente";
16    public static final String NOME_PAZIENTE = "nome_paziente";
17    public static final String COGNOME_PAZIENTE = "cognome_paziente";
18    public static final String SESSO = "sesso_paziente";
19    public static final String REPARTO = "reparto_paziente";
20    public static final String DATA_DI_NASCITA = "data_di_nascita";
21    public static final String ISOLAMENTO_DA_CONTATTO = "isolamento";
22    public static final String MOTIVAZIONE_ISOLAMENTO = "motivazione_isolamento";
23
24    public static final String TABLE_ALLERGIE = "allergie";
25    public static final String ID_ALLERGIA = "id_allergia";
26    public static final String NOME_ALLERGIA = "nome_allergia";
27    public static final String DESCRIZIONE_ALLERGIA = "descrizione_allergia";
28    public static final String DATA_ALLERGIA = "data_allergia";
```

4.2.3 File GestioneDB di CartellaClinica

Come ultimo file viene mostrato GestioneDB.java, il quale si occupa della definizione delle query che servono per trovare, inserire, modificare o eliminare i dati all'interno del database.

```
1   public Boolean inserisciPaziente(String nome,
2                         String cognome, String data_di_nascita,
3                                     String sesso, String reparto, String isolamento,
4                        String motivazione, String medico_curante){
5
6         SQLiteDatabase db = dbHelper.getWritableDatabase();
7
8         ContentValues contentValues = new ContentValues();
9         contentValues.put(DataBaseStrings.NOME_PAZIENTE, nome);
10        contentValues.put(DataBaseStrings.COGNOME_PAZIENTE, cognome);
11        contentValues.put(DataBaseStrings.DATA_DI_NASCITA, data_di_nascita);
12        contentValues.put(DataBaseStrings.SESSO, sesso);
13        contentValues.put(DataBaseStrings.REPARTO, reparto);
14        contentValues.put(DataBaseStrings.ISOLAMENTO_DA_CONTATTO, isolamento);
15        contentValues.put(DataBaseStrings.MOTIVAZIONE_ISOLAMENTO, motivazione);
16        contentValues.put(DataBaseStrings.USERNAME_MEDICO, medico_curante);
17
18        long ins = db.insert(DataBaseStrings.TABLE_PAZIENTI, null, contentValues);
19
20        if(ins==-1) return false;
21        else return true;
22    }
23
24
25   public int ritornaNumeroPazientiReparto(String reparto){
26        SQLiteDatabase db = dbHelper.getReadableDatabase();
27        Cursor cursor = db.rawQuery("select * from pazienti where
28        username=? and reparto_paziente=?", new String[] {reparto});
29
30        int numeroPazienti = 0;
31
32        if (cursor.moveToFirst()) {
33
34            do {
35                numeroPazienti++;
36            } while (cursor.moveToNext());
37        }
38        db.close();
39
40        return numeroPazienti;
41    }
```

Sono stati riportati due esempi di query eseguibili sul database : *inserisciPaziente(...)* e *ritornaNumeroPazientiReparto(...)*. La prima viene utilizzata per inserire un nuovo paziente all'interno del sistema e la seconda ritorna il numero di pazienti attualmente presenti all'interno dell'applicazione. Le funzioni vengono ovviamente chiamate nel momento in cui queste operazioni devono essere eseguite, ovvero quando un medico inserisce un nuovo paziente all'interno di un reparto o per svolgere controlli sul numero di pazienti.

La prima cosa da notare all'inizio dei metodi è un riferimento alla classe **DbHelper**, attraverso il quale sarà possibile lavorare su oggetti di tipo *SQLiteDatabase*. Vi sono due modi distinti per ottenere un riferimento al DbHelper, presenti anche nell'esempio

qui sopra : **getReadableDatabase()** e **getWriteableDatabase()**.

Le finalità dei metodi sono differenti:

- **getReadableDatabase() :** restituisce un riferimento al database "in sola lettura"
- **getWriteableDatabase() :** ne permette la sola modifica

I metodi implementati nella porzione di codice presentano tre operazioni primarie da svolgere sulle tabelle del db : *save* per salvare una nuova scadenza, *delete* per cancellarne una in base all'id e *query* per recuperarne l'intero contenuto.

Tornando alla variabile di tipo *SQliteDatabase*, in entrambi i casi denominata "db", è possibile svolgere quattro operazioni CRUD, fondamentali per garantire la persistenza del database : *Create, Read, Update e Delete).*

Nelle API Android per SQLite è presente almeno un metodo per ogni tipo di azione :

- **query :** esegue l'azione SELECT sui dati leggendo il contenuto delle tabelle. Mette a disposizione svariati overload che permettono
- **delete :** per la cancellazione di uno o più record della tabella
- **insert :** utilizzata per l'inserimento. Essa riceve in input una stringa contenente il nome della tabella e la lista di valori per inizializzare un nuovo record mediante la classe **ContentValues**. Essa è una struttura a mappa che prende in ingresso coppie chiave/valore dove la chiave rappresenta il nome del campo della tabella.
- **update :** il metodo associa i parametri usati nell'insert e nel delete.

Per utilizzare questi metodi non è necessario fare un uso esplicito di SQL, tuttavia vi sono anche le funzioni *execSQL* e *rawQuery* che permettono di scrivere interamente i propri comandi e query. Essi sono stati utilizzati all'interno del metodo *ritornaNumeroPazientiReparto*( ), in cui è stato definito un oggetto di tipo **Cursor** :

```
1 Cursor cursor = db.rawQuery("select * from pazienti
2                 where username=? and reparto_paziente=?", new String[] {reparto});
```

Viene quindi scritta per intero la query, specificando le condizioni e le clausole da applicare alla ricerca del dato desiderato.

Cursor è l'interfaccia che rappresenta una tabella bidimensionale di qualsiasi database. Quando si tenta di recuperare dei dati utilizzando l'istruzione SELECT il database creerà prima un oggetto **Cursor** e poi ne verrà restituito il riferimento. Il puntatore di questo riferimento punta alla posizione 0, la quale è anche chiamata "la prima posizione del cursore", in modo tale che, quando si vuole estrapolare un dato, è necessario spostarlo sul primo record, utilizzando la funzione moveToFirst().
Una volta invocato moveToFirst() il puntatore del cursore viene spostato nella prima location definita, dalla quale sarà possibile accedere al primo elemento oppure spostarsi avanti nei record attraverso il metodi *moveToNext()*, *moveToLast()* o *moveToPrevious()*.

4.2.4 Struttura del Database Generata da DB Browser for SQLite

Vengono riportate alcune parti d'esempio della struttura del Database generata dal software DB Browser for SQLite.
DB Browser for SQLite genera automaticamente una sorta di riassunto delle operazioni di creazione e delle proprietà degli oggetti all'interno del database, il documento viene generato sul momento e può essere anche stampato su carta.
Lo schema mostra, per ogni tabella all'interno del database, il comando che l'ha creata e le caratteristiche degli attributi appartenenti ad essa.

Nome	Tipo	Schema
esami_bioumorali		CREATE TABLE esami_bioumorali (id_esame INTEGER PRIMARY KEY AUTOINCREMENT, globuli_bianchi TEXT, emoglobina TEXT, piastrine TEXT, gr_neutrofili TEXT, P_T_INR TEXT, urea TEXT, creatinina TEXT, eGFR TEXT, bilirubina_totale TEXT, albumina TEXT, acido_lattico TEXT, gpt_alt TEXT, sodio TEXT, potassio TEXT, PCR TEXT, PCT TEXT, GOT TEXT, data_esame TEXT, id_paziente TEXT)
id_esame	INTEGER	"id_esame" INTEGER PRIMARY KEY AUTOINCREMENT
globuli_bianchi	TEXT	"globuli_bianchi" TEXT
emoglobina	TEXT	"emoglobina" TEXT
piastrine	TEXT	"piastrine" TEXT
gr_neutrofili	TEXT	"gr_neutrofili" TEXT
P_T_INR	TEXT	"P_T_INR" TEXT
urea	TEXT	"urea" TEXT
creatinina	TEXT	"creatinina" TEXT
eGFR	TEXT	"eGFR" TEXT
bilirubina_totale	TEXT	"bilirubina_totale" TEXT
albumina	TEXT	"albumina" TEXT
acido_lattico	TEXT	"acido_lattico" TEXT

Figura 4.3 : Struttura della Tabella esami_bioumorali

Nome	Tipo	Schema
gpt_alt	TEXT	"gpt_alt" TEXT
sodio	TEXT	"sodio" TEXT
potassio	TEXT	"potassio" TEXT
PCR	TEXT	"PCR" TEXT
PCT	TEXT	"PCT" TEXT
GOT	TEXT	"GOT" TEXT
data_esame	TEXT	"data_esame" TEXT
id_paziente	TEXT	"id_paziente" TEXT
medici		CREATE TABLE medici (username PRIMARY KEY, password not null)
username	TEXT	"username" TEXT
password	TEXT	"password" TEXT NOT NULL
microbiologia		CREATE TABLE microbiologia (id_microbiologia INTEGER PRIMARY KEY AUTOINCREMENT, isolamento TEXT, nome_battere TEXT, carica_microbica TEXT, antibiogramma TEXT, data_microbiologia TEXT, materiale_microbiologia TEXT, id_paziente TEXT)
id_microbiologia	INTEGER	"id_microbiologia" INTEGER PRIMARY KEY AUTOINCREMENT
isolamento	TEXT	"isolamento" TEXT
nome_battere	TEXT	"nome_battere" TEXT

Figura 4.4 : Struttura delle tabelle esami_bioumorali, medici e microbiologia

Nome	Tipo	Schema
carica_microbica	TEXT	"carica_microbica" TEXT
antibiogramma	TEXT	"antibiogramma" TEXT
data_microbiologia	TEXT	"data_microbiologia" TEXT
materiale_microbiologia	TEXT	"materiale_microbiologia" TEXT
id_paziente	TEXT	"id_paziente" TEXT
parametri		CREATE TABLE parametri (id_parametri INTEGER PRIMARY KEY AUTOINCREMENT, pressione_sistolica TEXT, pressione_diastolica TEXT, tipo_glicemia TEXT, valore_glicemia TEXT, valore_saturazione TEXT, frequenza_respiratoria TEXT, temperatura_corporea TEXT, data_parametri TEXT, id_paziente TEXT)
id_parametri	INTEGER	"id_parametri" INTEGER PRIMARY KEY AUTOINCREMENT
pressione_sistolica	TEXT	"pressione_sistolica" TEXT
pressione_diastolica	TEXT	"pressione_diastolica" TEXT
tipo_glicemia	TEXT	"tipo_glicemia" TEXT
valore_glicemia	TEXT	"valore_glicemia" TEXT
valore_saturazione	TEXT	"valore_saturazione" TEXT
frequenza_respiratoria	TEXT	"frequenza_respiratoria" TEXT
temperatura_corporea	TEXT	"temperatura_corporea" TEXT
data_parametri	TEXT	"data_parametri" TEXT

Figura 4.5 : Struttura della tabella parametri

4.3 Struttura di CartellaClinica

Figura 4.6 : Schema generale con focus sulla struttura del progetto

Avendo già illustrato la metodologia di organizzazione dei file e delle cartelle di Android Studio nel capitolo 2 verrà riportato semplicemente lo screenshot del workspace dell'applicazione, analizzandone le cartelle e i file.

Vengono mostrati in seguito i packages contenenti i file *.java*, organizzati in base alle loro caratteristiche e i compiti da loro svolti :

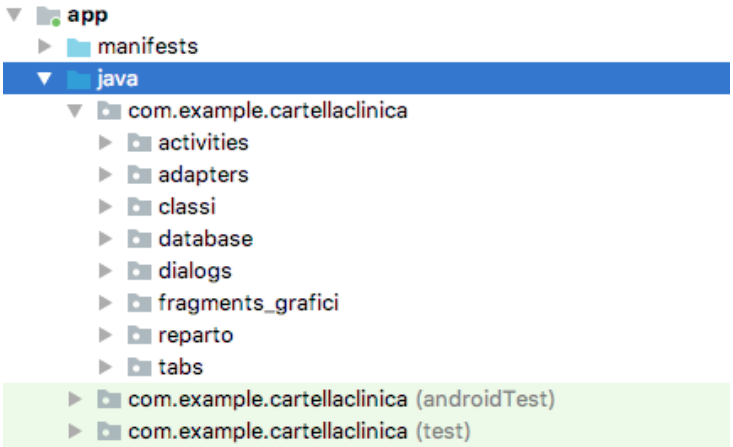

Figura 4.7 : Organizzazione dei file di CartellaClinica in AndroidStudio

- **Activities :** nella cartella "activities" sono presenti tutti i file .java direttamente collegati alle Activity dell'app, quindi le classi che si occupano del loro ciclo di vita (illustrato nel paragrafo 2.7), il caricamento dell'interfaccia, le risposte all'interazione dell'utente con l'applicazione, i dati relativi a quell'area del

progetto e molte altre operazioni.

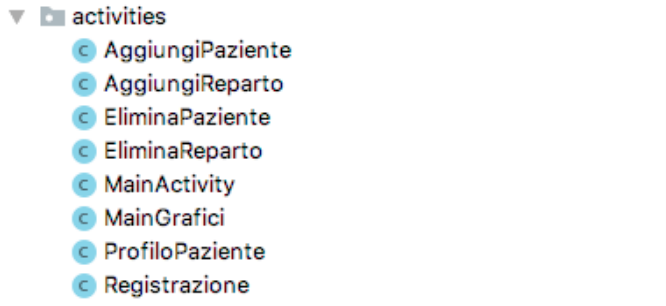

Figura 4.8 : Classi relative alle Activity dell'applicazione

- **Adapters :** in questa cartella sono stati raccolti tutti gli adapters, classi che fungono da "ponte" tra le componenti grafiche e le fonti di dati. Essi permettono di gestire il caricamento dei dati all'interno dell'UI (User Interface). Verranno approfonditi successivamente all'interno del capitolo, poiché fondamento del funzionamento di molti componenti dell'applicazione.

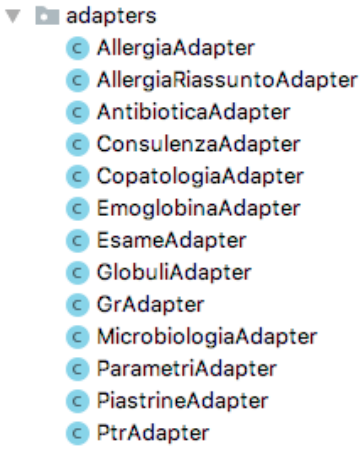

Figura 4.9 : Classi di tipo Adapter di CartellaClinica

- **Classi :** nella cartella Classi si trovano le classi rappresentanti gli oggetti utilizzati per la manipolazione dei dati, essi costituiscono più specificatamente la struttura dati dell'applicazione. Vi sono infatti, per esempio, le classi : *Paziente, Medico, Parametri, EsameBioumorale* etc. caratterizzate dagli attributi, e i corrispondenti metodi, associati alle entità spiegate nella progettazione del database. Vengono sfruttati *l'incapsulamento*, *l'ereditarietà* e il *polimorfismo* di java per avere a disposizione oggetti con attributi e metodi

raggruppati secondo una logica precisa.

Figura 4.10 : Classi rappresentanti gli oggetti di CartellaClinica

- **Database :** in questo package sono presenti i file *DBHelper.java*, *GestionDB.java* e *DataBaseStrings.java* per la gestione del database.

Figura 4.11 :File per la gestione del DB di CartellaClinica all'interno del package "database"

- **Dialogs :** cartella che raggruppa tutte le classi dedicate alla creazione di dialogs, finestre di piccole dimensioni che permettono all'utente utente di prendere una decisione o inserire informazioni aggiuntive.

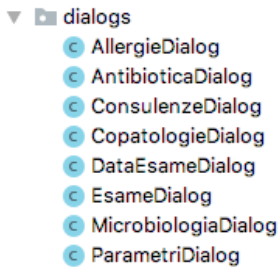

Figura 4.12 : Classi di tipo AppCompatDialogFragment che implementano i Dialogs di CartellaClinica

- **fragments_grafici :** vengono raggruppati tutti i fragment associati all'Activity di tipo **NavigationDrawer** implementata per la visualizzazione dei grafici degli esami Bioumorali. Ogni valore dell'esame possiede un proprio fragment, all'interno del quale si trova il grafico rappresentante il suo andamento generale e il record degli esami fatti, la data e il loro valore.

Figura 4.13 : Classi di tipo Fragment per i grafici degli esami bioumorali

- **Reparto :** vi sono i file che implementano il layout *MasterDetailFlow*, presente nella finestra dedicata alla visualizzazione dei reparti e dei pazienti al loro interno.

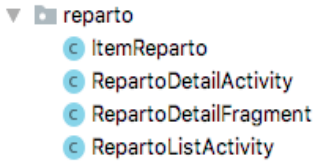

Figura 4.14 : File che implementano la finestra MasterDetailFlow dedicata ai Reparti

- **Tabs**: all'interno di tabs sono stati riuniti tutti i file di tipo Fragment che implementano le tab del profilo del paziente. Ogni Fragment è differente e indipendente, l'idea è quella di annidare più finestre all'interno della stessa Activity, grazie ad una Tab Bar che ne permette la navigazione.

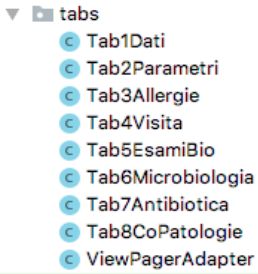

Figura 4.15 : Classi delle tabs che implementano il profilo del paziente

4.4 Fase di Registrazione e di Login

Figura 4.16 : Schema generale con focus sulle fasi di Login e Registrazione

All'avvio dell'applicazione viene aperta, come prima pagina, quella dedicata al Login dell'utente.

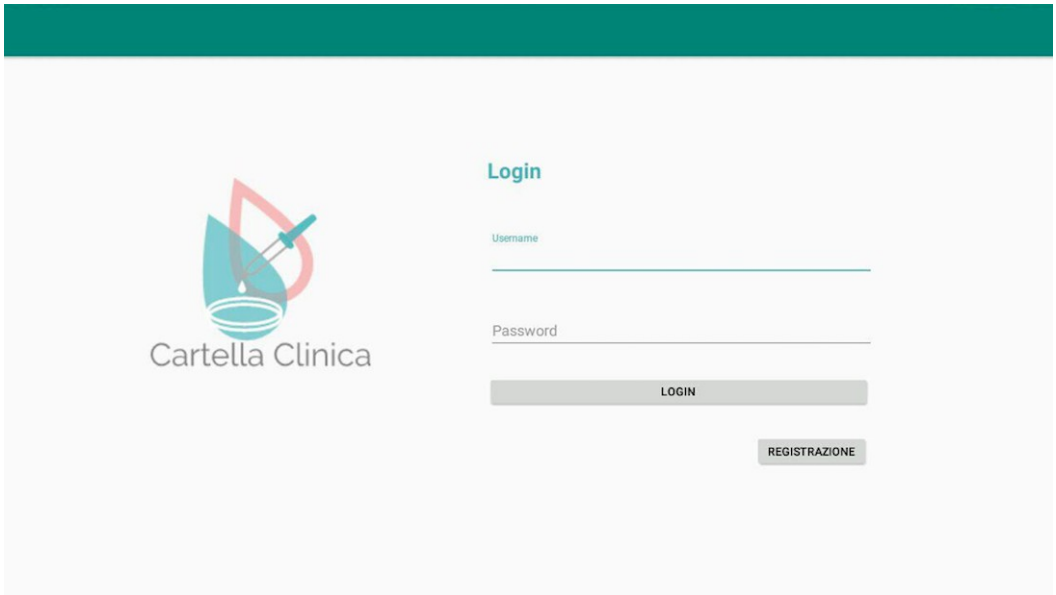

Figura 4.17 : Pagina di Login dell'applicazione

La grafica è piuttosto semplice : vengono richiesti username e password per accedere al portale o, in alternativa, l'utente può passare alla fase della Registrazione se non possiede ancora un account.

La fase di controllo del login viene effettuato dalla funzione

CheckUserPassword(String username, String password) definita nel file GestioneDB.java, alla quale vengono passate le credenziali come parametri.

```
1 public Boolean checkUserPassword(String username, String password){
2
3         SQLiteDatabase db = dbHelper.getReadableDatabase();
4
5         Cursor cursor = db.rawQuery("select * from medici where username=?
6                                 and password=?", new String[] {username,password});
7
8         if(cursor.getCount () > 0) return true;
9         else return false;
10     }
```

All'interno del metodo viene creato un riferimento a dbHelper di tipo "ReadableDatabase", viene composta la query che prende come argomenti lo username e la password inseriti dall'utente al momento del login e controlla se vi è almeno una coppia di questi valori all'interno del database (ovvero un utente registrato avente queste credenziali). Se il risultato è positivo il valore ritornato è True, di tipo Booleano, permettendo così l'accesso all'account, in caso contrario verrà mostrato un messaggio di errore all'utente.

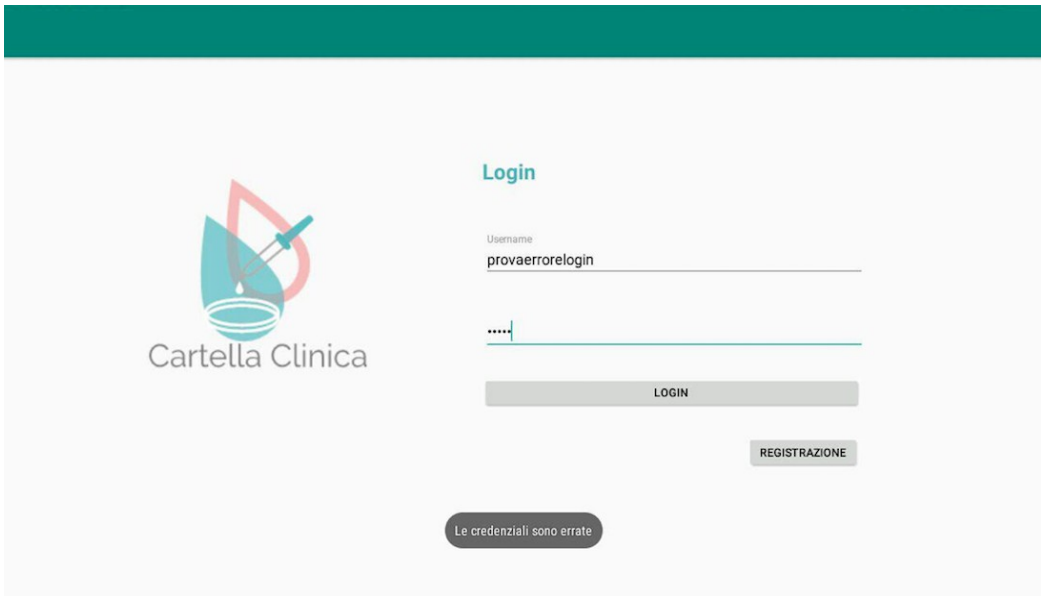

Figura 4.18 : Messaggio di Errore in caso di Login errato

Nel caso in cui l'utente non si sia ancora registrato è possibile farlo cliccando semplicemente sul bottone in basso a destra “Registrazione”, il quale ci porterà nella seguente Activity.

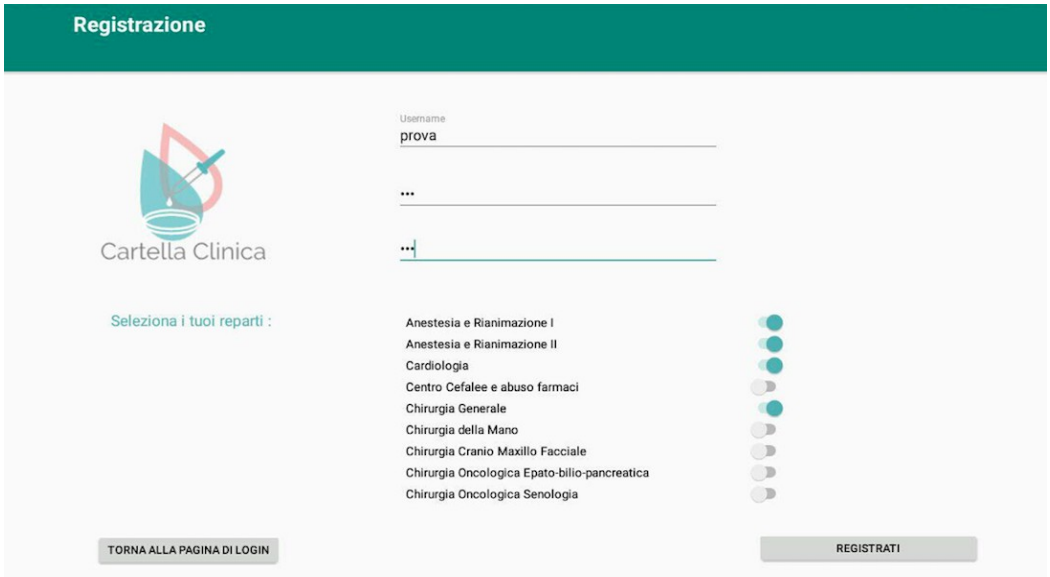

Figura 4.19 : Activity della Registrazione

Al momento della registrazione vengono richiesti all'utente alcuni dati obbligatori : lo **username** e la **password** (con la rispettiva conferma). Subito dopo le credenziali è stata inserita la lista rappresentante tutti i reparti all'interno del Policlinico, i quali possono essere selezionati o deselezionati, andando a specificare quali sono di proprio interesse.
Per realizzare la lista di reparti è stato utilizzato un LinearLayout Verticale inserito all'interno di una ScrollView per permetterne lo scorrimento.

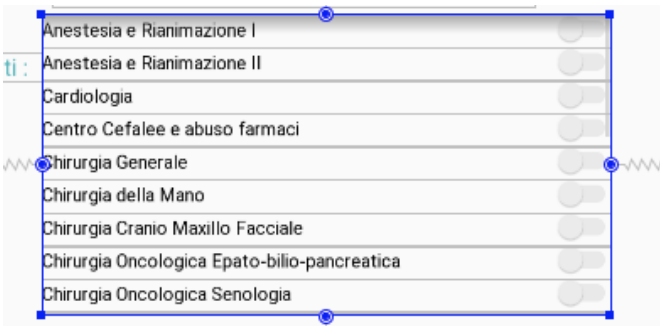

Figura 4.20 : ScrollView contenente i reparti selezionabili in fase di Registrazione

Ogni reparto è trattato separatamente infatti, nel momento in cui l'utente clicca sul pulsante **Registrati**, all'interno del listener del bottone *setOnClickListener*, vengono controllati uno ad uno gli Switch dei reparti, i quali :

-  : si presentano in questo modo se sono stati selezionati, il reparto corrispondente viene quindi aggiunto all'interno della tabella "reparti", avente come attributo "username" quello del medico che si sta registrando
-  : in caso contrario, invece, il reparto viene ignorato e non viene inserito nella lista del medico.

A livello di codice il controllo viene eseguito in questo modo, andando ad eseguire la query *inserisciReparto(String username, String reparto),* la quale apre il database in scrittura e inserisce il nuovo record

```
if (ane1.isChecked()){
    db.inserisciReparto(username,ane1.getText().toString());
}
if (ane2.isChecked()){
    db.inserisciReparto(username,ane2.getText().toString());
}
if (cardiologia.isChecked()){
    db.inserisciReparto(username,cardiologia.getText().toString());
}
if (cefalee.isChecked()){
    db.inserisciReparto(username,cefalee.getText().toString());
}
```

Una volta completata questa fase l'utente sarà registrato all'interno del database e potrà superare la fase del Login.

4.5 Visualizzazione dei Reparti e dei Pazienti

Figura 4.21 : Schema generale con focus sulla fase di Visualizzazione di Reparti e Pazienti

Ciò che viene mostrato all'utente dopo la procedura di Login è una finestra contenente i reparti scelti durante la registrazione (il medico ha anche la possibilità di non selezionare nessun reparto).

Alla base dell'aspetto di questa Activity c'è l'obiettivo di renderla il più pulita, semplice e intuitiva possibile.

Per realizzare un'interfaccia che rispettasse queste caratteristiche è stato utilizzato il **Master/Detail Flow**, un metodo di progettazione dell'UI in base al quale viene visualizzato un elenco di elementi a lato della pagina, denominato come "elenco principale". Selezionando un elemento dell'elenco le informazioni aggiuntive relative ad esso vengono presentate all'utente all'interno di un riquadro che occupa il resto dell'Activity. Per comprendere meglio l'aspetto del layout descritto viene mostrato in seguito uno screenshot della finestra :

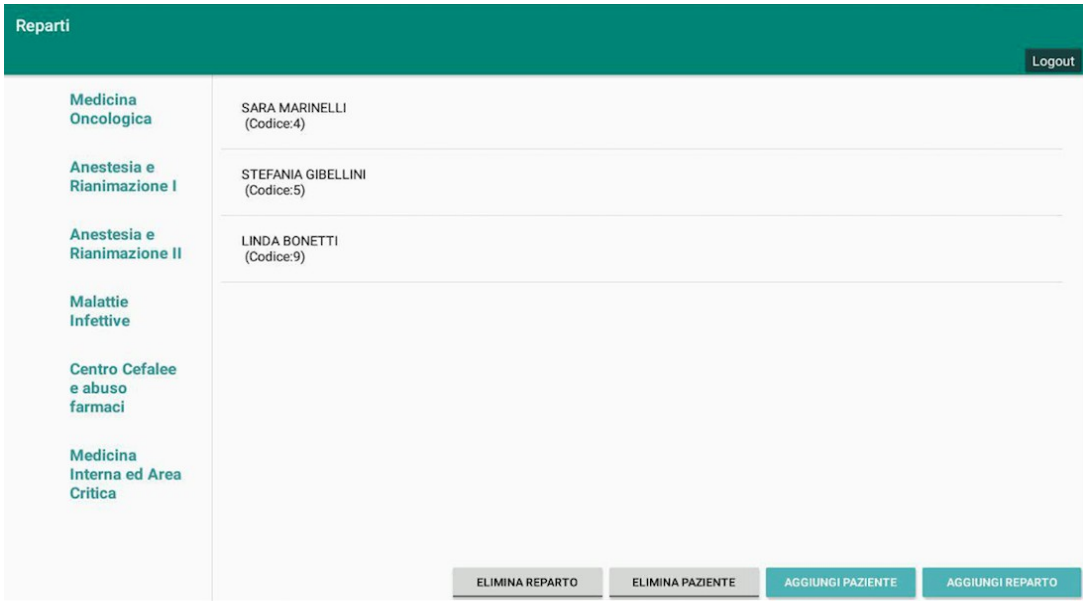

Figura 4.22 : Activity relativa alla visualizzazione dei Reparti

I reparti a sinistra sono stati precedentemente inseriti o modificati dal medico loggato, quando essi vengono cliccati vengono mostrati nel riquadro a destra i pazienti presenti all'interno del reparto.

4.5.1 Implementazione Master/Detail Flow

La creazione di questa pagina è avvenuta partendo dalla selezione del template in questo all'interno del menu messo a disposizione da AndroidStudio. Si è quindi scelto di non implementare l'Activity partendo da una finestra vuota (Empty Activity), ma utilizzando il supporto messo a disposizione dall'API.

Una volta scelto il template **Master/Detail Flow** dal menu dei template, strutturato come il menu mostrato in figura 2.2, è necessario inserire alcune informazioni riguardo gli oggetti che verranno trattati dall'interfaccia.

Nella prima casella bisogna inserire l'*Object Kind*, ovvero il tipo di oggetto generico (Item) che verrà mostrato all'interno del layout, nel nostro caso : **Reparto**.

Nella casella successiva si definisce nuovamente il tipo di oggetto, ma questa volta in forma plurale (Items) per rappresentare la lista di elementi che compone l'interfaccia.

Nel nostro caso avremo quindi una lista di reparti, i quali una volta cliccati, mostreranno nel contenitore a fianco i pazienti al loro interno.

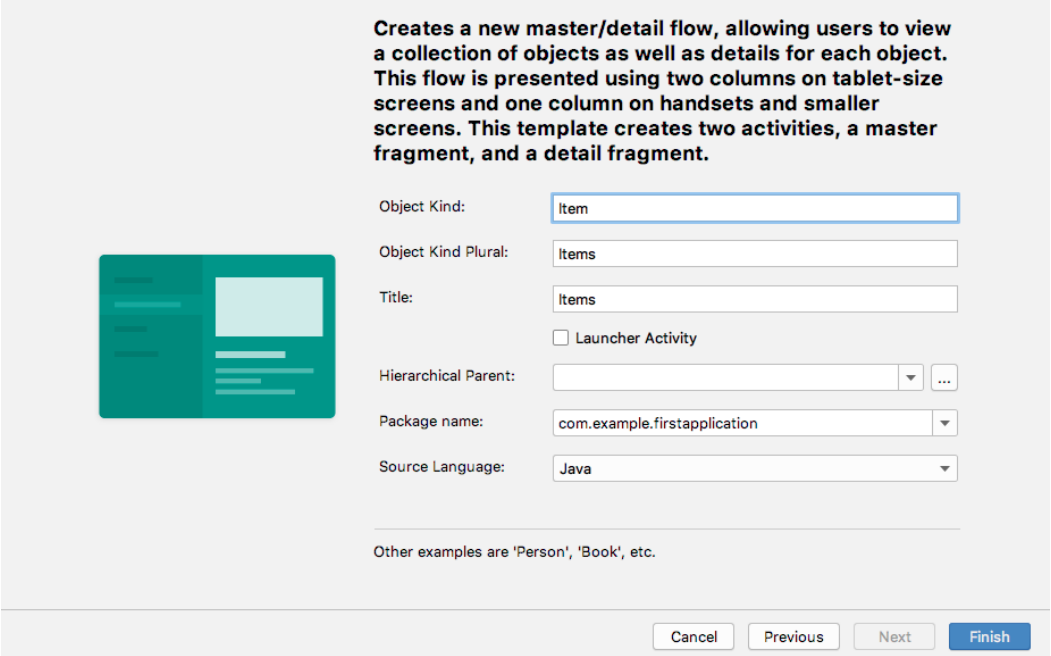

Figura 4.23 : Definizione degli oggetti utilizzati all'interno del Master/Detail Flow

Terminate le fasi di preparazione del layout e cliccato il pulsante "Finish"

AndroidStudio creerà automaticamente diversi file xml e java seguendo le informazioni specificate nel menu.

E' essenziale individuare e capire il compito di tutti i file che compongono il Master/Detail Flow, poiché diversamente sarebbe impossibile apportare modifiche al template di default a causa della loro complessità e della grande quantità di riferimenti e oggetti utilizzati al loro interno.

Prima di passare ad alcune nozioni teoriche riguardanti i file del master/detail flow viene mostrato uno schema per distinguerne le sezioni principali da consultare durante la loro descrizione :

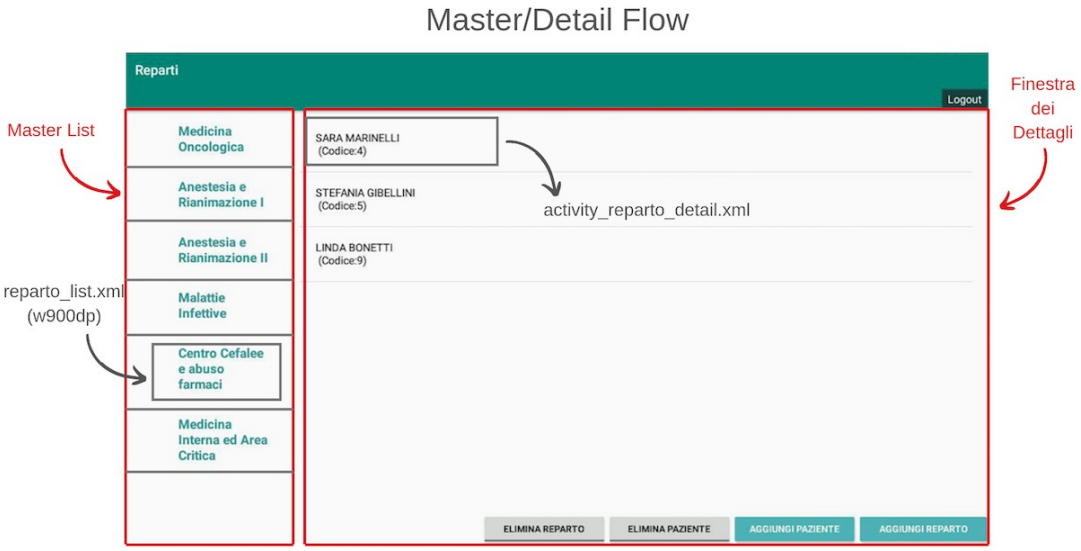

Figura 4.24 : Schema struttura interfaccia Master/Detail Flow

I file generati verranno elencati e spiegati in seguito :

- **activity_reparto_list.xml :** è il file che rappresenta il layout principale contenente la lista di oggetti Reparto (*master list*). è caricato dalla classe RepartoListActivity. Graficamente è composto da una *toolbar*, un *floating action button* e include il file *reparto_list.xml*, che verrà spiegato in seguito.

- **RepartoListActivity.java :** è la classe legata all'Activity responsabile della visualizzazione e della gestione della *master list* (dichiarata nel file activity_reparto_list.xml), si dedica inoltre alle risposte generate dopo la selezione degli elementi all'interno di tale elenco.

- **reparto_list.xml :** definisce il layout utilizzato per mostrare la lista di oggetti nella modalità "finestra singola", implementata sui telefoni e nell'utilizzo verticale del tablet, in cui la master list e l'area dei dettagli appaiono in schermi separati. Questo file non è altro che un oggetto di un RecyclerView, il quale verrà approfondito in seguito all'interno del capitolo, configurato per utilizzare il LinearLayoutManager. L'elemento RecyclerView dichiara che ogni oggetto della master list deve essere organizzato e mostrato come specificato nel file *reparto_list_content.xml*.

- **reparto_list.xml(w900dp) :** layout utilizzato per la master list inserita all'interno della modalità "two-pane", modalità in cui la master list e la sezione dettagliata appaiono uno a fianco all'altro (come in CartellaClinica). Questo file contiene un LinearLyout padre di tipo orizzontale, all'interno del quale risiede la master list, e un FrameLayout per contenere il contenuto della finestra per i dettagli.

- **reparto_content_list.xml** : contiene il layout utilizzato per organizzare i singoli elementi della lista. Di default si tratta di due oggetti di tipo TextView incorporati in un LinearLayout orizzontale, come utilizzato nell'applicazione, ma può essere modificato.

- **activity_reparto_detail.xml :** il layout di base usato per specificare la modalità di visualizzazione del contenuto della finestra con i dettagli della lista.

- **RepartoDetailActivity.java :** classe che carica il layout definito nel file

activity_reparto.detail.xml. Inoltre la classe inizializza e mostra il fragment contenente i dettagli definiti nei file *reparto_detail.xml* e *RepartoDetailFragment.java*.

- **RepartoDetailFragment.java :** è la classe di tipo Fragment responsabile per la visualizzazione e caricamento del file reparto_detail.xml, essa popola la finestra con il contenuto dei "dettagli". Il Fragment è inizializzato e mostrato dal file RepartoDetailActivity.java, il quale provvede anche a caricare il contenuto di activity_reparto_detail.xml per la modalità "single-pane" e il contenuto di reparto_list.xml per la modalità "two-pane".

- **DummyContent.java :** è una classe che serve a rappresentare i dati utilizzati all'interno del template. Questa classe può essere modificata o completamente rimpiazzata per incontrare le necessità dell'applicazione.

Per modificare i contenuti del template è quindi necessario modificare semplicemente gli attributi della classe *DummyContent.java*, nel nostro caso contenente i seguenti valori :

```
1  public static class DummyItem {
2          public final String id;
3          public final String reparto;
4          public final String numPazienti;
5
6          public DummyItem(String id, String reparto, String numPazienti) {
7              this.id = id;
8              this.reparto = reparto;
9              this.numPazienti = numPazienti;
10         }
11
12         @Override
13         public String toString() {
14             return reparto;
15         }
16     }
```

i valori scelti sono stati utili per rappresentare l'oggetto Reparto nella lista e per tenere traccia di quanti pazienti fossero presenti all'interno di ogni reparto.
All'interno della classe ItemReparto.java è stata definita la funzione

CaricaRepartiLogin(), la quale viene chiamata tutte le volte in cui l'utente apre la finestra composta dal Master/Detail Flow.

```
/**
 * numero pazienti viene contato grazie a una funzione nel database che rimanda
 *  quanti pazienti esistono per quel reparto
 */
public static void caricaRepartiLogin() {

    ITEMS.clear();
    ITEM_MAP.clear();

    //troviamo tutti i reparti per i quali lavora il medico attuale
    reparti =  MainActivity.db.trovaReparti(MainActivity.usernameLogin);


    for (int i = 0 ; i < reparti.size(); i++){

        //Grazie alla query so quanti pazienti sono presenti nel reparto selezionato
        int numeroPazienti = MainActivity.db.ritornaNumeroPazientiReparto(reparti.get(i))

        String indice = Integer.toString(i);
        String numPazientiString = Integer.toString(numeroPazienti);

        addItem(new DummyItem(indice, reparti.get(i), numPazientiString));
    }
}
```

Inizialmente viene svuotata la lista **ITEMS** di tipo DummyItem creata in precedenza per permettere alle possibili modifiche apportate dal medico di aggiornarsi tutte le volte che si accede alla pagina. La lista ITEMS è composta da tutti i reparti e i rispettivi pazienti attuali del medico.

All'interno della funzione caricaRepartiLogin() vengono eseguite due query :

- la prima è *trovaReparti(username),* la quale ritorna la lista di reparti attuali del medico loggato.
- La seconda è *ritornaNumeroPazientiReparto(reparto),* la quale restituisce,per ogni reparto scandito nel ciclo for, il numero di pazienti che vi sono all'interno.

Una volta ricavate queste informazioni viene estrapolata la posizione di ogni reparto nella lista e utilizzata come indice all'interno della *master list*. Dopodiché si crea un oggetto di tipo DummyItem, con parametri : indice, il reparto attuale e il numero di pazienti al suo interno.

Definita la *master list* vengono scanditi i pazienti all'interno di ogni elemento. E' stata

eseguita questa operazione all'interno del file RepartoDetailFragment.java nella funzione di creazione OnCreateView.

```
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                         Bundle savedInstanceState) {
    View rootView = inflater.inflate(R.layout.reparto_detail, container,  attachToRoot: false);

    List<String> pazienti = new ArrayList<~>();
    pazienti = MainActivity.db.trovaPazienti(RepartoListActivity.reparto);

    ListView listView;

    ArrayAdapter <String> adapter = new ArrayAdapter<~>(getContext(), android.R.layout.simple_list_item_1, pazienti);

    listView = (ListView) rootView.findViewById(R.id.reparti_detail);

    if (mItem != null) {
        listView.setAdapter(adapter);
    }
```

Una lista contenente i pazienti presenti nel reparto cliccato all'interno della master list viene salvata nella variabile, di tipo List<String>, *pazienti*. Dopodiché si inserisce la lista all'interno di un oggetto di tipo ArrayAdapter, il quale verrà settato nella listView corrispondente al layout reparti_detail. Gli Adapter servono per passare i dati all'interfaccia e organizzarli secondo una logica.

4.5.2 Operazioni di inserimento e eliminazione di Reparti e pazienti

Nella pagina appena descritta sono state implementate anche le operazioni di inserimento ed eliminazione di reparti e pazienti,

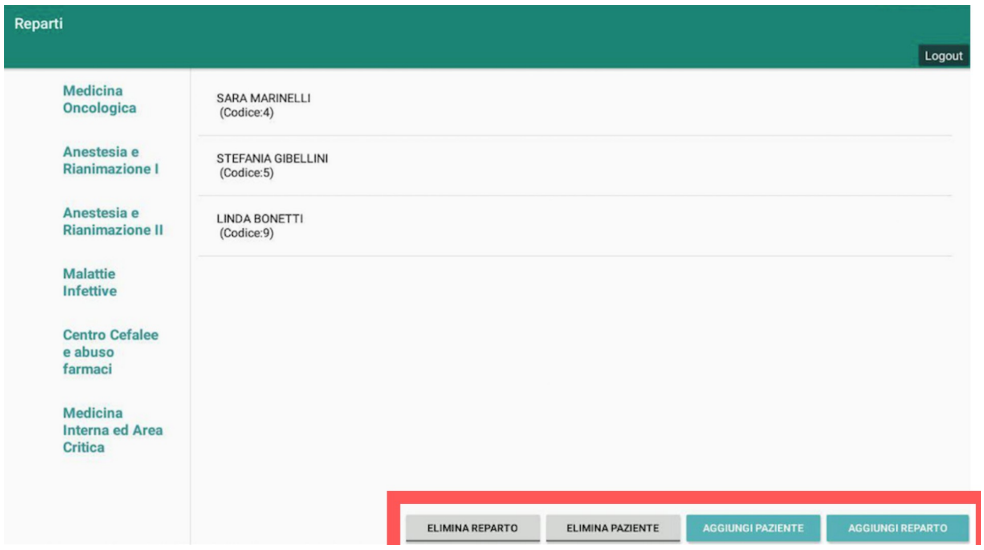

Figura 4.25 : Operazioni di modifica pazienti e reparti

L'implementazione di queste funzioni non verrà approfondita eccessivamente, poiché sono state realizzate semplicemente creando diversi menu a tendina composti dai reparti inseribili o eliminabili e i pazienti eliminabili, tra cui il medico può selezionare quello di suo interesse e attuare l'operazione. Verrà riportato in seguito, come esempio, la schermata che implementa l'eliminazione di un reparto :

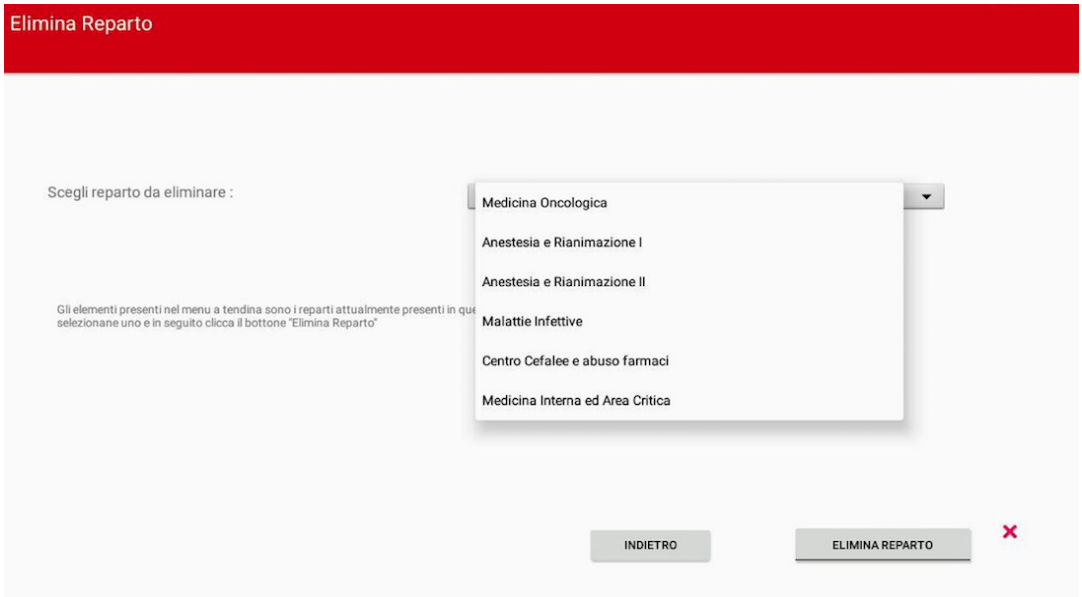

Figura 4.26 : Activity Eliminazione di un reparto

Gli elementi all'interno del menu sono solamente i reparti che il medico ha già inserito e che può quindi rimuovere dal proprio account.

Viene inoltre mostrata l'aggiunta di un paziente all'interno del sistema, attraverso un'Activity che permette al medico di inserire tutti i dati relativi a quest'ultimo.

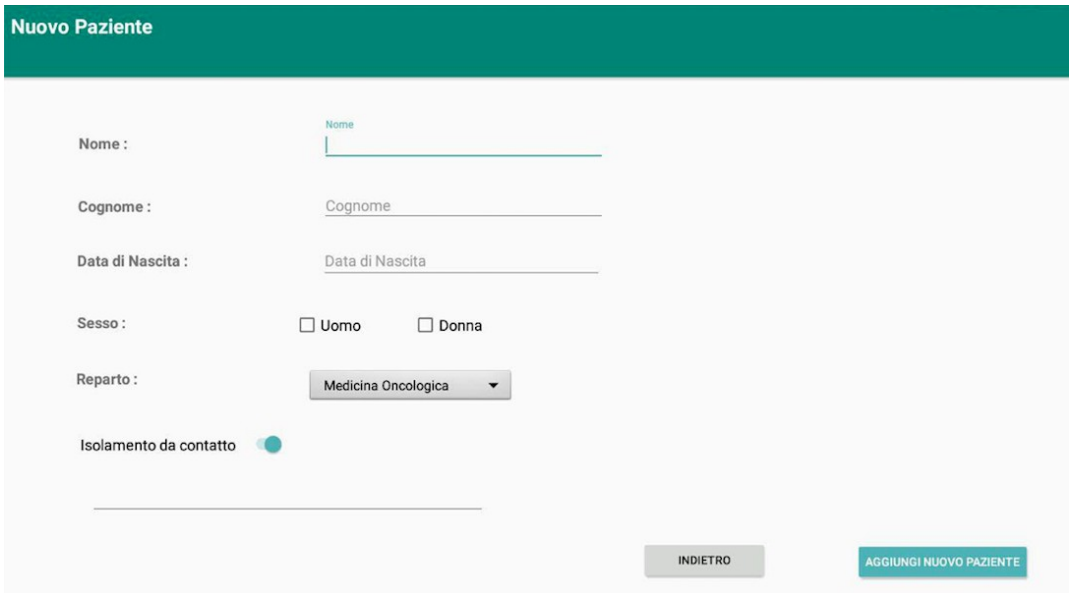

Figura 4.27 : Activity inserimento nuovo paziente all'interno del sistema

Dopo aver cliccato sul pulsante "Aggiungi nuovo paziente" e aver completato tutti i campi esso verrà inserito nel database ed associato a un codice id automatico dotato di autoincrementazione per essere rappresentato in maniera univoca. Tornando alla finestra dei reparti verrà richiamata la funzione caricaRepartiLogin(), grazie alla quale la lista di reparti viene eliminata e ricreata utilizzando i file aggiornati all'interno del database.

4.6 Visualizzazione profilo del paziente

Figura 4.28 : Schema generale con focus sulla fase di Visualizzazione del Profilo del Paziente

Selezionando un paziente ricoverato all'interno di un reparto il medico potrà accedere al suo profilo.

Il profilo dei pazienti è stato organizzato utilizzando una **Tabbed Activity**, la quale forma permette di annidare più pagine all'interno della stessa finestra facilitandone la navigazione. E' stato essenziale trovare un layout che potesse accogliere diverse finestre evitando di implementare un'Activity per ognuna di esse, poiché ogni paziente possiede 8 sezioni riguardanti i suoi dati anagrafici, parametri, allergie, consulenze, esami bioumorali, co-patologie, microbiolgia e terapie antibiotiche.

La pagina iniziale è riportata qui sotto nella figura 4.29, aperto il profilo del paziente il layout viene settato automaticamente sulla prima **Tab Dati**, all'interno della quale sono stati inseriti i dati anagrafici del paziente, alcune informazioni modificabili riguardanti un possibile isolamento da contatto, un grafico che illustra la temperatura corporea e una lista riassuntiva contenente i farmaci somministrati al paziente legati alle sue allergie se presenti.

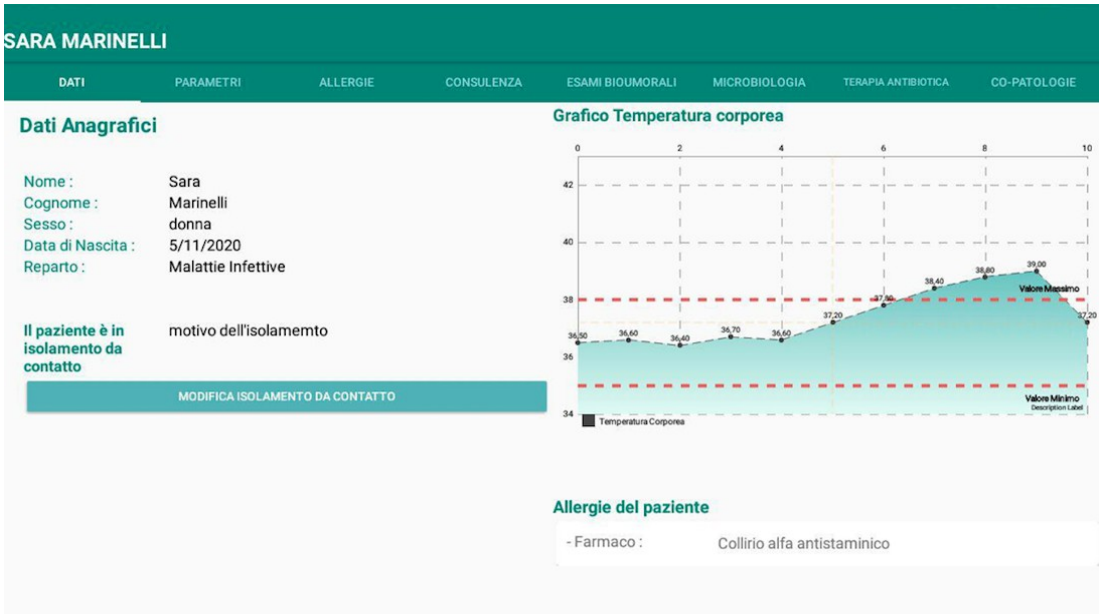

Figura 4.29 : Profilo del Paziente

Il caricamento dei dati relativi al paziente selezionato avviene grazie ad una variabile globale definita all'interno della funzione *onCreate(Bundle savedInstanceState)* dell'Activity ProfiloPaziente.java. Verrà approfondito in seguito il meccanismo che è stato utilizzato per mantenere il codice del paziente disponibile durante l'esecuzione delle diverse aree del profilo. E' essenziale avere sempre a disposizione il codice del paziente selezionato durante la navigazione all'interno del suo profilo per poterne caricare i dati.

4.6.1 Estrapolazione Codice Paziente

Per spiegare com'è stato gestito il codice riconoscitivo del paziente è necessario tornare indietro, all'interno del file *RepartoDetailFragment.java.*

Come è stato illustrato nel paragrafo 4.5.1, relativo all'approfondimento dei componenti dell'interfaccia Master/Detail Flow, all'interno di questo file vengono definiti i pazienti presenti nel reparto selezionato. Viene quindi creata una lista di nomi e caricata all'interno della finestra relativa ai dettagli. Ma come legare l'evento del Click sul paziente al caricamento del suo profilo personale?

Per fare ciò è stato definito, all'interno dell'evento *setOnItemClickListener* relativo alla *listView* dei pazienti, un semplice algoritmo che ha come scopo quello di estrapolare il codice del paziente che si trova in quella precisa posizione all'interno della lista.
Questa funzione viene quindi chiamata nel momento esatto in cui l'utente clicca su un paziente della lista .

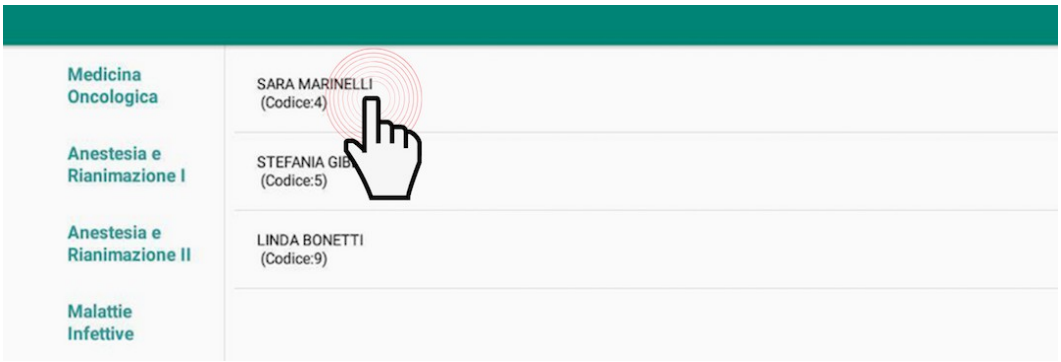

Figura 4.30 : Illustrazione che raffigura l'azione del click sulla lista dei pazienti

L'idea di base è quella di andare a prelevare il contenuto dell'oggetto della lista, in questo caso : **SARA MARINELLI (Codice:4)** e scandire la stringa fino a trovare il codice numerico del paziente, in questo caso : **4**. Il codice mostrato verrà commentato subito dopo :

```
listView.setOnItemClickListener((parent, view, position, id) → {
        String nome=(String)parent.getItemAtPosition(position);
        String codice=(String)parent.getItemAtPosition(position);

        Intent i = new Intent(getActivity(), ProfiloPaziente.class);

        int j = 0;

        for( ; j < nome.length(); j++){

            Character carattere = nome.charAt(j);

            if(carattere.toString().equals("(")){
                break;
            }
        }

        nome = nome.substring(0,j);

        int x = codice.length() - 1;

        for( ; x > 0 ; x--){
            Character carattere = codice.charAt(x);

            if(carattere.toString().equals(":")){
                x++;
                break;
            }
        }

        codice = codice.substring(x, codice.length()-2 );

        i.putExtra( name: "nomePaziente", nome);
        i.putExtra( name: "codicePaziente", codice);
        startActivity(i);
});
```

- La prima azione è quella di creare due stringhe : *nome* e *codice*
- La stringa denominata *nome* andrà a selezionare solamente il nome e il cognome del paziente (SARA MARINELLI), i quali verranno passati all'Activity del Profilo attraverso la funzione putExtra, come spiegato nel paragrafo introduttivo su AndroidStudio.
- La stringa *codice*, invece, conterrà solamente il codice numerico del paziente, il quale verrà utilizzato della query *trovaPaziente* per individuarlo all'interno del database. Il codice viene estrapolato grazie ad alcuni cicli for che scandiscono la stringa e ne prelevano solo la parte necessaria.
- Entrambe le stringhe vengono passate all'Activity *ProfiloPaziente*, all'interno della quale troveremo un'altra importante operazione.

Codice della classe ProfiloPaziente:

```
public class ProfiloPaziente extends AppCompatActivity {

    private TabLayout tabLayout;
    private AppBarLayout appBarLayout;
    private ViewPager viewPager;
    private TextView pazienteTV;
    public static String codice;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_profilo_paziente);
        tabLayout = (TabLayout) findViewById(R.id.tablayout_id);
        appBarLayout = (AppBarLayout) findViewById(R.id.appbarid);
        viewPager = (ViewPager) findViewById(R.id.viewpager_id);
        pazienteTV = (TextView) findViewById(R.id.pazienteTV);

        String nomePaziente =getIntent().getStringExtra( name: "nomePaziente");
        codice = getIntent().getStringExtra( name: "codicePaziente");

        pazienteTV.setText("  "+ nomePaziente);

        ViewPagerAdapter adapter = new ViewPagerAdapter(getSupportFragmentManager());
        adapter.AddFragment(new Tab1Dati(), Title: "Dati");
        adapter.AddFragment(new Tab2Parametri(), Title: "Parametri ");
        adapter.AddFragment(new Tab3Allergie(), Title: "Allergie");
        adapter.AddFragment(new Tab4Visita(), Title: "Consulenza");
        adapter.AddFragment(new Tab5EsamiBio(), Title: "Esami Bioumorali");
        adapter.AddFragment(new Tab6Microbiologia(), Title: "Microbiologia");
        adapter.AddFragment(new Tab7Antibiotica(), Title: "Terapia Antibiotica");
        adapter.AddFragment(new Tab8CoPatologie(), Title: "Co-Patologie");


        viewPager.setAdapter(adapter);
        tabLayout.setupWithViewPager(viewPager);

    }
```

Passando alla lettura del codice nella classe ProfiloPaziente si noterà che le variabili precedentemente create (*codice* e *paziente*) vengono prelevate dall'Intent e salvate in nuove variabili.

Il passaggio cruciale per il salvataggio dell'id del paziente è che la variabile *codice* è stata definita di tipo **public static**, sarà quindi possibile accedervi e modificarla da tutti i file.

Rendendo pubblica questa variabile essa può essere utilizzata per eseguire query all'interno di tutti i fragment che compongono il profilo del paziente. La variabile viene aggiornata tutte le volte in cui il medico esce dal profilo di un paziente e ne seleziona uno nuovo. Quindi il codice rimane valido e coerente durante tutta la navigazione all'interno del profilo.

Figura 4.31 : Estrapolazione del codice paziente

4.7 Struttura dell'Activity ProfiloPaziente

L'organizzazione dei componenti dell'Activity ProfiloPaziente ha richiesto un'ampia analisi delle possibilità messe a disposizione da AndroidStudio per sfruttare al meglio le potenzialità di questo IDE. Come per il Master/Detail flow ci si è affidati a una struttura di layout composta da più layout e molto utilizzata negli applicativi mobili Android : la **Tabbed Activity**.

Una **Tabbed Activity** implementa un layout a scorrimento, il quale consente di

spostarsi tra schermate di pari livello con un gesto orizzontale del dito o uno scorrimento laterale. Questo modello di navigazione viene anche definito “paginazione orizzontale” per il movimento compiuto dalle schede.

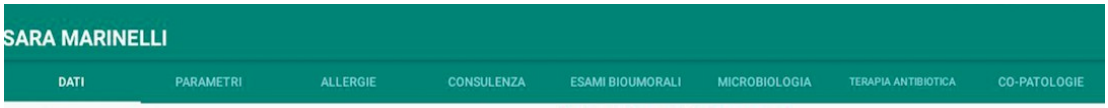

Figura 4.32 : Tabbed Layout

Ognuna delle tabs create è legata ad una scheda contenente dati di diverso tipo e scopo. Attraverso questa impaginazione l'utente potrà scorrerle velocemente e intuitivamente.

4.7.1 Creazione Tab View

Per comporre un'Activity contenente una Tabbed Bar come appena descritto è necessario, inizialmente, creare un semplice template di tipo “Tabbed Activity”.

Tabbed Activity

Nel momento della sua creazione bisogna specificarne lo Stile di Navigazione, il quale sarà “Action Bar Tabs (with ViewPager)” per poter supportare l'integrazione delle tabs.

All'interno del file creato da AndroidStudio si possono trovare sin da subito molti oggetti e funzioni tra cui : onCreate, onCreateOptionsMenu, onOptionSelectedItem etc. . Troviamo inoltre una inner static class di tipo placeholder contenente una semplice view, un palceholder fornisce un oggetto virtuale attraverso il quale si può posizionare un oggetto esistente.

Le schede trattate in questo layout vengono chiamate Fragments, ogni Fragment implementa una finestra indipendente dalle altre, con i propri attributi, dati e layout. Esse vanno quindi create e trattate separatamente.

Le classi sono create inizialmente come classi java di default, viene però impostata la superclasse a Fragment. Per ogni Fragment viene generato un file .xml per implementarne l'interfaccia.

All'interno della tabbed activity, ovvero ProfiloPaziente.java, vengono aggiunti i fragment uno ad uno, specificandone i titoli che verranno inseriti nella grafica.

```
    setContentView(R.layout.activity_profilo_paziente);
    tabLayout = (TabLayout) findViewById(R.id.tablayout_id);
    appBarLayout = (AppBarLayout) findViewById(R.id.appbarid);
    viewPager = (ViewPager) findViewById(R.id.viewpager_id);
    pazienteTV = (TextView) findViewById(R.id.pazienteTV);

    String nomePaziente =getIntent().getStringExtra( name: "nomePaziente");
    codice = getIntent().getStringExtra( name: "codicePaziente");

    pazienteTV.setText("  "+ nomePaziente);

    ViewPagerAdapter adapter = new ViewPagerAdapter(getSupportFragmentManager());
    adapter.AddFragment(new Tab1Dati(), Title: "Dati");
    adapter.AddFragment(new Tab2Parametri(), Title: "Parametri ");
    adapter.AddFragment(new Tab3Allergie(), Title: "Allergie");
    adapter.AddFragment(new Tab4Visita(), Title: "Consulenza");
    adapter.AddFragment(new Tab5EsamiBio(), Title: "Esami Bioumorali");
    adapter.AddFragment(new Tab6Microbiologia(), Title: "Microbiologia");
    adapter.AddFragment(new Tab7Antibiotica(), Title: "Terapia Antibiotica");
    adapter.AddFragment(new Tab8CoPatologie(), Title: "Co-Patologie");


    viewPager.setAdapter(adapter);
    tabLayout.setupWithViewPager(viewPager);

}
```

I Fragment vengono aggiunti all'adapter grazie alla funzione AddFragment, definita nella classe ViewPagerAdapter.

```
@Override
public CharSequence getPageTitle(int position) { return FragmentListTitles.get(position); }

public void AddFragment(Fragment fragment, String Title){
    fragmentList.add(fragment);
    FragmentListTitles.add(Title);
}
```

All'interno della quale vengono passati i parametri fragment, riferito al fragment da inserire nella tab, e Title, una stringa rappresentante il titolo della tab.
Il risultato di queste operazioni sarà una tabbed bar avente 8 tabs, ognuna delle quali distinguibili dal proprio titolo, e legate rispettivamente al layout caricato all'interno

della classe del fragment. L'utente potrà scorrere le schede attraverso un movimento orizzantale o cliccando sui loro titoli.

4.7.2 Fragments

I Fragment rappresentano una porzione di interfaccia grafica in un'Activity di tipo FragmentActivity. Si possono inserire più fragment all'interno della stessa Activity per creare un'interfaccia utente a più riquadri.
I Fragments hanno un ciclo di vita a sé stante, ricevono i propri eventi input e possono essere aggiunte o rimosse mentre l'Activity è in esecuzione, possono essere definite anche come “Activity secondarie”.

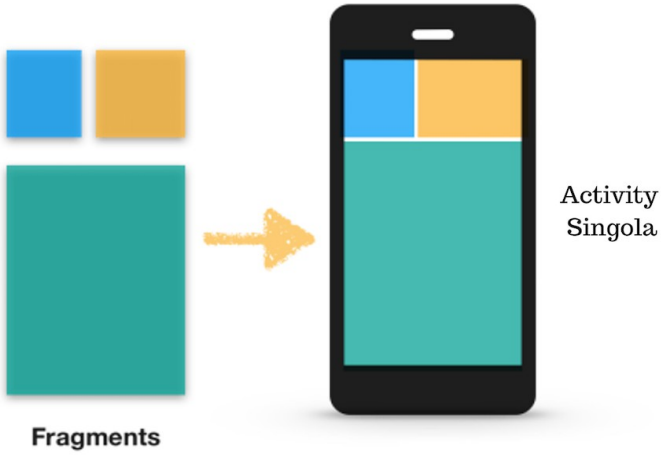

Figura 4.33 : Activity composta da più Fragments

Il ciclo di vita di un Fragment dipende direttamente dal ciclo di vita dell'Activity che lo contiene, quindi quando quest'ultima è in pausa anche i suoi fragment lo saranno e quando l'Activity verrà distrutta succederà lo stesso per i fragment al suo interno.
Tuttavia, durante l'esecuzione dell'Activity, è possibile interfacciarsi e manipolare ogni fragment in maniera indipendente.

I Fragments sono stati introdotti in Android 3.0 (API level 11) principalmente per supportare delle interfacce più dinamiche e flessibili per gli schermi grandi, come per esempio quelli dei tablet. Lo schermo di un tablet è molto spazioso e dà quindi la possibilità di combinare e organizzare più componenti in una singola finestra. Dividendo il layout di un'Activity in fragments è possibile modificare l'aspetto dell'interfaccia durante l'esecuzione dell'applicazione e mantenere possibili cambiamenti in un “back stack” gestito dall'Activity.

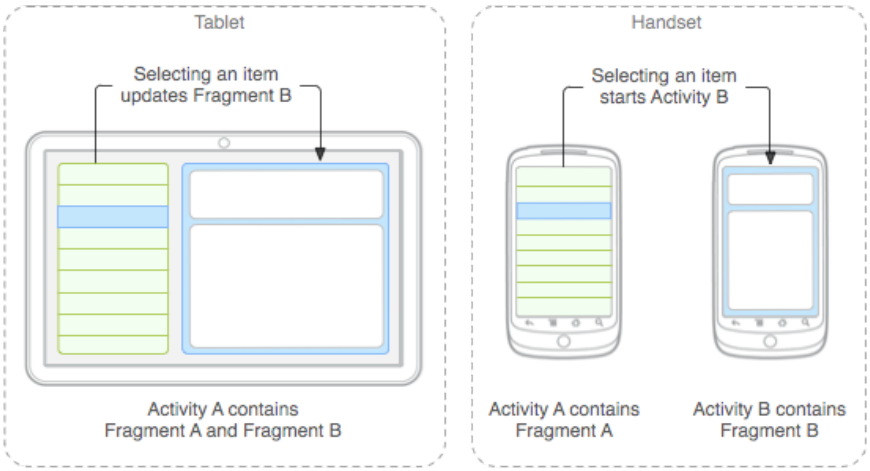

Figura 4.34 : Dinamicità e flessibilità dell'interfaccia grafica all'interno di un Tablet grazie ai Fragments

4.7.3 Creazione di un Fragment e il suo ciclo di vita

Per creare un Fragment bisogna, innanzitutto, creare una sottoclasse di tipo Fragment. Il codice di un Fragment è molto simile a quello di un'Activity, dal momento in cui contiene dei metodi di callback come onCreate(), onStart(), onPaure() e onStop().

- **OnCreate()** : il sistema chiama questa funzione durante la creazione del Fragment. Momento in cui è necessario inizializzare i componenti essenziali del fragment che si vogliono conservare quando il fragment è in pausa o in fase di stop.
- **OnCreateView()** : metodo chiamato nel momento in cui il fragment deve disegnare la sua interfaccia per la prima volta

- **onPause()** : onPause viene chiamato al primo segnale di chiusura del fragment.

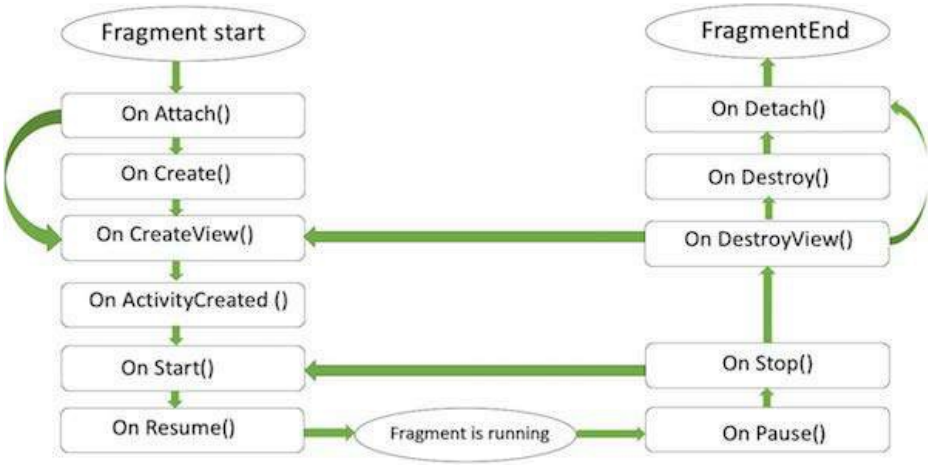

Figura 4.35 : Ciclo di Vita di un Fragment

4.7.4 Implementazione dei Fragments in CartellaClinica

Nel momento della creazione del primo fragment, Tab1Dati, dedicato ai dati anagrafici del paziente, la sua temperatura corporea e le sue allergie, è stato riscontrato un problema.

```
@Override
public View onCreateView(LayoutInflater inflater,
                         ViewGroup container, Bundle savedInstanceState) {
    paziente = MainActivity.db.ritornaPaziente(ProfiloPaziente.codice);


    return inflater.inflate(R.layout.tab1dati,
            container,  attachToRoot: false);



}
```

Definito il layout del fragment ed associato alla classe corrispondente risultava impossibile individuare i componenti dell'interfaccia grafica all'interno della funzione onCreateView. Non era quindi possibile regolarne il comportamento, modificare il contenuto dei componenti e gestirne gli eventi.

E' stato, tuttavia, individuato un metodo per superare questo ostacolo. I componenti non venivano riconosciuti poiché legati al ciclo di vita dell'Activity padre ProfiloPaziente, all'interno della quale si trova il fragment in questione. Era quindi impossibile accedervi in maniera diretta dalla classe del fragment.

Per sormontare questo ostacolo è stata implementata la funzione di callBack onStart(), la quale viene chiamata quando il fragment diventa visibile, momento in cui è effettivamente possibile accedere ai componenti della grafica poiché creati e pronti ad essere utilizzati.

```
@Override
public void onStart(){
    super.onStart();
    Activity a = (Activity)(getContext());

    isoBtn = a.findViewById(R.id.isolamentoBtn);
    modificaMotivazione = a.findViewById(R.id.isolamentoLayout);
    modificaMotivazione.setVisibility(View.INVISIBLE);
    contatto = a.findViewById(R.id.switchContatto);
    newMotivazione = a.findViewById(R.id.newMotivazione);
    motivazioneTV = a.findViewById(R.id.motivazioneTV);
    chartTemp = a.findViewById(R.id.chartTemperature);


    slvBtn = a.findViewById(R.id.salvaBtn);
```

All'inizio della funzione è stato prelevato il context dell'Activity padre. Il context è, come suggerisce il nome, il contesto dello stato corrente dell'applicazione o dell'oggetto preso in considerazione. Permette agli oggetti appena creati di capire cosa sta succedendo, viene in genere chiamato per ottenere informazioni su un'altra parte del programma (Activity, pacchetti o applicazioni). In questo caso viene quindi salvato all'interno della funzione a il contesto dell'Activity padre ProfiloPaziente. Il quale ci permetterà di andare a individuare e in seguito utilizzare i componenti della grafica. Come si può leggere nel codice riportato la funzione *findViewById* viene chiamata sull'oggetto di tipo Activity a.

4.8 Introduzione Tabs ProfiloPaziente.java

Figura 4.36 : Schema generale con focus sulle Tabs del profilo del paziente

Nei prossimi paragrafi verranno descritte le tab implementate all'interno del profilo del paziente. Esse hanno tutte la stessa struttura, poiché devono permettere al medico di inserire i dati in una determinata sezione e avere a disposizione lo storico di questi ultimi. Come look and feel della scheda si è pensato di dividerla verticalmente a metà ed implementare le operazioni appena descritte all'interno delle sue sezioni risultanti.

Per dividere il fragment a metà è stato creato un *LinearLayout orizzontale*, grazie al quale è stato possibile posizionare, dividendo precisamente a metà lo schermo, una ScrollView a sinistra e un RelativeLayout a destra.

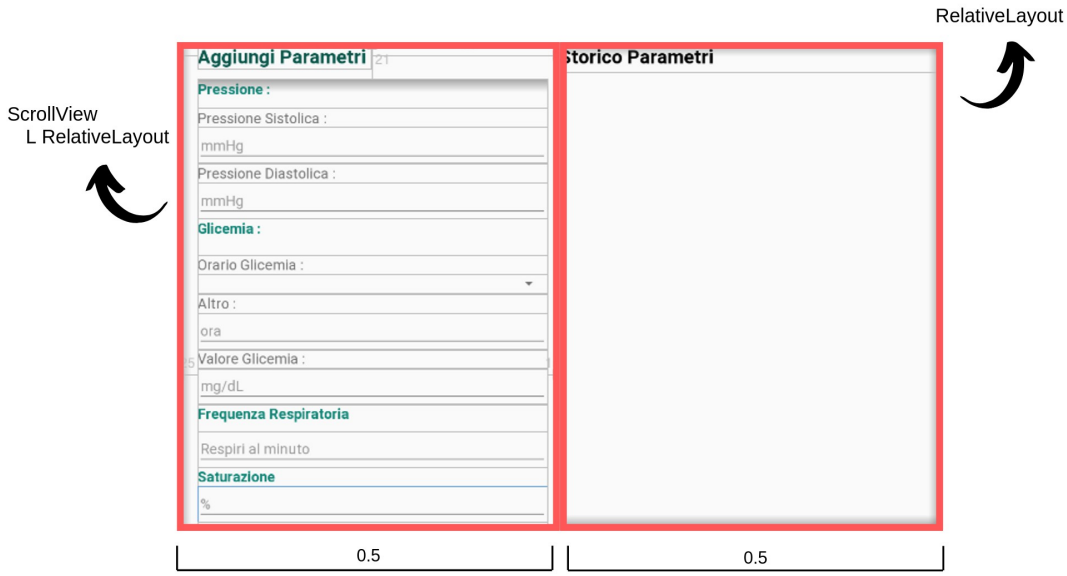

Figura 4.37 : Divisione dell'interfaccia delle schede del profilo del paziente

La sezione di sinistra è composta da una **ScrollView** e un **RelativeLayout** al suo interno. E' stato necessario rendere scorrevole il layout poiché la lista di dati da inserire è lunga e non sarebbe possibile posizionare tutte le label e le EditText all'interno della schermata statica.

A destra invece, all'interno del RelativeLayout, è stato inserito un **RecyclerView** che potesse contenere una lista dei dati inseriti in precedenza. Un **RecyclerView** permette di creare una lista di elementi contenente un sottoinsieme di informazioni che la caratterizzano, il ruolo e il funzionamento di questo componente verranno approfonditi nel paragrafo successivo.

4.9 Cos'è un RecyclerView e come viene utilizzato

Il **RecyclerView** è stato un componente chiave per lo sviluppo di CartellaClinica, ha infatti permesso di creare liste di dati di ogni tipo in maniera semplice e intuitiva. E' stato principalmente utilizzato, come già spiegato nel paragrafo precedente, per implementare le liste di valori inseriti precedentemente nel sistema, ovvero gli storici dei diversi pazienti.
RecyclerView dà la possibilità di creare una lista di oggetti aventi dettagli che la caratterizzano e la distinguono (come foto, titolo, descrizioni, date etc.). I dati inseriti nella lista vengono ottenuti dal database locale e caricati nell'interfaccia.

Nel momento della creazione della lista viene definito un insieme di *View* che verranno poi popolate con le informazioni e mostrate all'interno della *viewport,* ossia la porzione di schermo visibile all'utente. L'applicazione deve creare il layout per tutti gli elementi della lista, quindi anche quelli non visibili nella schermata, e deve memorizzare ogni elemento per poterlo mostrare in un secondo momento senza doverlo ricaricare.
Quando le liste create sono di grandi dimensioni questo metodo diventa obsoleto, poiché un grande insieme di elementi in una sola view porta ad una rapida *saturazione della memoria*.
E' qui che entra in gioco il **RecyclerView**, il quale mantiene in una coda, chiamata

recycler bin (cestino per il riciclaggio), alcuni degli elementi precedentemente visualizzati per riutilizzarli in un secondo momento. Durante lo scorrimento della lista il RecyclerView recupererà dalla coda un elemento e lo popolerà con le nuove informazioni da mostrare all'utente.

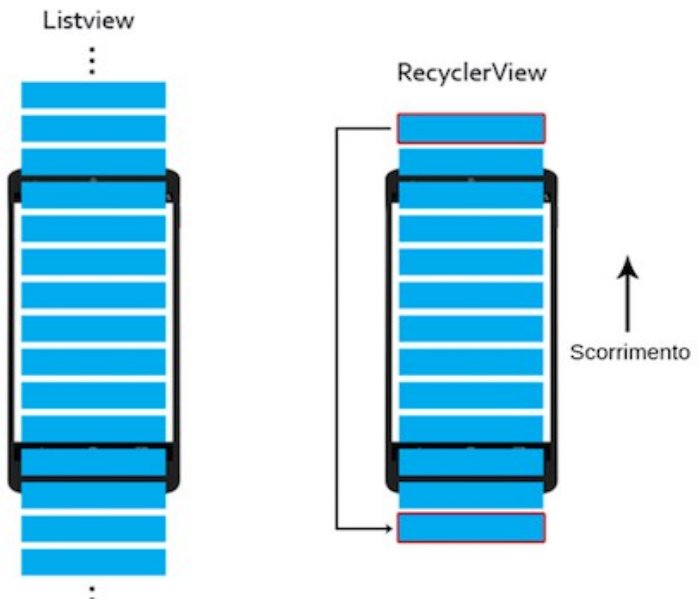

Figura 4.38 : Gestione dei layout degli elementi con ListView o RecyclerView

I vantaggi portati dal meccanismo di riciclo hanno fatto si che il RecyclerView si affermasse come soluzione principale per la creazione delle liste, sostituendo interamente l'utilizzo delle ListView.
Vengono elencati i pro e i contro dell'utilizzo del RecyclerView :

Pro	Riciclo delle viste attraverso l'utilizzo del pattern ViewHolder
	Supporto di liste orizzontali, verticali e sfalsate
	Support per lo scroll orizzontale e verticale
	Miglioramenti in termini di performance e occupazione di memoria, in quanto non è necessario creare il layout da popolare con le informazioni del data source per via del riutilizzo

	Integrazioni di animazioni per aggiungere, aggiornare e rimuovere oggetti
Contro	Aumento della complessità Mancanza di un metodo nativo per intercettare il click su un elemento della lista

La modificabilità e la possibilità di riutilizzazione del RecyclerView sono dovuti alla sua modularità, che permette di gestire ogni sua parte in maniera indipendente e dettagliata.

Gli elementi che compongono il RecyclerView sono quattro : **Adapter**, **ViewHolder**, **Layout Manager** e **DataSource.** Essi verranno spiegati nella seguente tabella:

Componente	Tipologia	Descrizione
Adapter	RecyclerView.Adapter	Estrae i dati dal database e li utilizza per creare e popolare i ViewHolder. Questi ultimi saranno poi inviati al Layout Manager del RecyclerView.Adapter
ViewHolder	RecyclerView.ViewHolder	E' il ponte che unisce il RecyclerView e l'Adapter, permette la riduzione nel numero di view da creare. Questo oggetto infatti fornisce il layout da popolare con i dati presenti nel database e viene riutilizzato dal RecyclerView per ridurre il numero di layout da creare.
Layout Manager	RecyclerView.LayoutManager	Crea e posiziona le view all'interno del RecyclerView.
DataSource	List	E' l'insieme di dati utilizzato per popolare la lista tramite l'Adapter.

Viene riportato un semplice schema di comunicazione tra i componenti coinvolti nella creazione di una lista tramite il RecyclerView

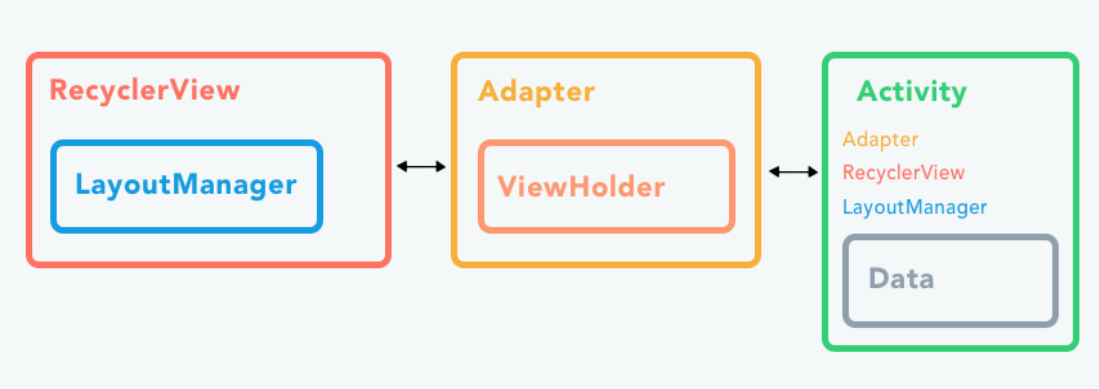

Figura 4.39: Schema della comunicazione tra componenti per la creazione di una lista tramite il RecyclerView

4.9.1 Modifica e creazione del Layout

Per realizzare una lista con Recycler View bisogna, innanzitutto, posizionare graficamente il componente *RecyclerView* all'interno dell'Activity in cui si desidera integrarlo. Viene quindi aperto il layout della finestra in questione e viene inserito il componente come segue :

```
1 <android.support.v7.widget.RecyclerView
2         android:id="@+id/rv_colored"
3         android:layout_width="match_parent"
4         android:layout_height="match_parent"/>
```

Viene riportato come esempio lo screenshot della tab Parametri, dove il RecyclerView deve essere posizionato nella porzione a destra all'interno del fragment :

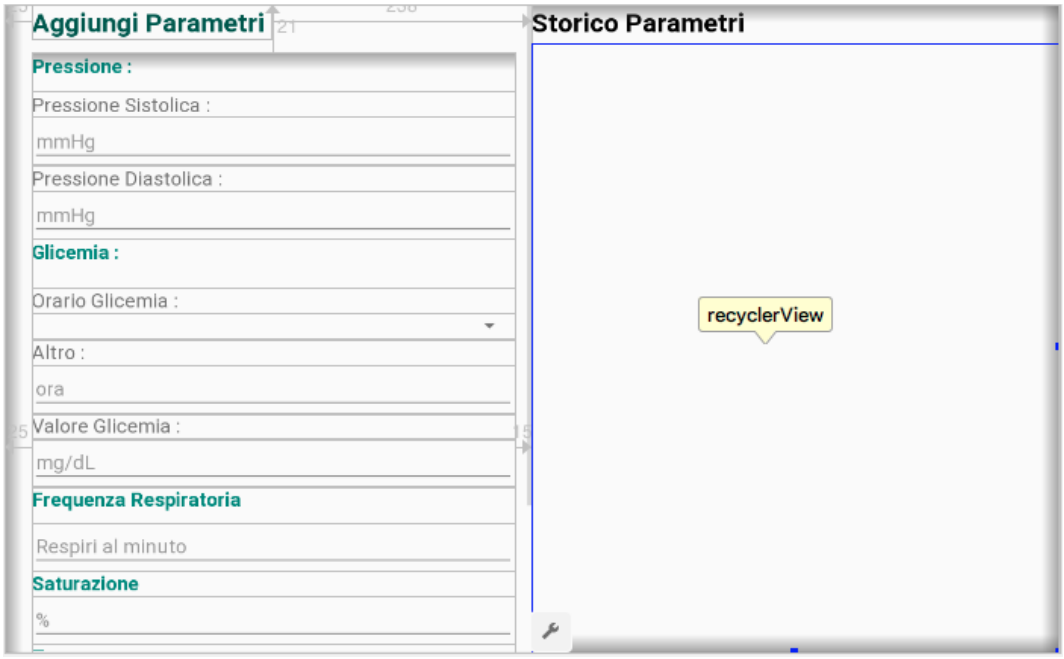

Figura 4.40 : Posizionamento recyclerView nella tab Parametri

Il secondo step è la creazione di un nuovo layout, il quale prenderà il nome di *list_item.xml*. Il list_item delinea la struttura e il contenuto di un singolo elemento della lista, il quale verrà utilizzato dal *ViewHolder* e popolato a run-time dall'*Adapter*.
Nel caso del RecyclerView della tab Parametri l'elemento list_item è stato creato con le seguenti caratteristiche :

- Ogni elemento della lista ha come titolo la stringa “Data Parametri”, seguita da una textView la quale verrà riempita con la data dell'esame corrispondente
- Subito dopo il titolo si trova una lista di label che contrassegnano i nomi dei valori calcolati in questo tipo di esame. Di fianco al titolo (es. Pressione Sistolica :) si trova una textView che viene aggiornata col valore di quel preciso parametro.
- Viene modificata la visibilità degli asterischi rossi in base al risultato dell'esame: se quest'ultimo non rientra nel range di “normalità” l'asterisco sarà impostato come “visibile” per segnalare un valore sballato.

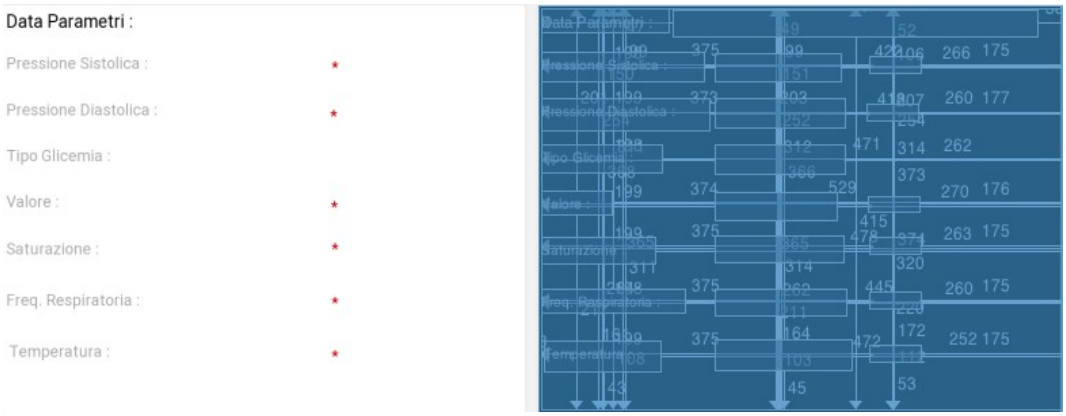

Figura 4.41 : Struttura dell'elemento item_list del RecyclerView all'interno della tab Parametri

4.9.2 Implementazione dell'Adapter e del ViewHolder

Dopo aver disegnato il layout per gli elementi della lista viene creata la classe di tipo **Adapter** e il suo relativo **ViewHolder**. L'Adapter prenderà in maniera parziale il nome dell'oggetto che verrà trattato per essere individuata più facilmente, ogni RecyclerView ha infatti *il proprio Adapter* per popolare la lista con i dati giusti.

Nel caso del RecyclerView della tab Parametri è stato denominato "**ParametriAdapter**" il quale, prima di essere implementato, deve attendere la creazione della inner classe **ParametriViewHolder**.

Un oggetto di tipo RecyclerView.Adapter crea delle views per gli items e sostituisce il contenuto di alcune viste con nuovi dati quando l'item iniziale non è più visibile.

La classe Adapter dispone di tre metodi essenziali : **onCreateViewHolder, onBindViewHolder** e **getItemCount,** i quali vengono implementati in base alle esigenze del RecyclerView.

All'interno del **ViewHolder** si determinano i componenti del layout che accoglieranno i

dati, dopodiché nel costruttore del ViewHolder le variabili verranno inizializzate grazie al parametro passato alla funzione (itemView), sul quale potremo chiamare il metodo *findViewById*.

```
//chiave di volta tra il RecyclerView e l'adapter e permette la riduzione del numero di View
//da creare. Questo oggetto fornisce il layout da popolare con i dati presenti nel DataSource
//e viene riutilizzato dal RecyclerView per ridurre il numero di layout da creare per
//popolare la lista
public static class ParametriViewHolder extends RecyclerView.ViewHolder{
    public TextView data, pressSistolica, pressDiastolica, tipoGlicemia, valGlicemia,
            saturazione, frequenza, temperatura, astPressDia, astPressSis, astGlicemia,
            astSatu, astFreq, astTemp;

    public ParametriViewHolder(View itemView, final OnItemClickListener listener){
        super(itemView);
        data = itemView.findViewById(R.id.dataItem);
        pressSistolica = itemView.findViewById(R.id.pressioneSistolicaItem);
        pressDiastolica = itemView.findViewById(R.id.pressioneDiastolicaItem);
        tipoGlicemia = itemView.findViewById(R.id.tipoGlicemiaItem);
        valGlicemia = itemView.findViewById(R.id.valoreGlicemiaItem);
        saturazione = itemView.findViewById(R.id.saturazioneItem);
        frequenza = itemView.findViewById(R.id.frequenzaItem);
        temperatura = itemView.findViewById(R.id.temperaturaItem);
        astPressDia = itemView.findViewById(R.id.astPressDia);
        astPressSis = itemView.findViewById(R.id.astPressioneSis);
        astGlicemia = itemView.findViewById(R.id.astGlicemia);
        astSatu = itemView.findViewById(R.id.astSaturazione);
        astFreq = itemView.findViewById(R.id.astFreqResp);
        astTemp = itemView.findViewById(R.id.astTemp);

        itemView.setOnClickListener((v) → {
              if (listener != null){
                  int position = getAdapterPosition();
                  if(position != RecyclerView.NO_POSITION){
                      listener.onItemClick(position);

                  }
              }
        });
    }
}
```

All'interno della funzione *ParametriViewHolder* è stato implementato il listener .**setOnClickListener**, il quale rende ogni elemento della lista cliccabile e ne gestisce le conseguenze dell'interazione con esso, verrà approfondito in seguito il funzionamento e lo scopo di questo comportamento.

Nel metodo **OnCreateViewHolder** si recupera, grazie al metodo Inflate, il layout che era stato costruito come struttura di un oggetto base, item_parametri. Dopo aver ricavato la View potremo costruire il nostro ViewHolder e utilizzarlo come valore di

ritorno nella funzione.\l

```
@Override
public ParametriViewHolder onCreateViewHolder(@NonNull ViewGroup viewGroup, int i) {
    View v = LayoutInflater.from(viewGroup.getContext()).inflate(R.layout.parametri_item, viewGroup,  attachToRoot: false);
    ParametriViewHolder pvh = new ParametriViewHolder(v, mListener);
    return pvh;
}
```

Infine, per quanto riguarda il popolamento delle variabili all'interno del layout, viene implementato il metodo **OnBindViewHolder**, al quale viene passato un oggetto di tipo Parametri contenente tutti i dati necessari.

```
@Override
public void onBindViewHolder(ParametriViewHolder parametriViewHolder, int i) {
    Parametri parametroCorrente = mParametriList.get(i);

    parametriViewHolder.astPressDia.setVisibility(View.INVISIBLE);
    parametriViewHolder.astPressSis.setVisibility(View.INVISIBLE);
    parametriViewHolder.astGlicemia.setVisibility(View.INVISIBLE);
    parametriViewHolder.astSatu.setVisibility(View.INVISIBLE);
    parametriViewHolder.astFreq.setVisibility(View.INVISIBLE);
    parametriViewHolder.astTemp.setVisibility(View.INVISIBLE);

    parametriViewHolder.
            data.setText(parametroCorrente.getData());
    parametriViewHolder.
            pressSistolica.setText(parametroCorrente.getPressioneSistolica() + " mmHg");
    parametriViewHolder.
            pressDiastolica.setText(parametroCorrente.getPressioneDiastolica()+ " mmHg");
    parametriViewHolder.
            tipoGlicemia.setText(parametroCorrente.getTipoGlicemia());
    parametriViewHolder.
            valGlicemia.setText(parametroCorrente.getValoreGlicemia()+ " mg/dL");
    parametriViewHolder.
            saturazione.setText(parametroCorrente.getSaturazione() +  "%");
    parametriViewHolder.
            frequenza.setText(parametroCorrente.getFrequenzaRespiratoria() + " respiri al minuto");
    parametriViewHolder.
            temperatura.setText(parametroCorrente.getTemperatura() + "°C");

    Double pressSis = Double.parseDouble(parametroCorrente.getPressioneSistolica());
    Double pressDia = Double.parseDouble(parametroCorrente.getPressioneDiastolica());
    Double glice = Double.parseDouble(parametroCorrente.getValoreGlicemia());
    Double satu = Double.parseDouble(parametroCorrente.getSaturazione());
    Double freq = Double.parseDouble(parametroCorrente.getFrequenzaRespiratoria());
    Double temp = Double.parseDouble(parametroCorrente.getTemperatura());


    if(pressSis >= 150.0 || pressSis <= 60.0){
        parametriViewHolder.astPressSis.setVisibility(View.VISIBLE);
    }
```

Come mostrato nella porzione di codice vengono inizialmente settati a “INVISIBLE” tutti gli asterischi relativi ai valori dell'esame, poiché la loro visibilità verrà modificata dopo aver effettuato dei controlli sul valore numerico dell'esame, se esso non rientra nel range il suo asterisco verrà reso visibile.

Le TextView del layout parametri_item come : *data, presSistolica, pressDiastolica,*

tipoGlicemia, valGlicemia etc., definiti precedentemente all'interno della funzione ParametriViewHolder, vengono popolate utilizzando i dati dell'oggetto di tipo Parametro corrente, inizializzato nel costruttore della classe :

```
public ParametriAdapter(ArrayList<Parametri> parametriList){
    mParametriList = parametriList;
}
```

Nella classe di tipo Fragment rappresentante la Tab in questione vengono definite e in seguito inizializzate tre variabili private: un RecyclerView, un Adapter (del tipo di dato in questione, in questo caso ParametriAdapter) e infine un RecyclerView.LayoutManager :

```
private RecyclerView mRecyclerView;
private ParametriAdapter mAdapter;
private RecyclerView.LayoutManager mLayoutManager;
```

Esse inizializzeranno gli oggetti appena descritti per realizzare la creazione e il popolamento degli oggetti della lista. La lista verrà creata, ovviamente, all'interno dello spazio dedicato al RecyclerView posizionato all'interno del fragment come mostrato nella figura 4.40.

```
//elementi per mRecyclerView
mRecyclerView = a.findViewById(R.id.recyclerView);
mRecyclerView.setHasFixedSize(true);
mLayoutManager = new LinearLayoutManager(a);
//creo l'oggetto adapter e gli passo la lista di oggetti che voglio mostrare
mAdapter = new ParametriAdapter(parametriList);

mRecyclerView.setLayoutManager(mLayoutManager);
mRecyclerView.setAdapter(mAdapter);
```

Come spiegato nella descrizione della classe Adapter viene passata la lista di elementi contenenti i dati utili al costruttore del ParametriAdapter, il quale si occuperà di caricare tutti i dati all'interno degli elementi della lista.

4.9.3 Gestione del clic su un oggetto della lista

Per permettere al medico di avere il completo controllo sui dati relativi ai pazienti è stata implementata la possibilità di eliminare oggetti all'interno degli storici contenenti dati precedentemente inseriti. Più specificatamente, è possibile selezionare un oggetto in un RecyclerView, individuarne l'id ed eliminarlo dal database, aggiornando anche il contenuto della lista.

Il RecyclerView, diversamente dal ListView, non offre un metodo nativo per intercettare il click su un elemento della lista, per aggiungere tale comportamento è necessario compiere alcuni passi aggiuntivi.

All'interno della classe ParametriAdapter viene creata l'interfaccia *ItemClickListener*, all'interno della quale viene definita una variabile statica per il listener ed è pronta a ricevere il messaggio di click:

```
public class ParametriAdapter extends RecyclerView.Adapter<ParametriAdapter.ParametriViewHolder> {
    private ArrayList<Parametri> mParametriList;
    private OnItemClickListener mListener;
```

In seguito si modifica il costruttore della classe *ParametriViewHolder* aggiungendo il listener, il quale verrà implementato all'interno della funzione. Ogni volta in cui l'utente cliccherà su un elemento della lista il metodo onClick verrà invocato.

```
public static class ParametriViewHolder extends RecyclerView.ViewHolder{
    public TextView data, pressSistolica, pressDiastolica, tipoGlicemia, valGlicemia,
            saturazione, frequenza, temperatura, astPressDia, astPressSis, astGlicemia,
            astSatu, astFreq, astTemp;

    public ParametriViewHolder(View itemView, final OnItemClickListener listener){
        super(itemView);
        data = itemView.findViewById(R.id.dataItem);
        pressSistolica = itemView.findViewById(R.id.pressioneSistolicaItem);
        pressDiastolica = itemView.findViewById(R.id.pressioneDiastolicaItem);
        tipoGlicemia = itemView.findViewById(R.id.tipoGlicemiaItem);
        valGlicemia = itemView.findViewById(R.id.valoreGlicemiaItem);
        saturazione = itemView.findViewById(R.id.saturazioneItem);
        frequenza = itemView.findViewById(R.id.frequenzaItem);
        temperatura = itemView.findViewById(R.id.temperaturaItem);
        astPressDia = itemView.findViewById(R.id.astPressDia);
        astPressSis = itemView.findViewById(R.id.astPressioneSis);
        astGlicemia = itemView.findViewById(R.id.astGlicemia);
        astSatu = itemView.findViewById(R.id.astSaturazione);
        astFreq = itemView.findViewById(R.id.astFreqResp);
        astTemp = itemView.findViewById(R.id.astTemp);

        itemView.setOnClickListener((v) → {
            if (listener != null){
                int position = getAdapterPosition();
                if(position != RecyclerView.NO_POSITION){
                    listener.onItemClick(position);

                }
            }
        });
    }
```

All'interno dell'Activity principale, in questo caso Tab2Parametri, viene invocato il metodo setOnItemClickListener(new ParametriAdapter.OnItemClickListener()), utilizzando la funzione OnItemClickListener definita nella classe ParametriAdapter :

```
public interface OnItemClickListener{
    void onItemClick(int position);
}
```

Viene passata a quest'ultima la posizione dell'elemento corrente, ovvero la posizione dell'elemento cliccato. A questo punto, per eliminare l'elemento dal database, è stata creata una finestra di tipo **Dialog** : creata nel momento in cui l'utente clicca su un elemento della lista, è una piccola finestra che occupa poco spazio e chiede all'utente di prendere la decisione di eliminare l'elemento selezionato o no.

```
mAdapter.setOnItemClickListener(new ParametriAdapter.OnItemClickListener() {
    @Override
    public void onItemClick(int position) {
        parametriList.get(position).getId_parametri();
        openDialog(parametriList.get(position).getId_parametri());

    }
});
```

Nella funzione setOnItemClickListener viene invocato il metodo openDialog definito all'interno della stessa classe, al quale viene passato come unico parametro l'id dell'elemento selezionato.

```
public void openDialog(String id){
    ParametriDialog parametriDialog = new ParametriDialog();
    Bundle bundle = new Bundle();
    bundle.putString("ID", id);
    parametriDialog.setArguments(bundle);
    parametriDialog.show(getFragmentManager(), tag: "Cancella parametri");

}
```

La funzione crea inizialmente un oggetto di tipo ParametriDialog e un Bundle che servirà per passare informazioni a quest'ultima, nel quale viene immagazzinata una stringa rappresentante l'id dell'elemento.

4.9.4 Cos'è una Dialog e come viene creata

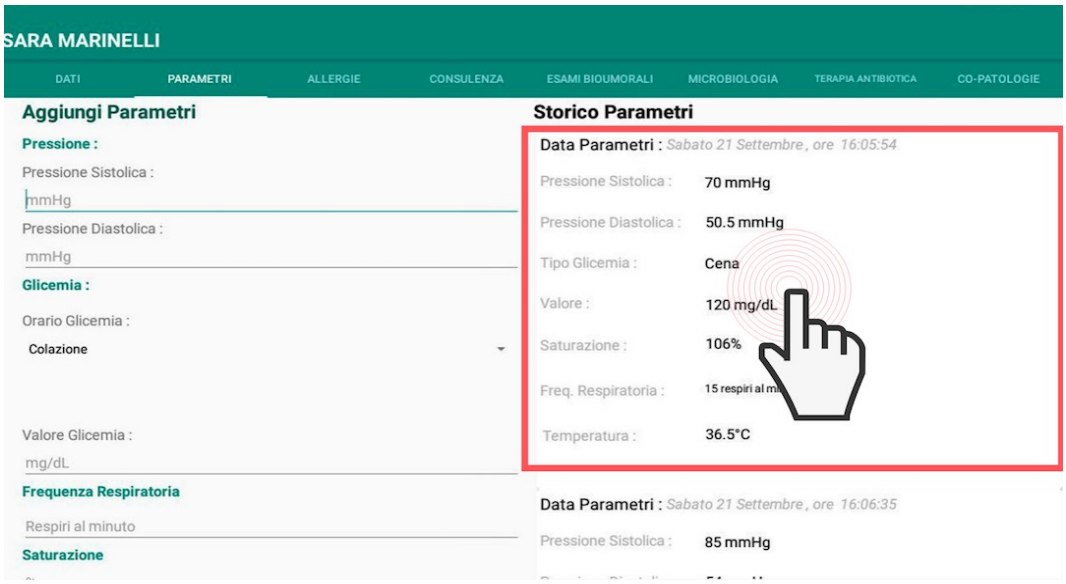

Figura 4.42 : Click su un elemento del RecyclerView

Nel momento in cui l'utente clicca su di un elemento della lista, come spiegato precedentemente, viene invocato un metodo che crea una finestra di tipo **Dialog** e alla quale viene passato l'id dell'oggetto cliccato.

La finestra appena citata si presenta in questo modo :

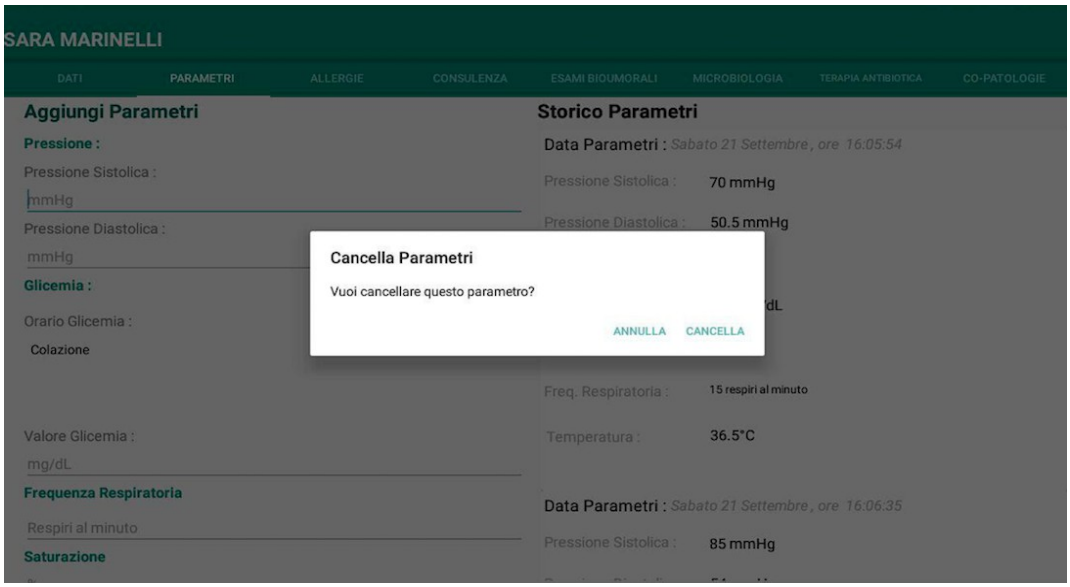

Figura 4.43 : Dialog relativa all'eliminazione di parametri

Essa viene creata istanziando un oggetto di tipo **DialogParametri**, classe contenente tutte le informazioni relative alla finestra che dovrà comparire sullo schermo (titolo, contenuto, bottoni etc.).

La classe **DialogParametri** è molto minimale, chiede in maniera diretta all'utente se è sicuro di voler eliminare i parametri selezionati, azione che può essere confermata o declinata cliccando sui bottoni sottostanti.

```
public class ParametriDialog extends AppCompatDialogFragment {
    private Button annullaBtn, cancellaBtn;
    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());


        LayoutInflater inflater = getActivity().getLayoutInflater();
        View view = inflater.inflate(R.layout.cancella_parametri_dialog,  root: null);

        builder.setTitle("Cancella Parametri")
                .setMessage("Vuoi cancellare questo parametro?")
                .setNegativeButton( text: "annulla", new DialogInterface.OnClickListener() {
                    @Override
                    public void onClick(DialogInterface dialog, int which) {

                    }
                })
                .setPositiveButton( text: "cancella", (dialog, which) → {
                        Activity a = (Activity)(getContext());

                        Bundle bundle = getArguments();
                        String id_parametro = bundle.getString( key: "ID");
                        MainActivity.db.eliminaParametro(id_parametro);

                        a.finish();
                        startActivity(a.getIntent());
                });

        annullaBtn = view.findViewById(R.id.annullaBtn);
        cancellaBtn = view.findViewById(R.id.cancellaBtn);
        return builder.create();
    }
}
```

All'interno della classe ParametriDialog, la quale estende la superclasse *AppCompatDialogFragment*, vengono definiti : il titolo della dialog, il messaggio mostrato e i due bottoni (negativo e positivo), le quali azioni e conseguenze vengono definite all'interno delle funzioni *.setNegativeButton* e *.setPositiveButton*.

La funzione *setNegativeButton* chiude la dialog, la funzione *setPositiveButton,* invece, estrapola l'id dell'oggetto cliccato dal bundle. quest'ultimo viene utilizzato all'interno della query "eliminaParametro(id_parametro)". La query individua nella tabella dei parametri l'esame il quale id corrisponde a quello passato alla funzione, una volta individuato viene eliminato dal database e di conseguenza anche dalla lista di parametri

all'interno dello storico.

```
public void eliminaParametro(String id){
    SQLiteDatabase db = dbHelper.getWritableDatabase();

    String where = "id_parametri=?";
    db.delete(DataBaseStrings.TABLE_PARAMETRI, where, new String[]{id}) ;


}
```

La possibilità di eliminare elementi dallo storico è stata implementata in tutte le tab all'interno del profilo del paziente. Sono state create quindi 8 dialog diverse che potessero eseguire query di eliminazione all'interno del database.

4.10 Tab Allergie

A fianco della Tab dei Parametri è posizionata la Tab delle Allergie.

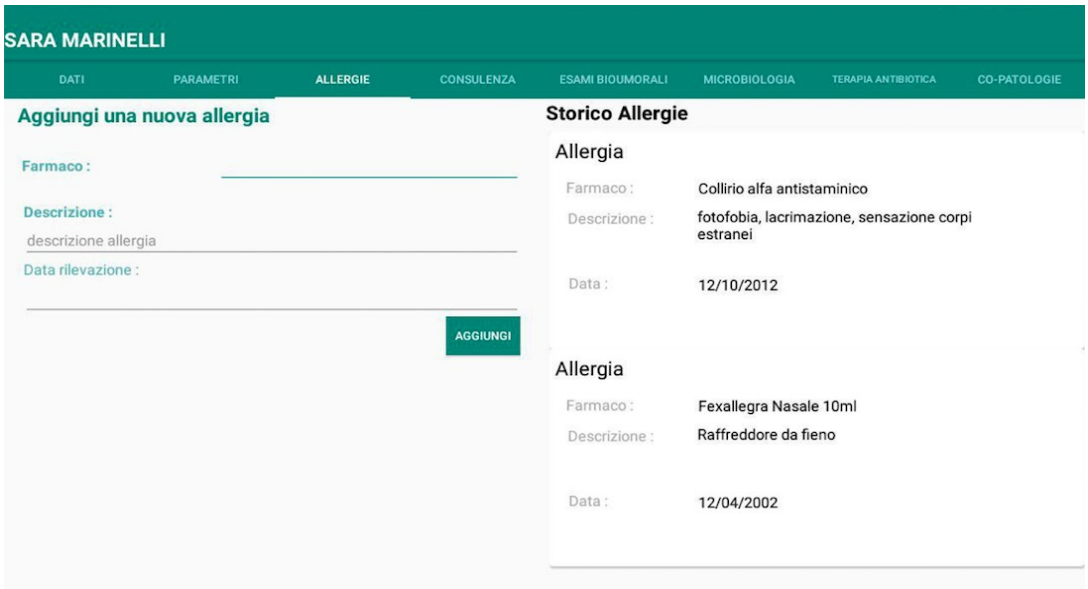

Figura 4.44 : Tab Allergie

Il medico può aggiungere manualmente un'allergia al profilo del paziente.
Viene richiesto :

- Il Farmaco utilizzato per curare l'allergia, il quale viene inserito all'interno di una semplice EditText

- La descrizione dell'allergia, ovvero i sintomi, come si è presentata la prima volta e i rischi maggiori
- La data della prima rilevazione dell'allergia, inserita all'interno di un'EditText avente InputType “date”

Di fianco alla sezione dell'inserimento di una nuova allergia si trova lo storico di quest'ultime, per ogni elemento vengono riportati i dati inseriti dal medico in precedenza.

La scheda è stata strutturata nello stesso modo della Tab Parametri, l'unica differenza è che il RelativeLayout a sinistra non è contenuto in una ScrollView, poiché i dati da inserire sono pochi e riescono a rientrare nella schermata statica.

4.11 Tab Consulenza

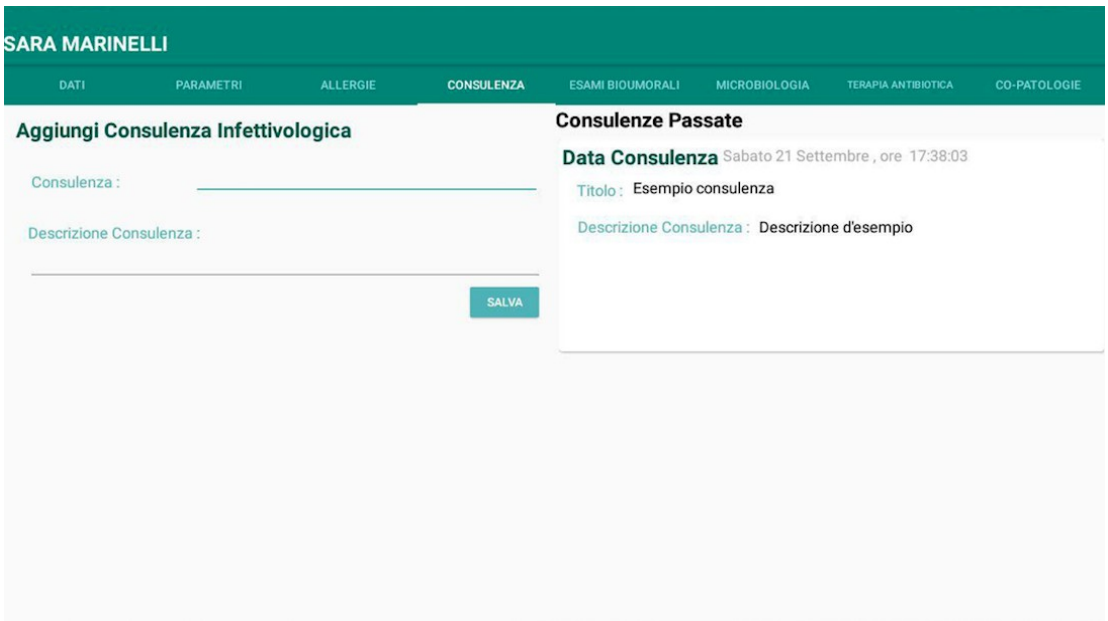

Figura 4.45: Tab Consulenze

Come suggerisce il nome della Tab, all'interno di quest'ultima, vengono inserite le consulenze fatte al paziente. I dati richiesti sono :

- Consulenza : una frase riassuntiva e descrittiva della consulenza effettuata
- Descrizione consulenza : una descrizione più dettagliata di quest'ultima

Una volta inserita vengono calcolate la data e l'ora attuale, le quali verranno riportate

all'interno dello storico assieme agli altri dati relativi alla consulenza.

4.12 Tab Microbiologia

La descrizione della Tab Esami Bioumorali verrà svolta subito dopo la descrizione delle tab “Microbiologia” e “Co-Patologie” poiché componenti al suo interno richiedono diversi approfondimenti trattati nell'ultima fase dell'implementazione, “Sezione Grafici”.

Tornando alla Tab Microbiologia, essa approfondisce gli esami svolti su campioni microbiologici dei pazienti :

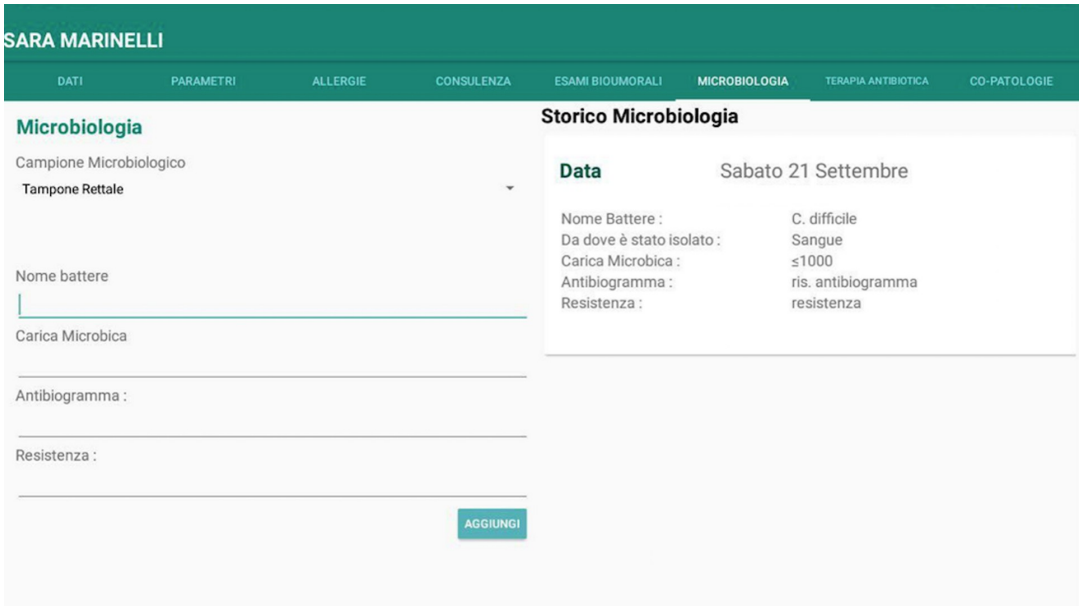

Figura 4.46 : Tab Microbiologia

Per selezionare il campione microbiologico dal quale è stato fatto l'esame è stata creato un menu a tendina da cui scegliere, al suo interno vi sono le scelte : tampone rettale, sangue, urine, urina da catetere, EspettoratoBal e “Altro”; selezionando la voce “Altro” apparirà una nuova EditText nella quale inserire a mano un campione microbiologico a scelta non presente nel menu. Viene inoltre richiesto il nome del battere, la carica microbica, l'antibiogramma e la resistenza. La struttura generale è la stessa delle tab

precedenti, è quindi anche possibile eliminare gli elementi dallo storico.

4.13 Tab Terapia Antibiotica

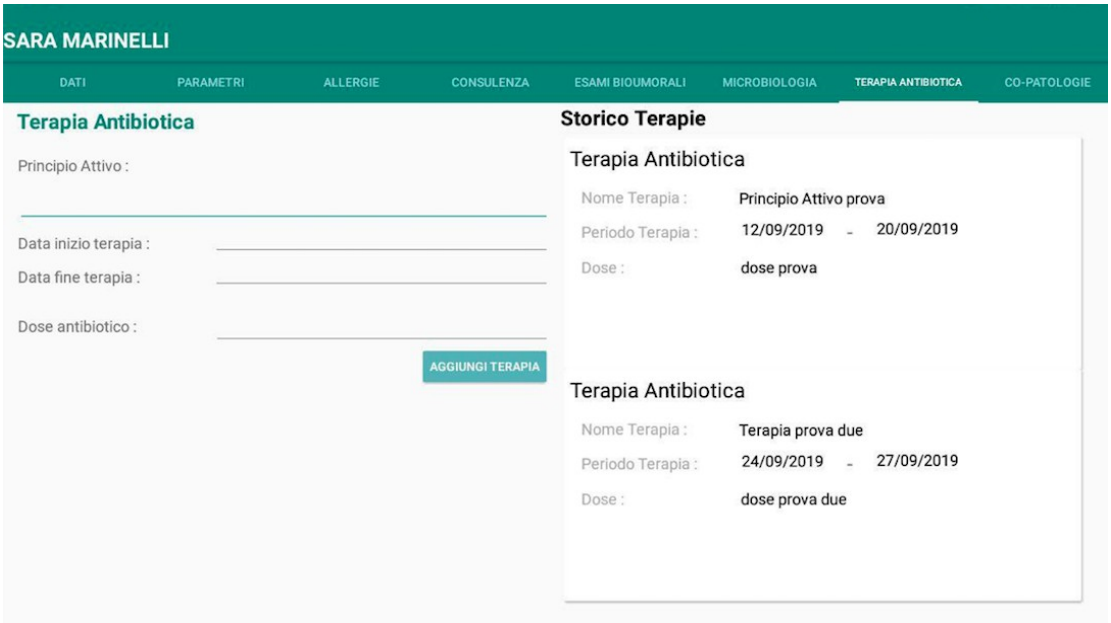

Figura 4.47 : Tab Terapia Antibiotica

A sinistra è possibile aggiungere una nuova terapia antibiotica, composta da un principio attivo, da un periodo delineato da una data iniziale ed una finale e infine la dose dell'antibiotico somministrato. La struttura del layout e la gestione dei dati è analoga alle tab precedentemente spiegate, troviamo a destra lo storico delle terapie ognuna all'interno di una “card” differente, la lista è scorrevole e cliccabile per eliminarne gli elementi singolarmente.

4.14 Tab Co-Patologie

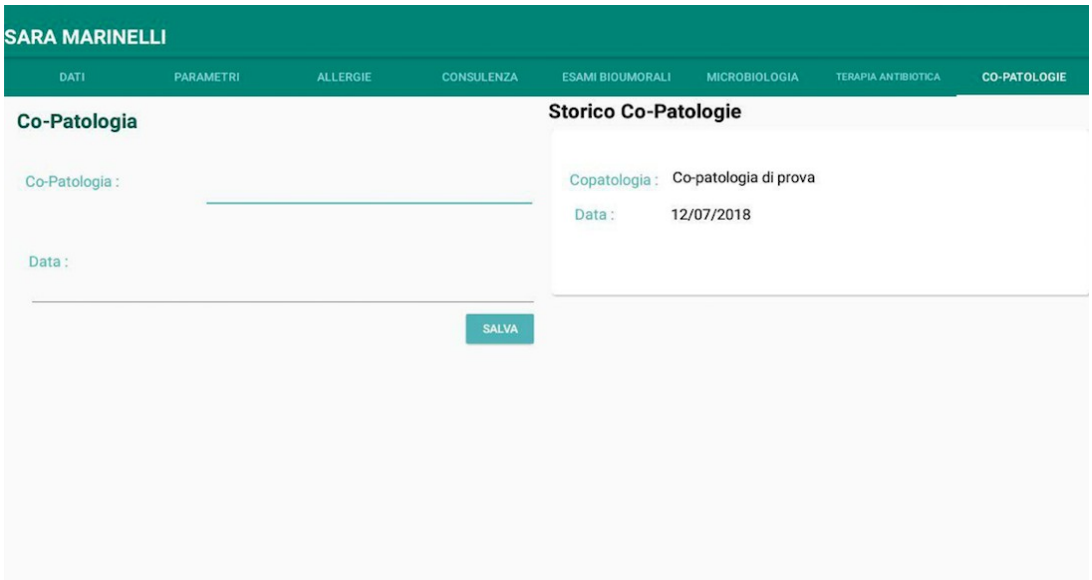

Figura 4.48 : Tab Co-Patologie

Infine troviamo la tab delle Co-Patologie, molto semplice e dai pochi campi. Una co-patologia è composta dalla variabile "co-patologia" e dalla data in cui è stata fatta. Gli elementi dello storico sono contenuti in un RecyclerView e possono essere selezionati ed eliminati.

4.15 Tab Esami Bioumorali

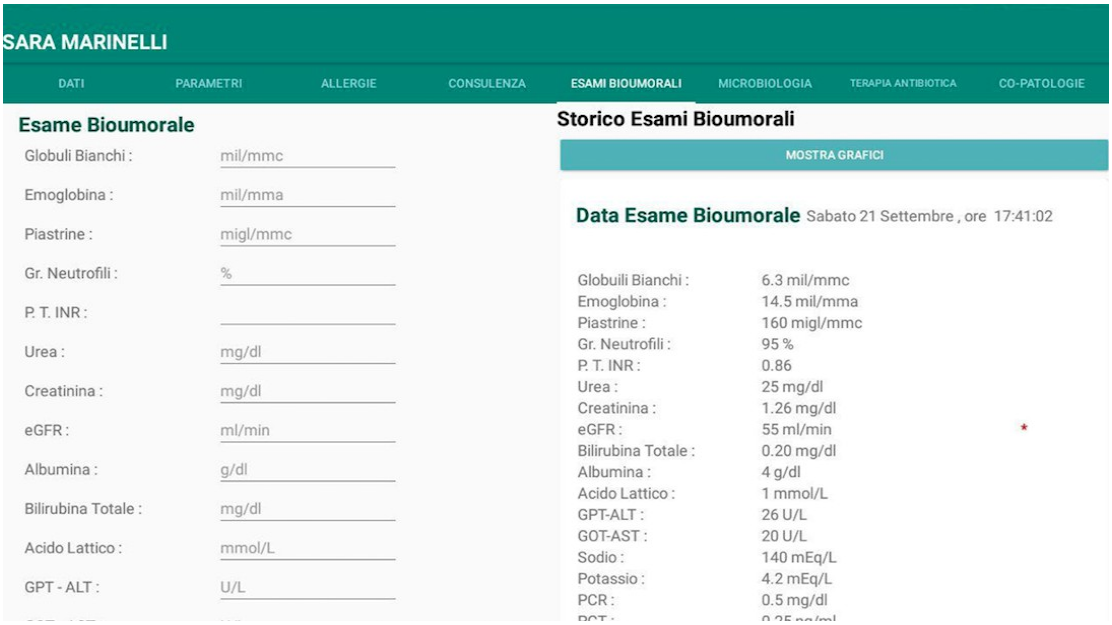

Figura 4.49 : Tab Esami Bioumorali

Un nuovo esame bioumorale viene inserito nel sistema quando il medico riempie tutti le EditText presenti nella sezione “Esame Bioumorale” e clicca sul pulsante “Salva”. Un esame bioumorale è quindi composto da 17 valori e la data in cui è stato effettuato.

Figura 4.50 : Tab Esami Bioumorali pt. 2

All'interno dello storico viene segnalato un valore sballato con la comparsa di un asterisco rosso al suo fianco. Il controllo di questi valori viene fatto all'interno dell'Adapter della Tab, il quale prende il nome di EsameAdapter. Quando l'elemento della lista viene creato viene effettuato un controllo su ogni valore inserito dal medico,

se quest'ultimo non rientra nel range di “normalità” l'asterisco al suo fianco viene settato come visibile.

Vengono mostrati alcuni controlli fatti su una parte dei valori :

```
if(globu <= 4 || globu >= 10.90){
    esameViewHolder.astGlob.setVisibility(View.VISIBLE);
}
if(emo <= 13.5 || globu >= 17.5){
    esameViewHolder.astEmo.setVisibility(View.VISIBLE);
}
if(pia <= 150 || pia >= 450){
    esameViewHolder.astPia.setVisibility(View.VISIBLE);
}
if(grNeut <= 90 || grNeut >= 150){
    esameViewHolder.astGrNeu.setVisibility(View.VISIBLE);
}
if(ptinr <= 0.85 || ptinr >= 1.25){
    esameViewHolder.astPT.setVisibility(View.VISIBLE);
}
if(urea <= 15 || urea >= 55){
    esameViewHolder.astUrea.setVisibility(View.VISIBLE);
}
if(creatinina <= 0.50 || creatinina >= 1.40){
    esameViewHolder.astCrea.setVisibility(View.VISIBLE);
}
if(eGFR <= 60){
    esameViewHolder.astEGFR.setVisibility(View.VISIBLE);
}
```

I range dei valori dell'esame sono stati forniti da Erica.

La tab Esami Bioumorali presenta una differenza rispetto alle altre schede, subito sotto il titolo “Storico Esami Bioumorali” è stato posizionato un bottone dal titolo “Mostra Grafici”:

Figura 4.51 : Bottone “Mostra Grafici” all'interno della tab Esame Bioumorale

Esso porta ad una nuova area dell'applicazione dedicata all'analisi di grafici creati sui valori degli esami bioumorali. Ogni valore dell'esame ha un grafico che ne disegna il comportamento nel tempo, avere a disposizione una rappresentazione grafica dell'andamento di questi esami è di grande supporto al medico che deve analizzarli.

4.16 Visualizzazione Grafici Esami Bioumorali

Figura 4.52 : Schema generale con focus sulla Sezione Grafici

E' stata dedicata un'intera area dell'applicazione alla visualizzazione dei risultati degli esami bioumorali per renderne la lettura e l'interpretazione più semplice e veloce possibile. Insieme a questa soluzione si era pensato di rendere i nomi degli esami cliccabili e, una volta selezionato uno di essi, mostrarne il grafico all'interno di una Dialog. Questo metodo è stato scartato per la ridotta dimensione delle finestre Dialog, le quali rendono la lettura dei valori all'interno dei grafici difficoltosa.

E' stato sfruttata la tipologia di menu "Navigation Drawer" molto diffuso e utilizzato negli applicativi Android. La scelta di avere questa gestione della pagina è dovuta alla semplicità di navigazione e dalla spaziosità delle schede che compongono il layout.

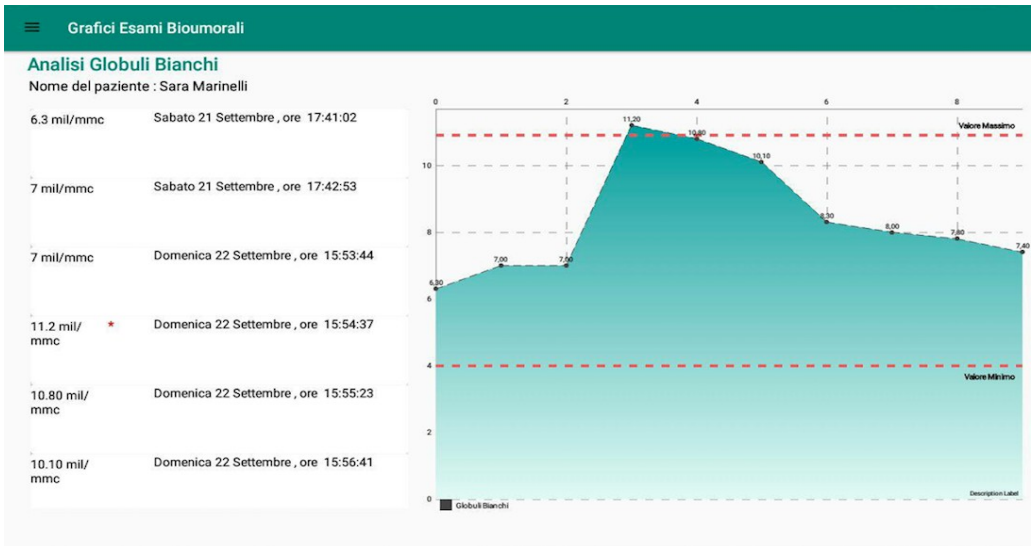

Figura 4.53 : Pagina iniziale Grafici Esami Bioumorali

Cliccando il bottone “Mostra grafici” nella Tab EsamiBioumorali l'utente verrà portato sulla finestra mostrata nella figura 4.53, contenente il grafico sull'andamento dei globuli bianchi e l'elenco degli esami effettuati con data, ora, valore e relativo asterisco se l'esame non rientra nel range.

Per ogni valore è stata costruita una scheda di questo tipo (Es. Globuli bianchi, emoglobina, piastrine, Gr.Neutrofili etc.), esse sono selezionabili dal menu a scomparsa posizionato a sinistra dello schermo :

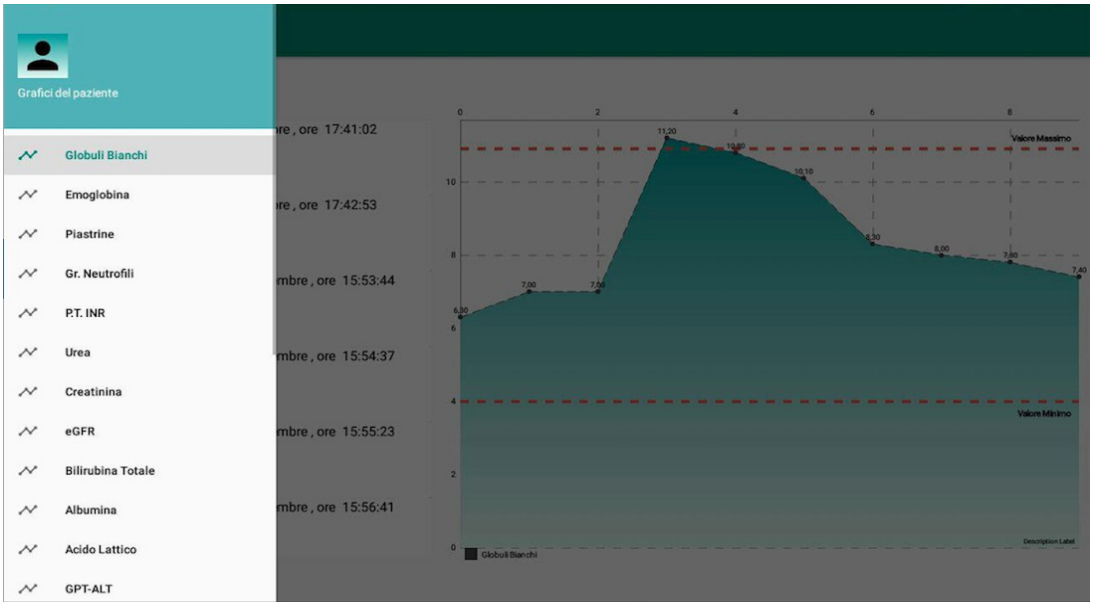

Figura 4.54 : Menu Navigation Drawer Finestra dei grafici

4.16.1 Il NavigationDrawer

Il **NavigationDrawer** può essere visualizzato dall'utente scorrendo il dito partendo dal bordo sinistro dello schermo oppure cliccando sull'icona composta da tre trattini di fianco al titolo dell'Activity. Dal momento in cui le “destinazioni” possibili partendo da questa Activity sono numerose (17, una per ogni valore) sono state elencate tutte all'interno del menu laterale, in modo da potervi accedere e selezionarle ordinatamente.

Inizialmente, per utilizzare il *DrawerLayout* e il *NavigationView* nel proprio progetto, è necessario importare le librerie di supporto e di design all'interno del file build.gradle :

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'com.android.support:design:28.0.0'
    implementation 'com.android.support:support-v4:28.0.0'
```

Per implementare il layout non è stato utilizzato il template messo a disposizione da AndroidStudio ma è stato realizzato da zero partendo da un'Empty Activity.

E' stato creato un file *nav_header.xml* contenente un LinearLayout verticale, un'immagine e una label per comporre l'header del NavigationDrawer :

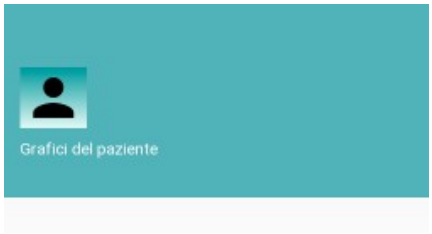

Figura 4.55 : navigation header

L'Activity *MainGrafici.java* rappresenta il contenitore principale del menu e dei fragment contenenti i grafici. All'interno del file .xml a lei collegato è stato inserito un DrawerLayout, una ToolBar, un FrameLayout e un NavigationView, il quale header è quello mostrato nella figura 4.55.

```
<android.support.v4.widget.DrawerLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:id="@+id/drawer_layout"
    android:fitsSystemWindows="true"
    app:layout_behavior="android.support.design.widget.AppBarLayout$ScrollingVie..."
    tools:context=".activities.MainGrafici"
    tools:showIn="@layout/activity_main_grafici"
    tools:openDrawer="start">

        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:orientation="vertical">

            <android.support.v7.widget.Toolbar
                android:layout_width="match_parent"
                android:layout_height="?attr/actionBarSize"
                android:background="@color/colorPrimary"
                app:titleTextColor="@color/tabcolor"
                android:id="@+id/toolbar"
                android:theme="@style/Base.ThemeOverlay.AppCompat.ActionBar"
                app:popupTheme="@style/Base.ThemeOverlay.AppCompat.Light"
                android:elevation="4dp"/>

            <FrameLayout
                android:id="@+id/fragment_container"
                android:layout_width="match_parent"
                android:layout_height="match_parent"></FrameLayout>
        </LinearLayout>

    <android.support.design.widget.NavigationView
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:id="@+id/nav_view"
        app:headerLayout="@layout/nav_header"
```

E' stato infine creato un file *drawer_menu.xml* nel quale sono state specificate tutte le diverse voci del menu con relative icone :

```xml
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:showIn="navigation_view">

    <group android:checkableBehavior="single">
        <item android:id="@+id/nav_1"
            android:icon="@drawable/ic_grafico"
            android:title="Globuli Bianchi"/>
        <item android:id="@+id/nav_2"
            android:icon="@drawable/ic_grafico"
            android:title="Emoglobina"/>
        <item android:id="@+id/nav_3"
        android:icon="@drawable/ic_grafico"
        android:title="Piastrine"/>
        <item android:id="@+id/nav_4"
            android:icon="@drawable/ic_grafico"
            android:title="Gr. Neutrofili"/>
        <item android:id="@+id/nav_5"
            android:icon="@drawable/ic_grafico"
            android:title="P.T. INR"/>
        <item android:id="@+id/nav_6"
            android:icon="@drawable/ic_grafico"
            android:title="Urea"/>
        <item android:id="@+id/nav_7"
            android:icon="@drawable/ic_grafico"
            android:title="Creatinina"/>
        <item android:id="@+id/nav_8"
            android:icon="@drawable/ic_grafico"
            android:title="eGFR"/>
        <item android:id="@+id/nav_9"
            android:icon="@drawable/ic_grafico"
            android:title="Bilirubina Totale"/>
        <item android:id="@+id/nav_10"
            android:icon="@drawable/ic_grafico"
            android:title="Albumina"/>
        <item android:id="@+id/nav_11"
            android:icon="@drawable/ic_grafico"
            android:title="Acido Lattico"/>
        <item android:id="@+id/nav_12"
```

Il risultato finale si presenta in questo modo :

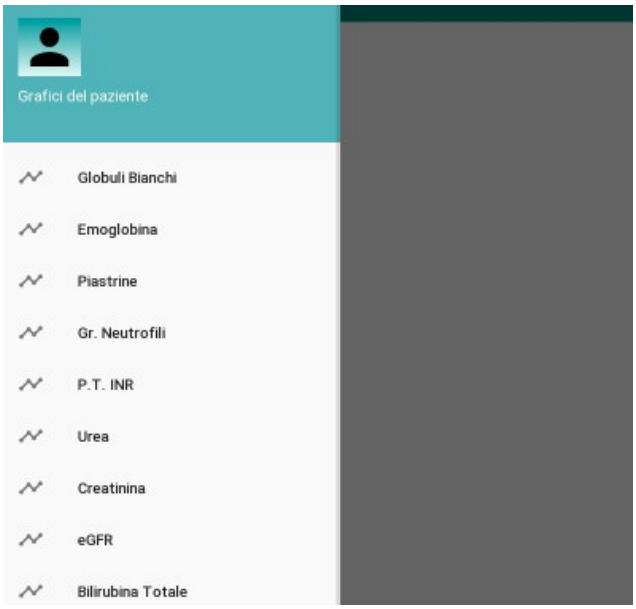

Figura 4.56 :Navigation Drawer Completo

All'interno del file *MainGrafici.java* sono stati definite le classi che implementano le varie schede dei grafici, in modo da collegare ogni elemento del menu al fragment contenente le informazioni del valore cliccato. E' stato implementato un costrutto switch per poter distinguere le varie casistiche :

```
public class MainGrafici extends AppCompatActivity implements NavigationView.OnNavigationItemSelectedListener {
    private DrawerLayout drawer;


    @Override
    public boolean onNavigationItemSelected(@NonNull MenuItem menuItem) {
        switch (menuItem.getItemId()){
            case R.id.nav_1:
                getSupportFragmentManager().beginTransaction().replace(R.id.fragment_container,
                        new GlobuliBianchiFragment()).commit();
                break;
            case R.id.nav_2:
                getSupportFragmentManager().beginTransaction().replace(R.id.fragment_container,
                        new EmoglobinaFragment()).commit();
                break;
            case R.id.nav_3:
                getSupportFragmentManager().beginTransaction().replace(R.id.fragment_container,
                        new PiastrineFragment()).commit();
                break;
            case R.id.nav_4:
                getSupportFragmentManager().beginTransaction().replace(R.id.fragment_container,
                        new GRNeutrofiliFragment()).commit();
                break;
            case R.id.nav_5:
                getSupportFragmentManager().beginTransaction().replace(R.id.fragment_container,
                        new PtInrFragment()).commit();
                break;
```

4.17 Creazione Grafici esami Bioumorali

Per realizzare i grafici relativi agli esami bioumorali è stata utilizzata la libreria **MPAndroid Chart**. Essa mette a disposizione numerosissimi tipi di grafici e gli strumenti per personalizzarli.

Per mostrare l'andamento degli esami è bastato utilizzare un semplice *Grafico Lineare.* I grafici a linee evidenziano il flusso temporale e la frequenza del cambiamento, anziché l'entità del cambiamento.

Sull'asse delle X è stato posizionato un indice sequenziale che aumenta all'aumentare dei valori inseriti all'interno dello storico, parte da 0 e arriva a n-1, dove n è il numero di valori all'interno del database.

Sull'asse delle Y, invece, viene segnato il valore vero e proprio dell'esame, in modo tale che, andando a congiungere l'indice di quest'ultimo, il suo valore e il punto successivo nel grafico si possa rappresentare una curva che descriva l'andamento nel tempo

dell'esame.

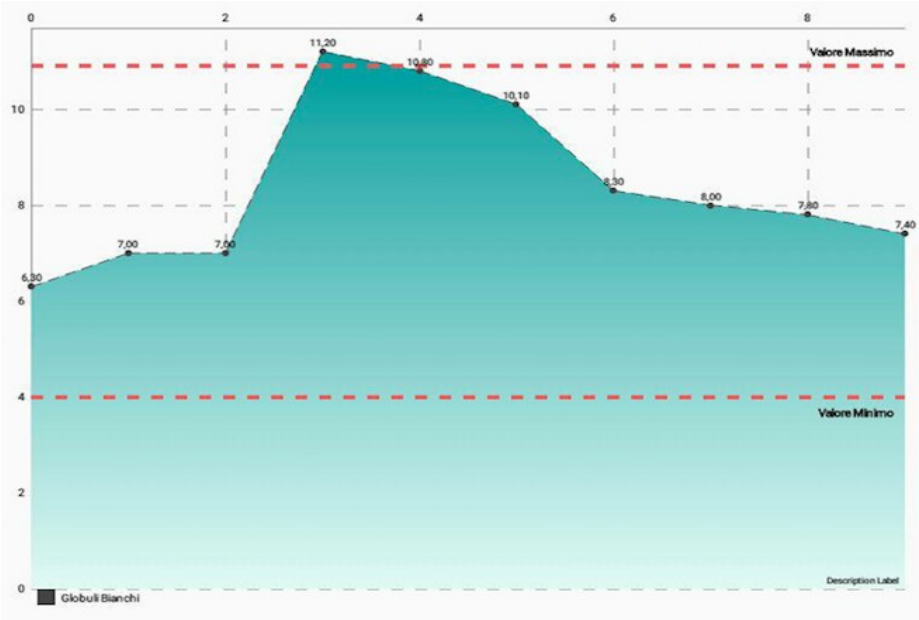

Figura 4.57 : Esempio di grafico lineare

MPAndroid Chart dà la possibilità di aggiungere, all'interno del grafico, una coppia di linee tratteggiate che delineino il range del valore massimo e del valore minimo all'interno del quale dovrebbero stare i valori dell'esame studiato.
In figura 4.57 il Valore minimo corrisponde a *4,00 mil/mmc* e il Valore massimo a *10.90 mil/mmc*, quando un punto del grafico esce dalle linee tratteggiate significa che il valore dell'esame è sballato, e non rientra nella normalità.

Analogamente alla Tabbed View spiegata nel paragrafo 4.7.1 ogni elemento all'interno del menu è legato ad un fragment indipendente, avente un layout proprio e contenuti diversi dagli altri.
Ogni Fragment ha la stessa struttura, l'unica cosa che li distingue sono i dati mostrati, poiché ognuno di essi è destinato a mostrare il grafico di un valore differente. Essendo tutti e 17 simili andremo ad analizzare soltanto uno di essi, quello dedicato all'analisi dei Globuli Bianchi.

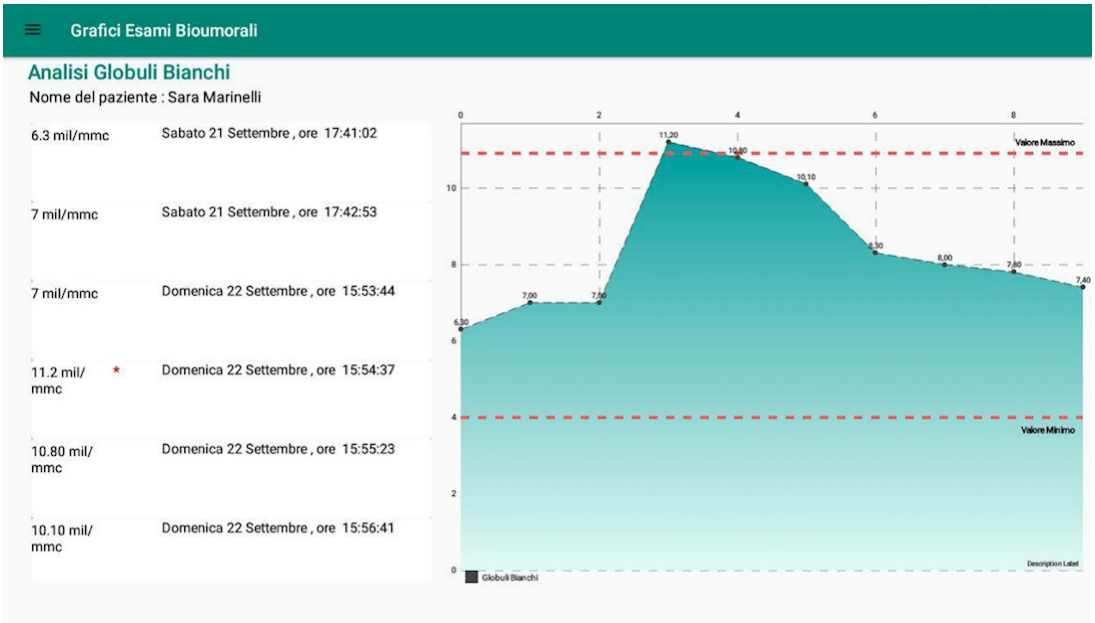

Figura 4.58 : Fragment grafici Globuli BIanchi

La pagina è stata suddivisa principalmente in due sezioni : la prima dedicata ad un elenco di record che riportano tutti gli esami fatti, specificando ovviamente soltanto il valore dei globuli bianchi, la data in cui sono stati fatti e un eventuale asterisco per segnalare un'anomalia.

La lista di record è stata implementata utilizzando il RecyclerView. Gli elementi non sono cliccabili, fungono soltanto da storico.

E' stato di conseguenza creato un Adapter, un layout per rappresentare il singolo elemento all'interno della lista e un componente di tipo RecyclerView all'interno del layout del fragment.

Per quanto riguarda il grafico, invece, è stata definita una variabile di tipo **LineChart** nella classe *GlobuliBianchiFragment.java*, ovvero il fragment dedicato ai risultati dei globuli bianchi, la quale rappresenta il grafico lineare e alla quale saranno applicate tutte le modifiche necessarie.

Subito dopo l'inizializzazione della variabile viene chiamata la funzione ***renderData()***, nella quale sono state specificate tutte le proprietà e l'aspetto del grafico.

```
public void renderData() {

    final ArrayList<Double> globuli_bianchi = MainActivity.db.trovaGlobuliBianchi(ProfiloPaziente.codice);
    Float max = (float) globuli_bianchi.size() -1 ;

    final XAxis xAxis = mChart.getXAxis();
    xAxis.enableGridDashedLine( lineLength: 10f,  spaceLength: 10f,  phase: 0f);
    xAxis.setAxisMaximum(30f);
    xAxis.setAxisMinimum(0f);
    xAxis.setDrawLimitLinesBehindData(true);
    xAxis.setAxisMaximum(max);


    LimitLine ll1 = new LimitLine( limit: 10.90f,  label: "Valore Massimo");
    ll1.setLineWidth(4f);
    ll1.enableDashedLine( lineLength: 10f,  spaceLength: 10f,  phase: 0f);
    ll1.setLabelPosition(LimitLine.LimitLabelPosition.RIGHT_TOP);
    ll1.setTextSize(10f);

    LimitLine ll2 = new LimitLine( limit: 4.00f,  label: "Valore Minimo");
    ll2.setLineWidth(4f);
    ll2.enableDashedLine( lineLength: 10f,  spaceLength: 10f,  phase: 0f);
    ll2.setLabelPosition(LimitLine.LimitLabelPosition.RIGHT_BOTTOM);
    ll2.setTextSize(10f);

    YAxis leftAxis = mChart.getAxisLeft();
    leftAxis.removeAllLimitLines();
    leftAxis.addLimitLine(ll1);
    leftAxis.addLimitLine(ll2);
    leftAxis.setAxisMinimum(0f);
    leftAxis.enableGridDashedLine( lineLength: 10f,  spaceLength: 10f,  phase: 0f);
    leftAxis.setDrawZeroLine(false);
    leftAxis.setDrawLimitLinesBehindData(false);

    mChart.getAxisRight().setEnabled(false);
}
```

Nella variabile *globuli_bianchi* vengono salvati tutti i valori relativi ai globuli bianchi presenti nel database, la lunghezza di questa lista verrà utilizzata per specificare all'interno del grafico quanti valori verranno inseriti.

Sono state definite due variabili di tipo XAxis e YAxis rappresentanti rispettivamente l'asse X e l'asse Y all'interno del grafico. Per entrambe vengono specificate alcune caratteristiche da cui dipenderà il loro aspetto nell'interfaccia.

- *enableGridDashedLine(lineLength, spaceLength, phase)* : consente di tracciare la linea della griglia in modalità tratteggiata "- - - -". lineLength controlla la lunghezza dei trattini della linea, spaceLength lo spazio tra le linee e phase segnala il punto iniziale
- *setAxisMaximum/Minimum()* : imposta un valore massimo/minimo personalizzato per l'asse su cui viene chiamato. Il valore massimo dell'asse delle

x corrisponde al numero di valori dei globuli bianchi presenti nel database.

- *AddLimitLine()* : permette di inserire una nuova “limitline” all'asse. In questo caso vengono aggiunge due limit line soltanto all'asse delle Y per rappresentare il range di normalità del valore dei globuli bianchi.

Le limit lines vengono create separatamente, ad ognuna di esse viene associata una label per distinguerle rispettivamente “Valore massimo” per la linea rappresentante il massimo e “Valore minimo” per la linea rappresentante il minimo.
Attraverso il metodo *setLabelPosition* vengono posizionate le label relative alle limitLines in posizione RIGHT_TOP o RIGHT_BOTTOM. Infine, anche queste linee vengono definite come “linee tratteggiate” attraverso il metodo enableDashedLines.

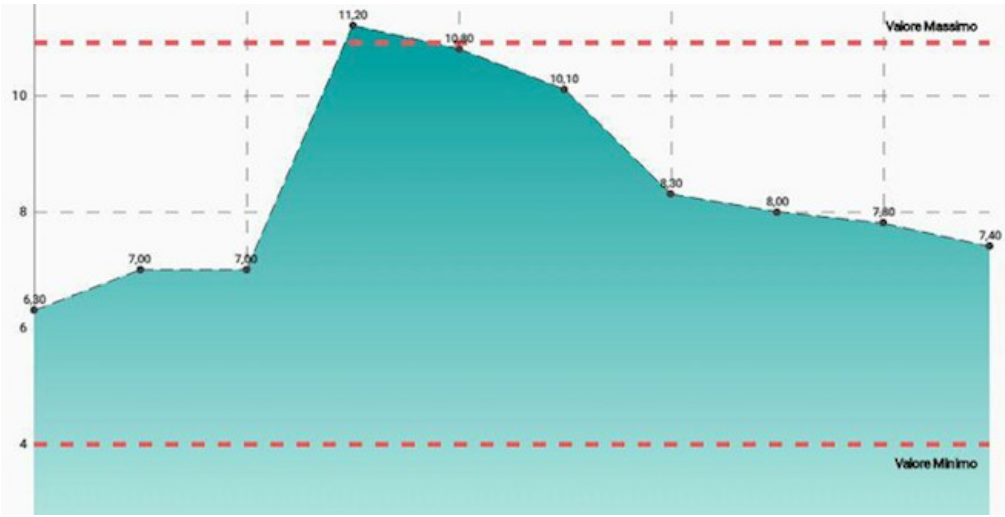

Figura 4.59 : Close-up sulle LimitLines all'interno di un grafico MPAndroid Chart

I grafici appena descritti sono ingrandibili e cliccabili. La possibilità di poter cliccare sugli elementi del grafico ha fatto sì che potesse essere implementata un'ultima funzionalità molto importante.
Cliccando su un preciso punto del grafico verrà mostrata una finestra Dialog contenente la data precisa in cui è stato effettuato l'esame d'interesse.

E' stata aggiunta questa possibilità di analisi degli esami per evitare di andare a cercare l'esame corrispondente all'interno dello storico a sinistra della scheda per poi individuarne la data. Questa operazione sarebbe risultata alquanto lenta e soggetta ad errori. Viene mostrata in seguito la Dialog mostrata nel momento del click su di un valore del grafico :

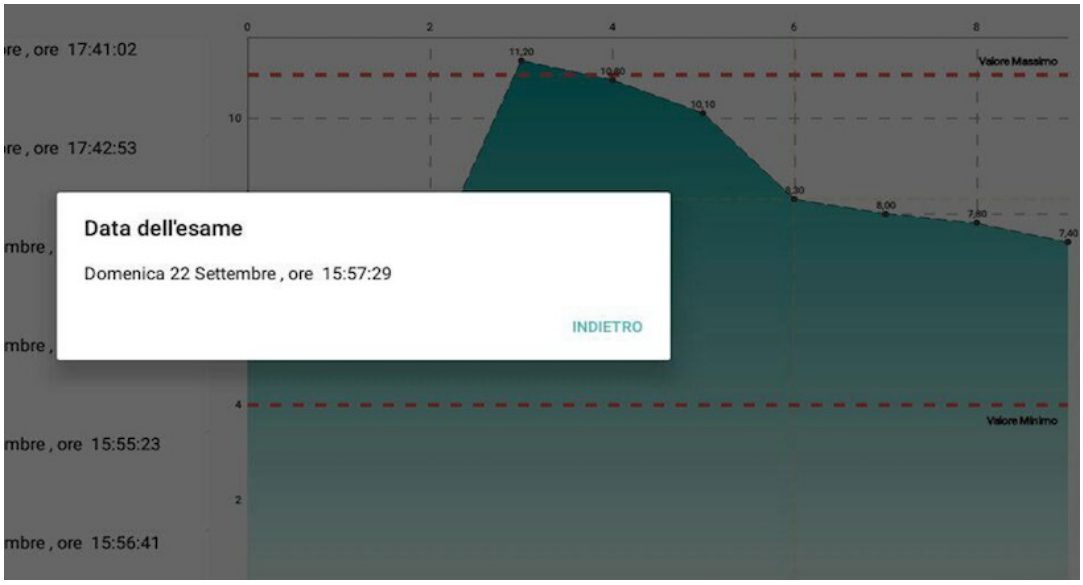

Figura 4.60 : Dialog contenente la data dell'esame cliccato

E' stata implementata una sola dialog per tutti i fragment riguardanti i grafici, ovvero è stato necessario creare una sola classe di tipo Dialog per tutti i grafici, senza doverne creare una per ogni tipo di valore.

Nel momento in cui l'utente clicca su un valore :

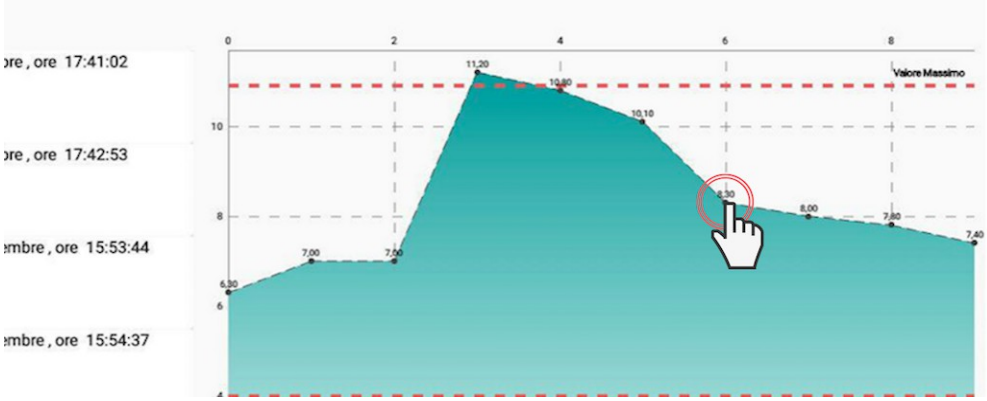

Figura 4.61 : Click su punto del grafico

Per gestire il click su un elemento del grafico è stato implementato il metodo *setOnChartValueSelectedListener* sulla variabile rappresentante il grafico. Il metodo viene chiamato tutte le volte in cui l'utente clicca su un punto del grafico, ma come distinguere un punto da un altro?

```
//funzione che preleva le coordinate x e y dell'oggetto, la x è essenziale per ricavare
//la posizione dell'oggetto cliccato all'interno della list. Una volta ricavato l'indice
//si preleva l'i-esimo oggetto dall'array date_globuli_bianchi
mChart.setOnChartValueSelectedListener(new OnChartValueSelectedListener() {
    @Override
    public void onValueSelected(Entry e, Highlight h) {
        Activity a = (Activity)(getContext());

        float x=e.getX();
        float y=e.getY();


        //l'indice della x è l'indice dell'oggetto nel recycler view, lo converto in int

        Integer indice_recycler = (int) x;

        openDialogEsami(date_globuli_bianchi.get(indice_recycler));
    }

    @Override
    public void onNothingSelected() {

    }
});
```

Alla funzione *onValueSelected* viene passato come parametro e, ovvero la Entry cliccata dall'utente. Avendo *e* è possibile ottenere il valore delle sue coordinate X ed Y all'interno del grafico. Il valore della coordinata X è essenziale poiché, come spiegato in precedenza, essa corrisponde esattamente all'indice dell'elemento salvato in maniera sequenziale e cronologica all'interno del database. Può essere quindi utilizzato come indice per andare a individuare esattamente il valore selezionato e quindi la sua data.

Prima di definire la funzione *setOnChartValueSelectedListener* è stato definito e inizializzato un ArrayList contenente tutte le date dei valori dei globuli bianchi. Dal momento in cui gli elementi sono ordinati in maniera analoga sia all'interno dell'array sia all'interno del grafico l'indice dell'asse X corrisponderà all'indice dell'elemento all'interno dell'array.

La data, come è possibile vedere nell'estratto di codice qui sopra, viene passata alla funzione openDialogEsami, nella quale viene inserita in un bundle e passata alla classe DataEsameDialog.java.

```java
public void openDialogEsami(String data){
    DataEsameDialog dataEsameDialog = new DataEsameDialog();
    Bundle bundle = new Bundle();
    bundle.putString("DATA", data);
    dataEsameDialog.setArguments(bundle);
    dataEsameDialog.show(getFragmentManager(),  tag: "Mostra data");

}
```

Infine, nella classe *DataEsameDialog.java*, la data viene mostrata nel messaggio della finestra. L'utente avrà solo la possibilità di visualizzare la data e tornare indietro al grafico ed eventualmente cliccare su un altro valore per visualizzarne la data.

```java
import ...

public class DataEsameDialog extends AppCompatDialogFragment {
    private Button annullaBtn, cancellaBtn;

    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());


        LayoutInflater inflater = getActivity().getLayoutInflater();
        Bundle bundle = getArguments();

        builder.setTitle("Data dell'esame")
                .setMessage(bundle.getString( key: "DATA"))
                .setNegativeButton( text: "Indietro", new DialogInterface.OnClickListener() {
                    @Override
                    public void onClick(DialogInterface dialog, int which) {

                    }

                });


        return builder.create();
    }
}
```

Capitolo 5
Conclusioni e Sviluppi Futuri

5.1 Conclusioni e Sviluppi futuri

La realizzazione di CartellaClinica si può vedere come un piccolo ma importante passo verso grandi migliorie nel mondo ospedaliero, campo che può essere totalmente rivoluzionato dalle potenzialità informatiche esistenti al giorno d'oggi. L'applicazione va incontro a quelli che sono i bisogni e le esigenze dell'infettivologo all'interno dell'ospedale, offrendo uno strumento del tutto portatile, semplice ed intuitivo da integrare alle azioni svolte quotidianamente.
Tuttavia una cartella clinica elettronica non ha il solo potere di sostituire totalmente i documenti cartacei, ma possiede tutte le risorse per arrivare ad ottimizzare operazioni di elaborazione dati, filtraggio e monitoraggio rendendo l'analisi di dati molto più rapida e corretta.
Con questo percorso è stata realizzata un'applicazione che può fungere da prototipo per un futuro software dalle grandissime potenzialità, l'obiettivo è stato infatti quello di costruire innanzitutto un'interfaccia pronta per poter accogliere grandi moli di dati e organizzarli al suo interno, ma le possibili operazioni che si possono integrare al

progetto sono svariate.

Una di esse, la più rilevante, è la procedura di estrapolazione dati provenienti direttamente dalla dorsale del Policlinico di Modena con l'integrazione della tecnologia NFC, andando quindi a eliminare tutte le operazioni di inserimento dati necessarie per popolare il database in questa versione del progetto. L'idea è quella di realizzare un software che possa prelevare dati relativi ai pazienti grazie al solo avvicinamento al loro letto dotato di un device passivo che possa trasmettere il codice del paziente al tablet. Con i medici Giovanni Guaraldi ed Erica Franceschini si è discusso molto sulla possibilità di poter estrarre dati relativi ai pazienti direttamente dalla dorsale principale dell'ospedale, progetto al quale purtroppo non sono riuscita a lavorare per motivi di tempistiche e fasi preliminari che bisognava compiere (realizzazione dell'interfaccia). Tuttavia, l'estrapolazione verrà sicuramente studiata ed approfondita in un possibile sviluppo futuro, dal momento in cui ciò che è stato fatto finora mira direttamente al completamento di questa funzionalità.

Infine, riprendendo quanto detto nel paragrafo 1.2 “Possibili progetto emersi durante gli incontri”, vi è la possibilità di espandere ulteriormente il progetto aggiungendo due aree in un menu principale : una dedicata al controllo della propagazione delle infezioni all'interno del reparto Malattie Infettive e l'altra riguardo la somministrazione di medicinali ai pazienti.

CartellaClinica può quindi essere ingrandita sotto moltissimi aspetti poiché, come è stato dimostrato dall'applicazione stessa, la medicina e l'informatica sono destinate ad evolversi e modellarsi a vicenda apportando un grande contributo alla salute umana.

Bibliografia

- Android Studio Documentation, https://developer.android.com/docs
- DB Browser for SQLite Documentation,https://github.com/sqlitebrowser/sqlitebrowser/wiki
- SQLite Documentation, https://www.sqlite.org/docs.html
- NFC, https://it.wikipedia.org/wiki/Near_Field_Communication
- MPAndroid Chart, https://github.com/PhilJay/MPAndroidChart
- RecyclerView Documentation, https://developer.android.com/guide/topics/ui/layout/recyclerview
- Tabbed Activity, https://developer.android.com/guide/navigation/navigation-swipe-view
- Master/Detail Flow, https://www.techotopia.com/index.php/An_Android_Master/Detail_Flow_Tutorial
- Navigation Drawer, https://developer.android.com/guide/navigation/navigation-ui
- Fragments, https://developer.android.com/guide/components/fragments