Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica per il Management

PROGETTAZIONE E IMPLEMENTAZIONE DI UN'APPLICAZIONE MOBILE PER IL MONITORAGGIO DI PAZIENTI CON SCOMPENSO CARDIACO

Relatore:
Chiar.mo Prof.
Marco Di Felice

Presentata da:
Andrea Lombardo

Correlatore:
Dott.
Luciano Potena

III Sessione
Anno Accademico 2019/2020

Abstract

Il follow up medico è un aspetto fondamentale nella gestione della qualità di vita di un paziente con scompenso cardiaco. Tuttavia, la pratica non è uguale per tutti, poichè ogni terapia scelta per il paziente ha una specifica fase di controllo che differisce dalle altre. Alcune di queste, che prevedono una comunicazione più continua tra medico e paziente, migliorano la condizione psicofisica di quest'ultimo, aumentando il suo benessere attraverso una maggiore consapevolezza e disciplina. Il contributo maggiore di questa tesi è quindi uno standard di monitoraggio remoto rappresentato da un'applicazione mobile in Android: discuteremo la sua progettazione e l'implementazione delle sue funzionalità nel dettaglio, per poi presentare due possibili esperienze d'uso da parte del paziente, mostrando il funzionamento di una soluzione che potrebbe effettivamente ridurre i costi della fase di follow up e garantire un'azione preventiva continua.

Indice

Introduzione **7**

1 Stato dell'arte **11**

1.1 Applicazioni mobili 11

1.1.1 Approcci per lo sviluppo di app 12

1.1.2 Sistemi operativi 13

1.2 Applicazioni mobili in ambito biomedicale 15

1.2.1 mHealth 15

1.2.2 Funzionalità 17

1.2.3 Obiettivi 17

2 Progettazione **21**

2.1 Obiettivi 21

2.2 Specifiche 22

2.3 Architettura 24

2.3.1 Adozione dell'architettura 29

2.3.2 Funzionalità 31

3 Implementazione **35**

3.1 Tecnologie 35

3.2 Componenti dell'architettura 37

3.2.1 Interfaccia grafica 38

3.2.2 Repository 44

3.2.3 Database 46

3.3 Funzionalità 50
3.3.1 Registrazione e autenticazione 51
3.3.2 Visualizzazione monitor 53
3.3.3 Inserimento parametri 54
3.3.4 Calendario 55
3.3.5 Visualizza giorno 61
3.3.6 Esportazione e invio report 62
3.3.7 Grafici 66
3.3.8 Notifica 68
3.3.9 Documentazione 71
3.3.10 Contapassi 73
3.3.11 Allarme 76
3.4 Testing 76

4 Validazione **79**
4.1 Registrazione e primo utilizzo 79
4.2 Utilizzo dopo due settimane 83

5 Conclusioni e sviluppi futuri **87**
5.1 Sviluppi futuri 88

Ringraziamenti **91**

Bibliografia **93**

Elenco delle figure

1.1 Architettura Android 14
1.2 Esempi di schermate di Quit Genius [21]. 18
1.3 Esempi di schermate di MobileHeart [16]. 19

2.1 Clean Architecture, struttura concentrica [22] 25
2.2 Clean Architecture, ruotata per identificare i tre strati 27
2.3 Clean Architecture, data flow. 28
2.4 Clean Architecture come tre strati, Dependency Rule. 29
2.5 I componenti del MVVM. 30
2.6 MVVM, Dependency Rule. 31
2.7 MVVM, flusso dei dati. 31

3.1 Implementazione del MVVM. 38
3.2 Ciclo di vita del ViewModel in relazione a quello dell'Activity [32]. 40
3.3 Rappresentazione grafica del layout di navigazione in XML tramite il Navigation Editor di Android Studio. 43
3.4 Le tre componenti di Room. 47
3.5 Meccanismo di funzionamento della RecyclerView: uso dell'Adapter. . . . 57

4.1 (a) Login (b) Registrazione, Fase 2 (c) Login effettuato, schermata Monitor 80
4.2 (a) FAB, paziente Scompenso Cardiaco (b) FAB, paziente LVAD (c) Inserisci parametri, paziente Scompenso Cardiaco (d) Inserisci parametri, paziente LVAD 81

4.3 (a) Contapassi (b) Segnala allarme (c) Chiamata numero di emergenza (numero censurato per privacy) (d) Monitor paziente LVAD aggiornato con tipologia di allarme e passi 82
4.4 (a) Menù di navigazione (b) Documentazione disponibile come PDF e immagini (c) Schermata per la scelta dell'applicazione (d) PDF 83
4.5 (a) Prima notifica quotidiana alle 16.00 (b) Seconda notifica alle 16.30 (c) Calendario (d) Visualizzazione di un giorno nel dettaglio 84
4.6 (a) Schermata Email (b) Invio della mail sull'applicazione scelta (c) Esempi di Grafici, Paziente Scompenso Cardiaco (d) Esempi di Grafici, Paziente LVAD 85

Introduzione

Le malattie cardiovascolari continuano ad essere, dal 2000, la prima causa di morte nel mondo [33]; nei paesi industrializzati sono inoltre la prima causa di scompenso cardiaco, patologia che riduce progressivamente l'autonomia del paziente fino a limitarlo anche nelle sue attività quotidiane. [9] Tale condizione è caratterizzata da un'elevata morbidità e mortalità, oltre a determinare un peggioramento della qualità di vita [30].

Il Policlinico Sant'Orsola-Malpighi di Bologna si occupa dello studio e della cura di tale patologia. Nel 2017 sono stati raccolti tutti i dati di pazienti con scompenso cardiaco avanzato (stadio gravato da un'elevata mortalità) ed inseriti nel protocollo REMEDIA-HF [30], approvato dal Comitato Etico AVEC.

Da tale archivio elettronico è stato effettuato, con l'unità operativa di psicologia clinica e sperimentale, uno studio sulla qualità di vita dei pazienti, dividendoli in due gruppi: pazienti con scompenso cardiaco avanzato e pazienti LVAD (Left Ventricular Assist Device) – ovvero dotati di dispositivi di assistenza ventricolare sinistra in grado di sostituire la funzione meccanica del cuore [30]. I risultati hanno presentato una discordanza tra condizione clinica e caratteristiche psicologiche. Nonostante i pazienti LVAD - cioè dotati di un dispositivo di assistenza ventricolare sinistro - avessero subito una chirurgia e avessero responsabilità maggiori riguardo la gestione del dispositivo, hanno presentato un profilo psicologico migliore rispetto ai pazienti con scompenso cardiaco con terapia medica [9].

Tale risultato mostra le conseguenze di due metodi differenti per seguire il paziente durante il suo decorso: il paziente LVAD viene seguito molto di più, poichè dotato di

monitoraggio remoto e telefonico che permette una comunicazione con il medico continuativa, oltre ad avere delle responsabilità di gestione del dispositivo da cui ne deriva una maggiore consapevolezza della propria condizione. Inoltre, viene seguito con controlli mensili e medicazioni settimanali.

Nasce quindi l'esigenza della creazione di uno standard per seguire allo stesso modo i due tipi di pazienti, con l'obiettivo finale di migliorare la qualità di vita dei pazienti con scompenso cardiaco avanzato ed offrire un monitoraggio a costo ridotto per quelli dotati di LVAD. Il Policlinico Sant'Orsola propone come mezzo per raggiungere tale obiettivo lo sviluppo di un'applicazione mobile. Tale scelta è motivata da una popolazione giovane di pazienti con scompenso cardiaco trattati al Policlinico - non superano i 65 anni - e da benefici tipici della telemedicina come la riduzione dei costi e la garanzia di un'azione preventiva continua.

L'obiettivo di questa tesi sarà dunque sperimentale: in seguito ad una discussione sullo stato dell'arte, ossia la letteratura nei riguardi dello sviluppo delle applicazioni mobili in generale e in contesti biomedicali, presenteremo e discuteremo in modo approfondito la progettazione e l'implementazione della nostra applicazione mobile come mezzo per la creazione di uno standard di monitoraggio remoto. Per concludere, mostreremo alcuni casi d'uso dell'applicazione con le relative schermate, come forma di validazione implicita del lavoro.

La tesi è pertanto divisa in 5 capitoli:

- Nel Capitolo 1 (Stato dell'Arte) introdurremo e discuteremo il tema delle applicazioni mobili: in una prima parte ci soffermeremo sulle tipologie e sui sistemi operativi disponibili; nella seconda ci concentreremo sul campo del Mobile Health, organizzando le applicazioni mobili appartenenti al settore in base alle funzionalità disponibili e ai loro intenti.

- Nel Capitolo 2 (Progettazione) presenteremo prima gli obiettivi del lavoro svolto in questa tesi, le specifiche ed i requisiti richiesti dal Policlinico; dedicheremo poi il resto del capitolo all'architettura dell'applicazione. In questa parte introdurremo

prima un'architettura di software design di riferimento, per poi adattarla ad un modello di software mobile. Infine, descriveremo tutte le funzionalità disponibili nell'applicazione.

- Nel Capitolo 3 (Implementazione) implementeremo l'architettura scelta, soffermandoci dapprima sulle tecnologie adottate. Proseguiremo presentando i dettagli implementativi di ciascuna componente architetturale, e mostreremo quindi il modo in cui ciascuna funzionalità è stata implementata. Nel corso del capitolo saranno introdotte le librerie utilizzate per implementare le componenti dell'architettura o determinate funzionalità.

- Nel Capitolo 4 (Validazione) presenteremo due simulazioni di un utilizzo dell'applicazione che raccolgono tutti i casi d'uso implementati. Nella prima simulazione il paziente interagisce con l'app per la prima volta; nella seconda, il paziente utilizza l'app da almeno due settimane (finestra temporale minima per l'uso di alcune funzionalità). In entrambi i casi, faremo distinzione tra le due tipologie di pazienti che presentano differenze a livello di funzionalità disponibili e operazioni da compiere.

- Nel Capitolo 5 (Conclusioni e Sviluppi Futuri) faremo un riepilogo di ciò che si è realizzato in questa tesi ed esporremo alcuni sviluppi futuri del lavoro.

Capitolo 1

Stato dell'arte

1.1 Applicazioni mobili

I dispositivi mobili appartengono ormai alla nostra realtà quotidiana. Nel 2019 si contano 3,2 miliardi di persone che utilizzano lo smartphone attivamente [4]. Circa il 93% della popolazione mondiale dispone di una connessione Internet [23]. Ad oggi Google Play conta quasi 2,9 milioni di applicazioni Android presenti nello store [28], mentre – secondo gli ultimi dati disponibili nel 2017 – l'Apple Store contiene 1,8 milioni di applicazioni [3]. Si stima che nel 2023 i guadagni mondiali del mercato delle app raggiungeranno i 935,2 miliardi di dollari statunitensi, aumentando di anno in anno (nel 2018 sono stati registrati 365,2 miliardi di dollari in guadagni) [12].

I dati mostrano una crescita continua dello sviluppo del settore, e aiutano a comprendere il passaggio - ormai avvenuto e consolidato - dal cellulare come semplice compositore di messaggi o dispositivo per chiamare a vero e proprio strumento poliedrico e dalle potenzialità smisurate.

I campi d'uso di uno smartphone comprendono il settore finanziario (come il mobile banking ed il mobile payment), il settore videoludico, ma anche servizi basati sulla geolocalizzazione ed in generale l'utilizzo di sensori hardware o virtuali (come l'accelerometro, il giroscopio, etc.). In particolare, grazie alla presenza di questi, si sono potute sviluppare applicazioni capaci di raccogliere dati e rielaborarli per diversi fini [20] (ad esempio il

context-awareness, che si specializza nella location awareness e nell'activity recognition [6]). Il campo investigato dalla tesi è quello del Mobile Health. Prima di analizzare il settore, riteniamo opportuno soffermarci velocemente sulle metodologie di sviluppo di applicazioni mobili e sui più importanti sistemi operativi attualmente disponibili.

Un'applicazione mobile è un software pensato e progettato per essere installato su un dispositivo mobile, quali smartphone e tablet [2]. Seppur simile in alcune caratteristiche ai software per desktop computer, presenta delle novità in termini di design, usabilità e user experience.

1.1.1 Approcci per lo sviluppo di app

Esistono tre approcci per lo sviluppo di un'applicazione mobile:

- Applicazione Nativa: viene eseguita nel sistema operativo del dispositivo, sarà quindi adattata al software su cui verrà installata. Lo sviluppo è realizzato attraverso uno specifico linguaggio di programmazione (Java o Kotlin per Android, Objective-C o Swift per iOS), utilizzando Application Programming Interfaces (API) e Software Development Kits (SDKs). I vantaggi di questo metodo riguardano la velocità di esecuzione dell'app, la possibilità di utilizzare le diverse funzionalità offerte dal dispositivo mobile – sensori, fotocamera, altre applicazioni, etc. -, la fedeltà e la coerenza con la piattaforma per cui è sviluppata e, nei riguardi del programmatore, la relativa semplicità di sviluppo attraverso una vasta documentazione, best practices e supporto dalla community. Negli svantaggi rientrano la frammentazione su più piattaforme – in parte risolto attraverso lo sviluppo di applicazioni cross-platform utilizzando piattaforme di sviluppo come Xamarin e Flutter -, la difficoltà nel riutilizzo e mantenimento del codice, l'alta spesa [1]

- Applicazione Web: è indipendente dal sistema operativo del dispositivo mobile, non necessita download ed installazione, è eseguita dal web browser presente nel dispositivo. È sviluppata utilizzando HTML, JavaScript e CSS, oltre ad una serie di framework di web application (come PHP). Tra i vantaggi troviamo la versatilità di utilizzo su più piattaforme – e quindi il costo e mantenimento ridotto – ed il

risparmio di memoria sul dispositivo. Lo svantaggio principale è la limitazione nell'utilizzo delle funzionalità del dispositivo mobile [1]

- Applicazione Ibrida: è installata sul dispositivo come applicazione nativa, ma viene eseguita utilizzando il browser del dispositivo. È in genere definita 'native-wrapped' mobile web application [1]. Il vantaggio più importante riguarda l'accesso alle funzionalità del dispositivo, lo svantaggio risiede in una performance non ottimale, dovuta all'uso del browser

1.1.2 Sistemi operativi

I due principali sistemi operativi delle applicazioni mobili, in termini di porzioni di mercato, notorietà del brand e avanzamento tecnologico, sono Android e iOS. Tutti gli altri sistemi operativi – come Windows Phone - insieme non raggiungono lo 0,1% di quota di mercato globale degli smartphone. [11]

- Android. Sistema operativo mobile di Google, è basato su un Kernel Linux e presenta un'architettura a più strati:
  - Linux Kernel: affidabile e sicuro, permette una facilità nella compilazione su diverse architetture hardware
  - Hardware Abstraction Layer (HAL): interfaccia che espone le funzionalità di basso livello del dispositivo con API di alto livello
  - Native C/C++ Libraries: librerie native scritte in C/C++ su cui molti componenti e servizi Android sono costruiti
  - Android Runtime: ambiente di runtime utilizzato da applicazioni e servizi di sistema
  - Java API Framework – set di funzionalità messo a disposizione attraverso un'API scritta in Java. Essenziale per lo sviluppo di applicazioni mobili in Android
  - System Apps: applicazioni standard del sistema operativo, come Calendario, Email, etc.

Una delle motivazioni per cui Android rappresenta il leader nel mercato globale degli smartphone, con una percentuale di share dell'86,6% nel Gennaio 2020 [11], è la presenza del livello di astrazione che funge da adattatore tra l'hardware di un produttore di dispositivi mobili ed il sistema operativo. L' "agnosticismo" di Android in relazione all'hardware, insieme alla sua natura open source, permette facilmente la sua adozione ed un rapido aggiornamento e miglioramento del software grazie al feedback continuo degli sviluppatori di applicazioni mobili.

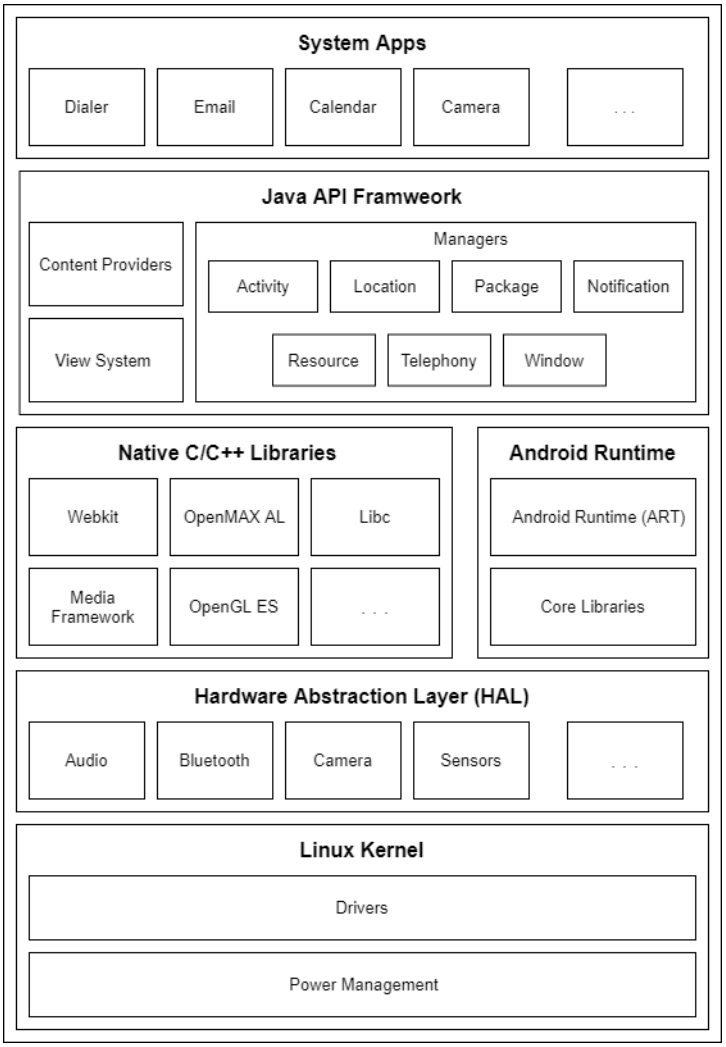


Figura 1.1: Architettura Android

- iOS. Sistema operativo mobile di Apple, basato su un Kernel XNU di Darwin. Anche la sua architettura è a strati:
  - Core OS: supporta funzionalità di basso livello come sicurezza, gestione della memoria, tecnologia bluetooth etc.
  - Core Services: servizi base ed essenziali come informazioni sulla localizzazione ed il movimento
  - Media: si occupa di funzionalità riguardanti animazione, grafica, audio e video
  - Cocoa Touch: ha il ruolo di incapsulare la parte software ed astrarla da quella hardware. Basato sul Model-View-Controller (MVC) design pattern, si occupa della gestione di funzionalità di alto livello come il multitasking e la gesture recognition [19]

  La natura proprietaria del sistema operativo e le famose politiche restrittive sullo sviluppo delle applicazioni in iOS contribuiscono al posizionamento del software al secondo posto nel mercato degli smartphone, con uno share del 13,4% nel Gennaio 2020 [11].

Entrambi costituiscono solo un punto di partenza nello sviluppo di un'applicazione mobile. Tale processo comprende più fronti, tra i quali: l'uso di determinate tecnologie, la scelta di modelli architetturali, ma anche aspetti più umani come la consapevolezza di chi utilizzerà l'applicazione e gli obiettivi che si vogliono raggiungere con la realizzazione del prodotto finale.

1.2 Applicazioni mobili in ambito biomedicale

1.2.1 mHealth

Il termine mHealth, abbreviazione di "Mobile Health", è stato coniato dal Professor Robbert S.H. Istepanian nel 2003, ed è definito come l'insieme dei mezzi di comunicazione mobili e delle tecnologie di rete per l'assistenza sanitaria [5]. La definizione non si limita solo alle applicazioni mobili, nonostante quest'ultime costituiscano una parte

fondamentale, dato il capillare uso di smartphones e tablet e le numerose funzionalità offerte da questi dispositivi. Si contano infatti più di 100.000 applicazioni negli store di Google ed Apple che promuovono la salute: alcune fungono come strumento di gestione e monitoraggio, altre informano l'utente riguardo malattie e rischi di vario genere [10]. Questi numeri non stupiscono, se si pensa che un'applicazione mobile è ottenibile in genere a basso costo – se non a costo zero – ed è accessibile ovunque ci si trovi, spesso anche offline.

In particolare, le applicazioni mobili in ambito biomedicale risultano essere:

- Discrete nel loro utilizzo in pubblico, se comparate ad altri strumenti di monitoraggio
- Comode, poiché permettono la gestione autonoma dei propri problemi sanitari
- User-friendly, perché semplici da utilizzare e richiedono un minimo investimento di tempo [29]

Nonostante queste caratteristiche e la consapevolezza che la salute sia la principale priorità nella vita dei pazienti, il problema tipico del settore è il mantenimento di alte percentuali di user retention, ossia l'uso continuativo dell'applicazione dopo averla installata sul proprio dispositivo (obiettivo perfettamente raggiunto da applicazioni social o di gaming, che raggiungono persino livelli di dipendenza). Tra il 30% ed il 40% di chi installa un'applicazione mobile per la propria salute continua ad utilizzarla in un periodo di sei mesi e più [25].

L'abbandono del loro utilizzo è dovuto a due fattori: il primo impatto del paziente con l'applicazione (facilità d'uso, comprensibilità, chiarezza negli obiettivi dell'applicazione, etc.) e la persistenza del paziente nei riguardi dei propri obiettivi per la sua salute [35]. Solo una relativamente bassa percentuale di applicazioni mobili in ambito biomedico risulta occuparsi in modo adeguato di tali fattori, permettendo all'utente un'esperienza piacevole e che facilita la trasformazione dell'applicazione in un "compagno" nella propria vita quotidiana.

1.2.2 Funzionalità

Per analizzare le applicazioni mobili nel mHealth, disponibili negli store o semplicemente sviluppate negli ultimi anni a fini di ricerca, è bene partire dalle sette principali funzionalità:

- Record: registrazione di parametri e informazioni da parte del paziente
- Display: elaborazione dei parametri e visualizzazione grafica (es.: lista, grafico)
- Inform: presentazione di informazione sotto vari formati
- Instruct: fornire istruzioni al paziente
- Remind: creazione di alert o notifiche con funzione di promemoria (es.: trattamento, medicine da prendere)
- Guide: elaborazione di diagnosi o trattamenti consigliati in base ai dati inseriti dal paziente
- Communicate: creazione di un canale comunicativo tra responsabili del servizio (medici, operatori sanitari) e paziente [5]

1.2.3 Obiettivi

Due applicazioni, pur offrendo le stesse funzionalità, possono distinguersi anche in base alla specificità degli obiettivi che si pongono. Ad esempio, alcune mobile health apps si concentrano sulla cura "a tutto campo" del paziente: si parla di well-being, self-care, ossia l'insieme di azioni che una persona compie per curare la propria salute fisica, mentale ed emotiva [5]. Possiamo accostare a questa categoria di app quella più specifica di applicazioni che si occupano di aspetti comportamentali come cattive abitudini da eliminare.

Dall'altra parte troviamo applicazioni mobili in campo biomedicale che hanno l'obiettivo di gestire una malattia o un disturbo del paziente, qualsiasi essa sia. Mentre quest'ultime si limitano a riprendere le maggiori funzionalità presenti nelle mHealth app,

specializzandosi nel campo d'interesse della patologia, le prime tendono ad implementare tecniche di gamification – inteso come l'inserimento di elementi grafici di gioco in contesti non ludici - durante lo sviluppo dell'app. L'impiego di livelli con premi finali e la creazione di un ambiente di gioco in cui l'utente può sentirsi stimolato a compiere azioni effettivamente difficili da un punto di vista psicologico – come preferire una scelta sana ad una pericolosa – risultano essere una strategia efficace per aumentare la fiducia in sé stessi [21].

Un esempio è l'applicazione Quit Genius, una mobile health app che rappresenta una versione digitalizzata di un programma di terapia cognitivo comportamentale, che in un sondaggio effettuato nel 2017 è riuscita con successo a far abbandonare al 36,3% dei suoi partecipanti (69/190) la cattiva abitudine di fumare [21].

Figura 1.2: Esempi di schermate di Quit Genius [21].

Le applicazioni mobili in campo biomedicale che gestiscono una malattia o un disturbo sono molto variegate. Solo per fare degli esempi, si segnalano:

- App per la gestione del disturbo bipolare – che comprende, tra le funzionalità: dare informazioni, monitorare e gestire il disturbo, offrire un supporto della community [27]

- App per pazienti con fibrosi cistica – l'applicazione comprende la funzionalità di reminder, il monitoraggio di parametri quotidiani e la possibilità di contattare il personale medico in caso di emergenza [31]

Infine, per avvicinarci al campo d'interesse dell'applicazione progettata e sviluppata in questa tesi, riteniamo opportuno citare forse l'app che più si avvicina a quanto realizzato, ossia MobileHeart, che ha l'obiettivo di monitorare pazienti con malattia coronarica durante il loro periodo di riabilitazione. Le funzionalità principali sono quella di controllare il paziente, offrire un modulo di apprendimento per aumentare la sua conoscenza del problema, e fornire una comunicazione a due vie tra il paziente e l'infermiere. È inoltre possibile ottenere delle prescrizioni su misura per il paziente, elaborate in seguito all'inserimento dei suoi dati personali (manualmente o tramite dispositivo portatile esterno) [16].

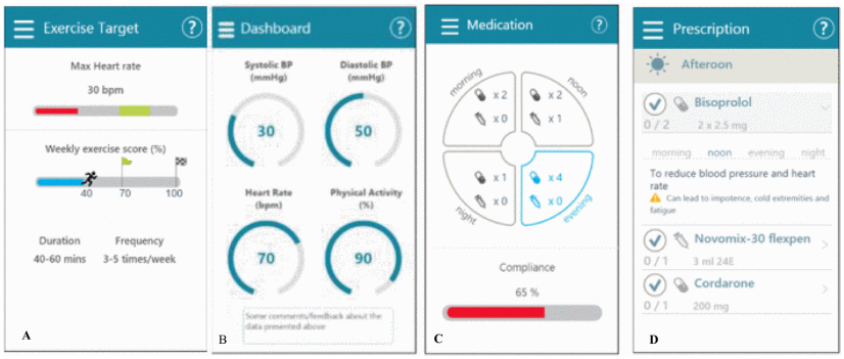


Figura 1.3: Esempi di schermate di MobileHeart [16].

L'analisi di questo capitolo considera le applicazioni mobili presenti nel mercato e non a livello globale, ma anche in Italia la situazione è analoga. Le funzionalità delle app presenti sul territorio italiano rientrano nelle sette segnalate in questo capitolo, e possono specializzarsi in un ambito medico o concentrarsi sullo stato di benessere del paziente. Di notevole impatto l'esperienza dell'ospedale di Trento e del suo sistema di raccolta dati chiamato TreC (Cartella Clinica del Cittadino), un unico backbone centrale a cui collegare mHealth apps con obiettivi diversi, come un'applicazione per il reparto di oncologia [17].

Risulta chiaro come il mondo del mHealth, seppur relativamente nuovo, presenti già delle basi promettenti per offrire ai pazienti un supporto al di fuori delle mura ospedaliere. In particolare evince da quest'analisi la necessità, durante lo sviluppo di un'applicazione, di una maggiore attenzione al punto di vista del paziente, fattore importante nella decisione finale di continuare o meno ad utilizzare l'applicazione dopo il primo utilizzo.

Capitolo 2

Progettazione

2.1 Obiettivi

Il lavoro svolto in questa tesi copre un'area del Mobile Health in parte esplorata: la gestione di pazienti con problemi cardiaci. Come abbiamo analizzato nel capitolo precedente, nonostante una crescita nel numero di applicazioni mHealth disponibili nel mercato e uno sviluppo costante della letteratura sull'argomento, sussiste il problema di riuscire a realizzare un'app che continui ad essere utilizzata dai pazienti in modo continuo.

Ciò che presentiamo in questa tesi è dunque un'applicazione mobile per l'assistenza di due tipologie di pazienti: quelli con scompenso cardiaco, e quelli portatori di LVAD. In particolare, l'applicazione verrà inizialmente utilizzata all'interno del Policlinico S.Orsola-Malpighi di Bologna, per cui sarà personalizzata adattandosi alle pratiche presenti nel reparto di cardiochirurgia e alle esigenze di relativi medici e pazienti.

L'applicazione rientra nella categoria di app che ricopre un settore specifico dell'ambito biomedicale, offrendo un set di funzionalità il più completo possibile e assistendo il paziente nella gestione della sua patologia.

Lo sviluppo ha quindi un duplice scopo: realizzare un'applicazione pilota funzionante, rapida e solida dal punto di vista progettuale e implementativo, facendo uso di best

practices e principi di design; porre il paziente al centro dello sviluppo per facilitargli l'utilizzo dell'applicazione prodotta.

Dedicheremo dunque questo capitolo alla discussione delle specifiche e dei requisiti richiesti dai medici del Policlinico, per poi esplorare le nostre decisioni architetturali, soffermandoci sulle componenti scelte e sulle funzionalità presenti.

2.2 Specifiche

Prima ancora di delineare un'architettura dell'applicazione, abbiamo intrapreso una serie di incontri con i responsabili del progetto al Policlinico per discutere sui requisiti necessari ed essenziali per il raggiungimento degli obiettivi.

In una prima fase abbiamo stabilito dei criteri a cui l'applicazione deve aderire: il paziente necessita di uno strumento facile da utilizzare e intuitivo nelle funzioni. Chi utilizza l'app non deve aver bisogno di una guida o un manuale, ma le varie schermate dovranno essere sufficientemente chiare per guidare in autonomia il paziente.

Questo requisito diventa ancora più importante nel momento in cui una percentuale dei fruitori dell'app si colloca nella fascia alta d'età, considerando che una persona anziana trova generalmente ancora difficoltà a rapportarsi con la tecnologia, in particolare con quelle mobili [24]. Ogni aspetto – design, funzionalità, interazione – andrà sviluppato tenendo presente queste considerazioni.

Chiarito questo requisito, abbiamo parlato dell'importanza dell'invasività da tenere al minimo durante lo sviluppo. Specialmente nel campo delle mHealth applications, qualsiasi elemento che possa generare insoddisfazione e desiderio di abbandono dello strumento va eliminato o quanto meno ridotto il più possibile.

Un esempio di applicazione invasiva è quella che visualizza, ripetutamente, pubblicità di vario tipo durante il suo utilizzo, interrompendo la normale esperienza dell'utente. Nel nostro caso, poiché il software è ovviamente privo di meccanismi di pubblicità, ci siamo concentrati sulla funzionalità di ricordare al paziente di compiere alcune operazioni tramite notifica, talvolta ritenuta fastidiosa se esagerata. Abbiamo così limitato la

frequenza delle notifiche durante l'implementazione - come vedremo meglio nel prossimo capitolo -, pur mantenendo la funzione di reminder per il paziente.

Un terzo aspetto molto importante, che andrà sicuramente rivisto e ampliato nelle fasi successive del progetto, riguarda la privacy. Oltre ad essere presente in letteratura una serie di studi ed analisi sui problemi di privacy e legalità durante lo sviluppo di mHealth apps [36], durante gli incontri ci siamo concentrati prima di tutto sull'implementazione di un meccanismo di protezione per il paziente e sulla sua percezione di sicurezza riguardo l'inserimento dei propri dati clinici all'interno dell'app.

Dopo aver concordato su questi elementi, abbiamo deciso a grandi linee cosa non sarebbe potuto mancare nell'applicazione pilota – in ambito di sviluppo software, si parla di Functional Requirements -. Al termine del lavoro di tesi, l'app potrà:

- Raccogliere i dati del paziente (funzionalità Record)
- Visualizzare - in vari formati - e modificare i dati raccolti (funzionalità Display)
- Ricordare al paziente di inserire i dati della giornata (funzionalità Remind)
- Esportare un report dei dati inseriti nel tempo (funzionalità Record)
- Inviare i dati al medico (funzionalità Communicate)
- Chiamare un numero di emergenza in caso di allarme (funzionalità Communicate)
- Fornire documenti relativi al disturbo ed eventualmente al dispositivo LVAD (funzionalità Inform)

Un dettaglio che non è stato sottovalutato durante gli incontri è l'adattabilità dell'applicazione mobile al tipo di paziente - scompenso cardiaco o portatore di LVAD - che la utilizzerà. Occorre progettare e implementare il software in modo tale che entrambe le tipologie abbiano tutti gli strumenti necessari per usufruire al meglio delle funzionalità disponibili.

2.3 Architettura

Individuati gli obiettivi e discussi i requisiti, il prossimo passo nello sviluppo di un software consiste nella scelta della sua architettura. Un'architettura robusta è ciò che permette ad un software di essere sviluppato, implementato e mantenuto facilmente.

Per il lavoro di questa tesi abbiamo scelto come punto di riferimento la Clean Architecture [22], un'architettura che segue i principi della programmazione ad oggetti. I primi cinque, che riguardano il design delle classi, sono conosciuti come i SOLID principles, e sono quelli a cui faremo riferimento in questo capitolo:

- Single responsibility principle: una classe può avere una sola responsabilità.
- Open/closed principle: una classe è aperta alle estensioni, ma chiusa alle modifiche. Ciò significa che può estendere il proprio comportamento senza modificare quello già esistente.
- Liskov substitution principle: una classe derivata può essere sempre sostituita ad una classe da cui deriva.
- Interface segregation principle: una classe deve dipendere solo da multiple e specifiche interfacce, non da una generica.
- Dependency inversion principle: una classe deve dipendere da astrazioni, non implementazioni.

Seguiamo queste best practices anche a livello di componenti: se una componente (o un modulo) di alto livello dipende direttamente da un'altra componente di basso livello, inseriamo un grado di separazione tra loro effettuando una Dependency Inversion. Nella pratica, inseriremo un'interfaccia che implementerà la componente di basso livello, e da cui dipenderà la componente di alto livello (Setting Boundaries). Questo rende inoltre ciascuna componente in grado di adempiere ad un solo compito – adoperando una Separation of Concerns. Le componenti risultano ora indipendenti, e possiamo quindi organizzarle secondo una struttura a strati.

La struttura risulta concentrica, con gli strati esterni di basso livello (instabili, tendenti a cambiare, dipendenti dalle tecnologie utilizzate) e quelli interni di alto livello (stabili e astratti, costituiscono il core del software).

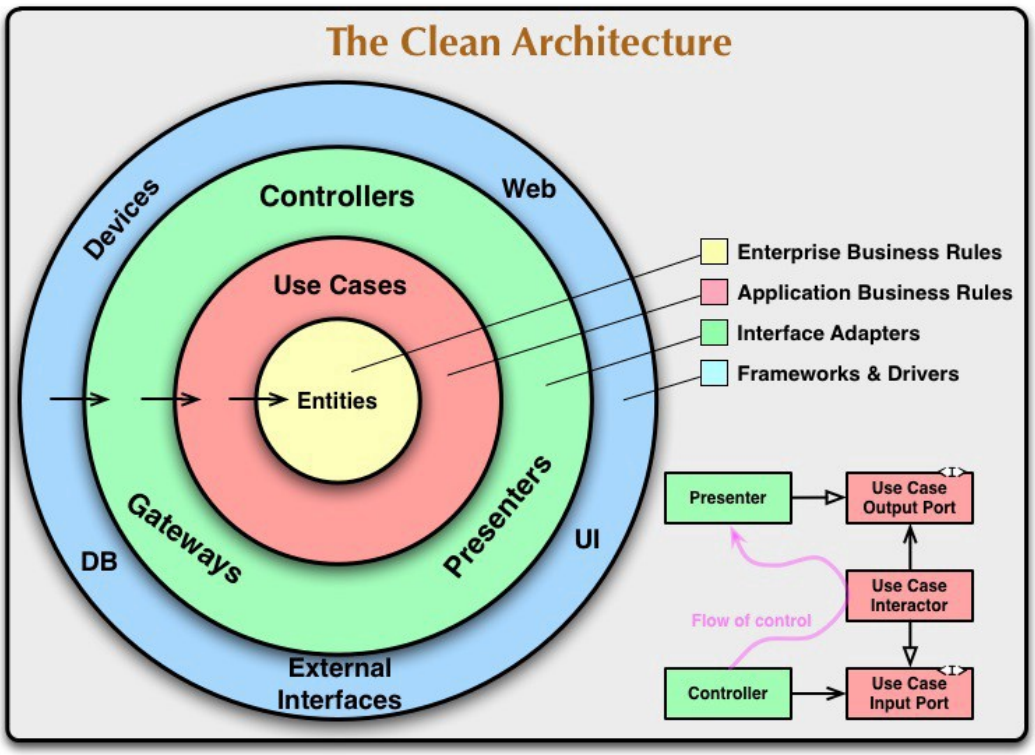


Figura 2.1: Clean Architecture, struttura concentrica [22]

Identifichiamo quattro strati, qui elencati dall'interno verso l'esterno:

- Entities: elementi critici nello sviluppo dell'applicazione. La particolarità risiede nell'indipendenza di tali elementi e dalla loro astrazione dal resto della struttura. Ad esempio, il Policlinico S.Orsola-Malpighi potrebbe utilizzare gli oggetti contenuti in questo strato per più applicazioni mobili.
- Use Cases: business rules a livello dell'applicazione. Organizzano il flusso di dati diretto e proveniente dalle entità, facendo in modo che vengano applicate le business rules specifiche dell'applicazione. Costituisce in pratica la business logic.

- Interfaces Adapters: interfacce che trasformano il flusso di dati nel modo più conveniente possibile in relazione allo strato ricevente. Se, ad esempio, vogliamo passare dei dati estrapolati da un database (ultimo strato, basso livello) allo strato delle entità e degli use cases (strati interni, alto livello), sarà compito delle interfacce trasformare i dati in un formato conveniente per le entità. Lo stesso vale per il flusso di dati nella direzione contraria. Questo permette l'astrazione dello strato interno dal modo in cui viene implementato lo strato esterno

- Frameworks & Drivers: insieme di strumenti, framework, tecnologie di basso livello. Qui risiedono la parte grafica, il database, il network etc.

Le relazioni tra strati sono delineate nella Dependency Rule, che afferma che le dipendenze a livello di codice si sviluppano dall'esterno verso l'interno. Uno strato può avere dipendenze solo con il suo strato successivo più interno. Uno strato di alto livello non saprà nulla degli strati di più basso livello.

L'efficacia di tale architettura risiede in quest'ultima regola: avendo una separazione totale tra componenti logiche del codice (modularizzazione delle componenti), siamo in grado di modificare a nostro piacimento le tecnologie, i framework, le implementazioni da noi scelte senza intaccare il core del nostro software. Un'importante conseguenza di ciò è che la business logic può essere testata indipendentemente dalla tecnologia implementata. In questo modo saremmo anche facilitati nel caso volessimo "trasportare" la nostra applicazione nativa da un sistema operativo ad un altro, poiché potremmo mantenere intatti gli strati più interni (Entities e Use Cases).

Un altro modo di vedere l'architettura è suddividendola in tre generici strati:

- Presentation Layer: strato che si occupa di gestire il modo in cui i dati vengono visualizzati e di catturare eventi generati dall'utente che produrranno determinati risultati (es.: invio di nuovi dati). Genericamente si tratta dell'interfaccia grafica

- Domain Layer: strato in cui vengono applicate le business rules, ossia tutte le regole a cui i dati devono adattarsi. Al suo interno i dati vengono processati e modificati in modo da ottenere un risultato conforme alle business rules

- Data Layer: strato in cui risiedono i dati e da cui è possibile accedervi

L'ordine della lista non è dato al caso, ma è rappresentativo della "posizione" degli strati. Ruotando la figura 2.1 di 90 gradi (in senso antiorario), otterremo, come viene mostrato nella figura 2.2, una suddivisione in tre strati, con al centro il Domain Layer ed ai lati gli altri due.

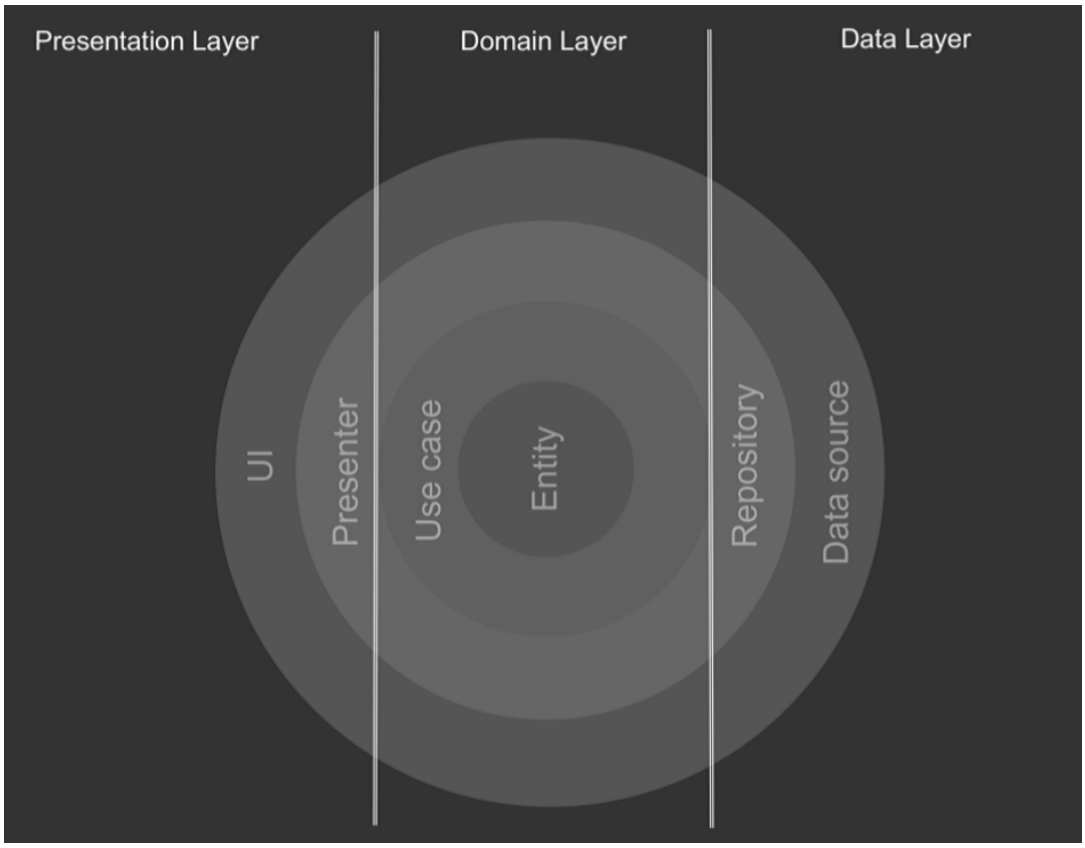


Figura 2.2: Clean Architecture, ruotata per identificare i tre strati

Questa struttura indica, se percorsa da sinistra verso destra, la direzione del flusso di dati dall'utente alla data source e viceversa.

Ma la stessa struttura rappresenta anche la Dependency Rule, poiché sia il Presentation Layer che il Data Layer dipendono dal Domain Layer. Per fare un esempio, nel

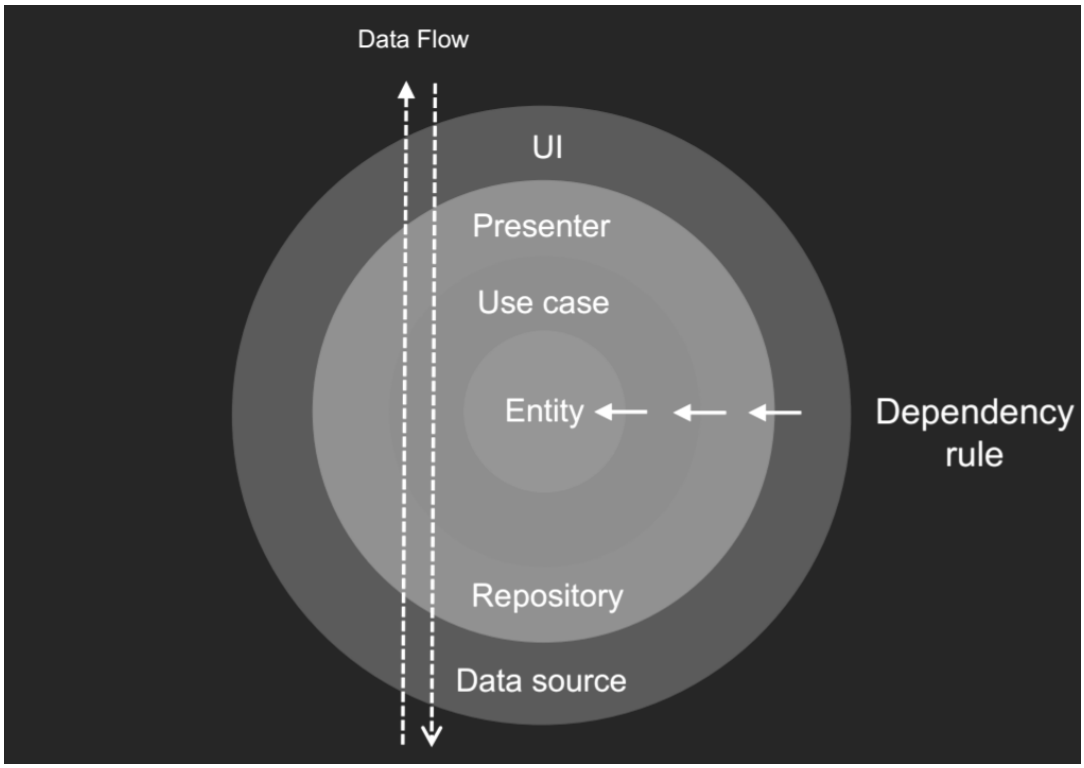


Figura 2.3: Clean Architecture, data flow.

primo caso il Presentation Layer invia dati inseriti dall'utente al caso d'uso specifico, che trasforma i dati nel modo più conveniente possibile per lo strato più interno (Entities), e nel secondo il Data Layer contiene i dettagli implementativi delle entità (astratte) contenute nel Domain Layer. Quest'ultimo non dipende da altri livelli perché costituisce lo strato più interno.

Ad un primo sguardo si potrebbe pensare che il Domain Layer dipenda dal Data Layer in quanto gli Use Cases, per svolgere la business logic, devono utilizzare – e quindi dipendere – dal meccanismo di gestione dei dati (es.: attraverso il Repository Pattern). Tale ragionamento è però sbagliato: gli Use Cases non sono dipendenti perché contengono l'interfaccia del meccanismo di gestione (dependency inversion principle), e ciò li rende indipendenti da qualsiasi implementazione si scelga.

Sintenticamente, il Presentation Layer si occupa solo di catturare un evento e la risposta all'evento; il Domain Layer si occupa solo di cosa deve succedere in seguito

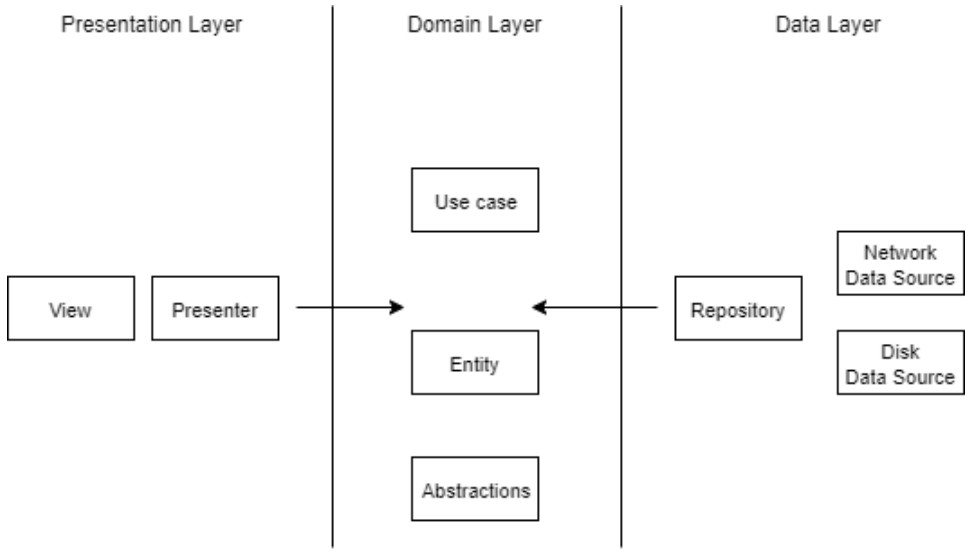


Figura 2.4: Clean Architecture come tre strati, Dependency Rule.

all'evento; il Data Layer si occupa solo di come deve succedere quel qualcosa imposto dal Domain Layer.

Abbiamo così un'architettura modulare, con gli strati più interni indipendenti, astratti e facilmente testabili. Così facendo sappiamo come il nostro software funzionerà (strati interni), senza interessarci di cosa utilizzeremo per farlo funzionare (strati esterni).

Il prossimo passo nello sviluppo dell'applicazione mobile è mettere in relazione quest'architettura generica – cioè implementabile per qualsiasi tipo di software – alle componenti tipiche di un'applicazione mobile. Proseguiremo descrivendo il processo di adattamento della Clean Architecture in un contesto di mobile software development.

2.3.1 Adozione dell'architettura

Per il lavoro di questa tesi abbiamo preso come riferimento l'architettura precedentemente descritta - modificata in base alle nostre esigenze e possibilità - e l'abbiamo combinata con il Model-View-ViewModel design pattern (MVVM), trovando numerosi punti di contatto e sovrapposizione tra le due architetture.

Il MVVM presenta tre componenti:

- Model: implementazione del Domain Layer. Comprende entità, business logic, Repository e data sources. Nella nostra applicazione l'unica data source è rappresentata da un database locale

- View: implementazione del Presentation Layer. Rappresenta l'interfaccia grafica (UI), lo strato con cui interagisce l'utente

- ViewModel: implementazione del Presentation Layer e intermediario tra Model e View. Gestisce lo stato della View. Da una parte riceve i dati provenienti dalla View per inviarli al Model nel formato corretto, dall'altra riceve i dati dal Model e li espone – anche qui dopo averli formattati correttamente – alla View [8]

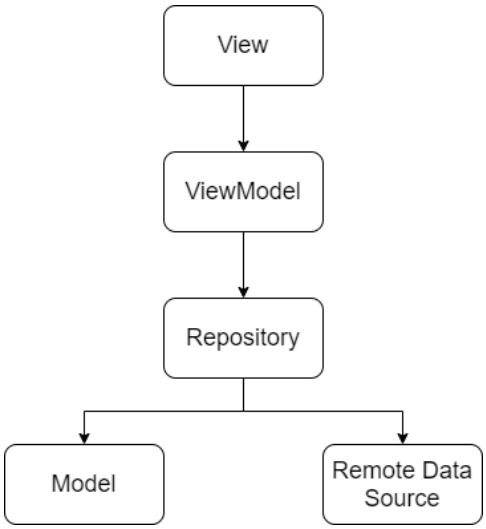


Figura 2.5: I componenti del MVVM.

La principale differenza con il Model-View-Presenter (MVP) è che, nonostante entrambi creino un'astrazione tra stato e comportamento della View, il ViewModel introduce un'indipendenza totale dalla View che lo utilizza.

Un dettaglio da sottolineare è, come si evince dalla figura 2.5, la presenza del Repository Pattern come intermediario tra il ViewModel e le data sources, creando così una Single Source of Truth.

Tale architettura permette ad ogni strato di essere dipendente solo dallo strato che si trova ad un livello più in basso.

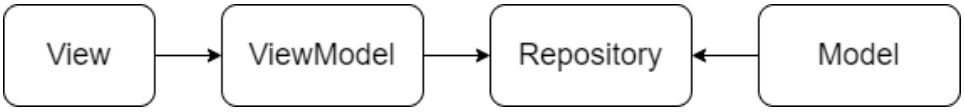


Figura 2.6: MVVM, Dependency Rule.

Il flusso dei dati è corrispondente a quello della Clean Architecture e la Dependency Rule è rispettata, in quanto View e ViewModel (Presentation Layer) dipendono dal Domain Layer, e lo stesso dicasi per il Model (Data Layer).

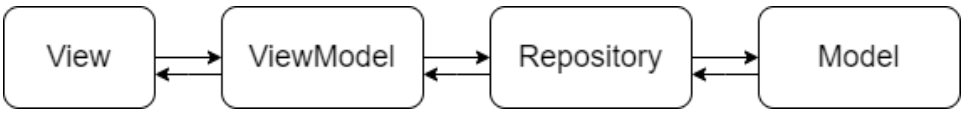


Figura 2.7: MVVM, flusso dei dati.

In questo modo, qualsiasi sia l'implementazione scelta per il Presentation Layer e per il Data Layer, avremo la certezza che il nucleo dell'applicazione sarà comunque funzionante - perché indipendente -, permettendo di testare in isolamento le prime due componenti e di modificarle senza il timore che un errore possa generare un effetto a cascata in tutta l'architettura.

Il risultato finale è quindi una struttura solida, decoupled, flessibile e modulare.

2.3.2 Funzionalità

Partendo dalle cinque macro funzionalità richieste nelle specifiche – Record, Display, Remind, Communicate, Inform - possiamo delineare l'insieme delle funzionalità effettivamente implementate nella nostra applicazione mobile. Fa eccezione la funzionalità di Autenticazione (rappresentata dalla Registrazione e dal Login), non appartenente ad alcuna delle cinque macro funzionalità, ma implementata come risposta a problematiche di sicurezza e privacy. Tale meccanismo è attualmente basato su un database locale.

Ciascuna funzionalità può essere considerata uno Use Case, poiché indipendente dalla sua implementazione (quindi dai framework e dalle librerie utilizzate), e appartenente al Domain Layer. Lasciando da parte i dettagli implementativi per il prossimo capitolo, procediamo con una descrizione di alto livello delle funzionalità individuate:

- Registrazione: il paziente inserisce tre tipologie di dati: dati identificativi (es. nome, cognome, etc.), dati "monitor" (es. patologia di base, tipologia paziente) e dati "account", ossia email e password
- Login: il paziente inserisce email e password scelti alla registrazione. Se corrispondono, può entrare nell'applicazione
- Monitor (Display): il paziente visualizza i suoi dati identificativi ed i parametri inseriti nel giorno corrente
- Inserisci Parametri (Record): il paziente inserisce un set di parametri quotidiani, che si differenziano in base alla tipologia di paziente (paziente Scompenso Cardiaco o paziente LVAD) e, se presente, in base al dispositivo LVAD (HM3 o HVAD)
- Contapassi (Record): il paziente visualizza i passi compiuti nel giorno corrente
- Allarme (Record, Communicate): registra una variazione pericolosa di alcuni parametri (nel caso di paziente LVAD) e chiama un numero di emergenza
- Calendario (Display): il paziente visualizza una lista di parametri, divisi e raggruppati per il giorno di inserimento. È possibile selezionare ciascun giorno per visualizzarne i dettagli
- Visualizza Giorno (Display): il paziente visualizza nel dettaglio i parametri inseriti, insieme ai passi compiuti e ad eventuali allarmi segnalati (solo nel caso di LVAD), nel giorno selezionato
- Documentazione (Inform): il paziente visualizza una serie di documenti PDF di carattere informativo
- Invia Report (Communicate): se il paziente registra i parametri quotidiani da almeno due settimane, può esportare il report bisettimanale ed inviarlo al medico

- Grafici (Display): il paziente visualizza dei grafici a linee della variazione di determinati parametri nel tempo
- Notifica (Remind): l'applicazione invia un reminder alle ore 16.00 al paziente che non ha inserito tutti i parametri quotidiani. Nel caso in cui il paziente ignori la notifica, l'app continuerà ad inviarne altre tre a distanza di mezz'ora

Tutte le funzionalità dell'applicazione mobile sono implementabili seguendo l'architettura sopra descritta. Per fare un esempio, la funzionalità "Inserisci Parametri" comprende un flusso di dati che parte dal Presentation Layer (View, poi ViewModel), prosegue verso il Domain Layer (Use Cases), per arrivare al Data Layer (Repository, poi Database). Il paziente inserisce i dati attraverso la View, il ViewModel formatta i dati inseriti in un formato conveniente per lo Use Case di riferimento. Questo elabora i dati in modo tale da generare una struttura dati adatta per la Repository, che a sua volta la registra nel database locale. Anche la Dependency Rule è rispettata (View e ViewModel dipendono dal Domain Layer, così come per il Model).

Nel prossimo capitolo parleremo di come effettivamente sono state implementate tali funzionalità, per arrivare ad avere una visione completa di tutta l'applicazione mobile sviluppata, dagli aspetti di alto livello a quelli di basso livello.

Capitolo 3

Implementazione

3.1 Tecnologie

Per il lavoro di questa tesi abbiamo scelto Android come piattaforma di sviluppo e sistema operativo su cui basarci, tenendo presente che con la progettazione discussa nel capitolo precedente saremmo in grado di sviluppare la nostra applicazione indipendentemente dalle tecnologie usate.

Le motivazioni principali di tale scelta sono state: la vasta documentazione messa a disposizione da Google, una maggiore personalizzazione nell'implementazione di Android rispetto ad iOS - data la sua natura Open Source - e l'adozione del software da parte della stragrande maggioranza dei produttori di dispositivi mobili.

Abbiamo usato Android Studio come ambiente di sviluppo e Gradle come builder. Il codice è scritto e compilato per dispositivi che utilizzano Android10, corrisponde all'API 29 (Target SDK e Compile SDK coincidono), tuttavia l'applicazione è funzionante su dispositivi che adottano una versione di Android successiva o corrispondente a Lollipop 5.0 (API 21). Ad oggi, circa l'89,3% dei dispositivi Android in tutto il mondo sarebbe in grado di installare la nostra applicazione [15]. Siamo soddisfatti di tale percentuale e preferiamo evitare una Minimum SDK precedente all'API 21, poichè l'applicazione utilizza funzionalità introdotte in questa release (es.: Material Design, accesso alle notifiche dalla schermata di blocco).

Abbiamo anche usato una serie di librerie e framework fondamentali per realizzare le funzionalità richieste. Prima di presentarle e discuterle nel dettaglio, vogliamo soffermarci sull'utilizzo del linguaggio di programmazione Kotlin invece che Java. Entrambi i linguaggi presentano dei vantaggi l'uno rispetto all'altro, ma abbiamo preferito il primo per alcune delle sue caratteristiche:

- La possibilità di gestire le null references direttamente dal Type System (Null-safety)
- L'invarianza degli array
- La sua concisione e la conseguente riduzione di codice necessario per l'implementazione
- Una serie di funzionalità non presenti in Java: lambda expressions, inline functions, extension functions, smart casts, string templates, data classes e coroutines [13]. Avremo modo di presentarle più accuratamente durante il capitolo.

Inoltre, Android Studio è in grado di convertire automaticamente il codice Java in Kotlin, funzione comoda nei casi in cui la documentazione presenta soltanto codice in linguaggio Java. In ogni caso, Kotlin è pensato per avere un'interoperabilità con Java: è infatti possibile utilizzare codice o librerie Java in una classe Kotlin. Ciò è stato fatto, per esempio, durante l'implementazione della funzionalità di esportazione del report.

Durante lo sviluppo è stato largamente utilizzato Android Jetpack, una collezione di librerie messa a disposizione da Android, che contiene anche numerosi strumenti e best practices da scegliere in base alle proprie esigenze. Ha permesso, inoltre, la semplificazione di codice e l'utilizzo intelligente di risorse. Un esempio è la sua libreria Android KTX, un set di estensioni di Kotlin per Android.

Naturalmente, presenteremo nel resto del capitolo tutte le librerie ed i framework utilizzati durante l'implementazione.

3.2 Componenti dell'architettura

La differenza principale tra un software desktop ed un software mobile risiede nella mancanza di un entry point specifico ed unico, un unico "main()" da cui il sistema può entrare. In Android, infatti, la struttura è più complessa: esistono più punti di entrata, in cui ogni componente può istanziare degli oggetti e può svolgere delle funzionalità ad essa confinate.

Una delle componenti principali è l'Activity, che fornisce all'utente uno spazio per compiere un'attività. Ciascuna Activity implementata può essere aperta da un'altra o - se ne ha il permesso - da un'Activity proveniente da un'altra applicazione. Inoltre, vi è una MainActivity che si occupa di gestire il flusso dell'applicazione. In questo modo un'applicazione potrà, per esempio, richiedere la funzionalità dell'Activity dell'app "Fotocamera", che gestisce l'uso dell'hardware del dispositivo per scattare delle fotografie, senza che l'applicazione richiedente sviluppi personalmente quella funzionalità. Questo rende il software mobile flessibile e indipendente.

La caratteristica principale di un'Activity, oltre a quelle già discusse, è la possibilità di entrare in diversi stati durante il suo ciclo di vita (lifecycle). Gli stati principali di tale componente sono riconducibili a:

- Active State: l'interfaccia grafica è visibile e l'utente può svolgere operazioni all'interno della componente
- Paused State: l'interfaccia grafica non è interagibile perché generalmente vi è un elemento sovrapposto ad essa (pop-up, dialog, etc.), ma è comunque visibile
- Stopped State: l'Activity entra in tale stato nel momento in cui clicchiamo il tasto Home o entriamo in un'altra applicazione. L'Activity non è distrutta, ma non è visibile l'interfaccia grafica né è possibile interagirci
- Destroyed State: il sistema distrugge l'Activity, perché richiesto da noi o da un'esigenza di memoria

Nel ciclo di vita dell'Activity sono presenti una serie di stati di transizione tra uno stato e l'altro e relativi metodi callbacks per gestirne al meglio il passaggio. Nello sviluppo

della nostra applicazione, la gestione del ciclo di vita è stata facilitata grazie a strumenti di Android Jetpack come i LiveData ed il ViewModel, di cui parleremo a breve.

Presentiamo adesso l'implementazione dell'architettura, eseguita secondo una logica consistente, in cui ogni componente dipende solo dal componente di un livello più in basso.

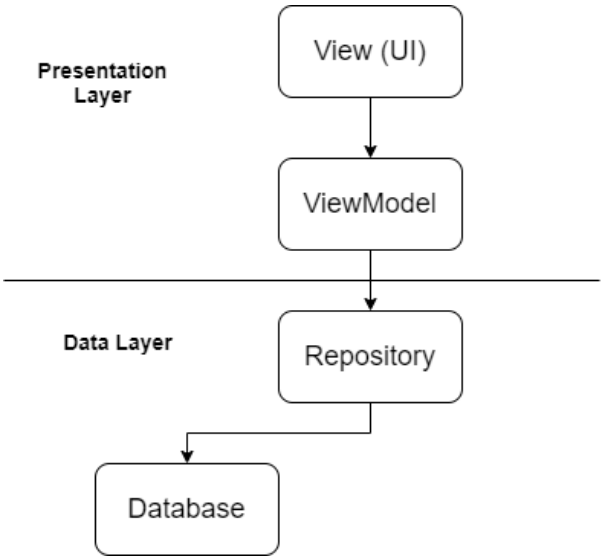


Figura 3.1: Implementazione del MVVM.

Nella figura 3.1, il Presentation Layer è rappresentato dalla coppia UI-ViewModel ed il Data Layer dalla coppia Repository-Database.

3.2.1 Interfaccia grafica

Nell'implementazione del Presentation Layer, corrispondente alla parte View-ViewModel del nostro modello di design, abbiamo scelto di utilizzare una sola Activity (MainActivity) - per la gestione della navigazione - ed una serie di Fragment, uno per ogni funzionalità dell'applicazione.

Un Fragment è una componente che appartiene ad un'Activity, è cioè dipendente dal ciclo di vita di quest'ultima. Ha tuttavia un ciclo di vita che si discosta leggermente da quello dell'Activity, ma che è approssimativamente sovrapponibile. Se un'Activity entra

in Paused State, anche il Fragment non è più interagibile, ma se un Fragment entra in Paused State, non è detto che lo sia anche l'Activity a cui fa riferimento.

La scelta di tale implementazione è dovuta alla flessibilità ed al dinamismo che offre il Fragment in termini di UI Design, oltre alla sua modularità e alla sua capacità di essere riutilizzato in più occasioni. Testare un Fragment, poi, è ancora più semplice che testare un'intera Activity.

Il Fragment è l'UI Controller nella nostra applicazione. Banalmente si potrebbe pensare di inserire la logica dei dati all'interno del Controller, ma questo porterebbe una serie di problemi non di poco conto. Innanzitutto, il framework Android gestisce il ciclo di vita del Fragment e dell'Activity, e potrebbe decidere di distruggere o ricreare il Controller, perdendo quindi la logica elaborata fino a quel punto. Anche un banale cambio della visuale da Portrait (verticale) a Landscape (orizzontale) della View – operazione chiamata Configuration Change - comporterebbe una tale perdita. Un altro problema è la gestione asincrona di chiamate al database o ad altre fonti di dati, che porterebbe a dei leaks nel momento in cui la View non è più disponibile per ricevere la callback, perché distrutta o ricreata nel frattempo.

Per ovviare a questi problemi e per seguire la best practice di mantenere l'UI Controller (sia l'Activity che i suoi Fragment) il più leggero possibile in termini di codice e responsabilità, accostiamo a ciascun Fragment un suo ViewModel, che si occuperà della logica dei dati. L'unica responsabilità dell'UI Controller è infatti quella di aggiornare la View al cambiare dei dati, o di avvisare l'arrivo di eventi da parte dell'utente.

Il ViewModel è una classe fondamentale nel nostro sviluppo dell'applicazione. Tale classe, appartenente alle componenti definite Lifecycle-Aware, sopravvive ai Configuration Changes, mantenendo un unico stato durante il ciclo di vita del Fragment, fino alla sua distruzione. Rispettiamo le dipendenze nella nostra architettura - il ViewModel si trova al di sotto dell'UI – e non inseriamo alcun riferimento al Fragment a cui è accostato. Il ViewModel è reso così completamente agnostico nei confronti della componente che lo utilizza.

Un'altra importante componente Lifecycle-Aware che abbiamo scelto di utilizzare

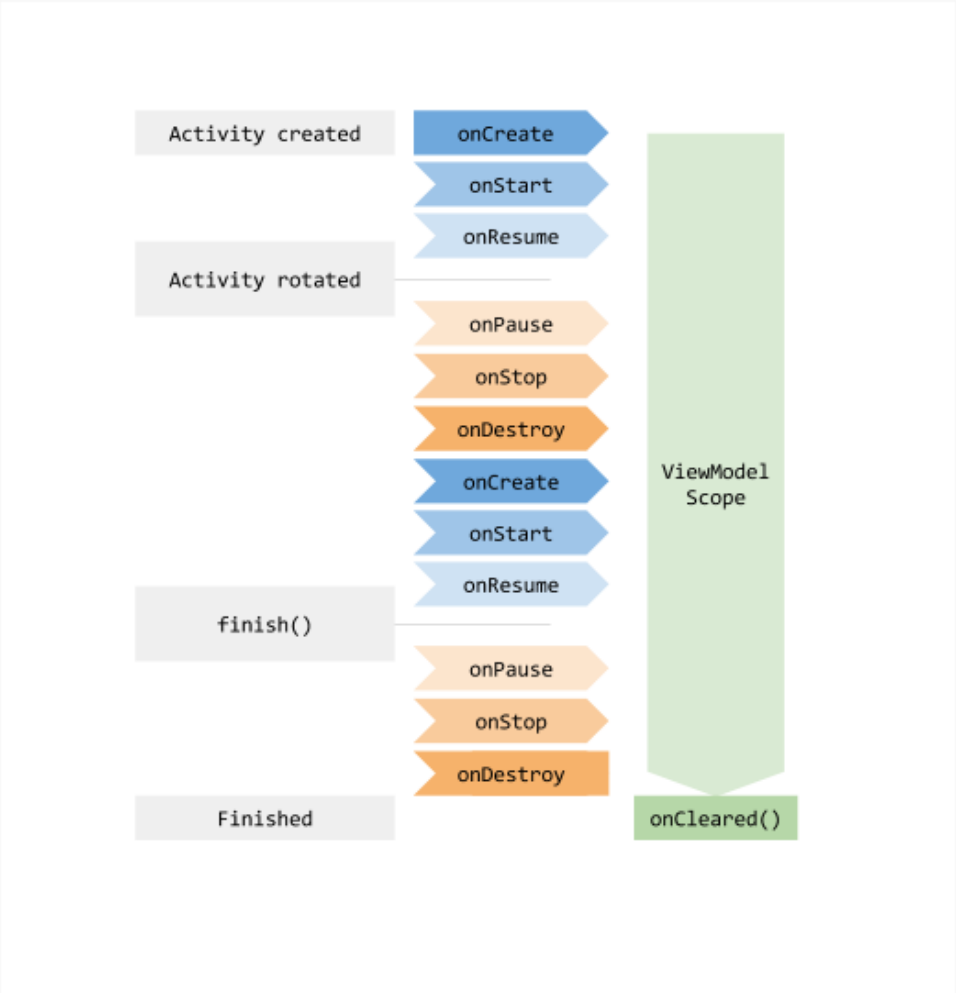


Figura 3.2: Ciclo di vita del ViewModel in relazione a quello dell'Activity [32].

è la data holder class LiveData, appartenente ad Android Jetpack. Questa classe ci permette di applicare l'Observer Pattern, ossia la possibilità di osservare i cambiamenti – in questo caso dei dati – sottoscrivendo il Fragment al ViewModel. Poiché è Lifecycle-Aware, l'observer verrà costruito nel momento in cui il Fragment si trova nello Started State, e distrutto nel Destroyed State.

Il ViewModel espone così i propri dati alla View attraverso degli Observables, e la gestione della loro allocazione e deallocazione è totalmente gestita dai LiveData (principale differenza che ci ha spinto verso l'adozione di questo contenitore di dati osservabili

invece di RxJava).

Per semplificare ulteriormente l'interfaccia grafica e privarla di codice boilerplate implementiamo la libreria Data Binding (anche questa contenuta in Android Jetpack), che collega direttamente elementi del layout alle fonti di dati con un approccio dichiarativo anziché imperativo.

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:android="http://schemas.android.com/apk/res/android">

    <data>
        <import type="android.view.View"/>

        <variable name="overviewViewModel"
            type="com.example.monitorcardiaco.overview.
    OverviewViewModel"/>
    </data>
            <androidx.constraintlayout.widget.ConstraintLayout
                android:layout_width="match_parent"
                android:layout_height="match_parent">

            <TextView
                android:id="@+id/user_name_textview"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="@{overviewViewModel.user.name}"
                android:textSize="16sp"
                app:layout_constraintStart_toStartOf="parent"
                app:layout_constraintTop_toBottomOf="@+id/
    subtitle_textview"/>
        </androidx.constraintlayout.widget.ConstraintLayout>

</layout>
```

Listing 3.1: Esempio di implementazione del Data Binding nel layout XML

Nei casi in cui determinati dati necessitano di un'elaborazione consistente e laboriosa - difficilmente implementabile nel layout XML - abbiamo creato delle clas-

si di aiuto chiamate Binder Adapters, che gestiscono quest'aspetto e alleggeriscono contemporaneamente il ViewModel.

Una caratteristica che differenzia l' implementazione dalla Clean Architecture è l'inserimento degli Use Cases all'interno del ViewModel - tecnicamente una componente del Presentation Layer - invece di creare delle classi apposite indipendenti sia da questo che dal Data Layer. Il numero di funzionalità e la loro complessità durante lo sviluppo di quest'applicazione, che ricordiamo essere un'app pilota di un progetto più ampio, ci permette di effettuare tale scelta senza rischiare di creare dei God ViewModels, ovvero delle classi ricche di responsabilità e codice. Come vedremo, i ViewModel delle funzionalità implementate risultano coerenti e lightweight.

Infine, la gestione della navigazione è totalmente assegnata alla MainActivity, facendo uso della Navigation Component (di Android Jetpack). Attraverso un file XML che riunisce tutte le informazioni relative alla navigazione, possiamo organizzare ogni transazione da un Fragment all'altro, offrendo all'utente un'esperienza fluida e consistente.

La MainActivity implementa anche un NavHost per le destinazioni dei Fragments ed un NavController per la gestione dei contenuti durante il passaggio tra Fragments.

In termini di codice, la Navigazione è implementata in ogni funzionalità allo stesso modo: l'utente genera un evento – generalmente clicca un bottone -, la View cattura l'evento e lo manda al ViewModel, il quale assegna un valore ad un oggetto LiveData. Poiché tale oggetto è osservato dalla View, essa riceve l'aggiornamento del valore, chiama il NavController, raccoglie eventuali dati richiesti dalla destinazione, li inserisce nel Bundle e permette il passaggio al Fragment successivo. L'unica eccezione è fatta per il Navigation Drawer, ma verrà discusso nella parte dedicata alla funzionalità "Visualizza Monitor".

```
1
2 class OverviewFragment : Fragment() {
3     override fun onCreateView(inflater: LayoutInflater, container:
    ViewGroup?,
4                               savedInstanceState: Bundle?): View? {
5         [...]
```

```
6         viewModel.navigateToAddEditParams.observe(this, Observer {
  event ->
7             event?.let {
8                 val bundle = bundleOf("email" to viewModel.user!!.value
  !!.email, "date" to getDate())
9                 this.findNavController()
10                    .navigate(R.id.
  action_overviewFragment_to_addEditParamsFragment, bundle)
11                viewModel.doneNavigatingAddEditParams()
12            }})
13        return binding.root
14    }}
```

Listing 3.2: Esempio di gestione della navigazione da parte del Fragment.

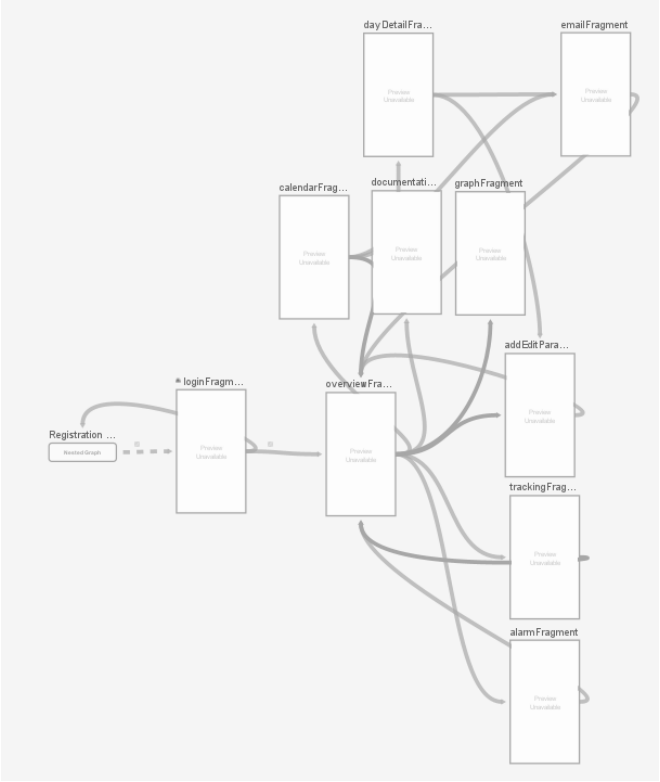


Figura 3.3: Rappresentazione grafica del layout di navigazione in XML tramite il Navigation Editor di Android Studio.

3.2.2 Repository

La Repository implementa parte del Data Layer (l'altra parte è implementata dal Database) e costituisce essa stessa l'implementazione del Domain Layer (dipende infatti da questa, come afferma la Dependency Rule).

La Repository ha la funzione d'intermediario tra il ViewModel che necessita di dati da esporre alla View e le varie sorgenti di dati – nel nostro caso, il Database – da cui ottenerli. Il primo vantaggio che deriva dalla sua implementazione è la creazione di un grado di astrazione aggiuntivo tra Use Cases e Data, che permette al ViewModel di non interessarsi a quale fonte verrà utilizzata per ottenere i dati necessari.

La Repository diventa quindi la Single Source of Truth della nostra applicazione, a cui faremo sempre affidamento. Sarà suo compito gestire molteplici flussi di dati.

Un comune problema derivante dalla sua implementazione consiste nel potenziale carico di risorse della nostra applicazione nel momento in cui ogni Fragment dovrà creare una nuova istanza di Repository e conoscere tutte le sue dipendenze.

Abbiamo risolto tale problema facendo uso del Service Locator Pattern, che permette di mantenere un'unica istanza della Repository in un registro centrale. In questo modo saremo anche facilitati nella fase di testing di tale modulo.

```
1  object ServiceLocator {
2
3      private var database: MonitorDatabase? = null
4      @Volatile
5      var repository: IRepository? = null
6      private val lock = Any()
7
8      fun provideUserRepository(context: Context): IRepository {
9          synchronized(this) {
10             return repository ?: createUserRepository(context)
11         }
12     }
13
14     private fun createUserRepository(context: Context): IRepository {
```

```
15          val newRepo = Repository(createUserDataSource(context),
    createDayDataSource(context))
16          repository = newRepo
17          return newRepo
18      }
19
20      private fun createUserDataSource(context: Context) :
    UserDatabaseDao {
21          val database = database ?: createDatabase(context)
22          return database.userDatabaseDao
23      }
24
25      private fun createDayDataSource(context: Context) : DayDatabaseDao
    {[...]}
26      private fun createDatabase(context: Context): MonitorDatabase
    {[...]}
```

Listing 3.3: Parte dell'oggetto Service Locator. Verrà istanziato nella MainActivity.

Il codice qui presentato mostra invece la classe Repository con alcuni metodi significativi.

```
1  package com.example.monitorcardiaco.repository
2  import [...]
3
4  class Repository(private val userDataSource: UserDatabaseDao,
5                   private val dayDataSource: DayDatabaseDao) :
6      IRepository {
7
8      override suspend fun insertUser(user: User) {
9          userDataSource.insert(user)
10     }
11
12     override fun getUserWithEmail(email: String): LiveData<User> {
13         return userDataSource.getUser(email)
14     }
15
16     override fun getUsersWithDays(): LiveData<List<UserWithDays>> {
17         return userDataSource.getUsersWithDays()
```

```
18      }
19
20      [...]
21
22      override fun getDay(date: String, userEmail: String): LiveData<Day
        >? {
23          return dayDataSource.getDay(date, userEmail)
24      }
25
26      override suspend fun updateStepsDay(email: String, date: String,
        steps: Int) {
27          dayDataSource.updateStepsDay(email, date, steps)
28      }
29  }}
```

Listing 3.4: Parte della classe Repository

Per eseguire funzioni che modificano il database (Insert, Update, Delete) utilizziamo le Suspend Functions di Kotlin. In breve, tali funzioni non bloccano il Thread fino al termine della loro esecuzione, ma possono essere ripetutamente sospese, lasciando che altre funzioni del Thread siano eseguite, per poi riprendere normalmente.

Poiché la nostra data source utilizza i LiveData, anche la Repository fa uso di questi, per garantire che i dati vengano aggiornati automaticamente dal Database fino al ViewModel.

3.2.3 Database

Android permette la creazione di un database SQL attraverso le API di SQLite, ma l'adozione di queste è sconsigliata perché di basso livello e dispendiose di tempo date le grandi quantità di codice da scrivere.

Nella nostra applicazione utilizziamo invece Room, una Object Relational Mapping Library, che fornisce un livello di astrazione su SQLite. La libreria è costituita da tre componenti:

- Entity: è l'implementazione delle Entities definite nella Clean Architecture, rappresenta un Plain Old Java Object – nel nostro caso una data class di Kotlin –

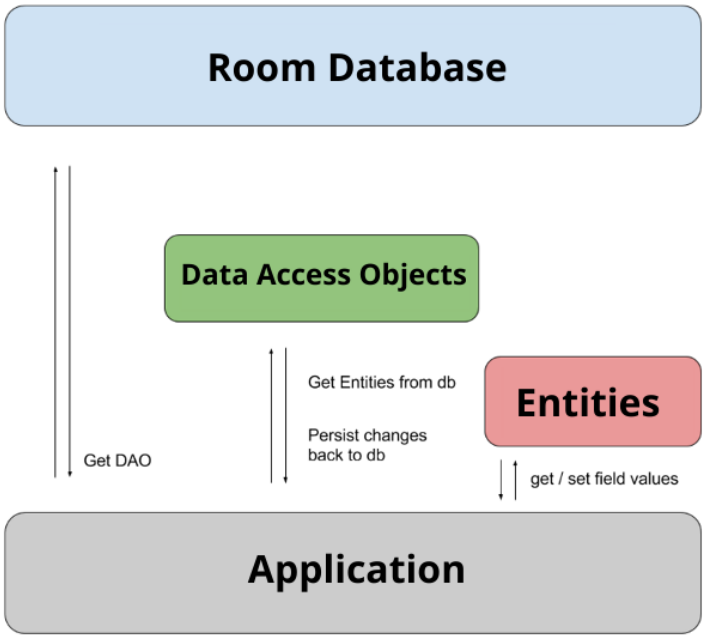


Figura 3.4: Le tre componenti di Room.

corrispondente alle tabelle del database. Room identifica una data class come Entity se presenta l'annotazione "@Entity". Nell'app sviluppata sono presenti due entità: User (il paziente) e Day (il giorno, contenente i parametri quotidiani).

```
1  package com.example.monitorcardiaco.database
2  import [...]
3
4  @Entity(tableName = "day_table", primaryKeys = ["date", "
       userCreatorEmail"])
5  data class Day(
6      val date: String,
7      val userCreatorEmail: String,
8      @Embedded var lvad: Lvad?,
9      @Embedded var bodyparams: BodyParams?,
10     @Embedded var cardioparams: CardioParams?,
11     var note: String,
12     var alarm: String?)
```

Listing 3.5: Entità Day. Le variabili annotate con "@Embedded" rappresentano altre entità contenenti parametri del paziente.

- DAO: il Database Access Object è un'interfaccia per accedere al database, per compiere facilmente operazioni CRUD (Create, Read, Update, Delete) e per gestire la persistenza dei dati. È ciò che la Repository usa per ottenere i dati richiesti dal ViewModel ed è riconoscibile dall'annotazione "@Dao" e dalle annotazioni dei metodi "@Insert", "@Delete", "@Query" (per query più complesse). Seguiamo la pratica di avere un'interfaccia DAO per ogni entità: ci sarà quindi uno UserDAO ed un DayDAO.

  Una peculiarità di Room è la possibilità di far restituire al DAO dei LiveData, facilitando tutto il meccanismo di aggiornamento e gestione del ciclo di vita delle componenti. Così facendo si crea una catena di subscriptions: la Repository riceve degli oggetti Observables dal DAO, il ViewModel fa lo stesso con la Repository e la View riceve sempre i dati aggiornati in tempo reale senza preoccuparsi di leaks e del proprio ciclo di vita. Il DAO offre inoltre la possibilità di definire suspend functions direttamente dal suo interno, in modo da compiere operazioni asincrone e non bloccanti.

```
1  package com.example.monitorcardiaco.database
2  import [...]
3
4  @Dao
5  interface DayDatabaseDao {
6
7      @Insert(onConflict = OnConflictStrategy.REPLACE)
8      suspend fun insert(day: Day)
9
10     @Query("SELECT * from day_table WHERE userCreatorEmail = :
       email AND date = :date")
11     fun getUserDay(email: String, date: String): LiveData<Day>
12
13     @Query("DELETE FROM day_table WHERE userCreatorEmail = :email
       ")
14     suspend fun deleteUserDays(email: String)
15
16     @Query("DELETE FROM day_table WHERE userCreatorEmail = :email
       AND date = :date")
```

```
17      suspend fun deleteDay(email: String, date: String)
18
19      @Query("UPDATE day_table SET alarm = :alarm WHERE
        userCreatorEmail = :email AND date = :date")
20      suspend fun updateAlarmDay(email: String, date: String, alarm:
        String)
21  }
```

Listing 3.6: Parte del DayDAO.

- Database: è una classe estensione di Room Database, si occupa di creare il database e la relazione tra entità ed interfaccia DAO.

```
1  package com.example.monitorcardiaco.database
2  import [...]
3
4  @Database(entities = [User::class, Day::class], version = 19,
       exportSchema = false)
5  @TypeConverters(Converters::class)
6  abstract class MonitorDatabase : RoomDatabase() {
7
8      abstract val userDatabaseDao: UserDatabaseDao
9      abstract val dayDatabaseDao: DayDatabaseDao
10
11     companion object {
12         @Volatile
13         private var INSTANCE: UserDatabase? = null
14
15         fun getInstance(context: Context): UserDatabase {
16             synchronized(this) {
17                 var instance =
18                     INSTANCE
19
20                 if (instance == null) {
21                     instance = Room.databaseBuilder(
22                             context.applicationContext,
23                             UserDatabase::class.java,
24                             "user_database"
25                     )
```

```
26                                  .fallbackToDestructiveMigration()
27                                  .build()
28                              INSTANCE = instance
29                          }
30                          return instance
31                      }
32                  }
33              }
34          }
```

Listing 3.7: Classe MonitorDatabase.

Il flusso di dati prosegue quindi in due sensi: nel primo, il paziente registra un dato o segnala un evento, la View (il Fragment) cattura l'evento o il dato, il ViewModel formatta il dato o elabora delle operazioni in seguito all'evento ed invia la richiesta – di qualsiasi natura – alla Repository, che si occupa di comunicare con il Database attraverso l'interfaccia DAO, il tutto facilitato dall'implementazione dei LiveData. Nel secondo accade la medesima cosa, con il Database come punto di partenza e la View, con i suoi dati aggiornati, come punto di arrivo.

Tale implementazione ci permette di proseguire nello sviluppo dell'applicazione mobile concentrandoci sulle singole funzionalità (Use Cases), avendo a disposizione strumenti efficaci e versatili.

3.3 Funzionalità

Il codice sorgente è diviso in packages in base alle funzionalità. In ogni package troviamo il Fragment, il ViewModel di riferimento, ed eventuali classi helper (ad esempio i BinderAdapter). Nella root sono presenti classi per l'implementazione di altre componenti – AlarmBroadcastReceiver, StepsService -, di una Factory per i vari ViewModel, il ServiceLocator Object ed altre Utils che menzioneremo nelle rispettive funzionalità.

La maggior parte dei Layout delle relative funzionalità è implementata utilizzando un ConstraintLayout, un ViewGroup che permette di posizionare i widgets in modo

flessibile - anche più di un RelativeLayout - basandosi sulla relazione tra questi. Per quelle schermate che non riescono a mostrare tutti i widgets, abbiamo deciso di inserire il ConstraintLayout dentro una NestedView, in modo da permettere lo scrolling della View.

3.3.1 Registrazione e autenticazione

La registrazione è composta da tre Fragment (e quindi da tre View), ciascuno con il compito di registrare i dati inseriti dal paziente. Tutti condividono un solo ViewModel, che mantiene i dati inseriti tra una View e un'altra.

Ciò è possibile grazie all'utilizzo del Factory Pattern, ossia una classe che crea un'istanza ViewModel quando richiesta da un Fragment. Se già presente, la classe si limita a restituire la stessa istanza. Per la precisione, la classe BaseFactoryUtil, in quanto generica, costituisce una "Factory di ViewModel Factories" e permette di creare o restituire un'istanza di qualsiasi ViewModel, con l'unica condizione che vengano fornite le dipendenze necessarie al costruttore del ViewModel (in genere l'interfaccia della Repository, la mail del paziente e la data del giorno).

```
1  package com.example.monitorcardiaco
2  import [...]]
3
4  class BaseFactoryUtil {
5
6      inline fun <reified T : ViewModel> Fragment.getViewModel(noinline
       creator: (() -> T)? = null): T {
7          return if (creator == null)
8              ViewModelProviders.of(this).get(T::class.java)
9          else
10             ViewModelProviders.of(this, BaseViewModelFactory(creator)).
       get(T::class.java)
11     }
12
13     inline fun <reified T : ViewModel> FragmentActivity.getViewModel(
       noinline creator: (() -> T)? = null): T {
14         return if (creator == null)
15             ViewModelProviders.of(this).get(T::class.java)
```

```
16          else
17              ViewModelProviders.of(this, BaseViewModelFactory(creator)).
    get(T::class.java)
18      }
19  }
```

Listing 3.8: La classe BaseFactoryUtil.

Il primo ed il secondo Fragment contengono un set di Material Widgets – libreria UI che permette di personalizzare l'interfaccia grafica e di dare una sensazione di continuità per tutta l'applicazione – in cui il paziente può inserire dati identificativi e riguardanti la sua tipologia (ad esempio se appartiene alla categoria paziente Scompenso Cardiaco o paziente LVAD). Al termine dell'inserimento dei dati un bottone permette il passaggio al Fragment successivo secondo il meccanismo di navigazione precedentemente discusso.

Il terzo Fragment contiene un bottone da premere al termine dell'operazione, subito dopo aver inserito email e password (dati necessari per il Login), che tramite Data Binding chiama il metodo nel ViewModel di raccolta dati, di creazione dell'entità Utente e di conseguente chiamata asincrona alla Repository per inserirla nel Database. La Repository comunica con il Database attraverso il DAO e porta a termine l'operazione. Tramite il meccanismo di navigazione, l'ultimo Fragment della Registrazione chiama il NavController e viene sostituito dal LoginFragment.

In questo momento il Database ha già registrato la nuova entità ed il paziente può provare ad autenticarsi inserendo, nei widgets appositi, la propria email e la propria password. Il meccanismo di autenticazione, infatti, è locale e basato sul Database. Nel caso in cui l'email o la password inserite dovessero risultare errate, un Toast Widget avvisa il paziente dell'errore e non permette il proseguimento della navigazione (in termini di codice l'evento non viene generato). Inserendo correttamente email e password, il paziente passa dal LoginFragment al Fragment relativo alla visualizzazione del proprio Monitor.

Da questo punto, tutte le operazioni svolte dal paziente verranno registrate nel Database facendo riferimento alla sua entità User e all'entità Day (corrente). Per questo

motivo, tutti i ViewModel associati ai Fragment dell'applicazione richiederanno l'email del paziente (per ottenere il corrispondente User dal Database) ed il giorno, sotto forma di data, a cui l'utente fa riferimento (per compire operazioni di lettura o scrittura sulla corrispondente entità Day).

Trattandosi di una funzionalità base, con il semplice compito di registrare alcuni dati, inviarli al Database e poi controllare che due variabili corrispondano, i tre Fragment ed il ViewModel risultano "lightweight". Alla creazione del Fragment si collega il Binding Object alla View, si acquisisce un'istanza del ViewModel, la si collega al Binding Object e si osservano i cambiamenti dei LiveData per la navigazione. Il ViewModel riceve i dati, li elabora e li invia alla Repository.

3.3.2 Visualizzazione monitor

Il Monitor costituisce la funzionalità più complessa tra quelle implementate in termini di possibilità di azione per il paziente. La sua principale funzione è quella di visualizzazione dei dati identificativi inseriti alla registrazione, oltre a presentare il riepilogo dei parametri inseriti dal paziente nel giorno corrente.

Tuttavia, abbiamo pensato il Monitor come la "home" del paziente, da cui può muoversi per compiere qualsiasi operazione. Due componenti grafici permettono potenzialmente lo spostamento in qualsiasi altro Fragment:

- Navigation Drawer: pannello laterale dell'interfaccia grafica che permette la navigazione verso funzionalità di carattere Display – come la visualizzazione dei grafici o la documentazione. La sua implementazione avviene nella MainActivity, di conseguenza vi si potrebbe accedere da qualsiasi Fragment, ma ciò è stato impedito e permesso solo nel Monitor per offrire una sensazione di ordine
- FAB (Floating Action Button): bottone animato che, una volta premuto, rivela altri tre FAB corrispondenti a tre funzionalità differenti: Inserimento parametri, Contapassi, Segnala allarme

Nel Fragment inseriamo l'email del paziente all'interno della SharedPreferences, interfaccia per accedere e modificare dati di piccola dimensione, in modo da averla disponibile

in ogni Fragment della navigazione (fino al Logout). Compiamo anche qui le operazioni di istanziazione dell'oggetto Binding e del ViewModel, ed osserviamo il LiveData Day per popolare la View con i parametri inseriti nella giornata (e contenuti nella rispettiva entità Day). La View è modificata dinamicamente in base alla presenza o meno dei parametri ed in base alla tipologia del paziente. Come vedremo nel capitolo 4, i due tipi di pazienti hanno delle variabili diverse e la View mostra solo quelle corrette.

Menzioniamo solamente la presenza di codice relativo alla funzionalità Notifica, inserito nel Monitor perché primo Fragment visitato dal paziente dopo l'autenticazione.

Il ViewModel risulta molto leggero, con il compito di ottenere l'entità User, l'entità Day dalla Repository ed assegnare valori ai LiveData nel caso di navigazione nei vari Fragments.

3.3.3 Inserimento parametri

Questa funzionalità permette al paziente di inserire o modificare parametri già inseriti nel giorno di riferimento. La modularità dei Fragment torna utile in questo caso, poiché possiamo utilizzarlo sia dal Monitor – in cui il paziente inserisce o modifica solo i parametri del giorno corrente -, sia dal Calendario, dove il paziente può modificare parametri inseriti in giorni precedenti, per ottenere un report bisettimanale privo di errori.

Il Fragment ha la funzione di prendere le variabili email e data dal Bundle proveniente dal Monitor (o dal Calendario), istanziare il Binding Object ed il ViewModel, ed osservare l'entità User – per modificare dinamicamente i parametri disponibili per l'inserimento o la modifica.

Il ViewModel, dopo aver ricevuto l'evento dalla View di inserimento dati nell'entità Day, formatta tali informazioni nel formato corretto ed invia la richiesta alla Repository, che tramite il DAO inserisce l'entità nel Database.

Al termine dell'operazione, il paziente ritorna nel Monitor.

3.3.4 Calendario

Questa funzionalità permette al paziente di visualizzare la lista dei giorni che ha inserito nel Database insieme ai rispettivi dati.

Visualizzare una lista di dati è una funzionalità comune nello sviluppo di un'applicazione mobile: diversi widgets permettono di farlo, come ListView e GridView, o un semplice LinearLayout. In questa implementazione abbiamo preferito invece fare uso del widget RecyclerView. Nonostante sia più complesso e più dispendioso di tempo per apprendere il funzionamento della sua API, permette vantaggi in termine di performance e flessibilità.

Oltre a poter visualizzare oggetti complessi – nel nostro caso, un'entità Day con all'interno una serie di parametri di vario tipo ed un'immagine rappresentativa -, è più efficiente poiché riutilizza e ricicla il più possibile gli elementi della lista.

Per implementare una RecyclerView abbiamo fatto uso dell'Adapter Pattern, che converte un'interfaccia in modo che sia utilizzabile con un'altra. In questo caso, i nostri dati provenienti da Room e dal ViewModel devono essere visualizzati come una lista. Costruiamo allora un Adapter che trasformi i nostri dati in qualcosa che la RecyclerView possa utilizzare.

All'interno dell'Adapter implementiamo tre metodi con tre funzioni fondamentali per la RecyclerView:

- Indicare quanti oggetti sono presenti nella lista
- Come creare un oggetto
- Come creare una View

```
1 package com.example.monitorcardiaco.calendar
2 import [...]
3
4 class DayAdapter(val clickListener: DayListener): ListAdapter<Day,
     DayAdapter.ViewHolder>(DayDiffCallback()) {
5
```

```
6       override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
    ViewHolder {
7           return ViewHolder.from(parent)
8       }
9
10      override fun onBindViewHolder(holder: ViewHolder, position: Int) {
11          holder.bind(getItem(position)!!, clickListener)
12      }
13
14      class ViewHolder private constructor(val binding:
    CalendarListItemDayBinding): RecyclerView.ViewHolder(binding.root) {
15
16          fun bind(item: Day, clickListener: DayListener) {
17              binding.day = item
18              binding.clickListener = clickListener
19              binding.executePendingBindings()
20          }
21
22          companion object {
23              fun from(parent: ViewGroup): ViewHolder {
24                  val layoutInflater = LayoutInflater.from(parent.context
    )
25                  val binding = CalendarListItemDayBinding.inflate(
    layoutInflater, parent, false)
26                  return ViewHolder(binding)
27              }
28          }
29      }
30
31      class DayListener(val clickListener: (dayDate: String) -> Unit) {
32          fun onClick(day: Day) = clickListener(day.date)
33      }
34  }
```

Listing 3.9: Classe DayAdapter, contenente la classe ViewHolder, collegata al layout XML del Calendario tramite Data Binding.

La View a cui fa riferimento l'interfaccia non è una vera e propria View, ma un

ViewHolder. Definiamo quindi un ViewHolder – dettaglio implementativo della RecyclerView - per la nostra entità Day che permetta di contenere una View ed alcune informazioni preziose per la RecyclerView.

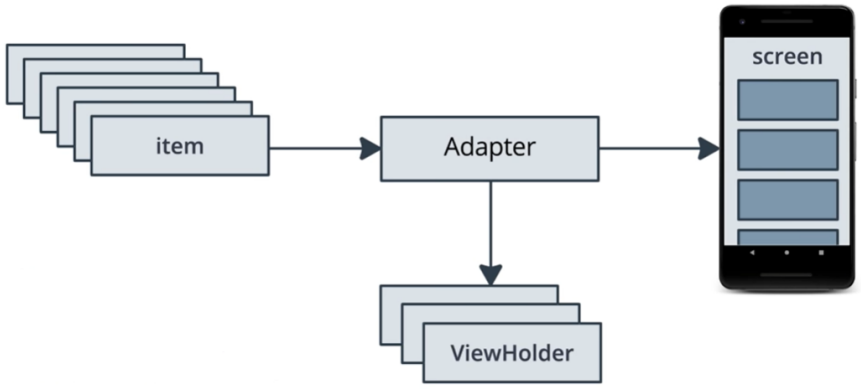


Figura 3.5: Meccanismo di funzionamento della RecyclerView: uso dell'Adapter.

L'ultimo aspetto da curare per far funzionare la RecyclerView è l'implementazione di un meccanismo per avvisare che un oggetto della lista è cambiato. Una possibilità è rappresentata dal metodo NotifyDataSetChanged(), che comunica alla RecyclerView che l'intera lista è potenzialmente invalida. La RecyclerView ricostruisce e ricollega ogni oggetto nella lista, anche se il cambiamento è stato minimo, persino di un solo parametro di un'entità Day. Tale implementazione comporta uno spreco notevole di risorse, con conseguenze negative anche sull'esperienza dell'utente.

Un'altra possibilità consiste nell'usare le API della RecyclerView per segnalare esattamente cosa sia cambiato e permettere alla RecyclerView di cambiare solo il necessario, ma è un meccanismo complesso e macchinoso.

Per questo, nella nostra implementazione, abbiamo preferito fare uso di DiffUtil, una classe helper per adapter di RecyclerView. La classe prende sia la vecchia lista che la nuova lista e scopre cos'è cambiato utilizzando l'algoritmo di Myers Diff, per trovare il minimo numero di cambiamenti necessari per trasformare la vecchia nella nuova [14]. Tra i benefici di questa scelta implementativa troviamo il risparmio di risorse e l'animazione di default dei cambiamenti.

```
package com.example.monitorcardiaco.calendar
import [...]

class DayDiffCallback :
    DiffUtil.ItemCallback<Day>() {
    override fun areItemsTheSame(oldItem: Day, newItem: Day): Boolean {
        return oldItem.date == newItem.date && oldItem.userCreatorEmail
    == newItem.userCreatorEmail
    }

    override fun areContentsTheSame(oldItem: Day, newItem: Day):
   Boolean {
        return oldItem == newItem
    }
}
```

Listing 3.10: La classe DayDiffCallback, inserita in DayAdapter.

Per finire, aggiungiamo il Data Binding anche alla RecyclerView. Implementiamo un BindingAdapter, un Adapter che trasforma i dati in un formato visualizzabile sullo schermo, in modo da gestire l'aggiornamento della View per rappresentarli. Così facendo, possiamo osservare e mostrare la lista dei giorni direttamente dal Database all'UI.

```

package com.example.monitorcardiaco.calendar
import [...]

@BindingAdapter("dayDateFormatted")
fun TextView.setDateFormatter(item: Day?) {
    item?.let {
        text = "Data: ${item.date}"

    }
}

@BindingAdapter("dayWeightString")
fun TextView.setWeightFormatter(item: Day?) {
    item?.let {
```

```
16          text = "${item.bodyparams?.weight ?: ""}kg"
17      }
18  }
19
20  [...]
21
22  @BindingAdapter("dayFluxString")
23  fun TextView.setFluxFormatter(item: Day?) {
24      item?.let { day ->
25          day.lvad?.let {
26              text = "${it.flux ?: ""}L/m"
27              visibility = View.VISIBLE
28          }
29      }
30  }
31
32  @BindingAdapter("dayDateImage")
33  fun ImageView.setDateImage(item: Day?) {
34      item?.let {
35          setImageResource(when (item.date.split("/")[0]) {
36              "01" -> R.mipmap.ic_day_1
37              "02" -> R.mipmap.ic_day_2
38              [...]]
39              "30" -> R.mipmap.ic_day_30
40              else -> R.mipmap.ic_day_31
41          })
42      }
43  }
```

Listing 3.11: Parte del BindingAdapter per formattare il testo di ogni oggetto della lista ed inserire a lato un'immagine rappresentativa del giorno.

Oltre alla visualizzazione della lista, la funzionalità Calendario permette di selezionare un singolo giorno (e navigare nel Fragment della funzionalità Visualizza Giorno), o esportare un report bisettimanale – solo nel caso in cui il numero di giorni inseriti lo permetta. Tale funzionalità è implementata attraverso la classe DayListener, presente nella classe DayAdapter.

Nel ViewModel, dopo aver fatto richiesta al Repository di ottenere la lista dei giorni inseriti da un paziente (ottenibile tramite il meccanismo di relazione uno a molti di Room), trasformiamo i dati nel formato corretto facendo uso di una Transformations Map.

```
package com.example.monitorcardiaco.calendar
import [...]

class CalendarViewModel(private val repository: IRepository,
                        private val email: String): ViewModel() {

    private var viewModelJob = Job()
    private val uiScope = CoroutineScope(Dispatchers.Main +
   viewModelJob)
    private val _usersWithDays = repository.getUsersWithDays()


    val userDays = Transformations.map(_usersWithDays) {
        it?.let {
            val userDays = it.first { it.user.email == email }
            val days = userDays.days
            Collections.sort(days, DayComparator())
            Collections.reverse(days)
            days
        }
    }
}
```

Listing 3.12: Parte del CalendarViewModel legata al recupero della lista dei giorni.

Nel Fragment istanziamo l'oggetto Data Binding ed il ViewModel, colleghiamo ViewModel al Data Binding e quest'ultimo alla RecyclerView (tramite BindingAdapter e DayAdapter), ed osserviamo la lista di giorni proveniente dal ViewModel, per passarla al DayAdapter.

```
package com.example.monitorcardiaco.calendar
import [...]

class CalendarFragment: Fragment() {
```

```
5     override fun onCreateView(inflater: LayoutInflater, container:
    ViewGroup?,
6                              savedInstanceState: Bundle?): View? {
7
8         val binding: FragmentCalendarBinding = DataBindingUtil.inflate(
9             inflater, R.layout.fragment_calendar, container, false
10        )
11        binding.setLifecycleOwner(this)
12        [...]
13
14        val viewModel = activity?.getViewModel()
15        { CalendarViewModel((requireContext().applicationContext as
    MonitorCardiacoApplication).repository, email!!) }
16
17        binding.calendarViewModel = viewModel
18
19        val adapter = DayAdapter(DayAdapter.DayListener { dayDate ->
20            viewModel!!.onDayClicked(dayDate)
21        })
22        binding.dayList.adapter = adapter
23        binding.dayList.layoutManager = LinearLayoutManager(context)
24
25        [...]
26        return binding.root
27    }
28 }
```

Listing 3.13: Parte del Fragment legato al collegamento di Data Binding, ViewModel e RecyclerView

3.3.5 Visualizza giorno

Questa funzionalità permette al paziente, proveniente dal Fragment del Calendario, di visualizzare tutti i parametri inseriti nel giorno selezionato dalla lista. Inoltre può spostarsi, tramite un bottone, al Fragment di modifica dei parametri, in modo da cambiare i parametri per quel giorno. Si tratta di una funzionalità semplice ma allo stesso

tempo di collegamento tra più Fragment, e permette ad un paziente di correggere errori di inserimento nei giorni passati prima di inviare un report bisettimanale al medico.

Il ruolo del Fragment e del ViewModel è il medesimo delle altre funzionalità, con la particolare attenzione di visualizzare solo i parametri relativi al tipo di paziente che utilizza l'applicazione, risultato ottenibile tramite l'osservazione dei LiveData dell'entità User e dell'entità Day nel Fragment.

3.3.6 Esportazione e invio report

Questa funzionalità può essere suddivisa in due parti.

Abbiamo implementato l'esportazione di un report bisettimanale, contenente le ultime due settimane (lista di 14 entità Day) di parametri inseriti da parte del paziente, in formato CSV tramite una libreria chiamata Commons CSV [34]. Tale libreria permette di semplificare il processo di scrittura attraverso un'astrazione delle operazioni più tediose.

```
fun writeCsvExternalStorage(root: String, file: File, header: List<
    String>?, records: MutableList<ArrayList<String?>?>?) {

    val newFile = file

    // Controllo della disponibilita' dell'ExternalStorage
    val state: String = Environment.getExternalStorageState()
    if (!Environment.MEDIA_MOUNTED.equals(state)) {
        return
    }

    var writer: FileWriter?
    try {
        newFile.createNewFile()
        writer = FileWriter(file.path)
        val csvPrinter = CSVPrinter(writer, CSVFormat.DEFAULT)
        csvPrinter.printRecord(header?.asIterable())
        records?.forEach {
```

```
18              csvPrinter.printRecord(it?.asIterable())
19          }
20          csvPrinter.flush()
21          csvPrinter.close()
22      } catch (e: Exception) {
23          e.printStackTrace()
24          Log.e("errorFile", e.toString())
25      }
26  }
```

Listing 3.14: Il metodo per esportare la lista di giorni in un file CSV.

Il metodo implementato, inserito nella classe Utils, e chiamato nel Fragment del Calendario, prende in input: una stringa con il path del File che verrà creato, un oggetto File (al momento generico e vuoto), una lista di String che costituisce l'header del CSV ed una MutableList di ArrayList di String, ossia una lista di parametri dei giorni che si vogliono esportare.

Dopo aver istanziato un oggetto FileWriter ed averlo collegato al path del File, istanziamo un oggetto CSVPrinter e diamo in input il nostro FileWriter ed un formato di default del file CSV (formato che gestisce il modo in cui il CSVPrinter crea una nuova riga o inserisce i parametri in una riga).

Inseriamo l'header precedentemente costruito nel Fragment del Calendario e iteriamo la lista di giorni. La libreria scriverà sul file i dati e renderà tale file un CSV. Terminata l'operazione, il file CSV sarà disponibile nella memoria dello smartphone, esattamente nel path corrispondente a quello dell'oggetto File dato in input.

Il file esportato appartiene alla categoria di "App-specific files", per questo motivo Android non necessita di particolari permessi di scrittura. In questo modo, altre applicazioni non possono accedere al report ed i file verranno rimossi durante la disinstallazione dell'app. Si potrebbe considerare quest'aspetto come negativo, ma nel nostro caso non rappresenta un problema: il report verrà subito inviato tramite email al medico dall'applicazione stessa, e l'accesso proibito ad altre app del file è solo un vantaggio aggiunto in termini di privacy e sicurezza.

La seconda parte della funzionalità consiste nell'invio dell'email da parte del paziente.

L'implementazione avviene nell'EmailFragment, dove si fa uso di un Intent. Precedentemente, in questo capitolo, abbiamo accennato alla possibilità di un'Activity di essere chiamata da un'altra Activity, anche appartenente ad un'altra applicazione. È ciò che avviene in questo caso: non implementiamo il meccanismo di invio email internamente al nostro software, ma utilizziamo applicazioni presenti nel dispositivo del paziente che sono già in grado di farlo – come Gmail, Outlook, etc. -.

Per farlo usiamo un Intent, un oggetto che chiede ad un'altra componente (in questo caso, ad un'altra Activity) di compiere un'azione (in questo caso, l'invio di una email).

Un oggetto Intent necessita di alcune tipologie di informazioni, che verranno usate dal sistema Android per recepire il destinatario (o i destinatari) della richiesta e fornire eventuali informazioni aggiuntive necessarie per compiere correttamente l'operazione:

- Component Name: informazione opzionale, ma obbligatoria nel caso in cui vogliamo creare un Explicit Intent, ovvero una richiesta ad una componente specifica. Nella nostra implementazione tale informazione è omessa di proposito, in modo che l'Intent creato sia implicito: vogliamo che sia l'utente a scegliere quale applicazione usare per inviare una email

- Action: stringa rappresentante il tipo di azione da chiedere. Inseriamo la stringa "ACTION_SENDTO" perché vogliamo inviare una email

- Data: l'URI che si riferisce ai dati da inviare o da utilizzare per l'operazione da compiere. Inseriamo l'URI "mailto:"

- Extra: coppie chiave-valore che corrispondono ad informazioni extra necessarie per il corretto funzionamento dell'operazione. Le coppie inserite riguardano la email del destinatario (nella nostra implementazione, già inserita perché unica), eventuali email da inserire in CC, l'oggetto, il testo, ed il report da inviare sottoforma di URI

- Flag: metadati per l'Intent. Nella nostra implementazione corrispondono ad una serie di Flag per avere il permesso di accesso al report nella memoria interna e di invio tramite email [18]

Inseriamo il set di informazioni nel nostro Intent e, nel momento in cui l'utente segnala al Fragment l'intenzione di voler inviare la Email, esso chiamerà il metodo startActivityForResult(), inserendo come input l'Intent creato ed un codice identificativo della richiesta. Quando l'email risulterà correttamente inviata, il Fragment della funzionalità Email verrà sostituito a quello del Monitor.

```
1  package com.example.monitorcardiaco.email
2  import [...]
3
4  class EmailFragment: Fragment() {
5      override fun onCreateView(inflater: LayoutInflater, container:
       ViewGroup?, savedInstanceState: Bundle?): View? {
6      [...]
7      val file = File(path)
8      val apkURI: Uri = FileProvider.getUriForFile(
9          context!!, context!!.applicationContext
10             .packageName.toString() + ".fileprovider", file)
11
12     viewModel.navigateToOverview.observe(this, Observer {
13
14         val emailSelectorIntent = Intent(Intent.ACTION_SENDTO)
15         emailSelectorIntent.data = Uri.parse("mailto:")
16
17         val emailIntent = Intent(Intent.ACTION_SEND).apply {
18             putExtra(Intent.EXTRA_EMAIL, viewModel.extraEmail)
19             putExtra(Intent.EXTRA_CC, viewModel.extraCCArray)
20             putExtra(Intent.EXTRA_SUBJECT, viewModel.extraSubjectText)
21             putExtra(Intent.EXTRA_TEXT, viewModel.extraEmailText)
22             putExtra(Intent.EXTRA_STREAM, apkURI)
23             addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION)
24             addFlags(Intent.FLAG_GRANT_WRITE_URI_PERMISSION)
25             addFlags(Intent.FLAG_ACTIVITY_NEW_TASK)
26             selector = emailSelectorIntent
27         }
28
29         if (emailIntent.resolveActivity(packageManager) != null) {
30             val chooser = Intent.createChooser(emailIntent, "Invia
       Email..")
```

```
31              startActivityForResult(chooser, EMAIL_REQUEST)
32          }
33      })
34      return binding.root
35      [...]
36  }
```

Listing 3.15: Parte del Fragment contenente il codice responsabile della creazione dell'Intent.

Il Fragment si occupa quindi di gestire i dati inseriti dall'utente, inviarli al ViewModel e poi osservarli tramite LiveData nel momento di creazione dell'Intent.

3.3.7 Grafici

Questa funzionalità è di tipo Display e fa uso della libreria MPAndroidChart [26]. La libreria offre la possibilità di visualizzare numerosi tipi di grafici, ma nella nostra implementazione ci limitiamo a mostrare dei grafici a linee di parametri d'interesse per il medico e per il paziente.

La parte più difficile nell'utilizzo della libreria è la formattazione dei dati in modo da renderli visualizzabili nel grafico. Per farlo, abbiamo creato un BindingAdapter per tale scopo. La classe contiene una serie di funzioni, ciascuna che riceve in input un oggetto LineChart (View di un grafico a linee) ed una lista di giorni. In base al dato che vogliamo inserire, scegliamo un nome, un colore della linea, ed un'unità di misura.

```
1  fun prepareLine(view: LineChart, data: List<Day>, entries: MutableList<
       Entry>, label: String, color: String, unit: String
2  ) {
3      val dataXAxis = mutableListOf<String>()
4      data.forEach {
5          val formatted = it.date.subSequence(0,5) as String
6          dataXAxis.add(formatted)
7      }
8
9      val xAxisFormatter = MyXFormatter(dataXAxis)
10     val labelCount = data.size-1
11
```

```
12      // Formattazione dei valori delle ascisse e delle ordinate
13      [...]
14
15      val dataSet = LineDataSet(entries, label)
16      dataSet.setColor(Color.parseColor(color))
17
18      val dataSets = mutableListOf<ILineDataSet>()
19      dataSets.add(dataSet)
20
21      val lineData = LineData(dataSets)
22
23      with(lineData) {
24          lineData.setValueTextSize(10f)
25          setValueFormatter(MyXFormatter(dataXAxis))
26      }
27      view.data = lineData
28  }
```

Listing 3.16: Metodo prepareLine(), cuore della libreria

Inseriamo i dati come Entries (formato richiesto dalla libreria), ossia una lista di coppie di valori, ciascuna composta da un valore nelle ascisse e dal corrispondente valore nelle ordinate.

Ottenuta la lista di dati, chiamiamo il metodo prepareLine() nel caso in cui il grafico non esista o refreshLine() in caso di dati da aggiornare. Il metodo prepareLine() si occupa di trasformare le Entries in un oggetto di tipo LineDataSet, sul quale sarà possibile compiere operazioni per formattare i dati inseriti e personalizzare il set di dati.

Quest'oggetto sarà poi inserito in una lista di LineDataSet – nel caso volessimo creare più linee nel grafico -, che sarà a sua volta inserito in un oggetto LineData.

Il resto dell'implementazione è semplice: il ViewModel richiede la lista di entità Day alla Repository, il Fragment osserva il LiveData ed assegna la lista al BindingAdapter, che si occuperà di formattare i dati e di visualizzare il grafico correttamente. Inoltre il Fragment mostrerà solo i grafici interessati al tipo di paziente (se non è un paziente LVAD, nasconderà i grafici relativi ai parametri del dispositivo).

3.3.8 Notifica

Questa funzionalità è l'unica dell'applicazione a non essere inserita in un package con un Fragment ed un ViewModel. Consiste nella creazione di una notifica mandata dall'applicazione al paziente, nel caso in cui non abbia inserito tutti i parametri obbligatori dell'entità Day corrispondente al giorno corrente.

```
1  package com.example.monitorcardiaco.overview
2  import [...]
3
4  class OverviewFragment : Fragment() {
5
6      override fun onCreateView(inflater: LayoutInflater, container:
       ViewGroup?, savedInstanceState: Bundle?): View? {
7
8          [...]
9          val intent = Intent(context, AlarmBroadcastReceiver::class.java
       )
10         val pendingIntent = PendingIntent.getBroadcast(context, 1,
       intent, 0)
11
12         val alarmManager = context?.getSystemService(Context.
       ALARM_SERVICE) as AlarmManager
13         alarmManager.cancel(pendingIntent)
14
15         val calendar: Calendar = Calendar.getInstance().apply {
16             timeInMillis = System.currentTimeMillis()
17             set(Calendar.HOUR_OF_DAY, 16)
18         }
19
20         viewModel.day?.observe(this, Observer {
21             if (it == null) {
22                 setAlarm(alarmManager, calendar, pendingIntent)
23             } [...]
24         })
25         return binding.root
26     }
27
```

```
28     private fun setAlarm(alarmManager: AlarmManager, calendar: Calendar
       , pendingIntent: PendingIntent) {
29         alarmManager.setInexactRepeating(
30             AlarmManager.RTC_WAKEUP,
31             calendar.timeInMillis,
32             1000 * 60 * 30,
33             pendingIntent
34         )
35     }}
```

Listing 3.17: Setting dell'allarme nel Fragment del Monitor. Il metodo setAlarm(...) manda un messaggio al BroadcastReceiver.

La sua implementazione - inserita nel Fragment del Monitor, primo Fragment visitato dal paziente dopo l'autenticazione - fa uso di diversi componenti:

- AlarmManager: classe che permette all'applicazione di essere eseguita in un momento specifico. Giunto l'orario prefissato, l'Intent programmato per quel momento viene inviato dal System che fa partire l'applicazione destinataria. Fissiamo quindi un allarme per le ore 16.00

- NotificationBuilder: oggetto appartenente alla libreria NotificationCompat che permette l'attuale costruzione della notifica.

  Prima di tutto scegliamo l'ID del canale in cui sarà contenuta la notifica. Tale implementazione permette al paziente di avere il controllo sulle notifiche in base al canale di appartenenza. L'oggetto Builder crea una notifica utilizzando il canale ed una serie di metodi per l'icona (setSmallIcon(icon)), il titolo (setContentTitle(text)), il contenuto (setContentText(text)) e la priorità della notifica (setPriority(priority)). Il canale viene creato attraverso un'istanza di NotificationChannel e con la sua registrazione nel sistema tramite il NotificationManager

- BroadcastReceiver: una delle maggiori componenti di Android, gestisce l'invio o la ricezione di messaggi broadcast con un meccanismo simile al Publish-Subscribe Design Pattern [7]. Nella nostra implementazione, all'orario specificato, viene creato un Intent che aziona il BroadcastReceiver. Questo ha il compito di azionare il meccanismo di creazione del canale e della notifica

Se le condizioni sopra riportate sono rispettate, una prima notifica sarà inviata all'utente alle 16.00 - orario concordato con i medici -, per poi ripetersi altre tre volte ogni mezz'ora (a meno che non abbia inserito tutti i parametri). In questo modo, la funzionalità rispetta l'obiettivo di fornire un promemoria al paziente senza essere troppo invasiva.

```
1  package com.example.monitorcardiaco
2  import [...]
3
4  class AlarmBroadcastReceiver: BroadcastReceiver() {
5
6      override fun onReceive(context: Context?, intent: Intent?) {
7          [...]
8          if (alarmCount == 3) {
9              [...]
10             alarmManager.cancel(pendingIntent)
11         } else {
12             showNotification(context)
13             [...]
14         }
15     }
16
17     fun showNotification(context: Context?) {
18         [...]
19         val mBuilder: NotificationCompat.Builder
20         val notificationIntent = Intent(context, MainActivity::class.
       java)
21         val bundle = Bundle()
22         notificationIntent.putExtras(bundle)
23         notificationIntent.flags = Intent.FLAG_ACTIVITY_CLEAR_TOP or
       Intent.FLAG_ACTIVITY_MULTIPLE_TASK
24         val contentIntent = PendingIntent.getActivity(context, 0,
       notificationIntent, PendingIntent.FLAG_UPDATE_CURRENT)
25         val mNotificationManager = context.getSystemService(Context.
       NOTIFICATION_SERVICE) as NotificationManager
26         [...]
27         mBuilder = NotificationCompat.Builder(context, channel)
28             .setSmallIcon(com.example.monitorcardiaco.R.mipmap.
```

```
    mc_launcher_red_foreground)
29              .setPriority(Notification.PRIORITY_HIGH)
30              .setContentTitle(context.getString(com.example.
    monitorcardiaco.R.string.notification_title))
31              .setContentIntent(contentIntent)
32              .setContentText("Ricorda di inserire i parametri!")
33              .setAutoCancel(true)
34          mNotificationManager.notify(1, mBuilder.build())
35      }
36  }
```

Listing 3.18: Parte del metodo onReceive() e metodo di costruzione della notifica.

Ogni volta che una notifica viene inviata, nel BroadcastReceiver – nel metodo onReceive() - si incrementa di un'unità una variabile addetta al conto, mantenuta nelle SharedPreferences. Se tale variabile vale 3, la notifica non viene inviata ed il PendingIntent viene cancellato. Sarà ricreato il giorno successivo alle ore 16.00.

3.3.9 Documentazione

Questa funzionalità implementa la specifica di avere una parte informativa all'interno dell'applicazione. All'interno del relativo Fragment, infatti, sono presenti una serie di PDF volti a sensibilizzare il paziente e a dare una serie di informazioni utili riguardo patologie e dispositivi (nel caso di paziente LVAD). La parte fondamentale della funzionalità è il modo in cui è possibile accedere ai file PDF. Abbiamo scartato l'opzione di visualizzare il file direttamente nell'interfaccia grafica perché pesante per l'applicazione e scomoda per l'utente. Abbiamo preferito procedere in questo modo:

- Inseriamo i file PDF all'interno della cartella Assets del codice sorgente, in modo che i pazienti possano automaticamente scaricarli durante l'installazione dell'applicazione sui propri dispositivi

- Alla creazione dell'interfaccia grafica, i file PDF sono copiati e scritti in una cartella appartenente all'applicazione (la stessa in cui vengono creati i report in formato CSV). Tale scelta ha una duplice motivazione: la prima riguarda la presenza dei

file negli Assets, che non sono direttamente accessibili durante l'esecuzione dell'applicazione. La seconda è relativa alla scelta di copiarli come "App-specific files" in modo da non aver bisogno di permessi specifici di scrittura

- Esattamente come nella funzionalità Email, un Intent implicito si occupa di mandare una richiesta per aprire il file PDF a qualsiasi applicazione presente nel dispositivo mobile che sia in grado di farlo

```
private fun copyAssets(asset: AssetManager?, root: String) {
        var files: Array<String>? = null
        val myDir = File(root)
        if (!myDir.exists()) {
            myDir.mkdirs()
        }
        try {
            files = asset?.list("")
        } catch (e: IOException) {
            Log.e("copy", "Failed to get asset file list.", e)
        }
        for (filename in files!!) {
            var `in`: InputStream?
            var out: OutputStream?
            try {
                `in` = asset?.open(filename)
                val outFile = File(myDir, filename)
                out = FileOutputStream(outFile)
                copyFile(`in`!!, out)
                `in`.close()
                `in` = null
                out.flush()
                out.close()
                out = null
            } catch (e: IOException) {
                Log.e("copy", "Failed to copy asset file: $filename", e
    )
            }
        }
```

```
29      }
```

Listing 3.19: Metodo in Java per copiare il file PDF dalla cartella Assets all'External Storage

Il Fragment relativo alla funzionalità si occupa di istanziare Data Binding e ViewModel, e di chiamare il metodo privato copyAssets(), che gestisce le prime due fasi (copiare i file e scriverli nella memoria interna).

3.3.10 Contapassi

Questa funzionalità permette al paziente di tenere traccia dei passi compiuti in una giornata.

Android mette a disposizione due tipi di sensori specifici per il tracking dei passi: lo Step Counter Sensor e lo Step Detector Sensor. Il primo restituisce il numero di passi compiuti dall'utente a partire dall'ultimo riavvio del dispositivo mobile, il secondo registra ogni passo compiuto dall'utente dal momento in cui è attivo.

Come primo approccio abbiamo scelto lo Step Detector Sensor, più vicino al nostro intento. Poiché è l'unica funzionalità indipendente da chi accede all'applicazione, possiamo dividere l'implementazione tra la MainActivity (in modo che il codice venga eseguito ancora prima del login del paziente) ed il package contenente Fragment e ViewModel (corrispondente all'interfaccia grafica in cui viene visualizzato il numero di passi compiuti nel giorno corrente).

Il codice del sensore è contenuto nel ViewModel: al suo interno creiamo una classe StepsLiveData, che estende la classe LiveData<String> ed implementa il SensorEventListener, interfaccia del sensore che si occupa di "ascoltare" l'arrivo di nuovi eventi (cioè passi registrati dal sensore). All'interno, solo se il LiveData è attivo, istanziamo il SensorManager, che ci permette di accedere al sensore interessato (TYPE_STEP_DETECTOR), e registriamo il Listener, in modo tale da registrare i passi compiuti. Quando il LiveData è inattivo (perché il ViewModel è distrutto), si deregistra il componente dal Listener.

Il problema di tale implementazione è la perdita dei passi compiuti dal paziente quando il LiveData è inattivo, cioè quando la MainActivity è distrutta, ovvero se l'applicazione non è aperta (o non è in background).

Il primo tentativo di superare tale problema è stato quello di creare una serie di allarmi durante la giornata che facessero partire un Intent ed aprissero ripetutamente la MainActivity in modo da mantenere attivo il Listener e continuare a tenere il conto dei passi. L'approccio non si è rivelato efficace a causa delle nuove politiche di Android secondo cui un allarme non ha il permesso di aprire un'applicazione, ma solo di far partire eventi come l'invio di una notifica (molto meno invasivo dell'apertura di un'intera applicazione e bloccabile attraverso la gestione dei canali).

Abbiamo quindi valutato di cambiare il tipo di sensore e di utilizzare lo Step Counter. Registriamo, nella MainActivity, il numero di passi in totale restituiti dallo Step Counter e salviamo tale numero nella SharedPreferences. Se nella giornata corrente è già presente un numero totale di passi, non lo sovrascriviamo.

Il senso di tale scelta è quello di tenere il conto del numero di passi che lo smartphone ha registrato prima che ne vengano compiuti degli altri. In questo modo, se il paziente apre l'applicazione – non importa che faccia il Login o compia operazioni -, la MainActivity segna il numero e lo mantiene per tutta la giornata, considerandolo il punto di partenza.

Inoltre, ogni volta che il paziente utilizza l'applicazione, il sensore registra i passi compiuti e ne aggiorna il numero totale, per poi sottrarre il valore corrispondente al punto di partenza. In questo modo otteniamo sempre il numero aggiornato di passi compiuti in un giorno, sia che il paziente abbia aperto l'applicazione o meno.

Infine, nel Fragment osserviamo il valore del LiveData – che aggiorna automaticamente l'interfaccia grafica perché collegato direttamente al sensore -, ed implementiamo una chiamata ad un metodo del ViewModel in modo da aggiornare il numero di passi quotidiani nel Database (per la precisione, nell'entità Day corrente) ogni volta che il valore viene aggiornato.

Per ottenere un risultato il più corretto possibile (ma non esatto, data la natura imprecisa dei dati forniti dai sensori di Android), il paziente dovrebbe aprire l'applicazione appena sveglio e prima di andare a dormire, in modo che l'applicazione registri tutti i passi compiuti nell'arco della giornata.

```
class TrackingViewModel(private val activity: FragmentActivity, private val repository: IRepository, private val date: String, private val email: String, private val initialSteps: Int) : ViewModel() {
    val steps = StepsLiveData()

    inner class StepsLiveData : LiveData<String>(), SensorEventListener {
        private val sensorManager
            get() = activity.getSystemService(Context.SENSOR_SERVICE) as SensorManager

        override fun onSensorChanged(event: SensorEvent) {
            postValue((event.values[0].toInt() - initialSteps).toString())
        }

        override fun onActive() {
            sensorManager.let { sm ->
                sm.getDefaultSensor(Sensor.TYPE_STEP_COUNTER).let {
                    sm.registerListener(this, it, SensorManager.SENSOR_DELAY_NORMAL)
                }
            }
        }
        override fun onInactive() {
            sensorManager.unregisterListener(this)
        }
    }[...]
}
```

Listing 3.20: Implementazione di StepsLiveData.

3.3.11 Allarme

L'ultima funzionalità dell'applicazione è accessibile dal Monitor solo per un paziente di tipo LVAD. Lo scopo è quello di registrare nel Database un allarme e di chiamare un numero di emergenza.

Il Fragment costruisce una View in cui il paziente può scegliere il tipo di allarme tramite un Material Widget e premere il bottone che istanzierà un oggetto Intent. L'Intent è più semplice di quelli implementati per la funzionalità Email e Documentazione, poiché sarà necessario inserire solamente l'azione "ACTION_DIAL", l'URI contenente il numero di emergenza, preceduto da "tel:" ed un codice da associare all'Intent.

Il Fragment azionerà l'Intent che attiverà la funzione di chiamata del dispositivo mobile, con il numero già impostato. In questo modo, oltre a gestire la situazione di emergenza, l'applicazione ha anche registrato l'allarme, dato che ritornerà utile ai medici durante la visione del report bisettimanale.

3.4 Testing

L'insieme dell'architettura e delle scelte implementative realizzate rende l'applicazione facilmente testabile. Durante lo sviluppo, abbiamo svolto test di vario "scope":

- Unit Tests: testano la logica di parti individuali di codice – anche una sola classe o un solo metodo. Hanno il vantaggio di essere veloci, specifici e facili da definire
- Integration Tests: testano un insieme di unità. Il nostro interesse è verificare che nella collaborazione tra unità non ci siano problemi
- End-to-End (E2E) Tests: testano un insieme di Views, di funzionalità, o l'intera applicazione

All'aumentare dello "scope" (quindi della portata del test), diminuisce la velocità e la fedeltà del test. Uno Unit Test è veloce ed affidabile, ma il test si riduce al massimo ad una classe. Un E2E test può testare anche tutta l'applicazione, ma sarà molto lento e probabilmente genererà flaky tests.

Per svolgere i test abbiamo utilizzato le librerie di JUnit, Espresso e Mockito (per creare Mock Objects).

Per alcuni Integration Tests abbiamo creato una Fake Repository ed un Fake Database, in modo da testare componenti che fanno uso di data sources escludendo problemi relativi all'ottenimento dei dati. In questo modo si ha controllo su quali dati vengano prodotti ed è possibile isolare il resto della funzionalità nella componente che si sta testando. Infine, abbiamo utilizzato la strategia del "Given, When, Then" per rendere i test più leggibili.

Il test viene suddiviso così in tre parti:

- Given: setup degli oggetti e dello stato dell'app necessario. Ad esempio, alcuni dati già forniti al momento della funzionalità da testare
- When: azione da compiere sull'oggetto da testare
- Then: controllo delle conseguenze dell'azione. È la parte in cui si verifica se il test è superato o è fallito. In genere contiene Assert Functions

```
1  package com.example.monitorcardiaco.registration
2  import [...]
3
4  @RunWith(AndroidJUnit4::class)
5  @MediumTest
6  @ExperimentalCoroutinesApi
7  class RegistrationFragmentTest {
8  private lateinit var repository: IRepository
9
10     @Before
11     fun initRepository() {
12         repository = FakeAndroidTestRepository()
13         ServiceLocator.repository = repository
14     }
15
16     @Test
17     fun clickRegisterButton_navigateToOverviewFragmentOne() =
    runBlockingTest {
```

```
18
19          repository.insertUser(User("Email","Name", "Surname","LVAD",
20              "Maschio", "18/11/1980", "Italy","Italy", "Italy", null,
    null, null))
21
22          // GIVEN - Siamo sulla schermata di Registrazione
23          val scenario = launchFragmentInContainer<
    RegistrationAccountFragment>(Bundle(), R.style.AppTheme)
24
25          val navController = mock(NavController::class.java)
26          scenario.onFragment {
27              Navigation.setViewNavController(it.view!!, navController)
28          }
29
30          // WHEN - Click sul bottone di registrazione
31          onView(withId(R.id.register_button))
32              .perform(click())
33
34          // THEN - Verifica la navigazione nella schermata di Monitor
35          verify(navController).navigate(
36              RegistrationAccountFragmentDirections.
    actionOverviewFragment())
37      }
38
39      @After
40      fun cleanupDb() = runBlockingTest {
41          ServiceLocator.resetRepository()
42      }
43  }
```

Listing 3.21: Esempio di test con la strategia GWT.

Capitolo 4

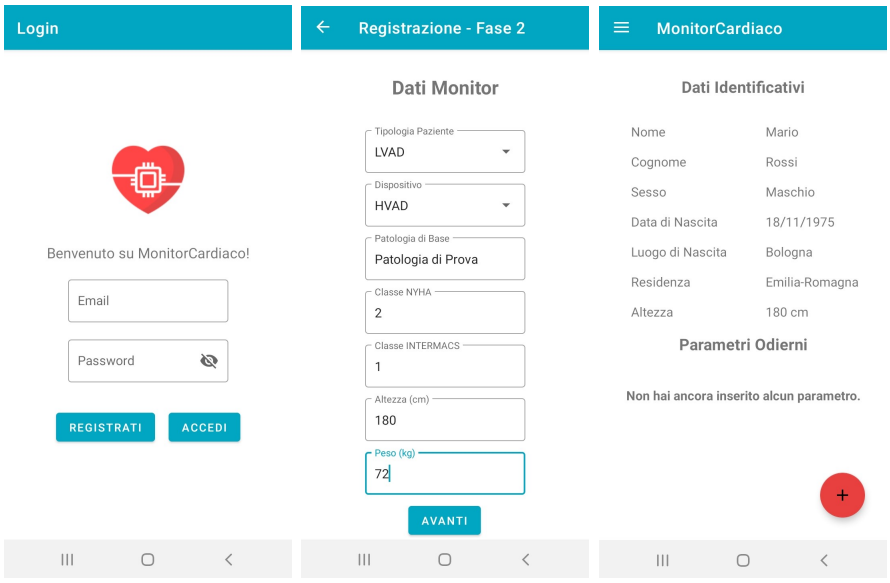
Validazione

Mostriamo adesso una serie di casi d'uso dell'applicazione sviluppata, raggruppati in due user experiences. Il paziente utilizza per la prima volta l'applicazione e dovrà quindi registrarsi e compiere alcune funzionalità. Come vedremo, però, non tutte saranno disponibili in questo momento, perché necessitano di una maggiore quantità di dati per essere eseguite – ad esempio, i grafici necessitano di almeno due giorni per mostrare una variazione delle variabili -. Per questo, nella seconda user experience mostreremo le varie schermate a due settimane dal primo utilizzo. In questo modo presenteremo i meccanismi di esportazione del report ed invio di una email, oltre ad avvicinarci maggiormente ad una situazione di uso reale.

Per alcune funzionalità, la schermata sarà diversa in base al tipo di paziente: il paziente LVAD ha più parametri da inserire e da visualizzare, per questo mostreremo entrambe le schermate per sottolineare la dinamicità dell'interfaccia grafica.

4.1 Registrazione e primo utilizzo

Il paziente apre per la prima volta l'applicazione e visualizza la schermata di Login. Non possedendo un account, clicca sul pulsante "Registrati", entra nella prima fase di registrazione, ed inserisce i suoi dati identificativi – nome, cognome, sesso, data di nascita, luogo di nascita, regione di residenza. Dopo aver compilato i campi, clicca su "Avanti".



(a) (b) (c)

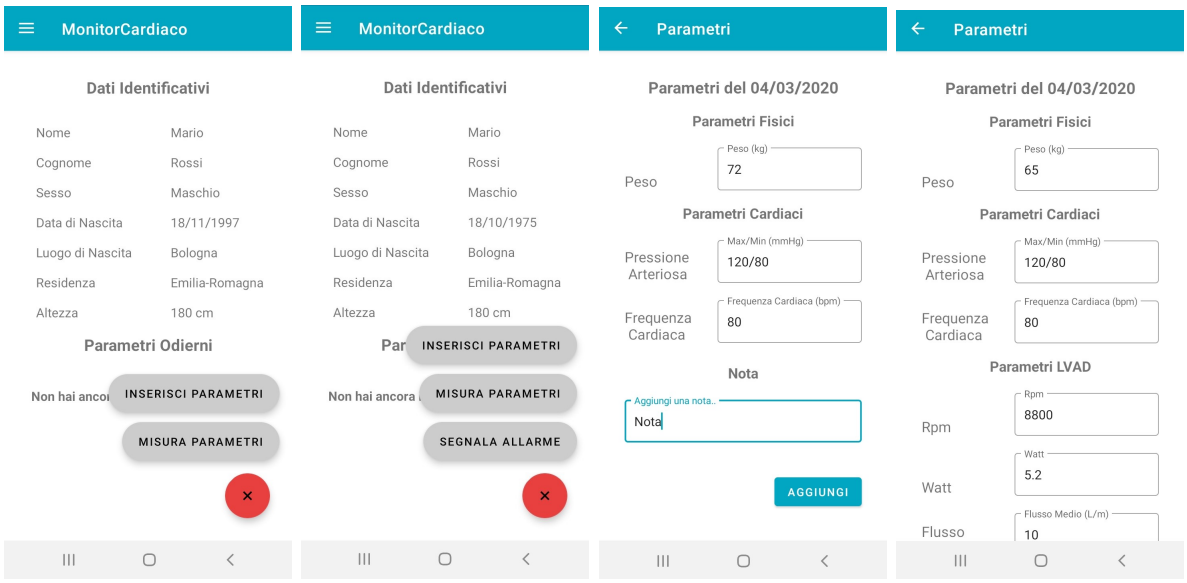
Figura 4.1: (a) Login (b) Registrazione, Fase 2 (c) Login effettuato, schermata Monitor

Nella seconda fase il paziente inserisce i suoi dati "monitor" – tipologia paziente (Scompenso Cardiaco, LVAD), patologia di base, classe NYHA, classe INTERMACS, altezza in centimetri -. Nel caso in cui il paziente scelga "LVAD" come tipologia paziente, la schermata mostra un nuovo campo relativo al dispositivo – HVAD o HM3 -. Tale scelta è importante per indicare all'applicazione quali parametri visualizzare, dato che i due dispositivi sono differenti.

Il paziente clicca nuovamente su "Avanti" e visualizza la terza ed ultima fase della registrazione. Inserisce indirizzo email e password, che userà per l'autenticazione. Clicca su "Registrati" e ritorna nella schermata di Login.

Il paziente compila i campi inserendo l'email e la password scelte alla registrazione e clicca su "Accedi". Il sistema gli impedisce di entrare perché ha inserito erroneamente i dati. Corregge l'errore, riprova e l'applicazione questa volta gli permette l'autenticazione.

Il paziente visualizza il Monitor con all'interno i dati identificativi ed i parametri inseriti nella giornata corrente. Al momento, la schermata segnala al paziente che non



(a) (b) (c) (d)

Figura 4.2: (a) FAB, paziente Scompenso Cardiaco (b) FAB, paziente LVAD (c) Inserisci parametri, paziente Scompenso Cardiaco (d) Inserisci parametri, paziente LVAD

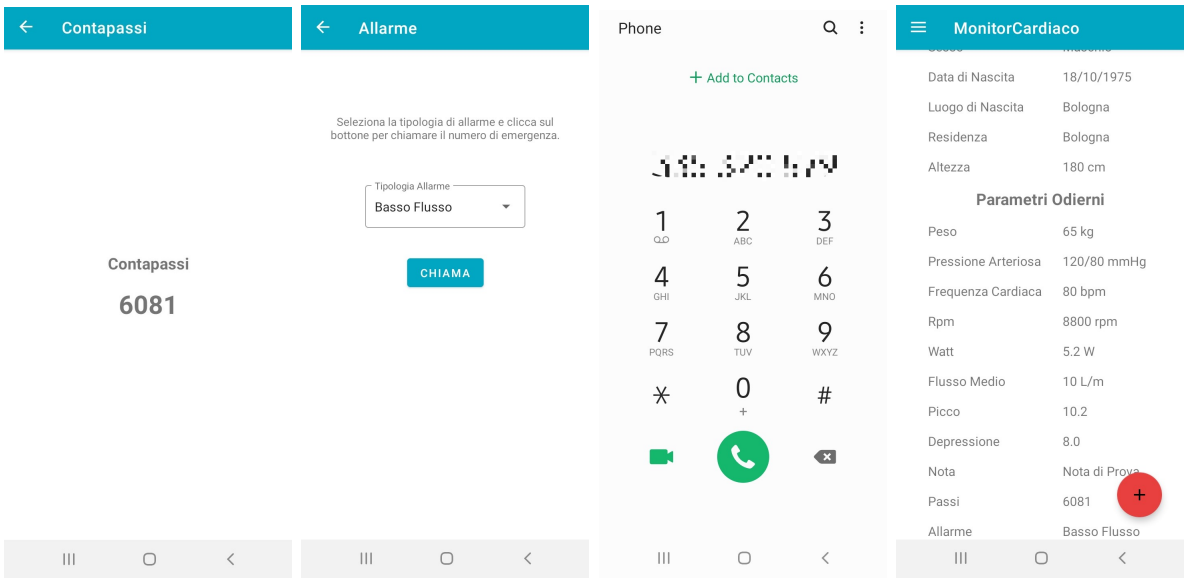
ha inserito alcun parametro.

Come prima azione, il paziente clicca il bottone in basso a destra e poi il primo bottone dall'alto, "Inserisci Parametri". Si apre una schermata indicante i campi compilabili per l'inserimento dei vari parametri.

Nel caso di paziente LVAD, la schermata permette di aggiungere dei parametri specifici, anche in base al dispositivo (Picco e Depressione se HVAD, PI se HM3). Il paziente compila tutti i campi e clicca "Inserisci".

L'applicazione salva i dati e rimanda il paziente nel monitor. Adesso, oltre ai dati identificativi, la schermata mostra la lista dei parametri precedentemente inseriti riferiti al giorno corrente.

Il paziente decide di controllare il numero di passi compiuti nella giornata. Dal monitor, clicca il bottone rosso e poi il bottone "Misura Parametri". Entra nella schermata del contapassi per visualizzare il numero a cui è interessato. Tornando indietro nel Monitor



(a) (b) (c) (d)

Figura 4.3: (a) Contapassi (b) Segnala allarme (c) Chiamata numero di emergenza (numero censurato per privacy) (d) Monitor paziente LVAD aggiornato con tipologia di allarme e passi

– tramite la freccia in alto a sinistra o premendo il tasto "Indietro" del dispositivo mobile -, l'applicazione aggiorna il parametro dei passi nel giorno corrente.

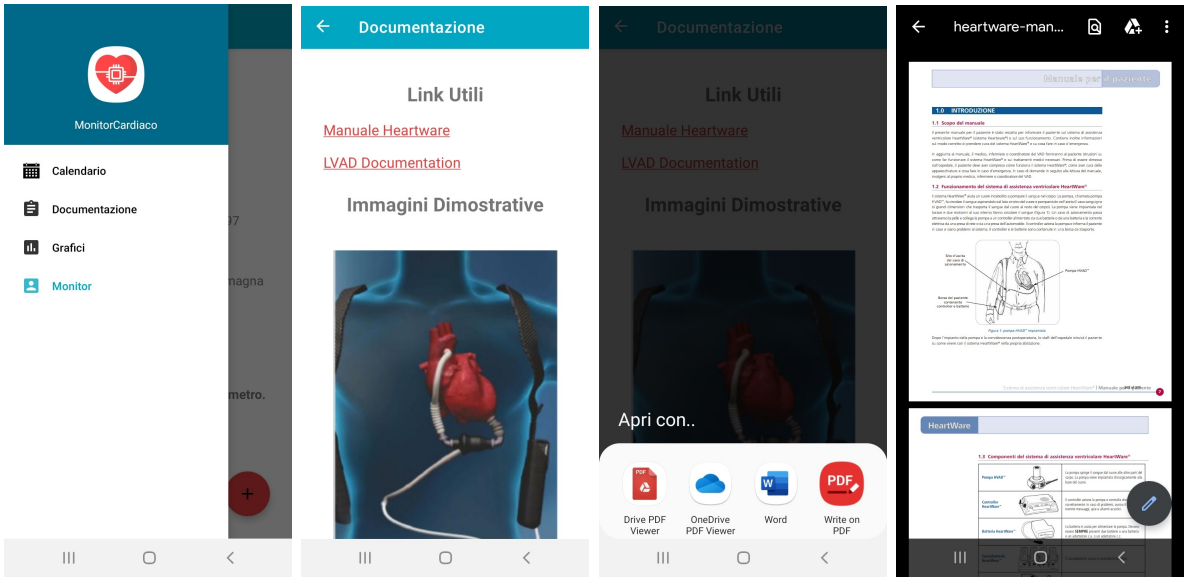
Ipotizzando che un paziente LVAD si trovi in una situazione di allarme – per esempio, il dispositivo segnala un wattaggio troppo alto rispetto al normale -, dal monitor clicca sul bottone rosso, poi sul bottone "Segnala Allarme". Entrato nella schermata apposita, il paziente sceglie la tipologia di allarme e clicca sul bottone "Chiama". L'applicazione inserisce automaticamente il numero di emergenza nella schermata di chiamata del proprio dispositivo mobile.

Terminata la chiamata, il sistema ha già salvato l'allarme, ed è ora visualizzabile tra i parametri inseriti nel giorno corrente.

Per verificare le operazioni da compiere in seguito a situazioni di emergenza, il paziente ritorna nella schermata principale del Monitor, apre il menù di navigazione tramite il

bottone in alto a sinistra, e clicca su “Documentazione”.

La schermata mostra una serie di documenti utili ed informativi. Il paziente clicca su “Manuale Heartware”, sceglie l’applicazione preferita nel proprio dispositivo per visualizzare il PDF, e si informa sulle operazioni da compiere in questa situazione specifica.



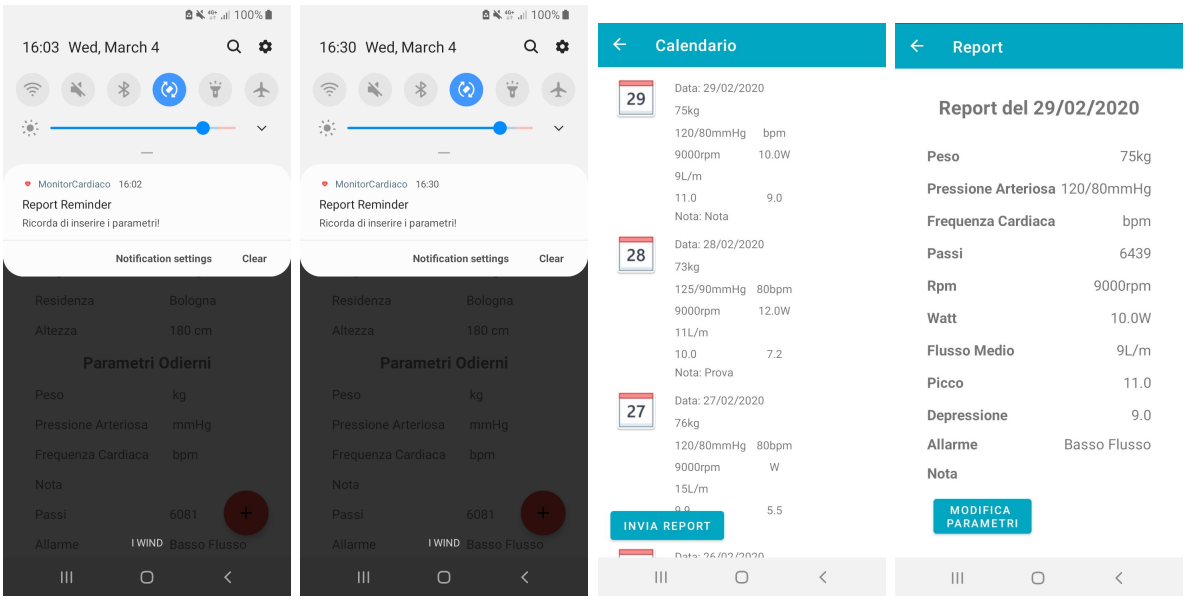
(a) (b) (c) (d)

Figura 4.4: (a) Menù di navigazione (b) Documentazione disponibile come PDF e immagini (c) Schermata per la scelta dell’applicazione (d) PDF

Le restanti funzionalità – Calendario e Grafici - al momento non risultano utili per il paziente perché necessitano di più dati (sono comunque navigabili).

4.2 Utilizzo dopo due settimane

Dopo due settimane di utilizzo quotidiano dell’applicazione, il paziente entra al mattino nel proprio monitor inserendo email e password. Questa volta si dimentica di inserire i parametri quotidiani e prosegue con la sua giornata. Alle ore 16.00 l’applicazione invia una notifica ricordandogli di compilare i parametri.



(a) (b) (c) (d)

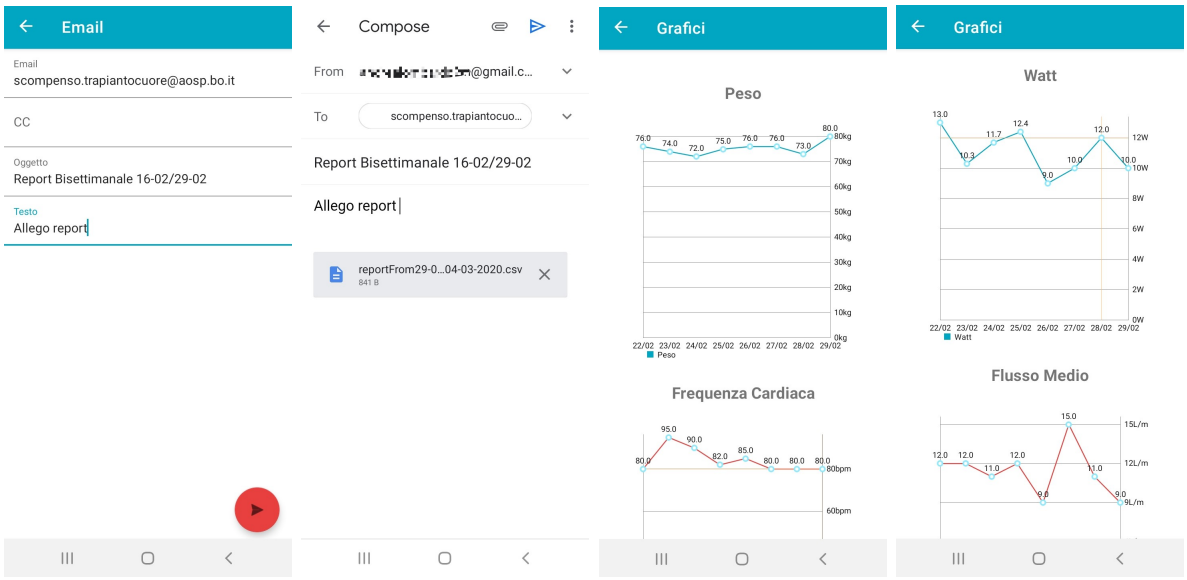
Figura 4.5: (a) Prima notifica quotidiana alle 16.00 (b) Seconda notifica alle 16.30 (c) Calendario (d) Visualizzazione di un giorno nel dettaglio

Il paziente decide di ignorare la notifica. Segue una seconda notifica alle 16.30: questa volta il paziente entra nell'applicazione ed inserisce tutti i parametri. Fino alla giornata successiva il paziente non riceverà più notifiche. Nel caso in cui ignorasse anche la seconda, ne seguirebbe un'altra alle 17.00 ed eventualmente un'ultima alle 17.30.

Il paziente apre il menù di navigazione e clicca su "Calendario". Visualizza una lista contenente tutti i giorni inseriti nel corso dell'utilizzo dell'app e decide di inviare al medico un report delle ultime due settimane.

Controllando la lista, nota che un giorno presenta alcuni parametri sbagliati. Clicca sul giorno, visualizza in dettaglio la lista di parametri e clicca su "Modifica Parametri" per correggere gli errori. L'applicazione salva i cambiamenti e riporta il paziente al calendario.

Assicuratosi che tutti i dati siano corretti, il paziente clicca su "Invia Report". L'applicazione esporta il report nella memoria interna, visualizza la schermata dell'Email,



(a) (b) (c) (d)

Figura 4.6: (a) Schermata Email (b) Invio della mail sull'applicazione scelta (c) Esempi di Grafici, Paziente Scompenso Cardiaco (d) Esempi di Grafici, Paziente LVAD

compila automaticamente il destinatario e permette al paziente di inserire oggetto della email e testo (ed eventuali indirizzi email in CC), allegando automaticamente il report.

Il paziente clicca il bottone rosso, sceglie la sua applicazione preferita per inviare la mail, e visualizza nell'app scelta tutti i dati da lui inseriti – insieme al report.

Al termine dell'operazione, l'applicazione visualizza il monitor, permettendo al paziente di proseguire la navigazione.

Il paziente apre nuovamente il menù di navigazione e clicca su "Grafici". Nel caso di paziente Scompenso Cardiaco, la schermata mostra il grafico della variazione nel tempo del peso e quello della frequenza cardiaca. Il paziente LVAD, invece, ha la possibilità di visualizzare anche i grafici relativi ai parametri del dispositivo – Watt e Flusso Medio, Picco e Depressione nel caso di HVAD, PI nel caso di HM3.

Capitolo 5

Conclusioni e sviluppi futuri

Questa tesi è nata con lo scopo di analizzare e presentare una soluzione concreta all'esigenza di uno standard di follow up medico remoto per pazienti con scompenso cardiaco e portatori di LVAD.

Nel capitolo 1 abbiamo discusso il contesto delle applicazioni mobili, mostrando le varie tipologie ed i maggiori sistemi operativi. Abbiamo poi concentrato la nostra attenzione sul campo del Mobile Health e delle applicazioni mobili più vicine alla nostra, per avere dei chiari riferimenti durante lo sviluppo.

Nel capitolo 2 abbiamo introdotto il punto di partenza del nostro lavoro analizzando gli obiettivi posti e le specifiche richieste per l'applicazione, comprese le funzionalità da realizzare. Abbiamo poi ampiamente discusso l'architettura sviluppata per il nostro software mobile, partendo da un'architettura general-purpose poi adattata alle nostre esigenze, applicando anche diversi design pattern. Il risultato finale è una struttura modulare, con ciascuna componente astratta e scollegata (decoupled) dalle altre, facilmente testabile e agnostica nei confronti dei dettagli implementativi.

Nel capitolo 3 abbiamo implementato ciascuna componente architetturale descrivendo le tecnologie, le librerie e i framework utilizzati, ma anche descrivendo estensivamente i dettagli implementativi scelti per l'applicazione. Abbiamo poi esposto l'implementazione di ciascuna funzionalità richiesta dalle specifiche, mostrando come questa rispecchi l'obiettivo architetturale di realizzare un software modulare e flessibile. Infine, abbiamo introdotto il nostro lavoro di testing delle varie componenti.

Nel capitolo 4 abbiamo presentato due user experiences, rappresentate dal primo utilizzo dell'applicazione da parte di un paziente (sia di tipo Scompenso Cardiaco che LVAD) e dall'utilizzo a due settimane dalla registrazione, per esplorare e dimostrare le funzionalità in azione ed il modo in cui ciascuna componente interagisce tra loro in una situazione di uso reale. Abbiamo quindi dimostrato la dinamicità dell'interfaccia grafica e della logica implementata mostrando le varie schermate che si adattano in base al tipo di paziente che utilizza l'applicazione mobile.

5.1 Sviluppi futuri

Il lavoro di questa tesi vuole essere solo il primo contributo per raggiungere un gold standard di follow up medico che, oltre a migliorare la qualità di vita, la consapevolezza e la disciplina del paziente in relazione al suo stato di salute, permetta ai medici una riduzione dei costi ed un monitoraggio continuativo. Il prossimo punto sarà la validazione clinica del software effettuata dal Policlinico Sant'Orsola-Malpighi e la conseguente scrittura di un articolo scientifico.

Concentrandoci sull'applicazione mobile, abbiamo identificato tre maggiori aspetti in cui l'applicazione continuerà ad essere sviluppata. Tale distinzione non ha carattere esclusivo ma solo organizzativo, poichè il perseguimento di ognuno contribuirebbe ad un miglioramento complessivo del software.

- Lato progettazione e implementazione: partendo dall'architettura, si potrebbe astrarre ancora di più il Domain Layer, estraendo gli Use Cases dal Presentation Layer. In questo modo la totalità dei dettagli implementativi potrebbe essere sostituita con facilità, semplificando un possibile futuro progetto di realizzare l'applicazione anche per sistemi iOS. Ancora, si potrebbe accompagnare (o sostituire del tutto) al database un server su cui salvare e mantenere i dati del paziente, ed implementare una cache locale in caso di connessione internet assente. Oltre a risparmiare memoria, la presenza di un server permetterebbe lo sviluppo di meccanismi di autenticazione più robusti e protetti.

- Lato funzionalità: potremmo aggiungere una serie di funzionalità per arricchire l'applicazione e aumentare le possibilità per il paziente. La prima potrebbe essere rappresentata dalla gestione delle terapie, con una schermata che permetta di inserire la terapia ed i vari orari di assunzione dei medicinali, con relativa notifica come reminder. Ancora, si potrebbe ampliare la funzione di tracking di parametri attraverso i sensori Android permettendo ad esempio al paziente di misurare la propria frequenza cardiaca, o calcolando il numero di calorie bruciate in un giorno. La funzionalità di invio del report bisettimanale potrebbe essere generalizzata e resa disponibile anche per altri scopi, come l'invio di fotografie di referti in caso di screening del paziente. Infine, una funzionalità che potrebbe fare la differenza nell'obiettivo di miglioramento della qualità di vita dei pazienti è rappresentata dalla presenza nell'applicazione di questionari di valutazione del paziente. Nella relativa schermata, il paziente potrebbe comunicare l'ammontare di fatica percepita durante la giornata, i piani di scale percorsi senza affanno, etc. per verificare i progressi della terapia.

- Lato interfaccia medico: potremmo creare un'interfaccia destinata all'uso da parte dei medici, con scopo di controllo e modifica della terapia. Rappresenterebbe un salto di qualità per l'applicazione mobile, poichè realmente si riuscirebbe a delocalizzare alcune operazioni che fino a questo momento necessitano della presenza fisica del paziente e del medico nella struttura ospedaliera. Assumendo che sia stato implementato un server, potremmo prendere i dati di tutti i pazienti gestiti da un medico e metterli a disposizione, per la visualizzazione o la modifica, nella sua interfaccia personale. Il medico potrebbe ad esempio monitorare in tempo reale gli ultimi parametri inseriti dai pazienti, e modificare opportunamente le varie terapie, aggiornando automaticamente i vari profili. I pazienti con le terapie modificate riceverebbero così una notifica di reminder della nuova terapia.

Ringraziamenti

Ringrazio il Prof. Di Felice per avermi dato l'opportunità di svolgere questo argomento di tesi, ma soprattutto di avermi dato fiducia nel riuscire a terminare il lavoro in tempi davvero ristretti.

Ringrazio Silvia, del Policlinico Sant'Orsola-Malpighi, per avermi dato con entusiasmo un punto di vista umano in quello che si sarebbe potuto limitare ad essere un freddo sviluppo di software. Il suo profondo desiderio di migliorare la qualità di vita dei suoi pazienti mi ha colpito molto, oltre ad essersi rivelato una forte motivazione durante le notti di lavoro di questi ultimi mesi.

Ringrazio la mia attuale coinquilina Giordana e Giulia, da poco entrate a far parte della mia vita, ma che mi hanno accompagnato come delle sorelle - e la domenica come delle nonne - durante tutto il percorso di tirocinio e tesi. Grazie a loro ho imparato che basta una risata condivisa, una storia raccontata o una gara di freccette per rendere piacevoli persino i momenti più difficili.

Ringrazio i miei genitori per avermi dato la libertà e la possibilità di vivere e studiare a Bologna ed i miei fratelli per avermi sempre sostenuto. Anche se tutti distanti, ogni giorno vi sento con me. Siete le solide fondamenta che mi permettono di rimanere in piedi qualsiasi avversità mi si presenti davanti.

Ringrazio Matteo (Deggi) per tutto ciò che mi ha insegnato in questi anni di amicizia sincera. Da un banco di scuola ad una pizza famiglia di martedì sera, dal primo euro guadagnato online alle risate senza fine dopo ore di riflessioni profonde sulla vita. Mettiamoci anche della sana competizione in tutto ciò che facevamo ed ecco che otteniamo

quell'amicizia rara se non unica, che in molti sognano senza mai riuscire ad averla. La conferma di ciò l'ho avuta quando, qualche mese fa, abbiamo intrapreso per la prima volta due percorsi diversi, in due città diverse. A parte non sentire più il suo naso tirato in su centinaia di volte al giorno a causa di un costante raffreddore, non è cambiato assolutamente nulla. E non c'è niente di più autentico di un'amicizia indipendente dalle circostanze.

Ringrazio le persone che sono state al mio fianco in questi anni, senza le quali non sarei la persona che sono ora. Parlo di quelle persone su cui posso sempre contare, di quelle per cui farei di tutto pur di vederli felici, e di quelle che anche solo per un breve periodo hanno deciso di condividere con me la propria vita e le proprie sfide. Non smetterò mai di esservi grato.

Ringrazio i ragazzi di Psicodizione, compagni di squadra formidabili nell'obiettivo comune di raggiungere la libertà comunicativa. Le esperienze condivise con voi, come la sensibilizzazione in piazza o nelle scuole, mi riempiono il cuore di gioia.

Infine, ringrazio tutte le persone che in qualsiasi modo hanno arricchito la mia vita, persino con una conversazione durata pochi minuti o uno sguardo denso di significato.

Approfitto di questo spazio per lasciare un messaggio a chiunque stia leggendo, me compreso: non diamo mai per scontate le persone che ci fanno stare bene. Il loro amore è ciò che ci fa andare avanti, sempre.

Bibliografia

[1] A. Ahmad et al. «An Empirical Study of Investigating Mobile Applications Development Challenges». In: *IEEE Access* 6 (2018), pp. 17711–17728.

[2] Domenico Amalfitano et al. «Chapter 1 - Testing Android Mobile Applications: Challenges, Strategies, and Approaches». In: a cura di Atif Memon. Vol. 89. Advances in Computers. Elsevier, 2013, pp. 1–52.

[3] *Apple unveils all-new App Store*. Giu. 2017. URL: `https://www.apple.com/newsroom/2017/06/apple-unveils-all-new-app-store/`.

[4] Holst Arne. *Smartphone Users Worldwide 2020*. Nov. 2019. URL: `https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/`.

[5] F. A. B. Azhar e J. S. Dhillon. «A systematic review of factors influencing the effective use of mHealth apps for self-care». In: *2016 3rd International Conference on Computer and Information Sciences (ICCOINS)*. Ago. 2016, pp. 191–196.

[6] Luca Bedogni, Marco Di Felice e Luciano Bononi. «By train or by car? Detecting the user's motion type through smartphone sensors data». In: *2012 IFIP Wireless Days*. IEEE. 2012, pp. 1–6.

[7] *Broadcasts Overview*. URL: `https://developer.android.com/guide/components/broadcasts`.

[8] Anderson C. *Pro Business Applications with Silverlight 5*. Berkeley, Apress, 2012. Cap. The Model-View-ViewModel (MVVM) Design Pattern, pp. 461–499.

[9] Lorenzo Campedelli. «Discrepanze tra qualità di vita percepita e dati clinico-strumentali nei pazienti con scompenso cardiaco avanzato». Tesi di laurea mag. Alma Mater Studiorum - Università di Bologna, Scuola di Medicina e Chirurgia, 2019, pp. 14-49–50.

[10] Jennifer K Carroll et al. «Who Uses Mobile Phone Health Apps and Does Use Matter? A Secondary Data Analytics Approach». In: *J Med Internet Res* 19.4 (apr. 2017), e125.

[11] M. Chau e R. Reith. *Smartphone Market Share*. Gen. 2020. URL: `https://www.idc.com/promo/smartphone-market-share/os`.

[12] J. Clement. *Mobile app usage - Statistics & Facts*. Ago. 2019. URL: `https://www.statista.com/topics/1002/mobile-app-usage/`.

[13] *Comparison to Java Programming Language*. URL: `https://kotlinlang.org/docs/reference/comparison-to-java.html`.

[14] *DiffUtil*. URL: `https://developer.android.com/reference/androidx/recyclerview/widget/DiffUtil`.

[15] *Distribution Dashboard*. URL: `https://developer.android.com/about/dashboards`.

[16] I. Frederix et al. «MobileHeart, a mobile smartphone-based application that supports and monitors coronary artery disease patients during rehabilitation». In: *2016 38th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. Ago. 2016, pp. 513–516.

[17] Enzo Galligioni et al. «Integrating mHealth in Oncology: Experience in the Province of Trento». In: *J Med Internet Res* 17.5 (mag. 2015), e114.

[18] *Intents and Intent Filters*. URL: `https://developer.android.com/guide/components/intents-filters`.

[19] Arshia Khan. «Objective-C and IOS Programming: a Simplified Approach to Developing Apps for the Apple IPhone and IPad». In: a cura di Cengage Learning. 2015. Cap. Working in the IOS Environment, p. 203.

[20] S. Krishnaswamy, J. Gama e M. M. Gaber. «Mobile Data Stream Mining: From Algorithms to Applications». In: *2012 IEEE 13th International Conference on Mobile Data Management*. Lug. 2012, pp. 360–363.

[21] Yuting Lin et al. «Effective Behavioral Changes through a Digital mHealth App: Exploring the Impact of Hedonic Well-Being, Psychological Empowerment and Inspiration». In: *JMIR Mhealth Uhealth* 6.6 (giu. 2018), e10024.

[22] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. 1st. USA: Prentice Hall Press, 2017.

[23] *Measuring Digital Development*. Rapp. tecn. 2019, pp. 1–8.

[24] Alexander Mertens et al. *A mobile application improves therapy-adherence rates in elderly patients undergoing rehabilitation: A crossover design study comparing documentation via iPad with paper-based control*. Set. 2016. URL: `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5023861/`.

[25] *mHealth Developer Economics*. Rapp. tecn. Mar. 2018, pp. 5–7. URL: `https://research2guidance.com/mhealth-app-developer-economics/`.

[26] *MPAndroidChart*. URL: `https://github.com/PhilJay/MPAndroidChart`.

[27] Jennifer Nicholas et al. «Mobile Apps for Bipolar Disorder: A Systematic Review of Features and Content Quality». In: *J Med Internet Res* 17.8 (ago. 2015), e198.

[28] *Number of Android Applications*. Feb. 2020. URL: `https://www.appbrain.com/stats/`.

[29] Hannah E Payne et al. «Behavioral Functionality of Mobile Apps in Health Interventions: A Systematic Review of the Literature». In: *JMIR mHealth uHealth* 3.1 (feb. 2015), e20.

[30] Dott. Luciano Potena et al. «Archivio elettronico per la terapia medica ed interventistica avanzata nello scompenso cardiaco severo». Apr. 2017.

[31] Isa Rudolf et al. «Assessment of a Mobile App by Adolescents and Young Adults With Cystic Fibrosis: Pilot Evaluation». In: *JMIR Mhealth Uhealth* 7.11 (nov. 2019), e12442.

[32] *The lifecycle of a ViewModel*. URL: https://developer.android.com/topic/libraries/architecture/viewmodel.

[33] *The top 10 causes of death*. URL: https://www.who.int/en/news-room/fact-sheets/detail/the-top-10-causes-of-death.

[34] *Using Apache Commons CSV*. URL: https://commons.apache.org/proper/commons-csv/.

[35] Isaac Vaghefi e Bengisu Tulu. «The Continued Use of Mobile Health Apps: Insights From a Longitudinal Study». In: *JMIR Mhealth Uhealth* 7.8 (ago. 2019), e12983.

[36] Y. Tony Yang e Ross D. Silverman. «Mobile Health Applications: The Patchwork Of Legal And Liability Issues Suggests Strategies To Improve Oversight». In: *Health Affairs* 33.2 (2014), pp. 222–227.