UNIVERSITÀ DI PISA

DIPARTIMENTO DI INFORMATICA

REALIZZAZIONE DI UN'APP PER IOS PER L'ACQUISIZIONE DI DATI DA DISPOSITIVI WEARABLE PER FINI SPERIMENTALI

TUTOR ACCADEMICO:
CORRADO PRIAMI

ELABORATO DI LAUREA DI:
SIMONE PICA

TUTOR AZIENDALE:
DAVIDE MORELLI

Anno accademico 2018/2019

Indice dei contenuti

CAPITOLO 1 **INTRODUZIONE** ... **1**

CAPITOLO 2 **BASI DI PARTENZA** ... **8**

CAPITOLO 3 **TECNOLOGIE UTILIZZATE** ... **9**

3.1 AMBIENTE DI SVILUPPO XCODE ... 9

3.2 LINGUAGGIO DI PROGRAMMAZIONE SWIFT ... 10

3.3 FRAMEWORK: COREGRAPHICS, COREBLUETOOTH, COREDATA ... 10

3.4 SLACK ... 11

3.5 SOURCE CONTROL E GITHUB ... 11

3.6 MICROSOFT EXCEL ... 13

CAPITOLO 4 **ANALISI DEI REQUISITI** ... **14**

4.1 GLOSSARIO DEI TERMINI ... 15

4.2 REQUISITI FUNZIONALI ... 16

4.3 REQUISITI NON FUNZIONALI ... 17

4.4 CASI D'USO ... 17

4.4.1 Casi d'uso con attore principale l'utente ... 17

4.4.1.1 Narrativa caso d'uso: Estrai dati ... 18

4.4.1.2 Narrativa caso d'uso: Visualizza dati ... 19

4.4.1.3 Narrativa caso d'uso: Acquisisci dati ... 20

4.4.1.4 Narrativa caso d'uso: Interrompi acquisizione dati ... 20

4.4.1.5 Narrativa caso d'uso: Connetti periferica ... 21

4.4.1.6 Narrativa caso d'uso: Disconnetti periferica ... 22

4.4.1.7 Narrativa caso d'uso: Ricerca periferiche ... 22

4.4.1.8 Narrativa caso d'uso: Interrompi ricerca periferiche ... 23

4.4.2 Casi d'uso con attore principale la periferica ... 24

4.4.2.1 Narrativa caso d'uso: Memorizza dato ... 24

4.5 DIAGRAMMA DELLE CLASSI DI ANALISI ... 25

CAPITOLO 5 **DIAGRAMMA DELLE CLASSI E DI SEQUENZA** ... **26**

5.1 DIAGRAMMA DELLE CLASSI DI PROGETTAZIONE 26
5.2 DIAGRAMMI DI SEQUENZA 27
5.2.1 Diagramma di sequenza: Estrai dati 28
5.2.2 Diagramma di sequenza: Acquisisci dati ed interrompi acquisizione 29
5.2.3 Diagramma di sequenza: Connetti periferica 30
5.2.4 Diagramma di sequenza: Disconnetti periferica 31
5.2.5 Diagramma di sequenza: Ricerca periferiche 32
5.2.6 Diagramma di sequenza: Interrompi ricerca periferiche 33
5.2.7 Diagramma di sequenza: Memorizza dato 33
5.2.8 Diagramma di sequenza: Visualizza dati 34
CAPITOLO 6 **ARCHITETTURA SOFTWARE ED IMPLEMENTAZIONE 35**
6.1 ORGANIZZAZIONE E SUDDIVISIONE DEI MODULI DEL PROGETTO 35
6.2 ARCHITETTURA DELL'APPLICAZIONE 36
6.3 PATTERN MODEL-VIEW-CONTROLLER 38
6.4 PATTERN PUBLISH-SUBSCRIBE 40
6.4.1 Infrastruttura publish-subscribe dell'applicazione 41
6.4.1.1 Definizione dei protocolli 41
6.4.1.2 Implementazione del dispatcher 42
6.5 FRAMEWORK COREBLUETOOTH 44
6.6 DRIVER DI COMUNICAZIONE CON LA PERIFERICA 45
6.6.1 Il driver e le dipendenze con il Protobuf di Google 48
6.6.2 Interazione tra Objective-C e Swift 49
6.6.3 Estensione del driver con nuove funzionalità 49
6.7 I CONTROLLORI DELL'APPLICAZIONE 51
6.7.1 Struttura del file JSON generato dall'applicazione 54
6.8 CICLO DI VITA DI UN'APPLICAZIONE IOS 58
6.8.1 Esecuzione dell'applicazione in background 59
6.8.1.1 Utilizzo di CoreBluetooth in background 60
CAPITOLO 7 **PROGETTAZIONE E REALIZZAZIONE DATABASE 64**
7.1 ANALISI DEI DATI 64
7.1.1 Collezione periferiche 65
7.1.2 Collezione sessioni 65
7.1.3 Collezione dati sonno 66

7.1.4 Collezione passi *66*
7.1.5 Collezione frequenze cardiache *67*
7.1.6 Collezione variabilità RR *67*
7.2 ANALISI DELLE OPERAZIONI 68
7.2.1 Operazioni periferiche *68*
7.2.2 Operazioni sessioni *68*
7.2.3 Operazioni sui dati *69*
7.2.3.1 Operazioni sulla frequenza cardiaca 69
7.2.3.2 Operazioni sui passi 70
7.3 PROGETTAZIONE CONCETTUALE 70
7.4 PROGETTAZIONE LOGICA 71
7.5 FRAMEWORK COREDATA 73
7.5.1 I contesti di CoreData ed esecuzione in background *75*
7.6 ESECUZIONE DELLE QUERY E RECUPERO DEI DATI 78
7.7 REALIZZAZIONE DEL DATABASE DELL'APPLICAZIONE 79
7.7.1 Organizzazione dei file *80*
7.7.2 Data Model dell'applicazione *81*
7.7.3 Gerarchia di classi e gerarchia di entità *84*
7.7.4 Gestore del database e contesti *86*
7.7.5 Alcune query con la relativa implementazione *89*
7.7.5.1 Recupero ultima periferica connessa 89
7.7.5.2 Inserimento nuovo passo 90
7.7.5.3 Recupero numero massimo di passi 92
7.7.5.4 Statistiche frequenza cardiaca 93
CAPITOLO 8 **PROGETTAZIONE E REALIZZAZIONE INTERFACCIA UTENTE 94**
8.1 FRAMEWORK COREGRAPHICS 94
8.1.1 Il contesto grafico *95*
8.1.1.1 Matrice di trasformazione e coordinate omogenee 96
8.1.2 Sistemi di coordinate di Quartz 2D *97*
8.2 STORYBOARD DELL'APPLICAZIONE 98
8.2.1 Elementi UIKit utilizzati nell'applicazione *100*
8.3 ORGANIZZAZIONE DEI FILE 101
8.3.1 Pacchetto TabBar *101*
8.3.2 Pacchetto Buttons *102*

8.3.3 Pacchetto Components *103*
8.3.4 Pacchetto Icons *105*
8.4 PANORAMICA DELL'INTERFACCIA GRAFICA 107
8.4.1 Pagina di monitoraggio *110*
8.4.2 Pagina per l'esportazione dei dati *111*
8.4.3 Pagina impostazioni *112*
8.4.4 Pagine per la visualizzazione dei dati *114*
8.4.5 Visualizzazione degli errori *116*
CAPITOLO 9 **TEST DELL'APPLICAZIONE** **118**
9.1 RECUPERARE LA RAPPRESENTAZIONE DI UNA PERIFERICA 119
9.2 FUNZIONE DI AGGREGAZIONE SULLA FREQUENZA CARDIACA 121
9.3 GENERAZIONE PREDICATO SUL PERIODO PER LE QUERY 123
CAPITOLO 10 **ANALISI DEI DATI ACQUISITI** **125**
10.1 ANALISI DEL NUMERO DI PASSI E DISTANZE PERCORSE 126
10.2 ANALISI DELLE FREQUENZE CARDIACHE 127
10.3 ANALISI DELLA QUANTITÀ E QUALITÀ DEL SONNO 128
CAPITOLO 11 **CONCLUSIONI** **130**
BIBLIOGRAFIA .... **I**

Indice delle figure

FIGURA 1 GITFLOW WORKFLOW DI VINCENT DRIESSEN [11] 12
FIGURA 2 CASI D'USO AVVIATI DALL'UTILIZZATORE DEL SISTEMA 18
FIGURA 3 CASI D'USO AVVIATI DALLA PERIFERICA 24
FIGURA 4 DIAGRAMMA DELLE CLASSI (ANALISI) 25
FIGURA 5 DIAGRAMMA DELLE CLASSI (PROGETTAZIONE) 26
FIGURA 6 DIAGRAMMA DI SEQUENZA PER ESTRARRE I DATI 28
FIGURA 7 DIAGRAMMA DI SEQUENZA PER ACQUISIRE I DATI 29
FIGURA 8 DIAGRAMMA DI SEQUENZA PER INTERROMPERE L'ACQUISIZIONE DEI DATI 29
FIGURA 9 DIAGRAMMA DI SEQUENZA ERRORE ACQUISIZIONE DATI 30
FIGURA 10 DIAGRAMMA DI SEQUENZA PER CONNETTERSI AD UNA PERIFERICA 30
FIGURA 11 DIAGRAMMA DI SEQUENZA PER DISCONNETTERSI DA UNA PERIFERICA 31
FIGURA 12 DIAGRAMMA DI SEQUENZA PER RICERCARE PERIFERICHE 32
FIGURA 13 DIAGRAMMA DI SEQUENZA PER INTERROMPERE LA RICERCA DI PERIFERICHE 33
FIGURA 14 DIAGRAMMA DI SEQUENZA PER MEMORIZZARE UN DATO 33
FIGURA 15 DIAGRAMMA DI SEQUENZA PER VISUALIZZARE I DATI 34
FIGURA 16 VISTA STRUTTURALE DI DECOMPOSIZIONE DELLA ROOT DEL PROGETTO 35
FIGURA 17 VISTA IBRIDA (DISLOCAZIONE E C&C) DELL'ARCHITETTURA DEL SISTEMA 36
FIGURA 18 RAPPRESENTAZIONE AD ALTO LIVELLO DELL'ARCHITETTURA DELL'APPLICAZIONE 37
FIGURA 19 RESPONSABILITÀ MVC 39
FIGURA 20 CONTENUTO PACCHETTO PUBLISHSUBSCRIBE 41
FIGURA 21 VISTA COMPORTAMENTALE PUBLISH-SUBSCRIBE 42
FIGURA 22 STRUTTURA DEL DISPATCHER 42
FIGURA 23 RAPPRESENTAZIONE DELLE CLASSI DI COREBLUETOOTH 44
FIGURA 24 ORGANIZZAZIONE DI UNA CBPERIPHERAL 45
FIGURA 25 INIZIALIZZAZIONE E UTILIZZO DEL DRIVER 46
FIGURA 26 VISTA STRUTTURALE (DECOMPOSIZIONE ED USO) DEL PACCHETTO DRIVEREXTENSION 49
FIGURA 27 VISTA STRUTTURALE DEI CONTROLLORI 52
FIGURA 28 RIASSUME I PASSI NECESSARI PER INTERPRETARE I DATI SONNO 53
FIGURA 29 MOSTRA AD ALTO LIVELLO I COMPONENTI COINVOLTI NELL'ESTRAZIONE DEI DATI 54
FIGURA 30 FILE JSON GENERATO DALL'APPLICAZIONE 56

FIGURA 31 CICLO DI VITA DI UN'APP IOS [17] 58
FIGURA 32 DISPATCHER DEGLI EVENTI IN FOREGROUND E BACKGROUND 61
FIGURA 33 GARANTIRE LA CONSISTENZA TRA LA VISTA ED IL MODELLO 62
FIGURA 34 SCHEMA CONCETTUALE 70
FIGURA 35 SCHEMA LOGICO 71
FIGURA 36 COREDATASTACK [21] 75
FIGURA 37 VISTA STRUTTURALE (DECOMPOSIZIONE ED USO) DEL PACCHETTO DATABASE 80
FIGURA 38 RELAZIONE TRA COREDATA E SQLITE 81
FIGURA 39 MODELLO DEI DATI IN COREDATA 83
FIGURA 40 MODELLO DEI DATI ALTERNATIVO 84
FIGURA 41 GERARCHIA DELLE CLASSI E DELLE ENTITÀ 85
FIGURA 42 PASSI PER OTTENERE L'OGGETTO JSON DI UN DATO 86
FIGURA 43 RUOLI DEL GESTORE DEL DATABASE 87
FIGURA 44 ARCHITETTURA DEL GESTORE DEL DATABASE (COREDATACONTROLLER) 87
FIGURA 45 IL CONTESTO GRAFICO È MODALE 95
FIGURA 46 RISULTATO DEL CLIPPING 96
FIGURA 47 SISTEMA DI COORDINATE DI QUARTZ 98
FIGURA 48 SISTEMA DI COORDINATE DI UIKIT 98
FIGURA 49 STORYBOARD DELL'APPLICAZIONE 99
FIGURA 50 VISTA STRUTTURALE DEL PACCHETTO USERINTERFACE 101
FIGURA 51 MOSTRA IL DISEGNO REALIZZATO DA UITABMENU 102
FIGURA 52 BOTTONE DI TIPO UIROUNDEDBUTTON 102
FIGURA 53 ANIMAZIONE DEL BOTTONE MENU 102
FIGURA 54 BOTTONI NAVIGAZIONE DEL MENU 103
FIGURA 55 BOTTONI UIARROWBUTTON PER SELEZIONATA LA DATA 103
FIGURA 56 INDICATORE REALIZZATO DA UIINDICATOR 103
FIGURA 57 GRAFICO A TORTA REALIZZATO DA UIPIECHART 104
FIGURA 58 ISTOGRAMMA REALIZZATO DA UIHISTOGRAM 104
FIGURA 59 RELAZIONE TRA UIABSTRACTCOMPONENT, UIDATA E UIPERIPHERAL 105
FIGURA 60 PROGETTAZIONE ICONA UIMONITORINGICON 106
FIGURA 61 PROGETTAZIONE ICONA UISHAREICON 106
FIGURA 62 PROGETTAZIONE ICONE UIHEARTHICON E UIPERIPHERALICON 106
FIGURA 63 PROGETTAZIONE ICONE UISETTINGICON ED UIERRORICON. 107
FIGURA 64 INTESTAZIONE DELL'APPLICAZIONE 107
FIGURA 65 MENU DELL'APPLICAZIONE 108
FIGURA 66 VISTA PRINCIPALE DELL'APPLICAZIONE 110
FIGURA 67 VISTA PER ESPORTARE I DATI NEL FORMATO JSON 111

FIGURA 68 VISTA IMPOSTAZIONI DI CONNESSIONE 112
FIGURA 69 VISTA VISUALIZZAZIONE DATI SULLA FREQUENZA CARDIACA DEL 13 LUGLIO 114
FIGURA 70 VISTA VISUALIZZAZIONE DATI SONNO DEL 11 LUGLIO 114
FIGURA 71 VISTA VISUALIZZAZIONE NUMERO PASSI DEL 12 LUGLIO 115
FIGURA 72 VISTA VISUALIZZAZIONE DISTANZA PERCORSA DEL 12 LUGLIO 115
FIGURA 73 VISTA PER MOSTRARE ALL'UTENTE GLI ERRORI 116
FIGURA 74 DIAGRAMMA DI FLUSSO PER RECUPERARE LA RAPPRESENTAZIONE DI UNA PERIFERICA 120
FIGURA 75 DIAGRAMMA DI FLUSSO PER LE FUNZIONI DI AGGREGAZIONE PER I DATI HR 122
FIGURA 76 DIAGRAMMA DI FLUSSO DEL METODO PER GENERARE IL PREDICATO SUL PERIODO 124
FIGURA 77 GRAFICO A LINEE RELATIVO AL NUMERO DI PASSI NEI MESI DI LUGLIO ED AGOSTO 126
FIGURA 78 GRAFICO A LINEE RELATIVO ALLE DISTANZE PERCORSE NEI MESI DI LUGLIO ED AGOSTO 126
FIGURA 79 GRAFICO A LINEE RELATIVO ALLA FREQUENZA CARDIACA NEL MESE DI LUGLIO 127
FIGURA 80 GRAFICO A LINEE RELATIVO ALLA FREQUENZA CARDIACA NEL MESE DI AGOSTO 128
FIGURA 81 ISTOGRAMMA ED AREOGRAMMA RELATIVI AL SONNO NEL MESE DI LUGLIO 128
FIGURA 82 ISTOGRAMMA ED AREOGRAMMA RELATIVI AL SONNO NEL MESE DI AGOSTO 129

Indice delle tabelle

Tabella 1 Glossario dei termini 15
Tabella 2 Requisiti funzionali 16
Tabella 3 Requisiti non funzionali 17
Tabella 4 Narrativa caso d'uso: Estrai dati 19
Tabella 5 Narrativa caso d'uso: Visualizza dati 19
Tabella 6 Narrativa caso d'uso: Acquisisci dati 20
Tabella 7 Narrativa caso d'uso: Interrompi acquisizione dati 21
Tabella 8 Narrativa caso d'uso: Connetti periferica 22
Tabella 9 Narrativa caso d'uso: Disconnetti periferica 22
Tabella 10 Narrativa caso d'uso: Ricerca periferiche 23
Tabella 11 Narrativa caso d'uso: Interrompi ricerca periferiche 23
Tabella 12 Descrizione chiavi contenute nel file JSON 57
Tabella 13 Attributi delle periferiche derivati dall'analisi 65
Tabella 14 Attributi delle sessioni derivati dall'analisi 66
Tabella 15 Attributi del sonno derivati dall'analisi 66
Tabella 16 Attributi dei passi derivati dall'analisi 67
Tabella 17 Attributi delle frequenze cardiache derivati dall'analisi 67
Tabella 18 Attributi della variabilità RR derivati dall'analisi 67
Tabella 19 Operazioni sulla collezione periferiche 68
Tabella 20 Operazioni sulla collezione sessioni 69
Tabella 21 Operazioni sulla collezione dati 69
Tabella 22 Operazioni sui dati relativi alla frequenza cardiaca 69
Tabella 23 Operazioni sui dati rilevati dal pedometro 70
Tabella 24 Vincoli non esprimibili con lo schema logico 72
Tabella 25 Elementi UIKit utilizzati nella storyboard 100
Tabella 26 Mostra l'animazione dalla scritta Monitoring a Settings. 109
Tabella 27 Batteria di test per il metodo getUIPeripheral 120
Tabella 28 Batteria di test per il metodo notify per l'evento GET_FUN_HR 123
Tabella 29 Batteria di test per il metodo datePredicate 124

Indice del codice

Codice 1 Protocollo del dispatcher 41
Codice 2 Protocolli per i publisher e subscriber 42
Codice 3 Implementazione del metodo publish 43
Codice 4 Gestione tentativo di connessione fallita 50
Codice 5 Metodo del driver per sospendere la ricezione dei pacchetti 51
Codice 6 Metodo adibito ad aggiungere gli intervalli di riposo all'interfaccia utente. 53
Codice 7 Metodo per generare la stringa JSON da un dizionario 55
Codice 8 Generazione del file JSON e condivisione 55
Codice 9 Aggiornamento variabile esecuzione in background 61
Codice 10 Recuperare una periferica da CoreData 79
Codice 11 Query generata da CoreData con entità DatoFisiologico 84
Codice 12 CoreDataController è conforme al pattern singleton 86
Codice 13 Implementazione query recupero ultima periferica connessa 90
Codice 14 Query generata da CoreData per recuperare l'ultima periferica connessa 90
Codice 15 Implementazione inserimento dati di tipo passo 91
Codice 16 Query generata da CoreData per inserire un nuovo passo 91
Codice 17 Implementazione numero massimo di passi 92
Codice 18 Query generata da CoreData per recuperare il numero massimo di passi 92
Codice 19 Implementazione statistiche HR 93
Codice 20 Query generata da CoreData per le statistiche HR 93
Codice 21 Utilizzo di CoreGraphics per disegnare una linea 95
Codice 22 Utilizzo di CoreGraphics per realizzare il clipping 96
Codice 23 Frammento di codice che realizza UITabMenu 102
Codice 24 Frammento di codice che realizza UIIndicator 103
Codice 25 Frammento di UIPieChart che realizza il grafico a torta 104
Codice 26 Realizzazione icona UIMonitoringIcon 106
Codice 27 Realizzazione icona UIShareIcon 106
Codice 28 Metodo per recuperare la rappresentazione di una periferica 119
Codice 29 Gestore richiesta funzioni di aggregazione per la frequenza cardiaca 121
Codice 30 Metodo per generare il predicato sul periodo 123

CAPITOLO 1
INTRODUZIONE

BioBeats è una startup internazionale che ha creato un'applicazione per monitorare lo stato di salute delle persone ed abbassare il loro livello di stress. Attualmente il prodotto è disponibile solo per le aziende, ma il team sta lavorando per portare il software anche ai singoli utenti. Il tirocinio si è svolto presso la sede aziendale in via San Lorenzo 6, Pisa ed è durato circa 300 ore totali.

L'obiettivo è la realizzazione di un'applicazione, per sistemi iOS, che si connetta via Bluetooth ad un dispositivo wearable, in particolare Lifesense Band 2, per l'acquisizione di dati al fine di condurre esperimenti scientifici con la collaborazione dell'ospedale Santa Chiara. Tale software dovrà attivare un processo in background attivo H24 in grado di recuperare e memorizzare i dati fisiologici.
In particolare si richiede di acquisire i battiti cardiaci, il numero di passi, la variabilità della frequenza cardiaca e dati relativi alla qualità del sonno. Inoltre sarà necessario fornire funzionalità per esportare le informazioni in un formato testuale per la condivisione. Specificatamente si è scelto di utilizzare lo standard JSON[1] in quanto risulta facile da leggere per le persone e per le macchine semplice da analizzare e generare. Tra le altre cose l'applicazione dovrà permettere all'utente di visualizzare i dati memorizzati localmente nel dispositivo.

[1] JSON è un semplice formato per lo scambio dei dati.

Prima della realizzazione dell'applicazione iOS il team di BioBeats aveva già sviluppato un'applicazione simile a quella richiesta, ma per la piattaforma Android. In questo modo gli sviluppatori hanno evitato di scontrarsi con dinamiche, generalmente più ostili, dei sistemi operativi Apple che abitualmente si incontrano quando si cerca di mantenere un processo sempre attivo. In particolare l'applicazione Android, da loro attualmente utilizzata, si compone di un unico bottone per abilitare l'acquisizione dei dati, senza dare la possibilità di estrarre e visualizzare le informazioni.

Il progetto è stato svolto in totale autonomia con la supervisione dei dipendenti dell'azienda, utilizzando l'ambiente di sviluppo integrato Xcode. Tale software, completamente sviluppato e mantenuto da Apple, contiene una suite di strumenti utili alla produzione di applicazioni. Inoltre BioBeats ha lasciato piena libertà nel scegliere quale linguaggio di programmazione utilizzare: Objective-C oppure Swift.
L'attuale applicazione dell'azienda è completamente sviluppata in Objective-C, ma dopo un'attenta riflessione si è scelto di utilizzare il linguaggio Swift. Questo perché non solo è stato concepito per coesistere con il linguaggio Objective-C, ma anche progettato per essere più resiliente dagli errori nel codice. Inoltre l'azienda stessa sta cercando di avviare un programma che avvicini i suoi programmatori a questo nuovo linguaggio. Oltre a ciò, poiché tra le varie attività della società vi è quella di trasportare l'intera logica di business dell'applicazione in TypeScript[2], nelle prime settimane del tirocinio mi è stata mostrata una panoramica del linguaggio e dei problemi che si incontrano nell'interoperabilità con Swift. In particolare si è partecipato ad alcune delle discussioni aziendali sulle questioni riguardanti il garbage collector e memory leak. Inoltre, per comprendere maggiormente alcuni aspetti di TypeScript, sono state visionate nozioni del framework JavaScriptCore che offre la possibilità di valutare codice JavaScript da applicazioni Swift. In seguito questi aspetti non sono stati approfonditi poiché non rilevanti per lo sviluppo dell'applicazione. Di

[2] TypeScript è un linguaggio che estende le potenzialità di JavaScript.

conseguenza non verranno trattati nella relazione. La scelta di non sviluppare logica di business dell'applicazione in TypeScript deriva dalla mancanza di tempo.

Per quanto riguarda la fase implementativa, l'applicazione è stata sviluppata sul Macbook Pro personale dotato di macOS Mojave e testata su iPhone X, iPhone 7 e iPhone SE. Invece il dispositivo wearable Lifesense Band 2 è stato fornito dall'azienda presso la quale si è svolto il tirocinio.

Per quanto riguarda i motivi che mi hanno spinto a scegliere questa esperienza rispetto ad altre, offerte sempre dall'Università di Pisa, è stata la possibilità di incrementare le conoscenze nel campo dello sviluppo di applicazioni mobile, in particolare per i sistemi Apple. Oltre a quanto è stato detto, prima dell'inizio del tirocinio, avevo una conoscenza base del linguaggio Swift, degli strumenti di Xcode e del framework CoreGraphics che sfrutta la potenza della tecnologia Quartz per eseguire rendering 2D leggeri. Questa esperienza mi ha consentito di incrementare le conoscenze precedentemente menzionate ed imparare altri framework, come ad esempio CoreBluetooth, CoreData e CoreAnimation.
In particolare CoreBluetooth fornisce le classi necessarie per comunicare con dispositivi dotati di tecnologia wireless Bluetooth Low Energy[3] (BLE). Invece CoreData consente di memorizzare in modo permanente i dati nel dispositivo. Per quanto concerne CoreAnimation fornisce le classi per creare animazioni fluide senza appesantire la CPU e rallentare l'applicazione. Oltre a ciò è stato fondamentale analizzare e risolvere le problematiche che si verificano nel mantenere un processo in background attivo H24.

Questo documento intende riportare i concetti affrontati al fine di realizzare l'applicazione. In primo luogo verrà effettuata un'analisi della realtà d'interesse con lo scopo di individuare gli elementi del dominio e determinare i principali casi d'uso.

[3] BLE è una tecnologia wireless commercializzata con lo scopo di fornire un consumo energetico ridotto, mantenendo un intervallo di comunicazione simile al classico Bluetooth.

In secondo luogo si mostrerà alcuni schemi derivati in fase di progettazione come il diagramma delle classi e di sequenza. Essi risultano fondamentali per iniziare a comprendere quali sono le componenti e quali messaggi utilizzano per la comunicazione. Dopodiché mostreremo l'architettura del sistema definendo anche come avviene la comunicazione tra i vari moduli. In particolare sarà argomentato il driver che permette di cooperare con la periferica remota. Oltre a ciò descriveremo il ciclo vita di un'applicazione iOS, focalizzando l'attenzione sulle nozioni fondamentali a mantenere un'attività a lungo termine. In parallelo verranno determinate le informazioni che devono essere recuperate e memorizzate localmente. Specificatamente, mediante un'analisi, saranno definiti i dati e le operazioni su di essi. Dopodiché sarà progettata la base di dati giustificando le varie scelte e riportando le fasi relative all'implementazione di quest'ultima. In seguito daremo informazioni per quanto riguarda la progettazione dell'interfaccia utente, offrendo anche una panoramica delle viste realizzate. Successivamente andremo a definire le tecniche utilizzate per testare l'applicazione, riportando anche alcuni esempi. Infine verranno mostrati e descritti dei grafici generati dai dati acquisiti dall'applicazione nei mesi di luglio ed agosto.
Oltre a ciò verranno analizzate le problematiche incontrate e le relative soluzioni, giustificando ed argomentando le scelte che ne hanno determinato la risoluzione. Inoltre il documento cerca di introdurre tutti i concetti necessari alla comprensione degli aspetti trattati. In questo modo anche un lettore poco informato sulle tecnologie utilizzate, ma con conoscenze informatiche, potrà comprendere quanto scritto. Invece per gli utenti estranei al settore potrebbe essere necessario integrare il contenuto con i riferimenti inseriti nella bibliografia.

Per quanto riguarda gli obiettivi raggiunti, al termine del tirocinio, sono riuscito a sviluppare l'intera applicazione rispettando le specifiche richieste. Oltre a ciò, in accordo con il tutor aziendale, sono state realizzate delle viste personalizzate invece di utilizzare i componenti grafici predefiniti. Ciò ha reso l'applicazione visivamente più curata ed elaborata.

Infine, per concludere questa introduzione, si elencano tutti i capitoli con una descrizione sui contenuti trattati. In questo modo si facilita il lettore nel reperire gli argomenti che ritiene più interessanti.

Capitolo 1. Introduzione:

Viene delineata una panoramica del progetto e dell'ambiente in cui è stato sviluppato. Inoltre si riportano le principali giustificazioni che hanno portato ad effettuare determinate scelte.

Capitolo 2. Basi di partenza:

Vengono definite le conoscenze di partenza e cosa esisteva prima del tirocinio.

Capitolo 3. Tecnologie utilizzate:

Vengono descritte tutte le tecnologie utilizzate chiarendo a cosa servono, dove vengono utilizzate e perché sono state preferite ad altre. In particolare si parla dell'ambiente di sviluppo, del linguaggio di programmazione ed eventuali framework utilizzati. Inoltre vengono descritti i principali software utilizzati per la condivisione, il controllo delle versioni e per la comunicazione con i dipendenti aziendali.

Capitolo 4. Analisi dei requisiti:

Viene riportata una descrizione formale dell'applicazione da realizzare, mettendo in evidenza i requisiti funzionali, non funzionali ed i casi d'uso.

Capitolo 5. Diagramma delle classi e di sequenza:

Vengono individuate le classi di progettazione e si riportano i principali diagrammi di sequenza al fine di individuare i messaggi e le interazioni tra i moduli. Questo capitolo rappresenta la base per definire una buona architettura del software.

Capitolo 6. Architettura software ed Implementazione:

Si descrive l'architettura complessiva del software incentrata sullo stile Model-View-Controller[4] (MVC) e sul pattern publish-subscribe[5] per le comunicazioni tra i componenti. Inoltre si delinea l'interazione, basata sul famework CoreBluetooth, tra periferica ed applicazione. In particolare si riporta il comportamento del driver e le principali problematiche affrontate. Infine verrà data una visione generale degli aspetti relativi al ciclo di vita di un'applicazione iOS, con particolare attenzione all'esecuzione in background.

Capitolo 7. Progettazione e realizzazione del database:

Viene descritta la progettazione del database e la sua realizzazione. Comprende informazioni sul framework CoreData adibito per il salvataggio ed il recupero dei dati, oltre che alla gestione dello schema. Inoltre saranno riportate le problematiche, con le relative soluzioni, derivanti dalle scelte implementative del framework. Particolare attenzione sarà data all'utilizzo concorrente delle risorse di CoreData.

Capitolo 8. Progettazione e realizzazione interfaccia utente:

Viene mostrata l'interfaccia utente dell'applicazione. In primo luogo verrà introdotto il framework CoreGraphics, utilizzato per il disegno 2D basato su Quartz, al fine di produrre un elaborato comprensibile anche al lettore meno esperto. In secondo luogo sarà mostrata l'organizzazione dei file e la progettazione degli elementi grafici utilizzati nell'applicazione. In particolare, per i componenti più semplici, riporteremo l'implementazione in cui è possibile notare l'utilizzo di primitive grafiche. Infine daremo una panoramica delle viste, che realizzano l'interfaccia utente, descrivendo per ciascuna di esse quali informazioni mostrano all'utilizzatore.

[4] MVC è un pattern architetturale in grado di separare la logica di presentazione dei dati dalla logica di business.

[5] Publish-Subscribe è un design pattern utilizzato per la comunicazione asincrona tra oggetti.

Capitolo 9. Test dell'applicazione:

Vengono delineate le tecniche utilizzate per testare l'applicazione. Inoltre si riportano, a titolo esemplificativo, le batterie di test effettuate su alcuni metodi.

Capitolo 10. Analisi dei dati acquisiti:

Vengono mostrati e descritti grafici che riportano i dati acquisiti dall'applicazione nei mesi di luglio ed agosto.

Capitolo 11. Conclusioni:

Vengono descritti gli obiettivi raggiunti, definendo anche le capacità acquisite durante il corso di questa esperienza.

CAPITOLO 2
BASI DI PARTENZA

Questo capitolo intende definire in modo chiaro e conciso la situazione ambientale prima del tirocinio. In particolare BioBeats aveva già sviluppato un'applicazione Android in grado di acquisire dati dal dispositivo remoto. Quest'ultima si componeva di un unico bottone per avviare il monitoraggio. Lo scopo del tirocinio è creare un'applicazione analoga per i dispositivi iOS includendo, oltre alla funzionalità di acquisizione, la possibilità di estrarre e visualizzare i dati. Per fare ciò sono stati impiegati diversi framework tra i quali CoreBluetooth, CoreData e CoreGraphics. Per quanto riguarda i primi due non li avevo mai utilizzati, quindi è stato necessario uno studio preliminare per capirne i meccanismi di utilizzo e le funzionalità offerte. Invece CoreGraphics lo avevo già sperimentato in attività personali, di conseguenza questa esperienza è stata una buona occasione per acquisirne più consapevolezza e praticità. Per quanto concerne lo sviluppo è stata creata una nuova applicazione costruendo tutti i moduli necessari al corretto funzionamento. A causa della mancanza di tempo, su suggerimento del tutor aziendale, l'unico componente utilizzato e sviluppato dall'azienda ospitante è stato il driver della periferica. In particolare lo scopo di quest'ultimo è recuperare i frame dal dispositivo remoto ed interpretarli secondo il protocollo definito dal produttore dell'orologio. Nonostante ciò è stato necessario leggerne e comprenderne il codice per risolvere problemi relativi alla compatibilità. Inoltre sono stati studiati strumenti per la condivisione ed il controllo delle versioni del codice. Tali aspetti saranno dettagliati nel prossimo capitolo, specificando le funzionalità ed il livello di conoscenza prima del tirocinio.

CAPITOLO 3
TECNOLOGIE UTILIZZATE

In questo capitolo verranno elencate le tecnologie utilizzate. In particolare per ciascuna di esse sarà definito l'utilizzo che ne è stato fatto e le varie motivazioni che ne giustificano la scelta, discutendo eventualmente anche le principali alternative.

3.1 Ambiente di sviluppo Xcode

L'ambiente di sviluppo Xcode permette di creare applicazioni per i sistemi macOS, iOS, watchOS e tvOS, fornendo una suite di strumenti utili alla produzione di software. Dato che l'obiettivo è quello di creare un'applicazione iOS è stata immediata la scelta di questo IDE[6]. In particolare questa esperienza mi ha permesso di acquisire più consapevolezza ed ampliare la conoscenza sugli strumenti offerti da Xcode. Infatti, prima del tirocinio, si aveva solo una competenza superficiale delle principali funzionalità. Invece durante l'esperienza ho imparato ad utilizzare strumenti più professionali dei quali ignoravo l'esistenza. Tra i principali menzioniamo Playground e Source Control. Il primo è un foglio intelligente capace di analizzare ed eseguire immediatamente il codice scritto. Invece Source Control permette di archiviare i sorgenti in un repository così da poter gestire le vecchie versioni e tenere traccia delle modifiche nel corso del ciclo di sviluppo. Il lettore che

[6] È l'acronimo di ambiente di sviluppo integrato. Generalmente riferito con il termine in lingua inglese integrated development environment.

intende approfondire questo IDE può consultare il riferimento [1] inserito nella bibliografia.

3.2 Linguaggio di programmazione Swift

Per quanto riguarda il linguaggio di programmazione è stato possibile scegliere liberamente tra Objective-C e Swift. Dopo un'accurata riflessione è stato preferito l'utilizzo di quest'ultimo principalmente perché ha una sintassi simile ai linguaggi studiati nel corso di laurea in informatica, come ad esempio Java. Di conseguenza, in breve tempo, sono riuscito a prendere dimestichezza con questo linguaggio consultando l'e-book fornito dall'azienda Apple [2]. Inoltre, come specificato in [3], risulta più resiliente dagli errori e fino a 2.6 volte più veloce di Objective-C.

3.3 Framework: CoreGraphics, CoreBluetooth, CoreData

I principali framework utilizzati per sviluppare l'applicazione sono CoreGraphics, CoreBluetooth e CoreData. Tale scelta non è stata imposta dal tutor aziendale, ma è ragionevole cimentarsi in tali librerie quando si vuole realizzare un'applicazione per dispositivi iOS. In particolare il framework CoreGraphics, del quale avevo una conoscenza base, come specificato in [4] offre un rendering 2D a basso livello e leggero per gestire il disegno e le trasformazioni. Nell'applicazione è stato utilizzato per l'interfaccia utente al fine di creare le icone, determinati widget ed alcune animazioni gestite manualmente con i timer. Inoltre, a causa della mancanza di tempo, altre animazioni sono state realizzate mediante CoreAnimation [5], con il quale non avevo familiarità.

CoreBluetooth e CoreData sono gli altri due framework che non conoscevo. Il primo, come specificato in [6], viene utilizzato nelle comunicazioni Bluetooth Low Energy in cui l'applicazione svolge il ruolo di gestore centrale alla quale la periferica si connette per inviare i dati. Mentre quest'ultimo viene impiegato per la connettività ed il recupero delle informazioni dall'orologio, il framework CoreData [7] viene utilizzato per memorizzare permanentemente i dati nel dispositivo, offrendo la possibilità di recuperarli filtrati secondo particolari criteri.

Il lettore troverà maggiori dettagli nei capitoli opportuni, in cui ciascun framework verrà descritto in relazione all'utilizzo che ne è stato fatto.

3.4 Slack

Slack è un software che fornisce strumenti di collaborazione aziendale permettendo una messaggistica istantanea con tutti i membri del team. Come riportato in [8], tra le funzionalità offerte vi è la possibilità di creare dei canali per condividere idee, file ed effettuare videochiamate. Tale applicazione è stata usata per comunicare con il tutor aziendale ed alcuni suoi colleghi permettendo una collaborazione reciproca in qualsiasi momento della giornata.

3.5 Source Control e Github

Lo strumento Source Control [9], presente in Xcode, permette di salvare il progetto in un repository di Github [10] dando la possibilità di condividere il codice con altre persone. In particolare, questo software, è stato utile sia per tenere traccia delle varie versioni dell'applicazione, ma principalmente per permettere al tutor aziendale di supervisionare la stesura del codice. Il funzionamento di Source Control è equivalente a quello di Github. Ovvero per ciascuna funzionalità viene creato un ramo, comunemente riferito con il termine inglese branch. Di default il repository[7] ha un unico branch chiamato master. Quando un nuovo ramo viene creato, esso risulta una copia di quest'ultimo. A questo punto è possibile modificare il contenuto del nuovo ramo senza alterare il master. Ogni qualvolta che le modifiche devono essere riportate è necessario effettuare un commit[8]. È possibile fare più commit e ripristinare vecchie versioni nel caso in cui siano stati introdotti dei bug nel codice. Inoltre Xcode, come Github, permette di comparare versioni diverse facendo vedere quali istruzioni sono state aggiunte, rimosse o modificate. Infine, quando si decide di

[7] Struttura dati utilizzata per tracciare la cronologia di un set di file.
[8] Operazione mediante la quale tutti i cambiamenti vengono salvati, diventando parte della cronologia di un repository.

far vedere le modifiche agli altri sviluppatori, è possibile effettuare una pull request[9]. Le pull request sono il cuore delle collaborazioni in quanto significano che si stanno proponendo delle modifiche. Di conseguenza i collaboratori possono revisionare il codice proposto e suggerire eventuali correzioni. Quando siamo convinti dei cambiamenti che stiamo apportando è possibile effettuare una merge pull request, la quale riporta le modifiche sul ramo master. A questo punto, nel caso in cui non sia più necessario, è ragionevole eliminare il branch creato allo scopo di tenere il repository il più ordinato possibile. Come riportato in [11] , uno tra i modelli di ramificazione maggiormente utilizzati per Git è GitFlow creato da Vincent Driessen.

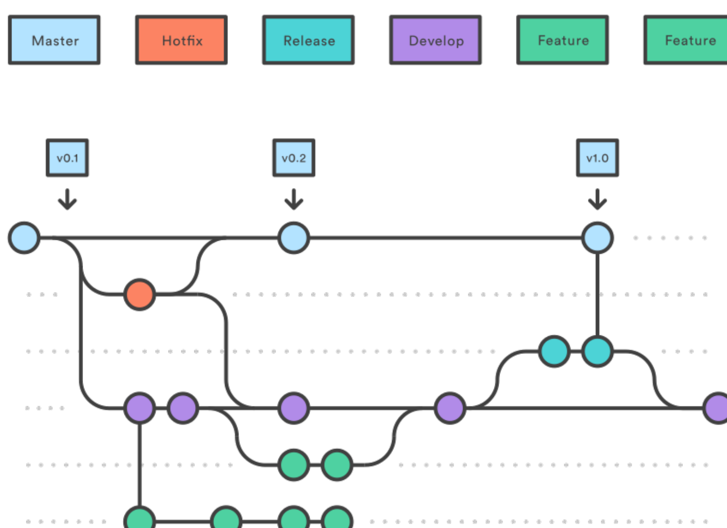


Figura 1 Gitflow Workflow di Vincent Driessen [11]

Il modello prevede la creazione di un branch dal ramo master, che prende il nome di develop, utilizzato per l'integrazione di nuove funzionalità. Invece il branch master memorizza la cronologia dei rilasci ufficiali. Quando una nuova funzionalità deve essere sviluppata viene creato un feature branch che usa il ramo develop come genitore. Quando quest'ultima è completata, tutte le modifiche vengono riportate sullo sviluppo. Una volta che il ramo develop ha raggiunto sufficienti funzioni per un rilascio viene creato un nuovo release branch. La creazione di questo ramo avvia il prossimo ciclo di rilascio. Di conseguenza non sarà più possibile aggiungere funzionalità, ma gli unici commit riguarderanno la correzione di errori e la

[9] Una pull request è una notifica ad altri che sono state apportate alcune modifiche ad un repository.

generazione della documentazione. Quando il software è stato sufficientemente testato il ramo release viene unito al ramo master e develop, rilasciando una nuova versione del prodotto. Se dopo il rilascio si manifesta un bug nel codice è possibile creare un ramo hotfix, da quello master, per risolvere l'errore. Al termine della manutenzione tale ramo viene unito a quello di develop e master.

3.6 Microsoft Excel

Il programma Microsoft Excel è stato impiegato per realizzare grafici su larga scala partendo da un insieme di dati estratti dall'applicazione nel formato JSON. Tale attività non è stata richiesta dal tutor aziendale, ma è stata ritenuta necessaria al fine di inserire in questa relazione il modo in cui potrebbero essere utilizzate le informazioni rilevate dall'applicazione. In particolare è stato scelto Excel, rispetto a molte altre valide alternative, sia per la personale esperienza relativa al programma, ma anche per la varietà di grafici che permette di realizzare. Inoltre, tale software, offre strumenti sofisticati per importare dati da sorgenti esterne in modo semplice e veloce. Una volta elaborate le informazioni cercheremo di ricavare informazioni sullo stato e sulle abitudini della persona.

CAPITOLO 4
ANALISI DEI REQUISITI

Lo scopo di questo capitolo è riportare tutte le specifiche richieste dall'azienda e che il software deve soddisfare. In particolare è stato commissionato lo sviluppo di un'applicazione, per i dispositivi iOS, in grado di connettersi allo smartband Lifesense Band 2 per ottenere dati fisiologici della persona che lo indossa. Ciascun dato sarà inviato al gestore dati che provvederà a memorizzarlo permanentemente nel dispositivo locale. Nel dettaglio siamo interessati alle informazioni relative alla frequenza cardiaca, numero di passi, variabilità RR[10] e qualità del sonno. Il lettore interessato potrà trovare una descrizione più dettagliata dei dati nel CAPITOLO 7. Inoltre l'applicazione dovrà fornire un'interfaccia molto semplice che consentirà all'utente di effettuare la connessione all'orologio, con un'attesa massima di 60 secondi. Oltre a ciò fornirà gli elementi necessari a visualizzare i dati ed esportarli in un formato testuale. In particolare la pagina iniziale sarà dotata di un bottone che permetterà di avviare ed interrompere l'acquisizione dei dati dalla periferica connessa. Invece la scheda di impostazioni dovrà permettere di mostrare i dispositivi remoti precedentemente connessi, identificare chiaramente quello attualmente connesso e fornire la possibilità di ricercare nuove periferiche. Per quanto riguarda la pagina di visualizzazione dei dati si vuole mostrare le ultime rilevazioni, includendo opzionalmente eventuali widget[11] per visualizzare informazioni di carattere generale. In riferimento alla scheda di estrazione dei dati è richiesta la possibilità di

[10] Misura l'intervallo di tempo tra due battiti successivi.
[11] Con il termine widget ci si riferisce ad un componente grafico.

poterli filtrare per periodo e per tipo di dato. Ad esempio si deve poter estrarre solo i dati relativi al battito cardiaco rilevati nell'ultimo mese.
Infine è richiesto che l'applicazione resti attiva in background H24, gestendo le varie problematiche che possono sorgere quando il sistema operativo decide di terminare il processo. Questa applicazione verrà utilizzata per condurre esperimenti scientifici con la collaborazione dell'ospedale Santa Chiara.

4.1 Glossario dei termini

Di seguito viene riportato il glossario dei termini per disambiguare il significato di alcune parole e definire, in modo chiaro e conciso, a cosa fanno riferimento.

Termine	Definizione
1. Dati fisiologici:	si riferisce ad un dato generico al quale l'applicazione è interessata. Ad esempio possono essere i battiti cardiaci, il numero di passi, la variabilità RR e la qualità del sonno.
2. Dispositivo locale:	si riferisce allo smartphone sul quale è stata installata l'applicazione.
3. Dispositivo remoto, periferica:	si riferisce allo smartband che invia i dati della persona che lo indossa all'applicazione.
4. Interfaccia:	si riferisce all'interfaccia utente che consente l'interazione reciproca tra l'uomo e la macchina.
5. Widget:	si riferisce ad eventuali grafici o indicatori che mostrato informazioni di carattere generale e che potrebbero essere inseriti nell'applicazione. Ad esempio con il termine widget possiamo fare riferimento ad un componente che mostra un areogramma relativo ai battiti cardiaci suddivisi in intervalli di valori prestabiliti.
6. Background:	si riferisce alla modalità di esecuzione dell'applicazione che non richiede l'intervento dell'utente.

Tabella 1 Glossario dei termini

4.2 Requisiti funzionali

Questo paragrafo intende raggruppare l'insieme delle funzionalità che il sistema deve fornire. Inoltre verrà utilizzata un'analisi MoSCoW[12] al fine di attribuire la giusta importanza al raggiungimento di ciascun requisito.

1.	Se l'utente clicca il bottone per ottenere i dati e la periferica è correttamente connessa allora il sistema deve acquisire i valori dallo smartband. [Must have]
2.	Se l'utente clicca il bottone per interrompere l'acquisizione dei dati allora il sistema deve sospendere la ricezione dei dati. [Must have]
3.	Se l'utente clicca il bottone per ricercare dispositivi allora il sistema deve individuare le periferiche disponibili. [Must have]
4.	Se l'utente clicca il bottone per interrompere la ricerca dei dispositivi allora il sistema deve sospendere l'attività per individuare le periferiche. [Must have]
5.	Se l'utente clicca su una periferica disponibile il sistema deve tentare la connessione alla periferica. [Must have]
6.	Se l'utente clicca sul bottone disconnetti il sistema deve disconnettersi dalla periferica connessa. [Must have]
7.	Il sistema deve memorizzare permanentemente il dato fisiologico includendo la data di rilevazione. [Must have]
8.	Il sistema deve ottenere informazioni riguardo la frequenza cardiaca, il numero di passi, la variabilità RR e dati relativi al sonno. [Must have]
9.	Il sistema deve mostrare le ultime rilevazioni effettuate. [Could have]
10.	Il sistema deve recuperare dal database i dati memorizzati filtrandoli per periodo e/o tipo di dato. [Should have]
11.	Se l'utente clicca sul bottone per estrarre i dati il sistema deve generare un file, in formato testuale, che contiene i dati filtrati secondo i criteri stabiliti dall'utente. [Must have]

Tabella 2 Requisiti funzionali

[12] Acronimo di Must - Should - Could - Won't in cui le "o" sono aggiunte per rendere pronunciabile la parola.

4.3 Requisiti non funzionali

Il seguente paragrafo raggruppa i requisiti non funzionali, ovvero quei requisiti che rappresentano i vincoli e le caratteristiche relative del sistema, come vincoli di natura temporale, vincoli sul processo di sviluppo e sugli standard da adottare.

1.	Il file testuale per esportare i dati deve rispettare lo standard JSON.
2.	L'applicazione deve restare attiva H24 in background.
3.	La connessione con la periferica deve avvenire entro 60 secondi.
4.	Il sistema può mostrare dei widget con informazioni di carattere generali come numero di dati e grafici.
5.	L'applicazione deve essere compatibile con almeno iOS 11 e versioni successive.
6.	L'interfaccia utente deve essere ben definita e molto semplice da utilizzare.

Tabella 3 Requisiti non funzionali

4.4 Casi d'uso

In questo paragrafo verranno mostrati i principali casi d'uso del sistema. I casi d'uso catturano le funzionalità di un sistema visto dall'esterno, ovvero dal punto di vista dell'utilizzatore. Sono espressi in termini di scenari, vale a dire come una sequenza di scambi di messaggi tra attori e sistema.

4.4.1 Casi d'uso con attore principale l'utente

L'utente dell'applicazione può avviare l'operazione di estrazione dei dati, indicando eventualmente criteri di filtraggio relativi al periodo ed ai tipi di informazioni.

Inoltre potrà connettersi e disconnettersi dalla periferica in cui l'operazione di connessione deve essere limitata ad un massimo di 60 secondi, oltre i quali deve essere considerata fallita. Qualora il dispositivo non fosse presente è possibile avviare un'operazione di ricerca ed interromperla quando la periferica viene trovata. Oltre a ciò, dopo che la connessione verso il dispositivo è stata stabilità, l'utente può avviare l'azione per acquisire i dati ed eventualmente sospenderla quando lo ritiene

opportuno. Infine l’utente deve poter visualizzare informazioni relative ai dati presenti nel dispositivo.

Di seguito viene mostrata una figura dei casi d’uso:

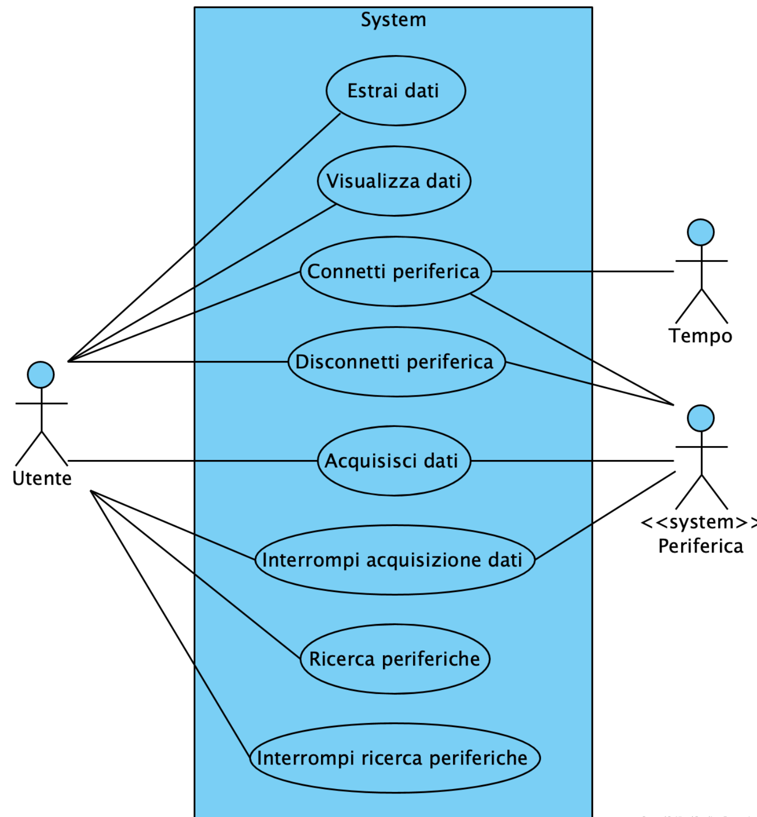


Figura 2 Casi d'uso avviati dall'utilizzatore del sistema

4.4.1.1 Narrativa caso d’uso: Estrai dati

Nome:	Estrai dati
Breve descrizione:	L’utente desidera estrarre i dati fisiologici memorizzati.
Attori primari:	Utente
Attori secondari:	Nessuno
Precondizione:	Nessuna
Sequenza degli eventi principali:	1. L’utente richiede al sistema di estrarre i dati fisiologici. 2. Il sistema verifica che i filtri inseriti siano consistenti. 3. Il sistema recupera i dati consistenti con i criteri dell’utente. 4. Per ciascun dato: 4.1. Crea la sua rappresentazione JSON.

	5. Inserisce la rappresentazione JSON nel file di output. 6. Il sistema mostra all'utente le opzioni per condividere il file generato.
Postcondizione:	File testuale con i dati fisiologici generato correttamente.
Sequenze alternative degli eventi:	• Filtri inseriti non sono consistenti, ad esempio nessun tipo di dato è stato selezionato oppure il periodo non è valido. • Nessun dato che rispetti i vincoli è disponibile. • Errore nella generazione del file JSON. In questi casi verrà mostrato un opportuno messaggio di errore.

Tabella 4 Narrativa caso d'uso: Estrai dati

4.4.1.2 Narrativa caso d'uso: Visualizza dati

Nome:	Visualizza dati
Breve descrizione:	L'utente desidera visualizzare le ultime rilevazioni.
Attori primari:	Utente
Attori secondari:	Nessuno
Precondizione:	Nessuna
Sequenza degli eventi principali:	1. L'utente richiede al sistema di ottenere le ultime rilevazioni. 2. Il sistema recupera gli ultimi dati inseriti. 3. Per ciascun dato 3.1. Il sistema mostra il dato all'utente.
Postcondizione:	I dati sono stati correttamente visualizzati.
Sequenze alternative degli eventi:	• Nessuna dato da visualizzare. In questi casi verrà mostrato un opportuno messaggio di errore.

Tabella 5 Narrativa caso d'uso: Visualizza dati

4.4.1.3 Narrativa caso d'uso: Acquisisci dati

Nome:	Acquisisci dati
Breve descrizione:	L'utente desidera acquisire i dati fisiologici dalla periferica.
Attori primari:	Utente
Attori secondari:	Periferica
Precondizione:	L'applicazione è connessa ad una periferica.
Sequenza degli eventi principali:	1. L'utente richiede al sistema di acquisire i dati fisiologici. 2. Il sistema si sottoscrive alla caratteristica della periferica.
Postcondizione:	La periferica invierà i cambiamenti dei valori delle caratteristiche.
Sequenze alternative degli eventi:	• La connessione verso la periferica viene persa. • L'utente disattiva il Bluetooth. In questi casi verrà mostrato un opportuno messaggio di errore.

Tabella 6 Narrativa caso d'uso: Acquisisci dati

4.4.1.4 Narrativa caso d'uso: Interrompi acquisizione dati

Nome:	Interrompi acquisizione dati
Breve descrizione:	L'utente desidera interrompere l'acquisizione dei dati fisiologici dalla periferica.
Attori primari:	Utente
Attori secondari:	Periferica
Precondizione:	L'applicazione è connessa ad una periferica.
Sequenza degli eventi principali:	1. L'utente richiede al sistema di interrompere l'acquisizione dei dati fisiologici. 2. Il sistema notifica alla periferica la volontà di non ricevere ulteriori valori.
Postcondizione:	La periferica non invia dati all'applicazione.

Sequenze alternative degli eventi:	• Impossibile comunicare con la periferica. • L'applicazione non si trova nello stato di acquisizione dati. In questi casi verrà mostrato un opportuno messaggio di errore.

Tabella 7 Narrativa caso d'uso: Interrompi acquisizione dati

4.4.1.5 Narrativa caso d'uso: Connetti periferica

Nome:	Connetti periferica
Breve descrizione:	L'utente desidera connettersi ad una periferica.
Attori primari:	Utente
Attori secondari:	Periferica, Tempo
Precondizione:	L'applicazione conosce la periferica alla quale connettersi e non è connessa ad un altro dispositivo.
Sequenza degli eventi principali:	1 L'utente richiede al sistema di connettersi ad una periferica. 2 Il sistema avvia un timer di 60 secondi. 3 Il sistema invia la richiesta per connettersi alla periferica. 4 La periferica si connette al sistema. 5 Se (il sistema non conosce la periferica) 5.1. Il sistema memorizza la periferica. 6 Include Acquisisci dati.
Postcondizione:	La periferica è stata correttamente connessa al dispositivo e sta inviando dati.
Sequenze alternative degli eventi:	• Impossibile stabilire la connessione con la periferica. • Il dispositivo ha il Bluetooth spento. • Il timer invia il segnale che il tempo è scaduto.

	In questi casi verrà mostrato un opportuno messaggio di errore.

Tabella 8 Narrativa caso d'uso: Connetti periferica

4.4.1.6 Narrativa caso d'uso: Disconnetti periferica

Nome:	Disconnetti periferica
Breve descrizione:	L'utente desidera disconnettersi dalla periferica.
Attori primari:	Utente
Attori secondari:	Periferica
Precondizione:	L'applicazione è connessa ad una periferica.
Sequenza degli eventi principali:	1 L'utente richiede al sistema di disconnettersi dalla periferica alla quale è connesso. 2 Include Interrompi acquisizione dati. 3 Il sistema informa la periferica del tentativo di disconnessione.
Postcondizione:	La periferica è stata disconnessa correttamente.
Sequenze alternative degli eventi:	• Impossibile disconnettersi dalla periferica. In questi casi verrà mostrato un opportuno messaggio di errore.

Tabella 9 Narrativa caso d'uso: Disconnetti periferica

4.4.1.7 Narrativa caso d'uso: Ricerca periferiche

Nome:	Ricerca periferiche
Breve descrizione:	L'utente desidera trovare le periferiche alle quali connettersi.
Attori primari:	Utente
Attori secondari:	Nessuno
Precondizione:	Il dispositivo ha il Bluetooth nello stato attivo e non è connesso a nessuna periferica.
Sequenza degli eventi principali:	1 L'utente richiede al sistema di trovare le periferiche alle quali connettersi.

	2 Per ciascuna periferica trovata a. Il sistema mostra all'utente la periferica.
Postcondizione:	L'utente può interagire con le periferiche mostrate dal sistema.
Sequenze alternative degli eventi:	• Nessuna periferica viene trovata. In questi casi verrà segnalata la situazione all'utente.

Tabella 10 Narrativa caso d'uso: Ricerca periferiche

4.4.1.8 Narrativa caso d'uso: Interrompi ricerca periferiche

Nome:	Interrompi ricerca periferiche
Breve descrizione:	L'utente desidera non visualizzare altre periferiche alle quali connettersi.
Attori primari:	Utente
Attori secondari:	Nessuno
Precondizione:	Il dispositivo sta ricercando periferiche.
Sequenza degli eventi principali:	1 L'utente richiede al sistema di interrompere la ricerca di nuove periferiche. 2 Il sistema sospende la ricerca.
Postcondizione:	Nessun'altra periferica viene mostrata all'utente.
Sequenze alternative degli eventi:	• Impossibile sospendere l'attività di ricerca. In questi casi verrà mostrato un opportuno messaggio di errore.

Tabella 11 Narrativa caso d'uso: Interrompi ricerca periferiche

4.4.2 Casi d’uso con attore principale la periferica

Quando l’utente si è connesso alla periferica ed ha espresso la volontà di ricevere periodicamente eventuali dati fisiologici, il dispositivo può avviare il caso d’uso per memorizzare il dato.

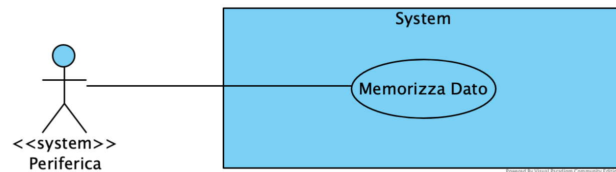


Figura 3 Casi d'uso avviati dalla periferica

4.4.2.1 Narrativa caso d’uso: Memorizza dato

Nome:	Memorizza dato
Breve descrizione:	La periferica invia al sistema una nuova rilevazione.
Attori primari:	Periferica
Attori secondari:	Nessuno
Precondizione:	Il dispositivo è connesso ad una periferica e si è sottoscritto ad alcune caratteristiche.
Sequenza degli eventi principali:	1 La periferica invia un nuovo dato al sistema. 2 Il sistema verifica l’integrità del pacchetto. 3 Il sistema costruisce il dato. 4 Il sistema memorizza permanentemente il dato nel dispositivo.
Postcondizione:	Il dato ricevuto è memorizzato correttamente.
Sequenze alternative degli eventi:	• Il dato non è valido. In questi casi è opportuno terminare il caso d’uso ignorando l’errore.

4.5 Diagramma delle classi di analisi

La seguente figura riporta il diagramma delle classi derivato dall'analisi dei requisiti. Al fine di non appesantire la lettura dello schema verranno omesse le proprietà e le operazioni che non sono riportate nella descrizione del dominio.

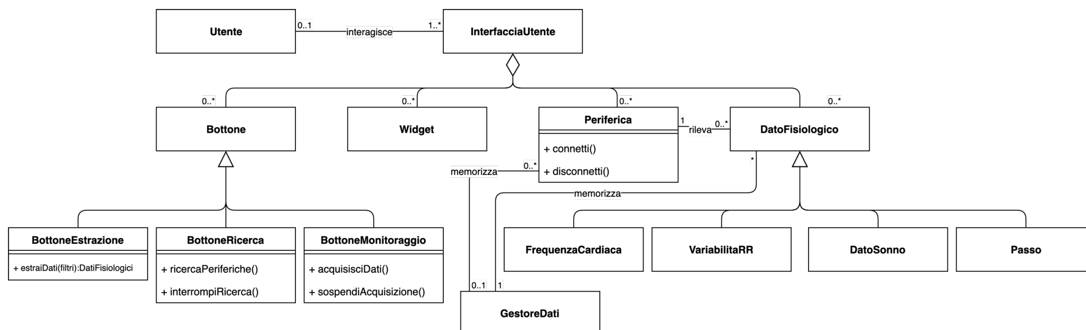


Figura 4 Diagramma delle classi (Analisi)

Come descritto nella realtà di interesse, l'utente interagisce con una serie di viste che sono un'aggregazione di bottoni, widget, periferiche e dati fisiologici da visualizzare. Nell'applicazione devono esistere diversi tipi di bottoni al fine di estrarre le informazioni, avviare ed interrompere la ricerca di periferiche, acquisire e sospendere la ricezione dei dati. Questi ultimi possono essere valori relativi alla frequenza cardiaca, alla variabilità RR, al sonno oppure al numero di passi. Inoltre un gestore, interno al sistema, si deve occupare di memorizzare tali informazioni permanentemente nel dispositivo locale. Equivalentemente, al fine di ricordarsi quali sono le periferiche utilizzate, queste ultime vengono archiviate in memoria. Infine, mediante opportune operazioni, deve essere possibile connettersi e disconnettersi da una periferica.

Il prossimo capitolo, basandosi sulle informazioni riportate nell'analisi, descriverà i passi principali della progettazione del software.

CAPITOLO 5
DIAGRAMMA DELLE CLASSI E DI SEQUENZA

In questo capitolo mostreremo come prima cosa il diagramma delle classi realizzato in fase di progettazione. Tale schema è direttamente derivato dalla Figura 4 aggiungendo elementi non menzionati nel dominio, ma necessari a rispettare i vincoli di progettazione. In fase di sviluppo ciascun elemento del diagramma sarà mappato in una o più classi d'implementazione. Dopodiché saranno riportati i diagrammi di sequenza dei casi d'uso, definiti nel capitolo precedente, al fine di determinare chiaramente quali sono i messaggi scambiati tra le varie istanze.

5.1 Diagramma delle classi di progettazione

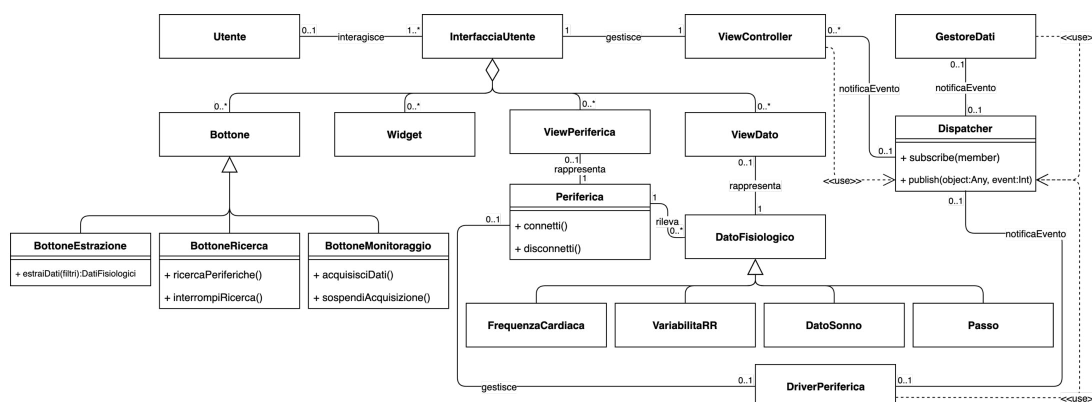


Figura 5 Diagramma delle classi (Progettazione)

La figura rappresenta il diagramma delle classi derivato dalla progettazione in cui si evidenziano moduli non menzionati nel dominio. In particolare, per quanto riguarda

le periferiche ed i dati, l'interfaccia mostra una loro rappresentazione che astrae dalla logica. In questo modo si ha una netta separazione tra l'oggetto della logica di business, utilizzato per il corretto funzionamento del programma, e l'elemento grafico con il quale si interfaccia l'utente. Oltre a ciò, poiché il software verrà sviluppato secondo il pattern Model-View-Controller, ogni vista è gestita da un controllore che dovrà essere specializzato in relazione agli elementi presenti. Tale modulo servirà a trasformare le richieste dell'utente in messaggi per il modello e viceversa. Per quanto riguarda le funzionalità è stato inserito un driver che gestisce le interazioni con il dispositivo connesso. In particolare dovrà fornire metodi per ricercare le periferiche, connettersi ad esse e garantire l'integrità dei dati rilevati dal dispositivo remoto. Inoltre questi ultimi devono essere memorizzati permanentemente nell'applicazione fornendo la possibilità di recuperarli in qualsiasi momento. Tali operazioni saranno garantite da un modulo adibito alla gestione dei dati. Infine la classe "Dispatcher" ha lo scopo di realizzare la comunicazione secondo il pattern publish-subscribe. Ovvero dovrà farsi carico degli eventi provenienti dai vari moduli e notificarli a tutti i componenti preventivamente abbonati. Il lettore può ottenere una specifica più dettagliata del pattern Model-View-Controller e publish-subscribe nel CAPITOLO 6.

5.2 Diagrammi di sequenza

Questo paragrafo intende riportare i diagrammi di sequenza relativi ai casi d'uso descritti nel paragrafo 4.4. Tali schemi hanno lo scopo di mostrare i messaggi scambiati tra i sottosistemi e rappresentano la base per implementare la comunicazione tra i moduli.

5.2.1 Diagramma di sequenza: Estrai dati

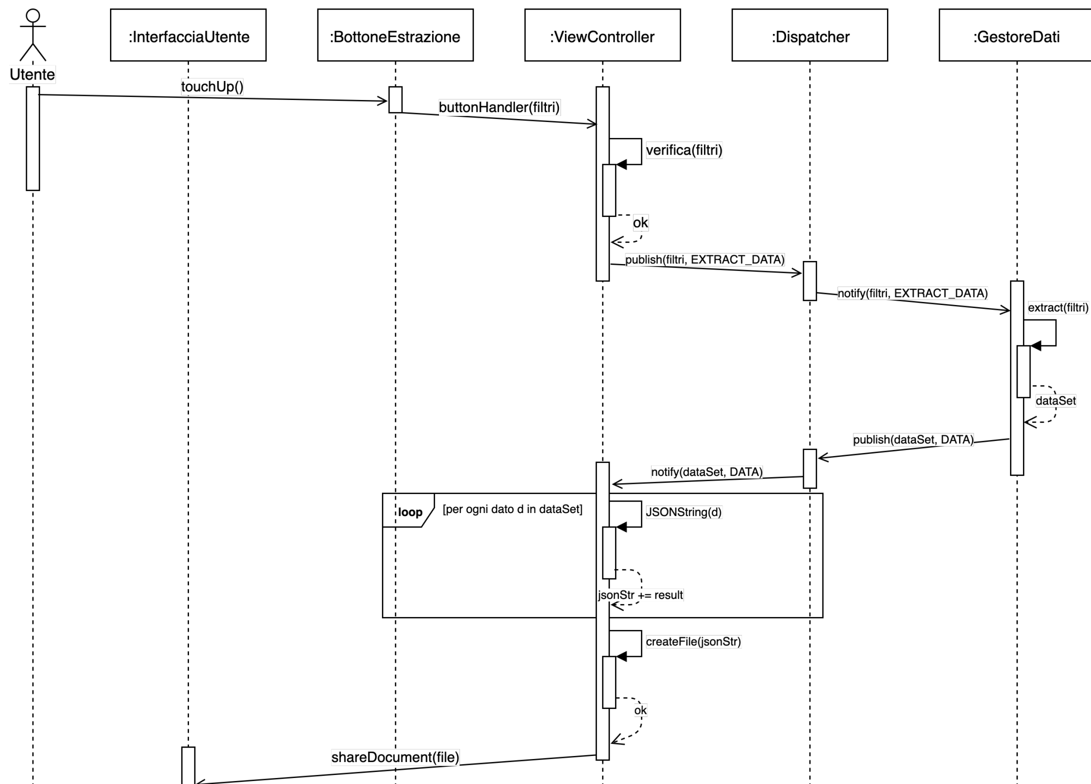


Figura 6 Diagramma di sequenza per estrarre i dati

La figura mostra il diagramma di sequenza relativo all'estrazione dei dati nell'ipotesi in cui non si verifichino errori. Come si può notare dal disegno la richiesta viene gestita in due fasi. La prima fase riguarda la trasformazione dell'input per comunicare l'operazione al modello. Infatti, quando l'utente preme il bottone per estrarre i dati, il view controller riceve la richiesta ed effettua la verifica dei filtri. Nel caso in cui i filtri non fossero consistenti il controllore mostra un messaggio sull'interfaccia utente terminando il caso d'uso. Contrariamente, se il controllo termina con successo, viene inviato un messaggio al dispatcher che determina l'inizio della seconda fase. Tale sequenza include il recupero e la trasformazione dei dati. Infatti il dispatcher notifica la richiesta al gestore dei dati, il quale provvederà a recuperare le informazioni. A termine dell'attività il gestore invia al dispatcher il relativo evento di risposta che viene notificato al controllore. Di conseguenza quest'ultimo provvede a creare il file con la rappresentazione JSON dei dati e termina il caso d'uso mostrando sull'interfaccia utente le opzioni di condivisione.

5.2.2 Diagramma di sequenza: Acquisisci dati ed interrompi acquisizione

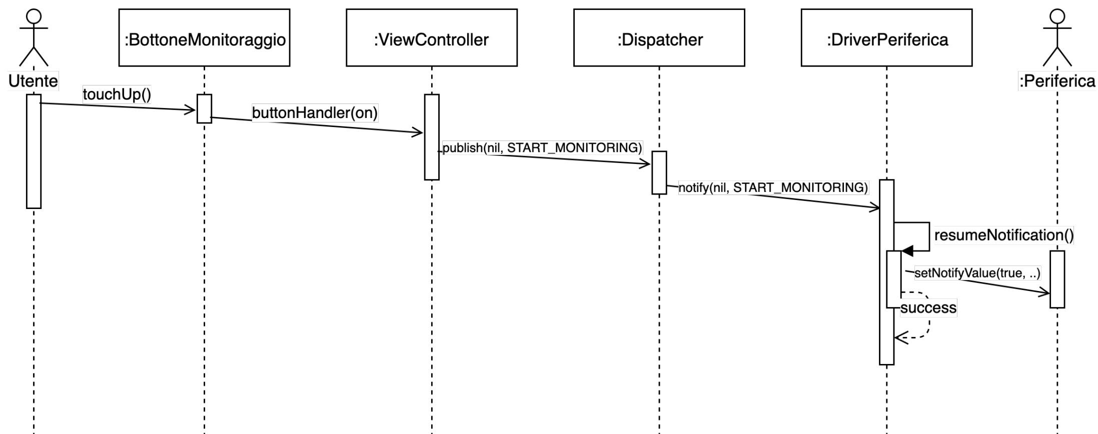


Figura 7 Diagramma di sequenza per acquisire i dati

Questo paragrafo descrive il diagramma di sequenza relativo all'acquisizione dei dati. Quando l'utente preme sul bottone per avviare il monitoraggio, l'input viene segnalato al controllore opportuno che provvede a generare una richiesta verso il modello. In particolare invia al dispatcher un evento di inizio monitoraggio, il quale viene percepito dal driver che gestisce le comunicazioni verso la periferica. Tale modulo provvederà, utilizzando segnali radio, ad informare il dispositivo remoto di notificare i cambiamenti di determinate caratteristiche.

In modo analogo viene trattato il caso d'uso per interrompere l'acquisizione dati, mostrato nella seguente figura.

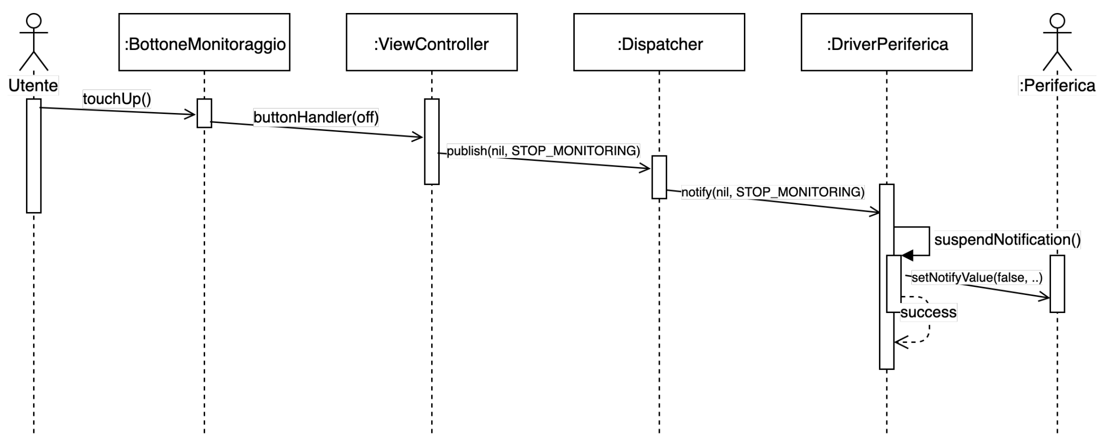


Figura 8 Diagramma di sequenza per interrompere l'acquisizione dei dati

Come si può notare il diagramma di sequenza è equivalente al precedente. L'unica differenza è che il driver comunica alla periferica di interrompere l'invio di pacchetti relativi ai cambiamenti dei valori delle caratteristiche.

Inoltre, poiché anche in questo caso si possono verificare errori, si riporta il diagramma di sequenza che mostra come segnalare la situazione all'utente.

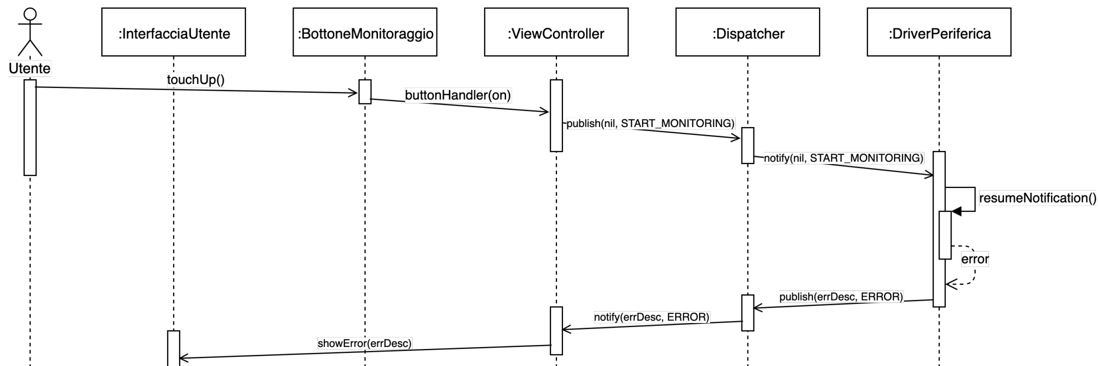


Figura 9 Diagramma di sequenza errore acquisizione dati

Una situazione di errore molto frequente potrebbe derivare dal fatto che l'utente richieda di acquisire i dati quando ancora la periferica non risulta connessa.

5.2.3 Diagramma di sequenza: Connetti periferica

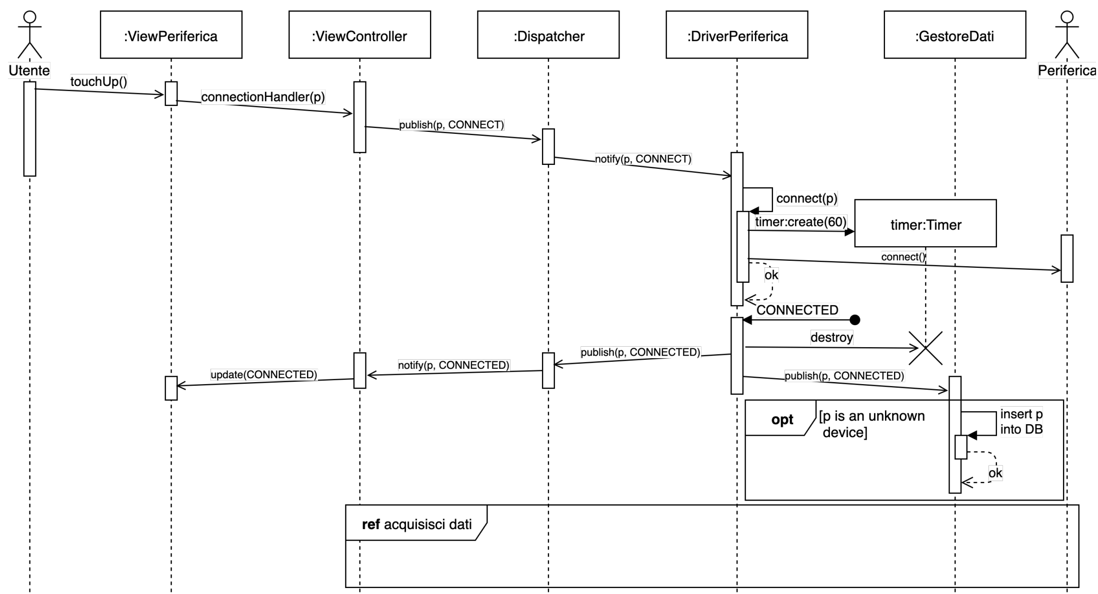


Figura 10 Diagramma di sequenza per connettersi ad una periferica

La figura mostra il diagramma di sequenza per connettersi ad una periferica. Analogamente ai casi precedenti l'utente, interagendo con l'interfaccia della periferica, genera l'evento che viene notificato al controllore, il quale lo propaga verso il modello inviando un messaggio. A questo punto il driver avvia un timer di 60 secondi e comunica alla periferica, mediante connessione Bluetooth, la richiesta di connessione. Quando a quest'ultimo viene notificato che il dispositivo si è connesso

correttamente, interrompe il timer e propaga l'evento al dispatcher così da aggiornare la vista e memorizzare la periferica nel database, qualora non fosse già presente. Altrimenti, se il timer scadesse prima di ricevere la notifica, verrebbe generato un messaggio di errore ed eventuali connessioni successive sarebbero rifiutate. Oltre a ciò il driver può generare errori anche prima di richiedere la connessione qualora un dispositivo risulti già correttamente connesso.

5.2.4 Diagramma di sequenza: Disconnetti periferica

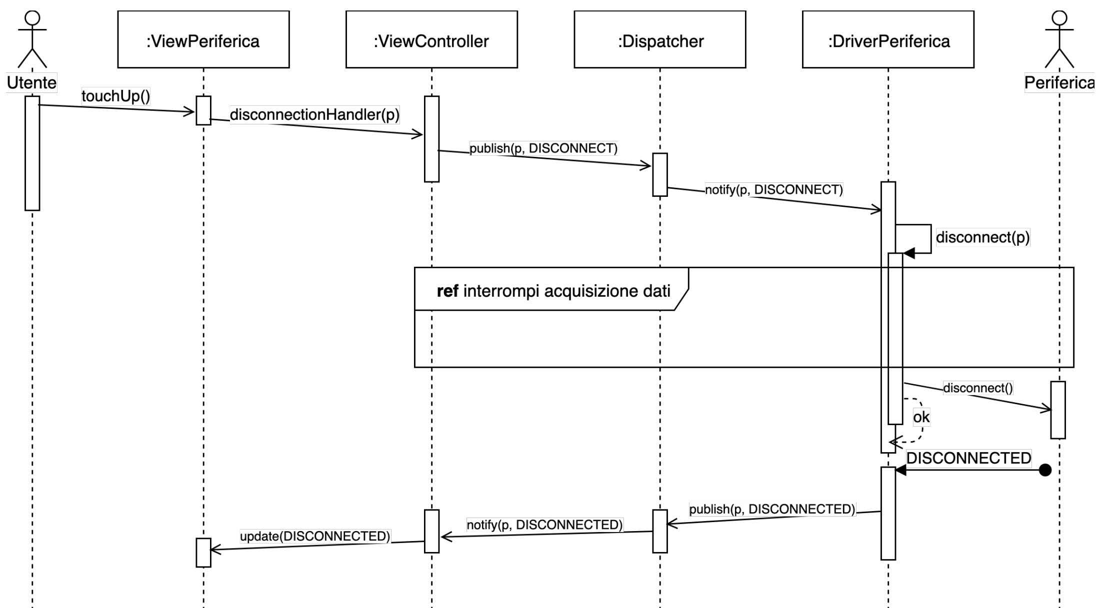


Figura 11 Diagramma di sequenza per disconnettersi da una periferica

Questo paragrafo descrive lo scambio dei messaggi al fine di disconnettere la periferica remota. Equivalentemente ai casi precedenti l'utente, toccando un apposito pulsante nella vista della periferica, avvia il caso d'uso. Tale azione viene rilevata dal controllore che genera il messaggio da propagare agli altri moduli. A questo punto il driver, al quale è stata notificata la richiesta, provvede a disconnettere il dispositivo. In particolare, come prima cosa, interrompe l'acquisizione dei dati e successivamente comunica alla periferica il tentativo di disconnessione. Poiché la periferica stava comunicando con l'applicazione si assume che, entro un tempo ragionevolmente breve, il driver riceverà la risposta che comunica lo stato dell'operazione. Di conseguenza è stato ritenuto inutile attivare un timer. Contrariamente quando la periferica viene allontanata dal dispositivo e si

verifica una perdita di comunicazione, tale evento sarà trattato come una richiesta di connessione. Quindi, per quanto riportato nel paragrafo 5.2.3, dopo 60 secondi se la periferica non risponde verrà dichiarata disconnessa. Per terminare il caso d'uso il driver pubblicherà un evento con il risultato dell'operazione. In caso di successo il messaggio causerà l'aggiornamento della vista, dando la possibilità di connettersi ad un'altra periferica. Contrariamente, se si verifica un errore, il caso d'uso terminerà mostrando un messaggio all'utente con una sequenza simile a quella in Figura 9.

5.2.5 Diagramma di sequenza: Ricerca periferiche

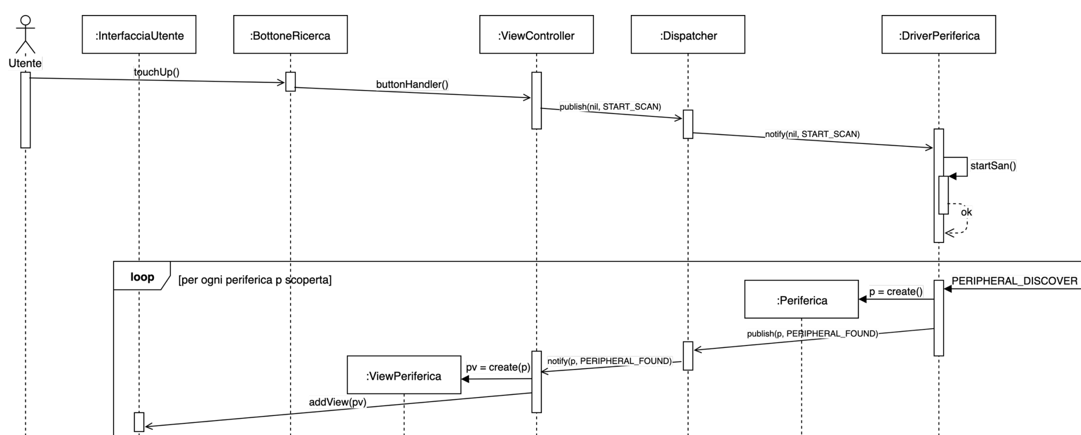


Figura 12 Diagramma di sequenza per ricercare periferiche

Il diagramma di sequenza mostra i messaggi scambiati per realizzare il caso d'uso relativo alla ricerca di periferiche. L'azione viene avviata dall'utente che preme su un bottone. Di conseguenza il controllore riceve la richiesta e genera l'evento verso il modello. A questo punto il driver effettua l'operazione che permette di recuperare le periferiche che stanno pubblicizzando annunci. Per ciascuna periferica scoperta il driver riceve un messaggio. Di conseguenza crea l'oggetto che rappresenta il dispositivo remoto nella logica di business e pubblica un evento per informare gli altri moduli. In particolare il controllore riceve il messaggio, quindi crea l'elemento grafico che incapsula la periferica e lo aggiunge all'interfaccia utente. Così facendo l'utente può interagire con il dispositivo trovato, ad esempio per connettersi ad esso. Anche in questo caso possono verificarsi errori che dovranno essere gestiti, con tecniche analoghe a quelle descritte nei paragrafi precedenti, al fine di comunicare la situazione all'utente.

5.2.6 Diagramma di sequenza: Interrompi ricerca periferiche

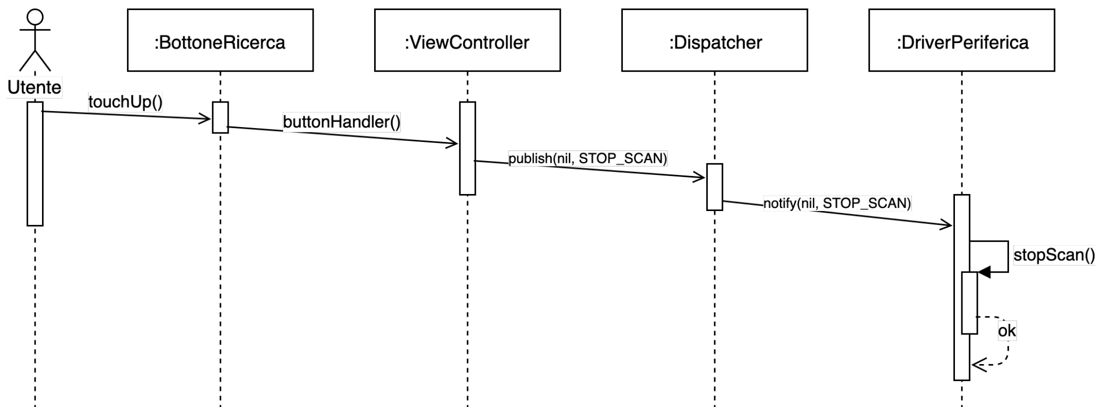


Figura 13 Diagramma di sequenza per interrompere la ricerca di periferiche

La figura mostra lo scambio dei messaggi con l'obiettivo di interrompere la ricerca di nuove periferiche. L'azione parte dall'utente e viene propagata all'driver il quale interrompe la scansione. Nonostante il semplice scenario si possono comunque verificare degli errori, ad esempio quando l'utente avvia il caso d'uso mentre non è in esecuzione nessuna ricerca. In tal caso deve essere mostrato un messaggio di errore all'utente.

5.2.7 Diagramma di sequenza: Memorizza dato

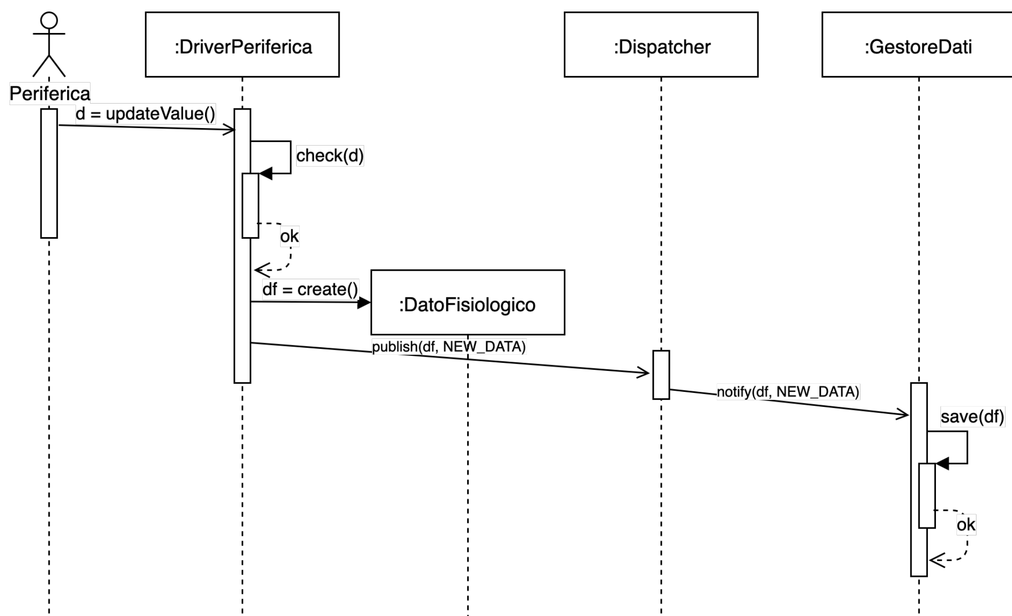


Figura 14 Diagramma di sequenza per memorizzare un dato

Il diagramma mostra i messaggi scambiati tra i vari moduli al fine di memorizzare un dato permanentemente nel dispositivo. In particolare il caso d'uso viene avviato

dalla periferica remota che comunica un nuovo valore. Tale messaggio viene recepito dal driver che controlla l'integrità del pacchetto, alla quale segue la creazione dell'oggetto. A questo punto il driver pubblica l'evento che viene notificato al gestore dei dati, il quale provvede a memorizzare l'informazione nel dispositivo locale. Poiché l'intero caso d'uso è nascosto all'utente e la perdita di un dato non è significativa, in caso di errore è possibile interrompere brutalmente il protocollo di memorizzazione. Di conseguenza il dato non verrà registrato e si attenderà la notifica di un nuovo valore.

5.2.8 Diagramma di sequenza: Visualizza dati

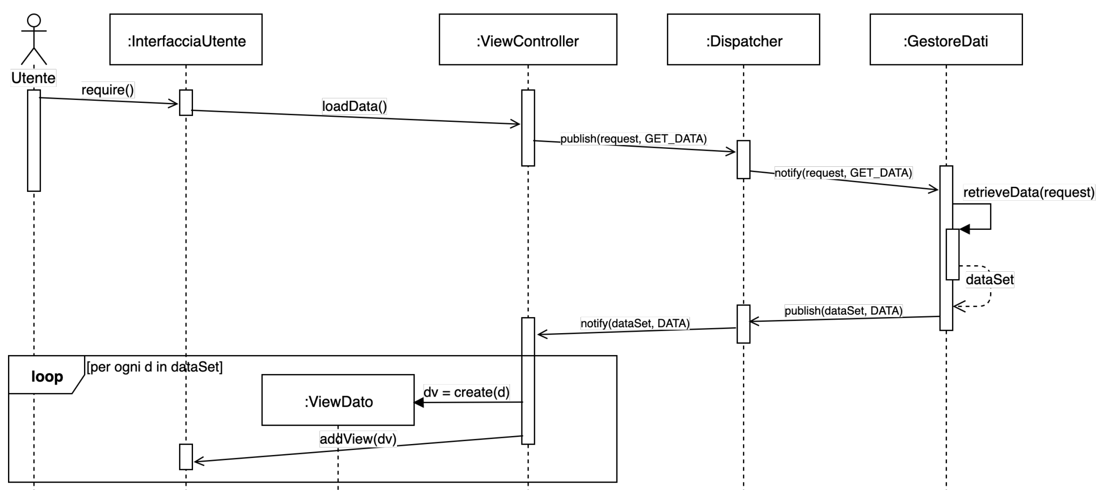


Figura 15 Diagramma di sequenza per visualizzare i dati

Il diagramma mostra i messaggi scambiati tra i vari oggetti al fine di mostrare i dati. Generalmente il caso d'uso inizia quando l'utente vuole visualizzare una vista. Tale attività avvia nel controllore l'operazione di caricamento dati, la quale propaga l'evento generando un messaggio verso il modello. A questo punto il gestore dei dati riceve la richiesta, recupera le informazioni e le invia pubblicando un messaggio. Di conseguenza il controllore riceve l'insieme degli oggetti da mostrare all'utente. Quindi incapsula ciascun dato in un elemento grafico e lo inserisce nell'interfaccia utente. Nell'ipotesi in cui non ci sia nessuna informazione da visualizzare, a seconda della richiesta, dovrà essere generato un errore o in alternativa un componente grafico che informa l'utente della situazione corrente.

CAPITOLO 6
ARCHITETTURA SOFTWARE ED IMPLEMENTAZIONE

In questo capitolo verrà analizzata l'architettura del sistema mostrando le componenti principali. In primo luogo verranno descritti i pattern Model-View-Controller e publish-subscribe con riferimenti a dettagli implementativi. In secondo luogo verrà introdotto il driver e le sue funzionalità. Inoltre si riporterà alcune caratteristiche del framework CoreBluetooth allo scopo di facilitare la comprensione al lettore di alcuni concetti. Dopodiché verranno determinati i controllori e la struttura del file JSON generato. Infine daremo una panoramica dei concetti utilizzati per mantenere l'applicazione in background H24.

6.1 Organizzazione e suddivisione dei moduli del progetto

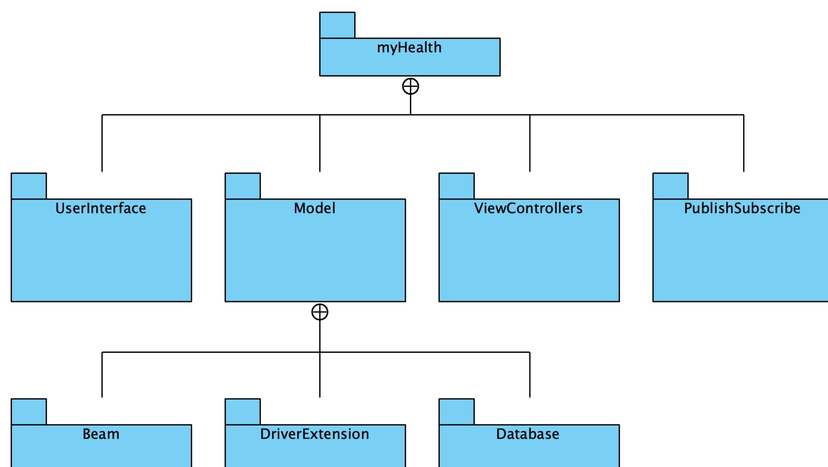


Figura 16 Vista strutturale di decomposizione della root del progetto

La Figura 16 mostra una vista strutturale di decomposizione riportando i pacchetti principali. “UserInterface” definisce le varie viste personalizzate usate per mostrare i dati all’utente e creare l’interfaccia grafica. Invece “Model” raccoglie tutti i moduli necessari per comunicare con la periferica al fine di recuperare e salvare i dati. In particolare si suddivide a sua volta in pacchetti secondari: “Beam” contiene i moduli del driver fornito dall’azienda, “DriverExt” definisce l’estensione di quest’ultimo per offrire ulteriori funzionalità ed infine “Database” raggruppa le classi necessarie alla gestione dei dati. Per quanto riguarda il pacchetto “ViewControllers” contiene i vari moduli utilizzati per ricevere ed interpretare le richieste dell’utente, eventualmente cambiando lo stato dell’applicazione.
Infine “PublishSubscribe” definisce i sorgenti utili a realizzare la comunicazione tra componenti al fine di creare classi coese e disaccoppiate.

Si informa il lettore che tali pacchetti verranno affrontati singolarmente e descritti nel dettaglio nei paragrafi successivi, mostrando i vari moduli e le loro funzionalità.

6.2 Architettura dell’applicazione

In questa sezione verrà mostrata l’architettura complessiva del sistema. Inoltre saranno descritte le principali responsabilità di ciascun componente.

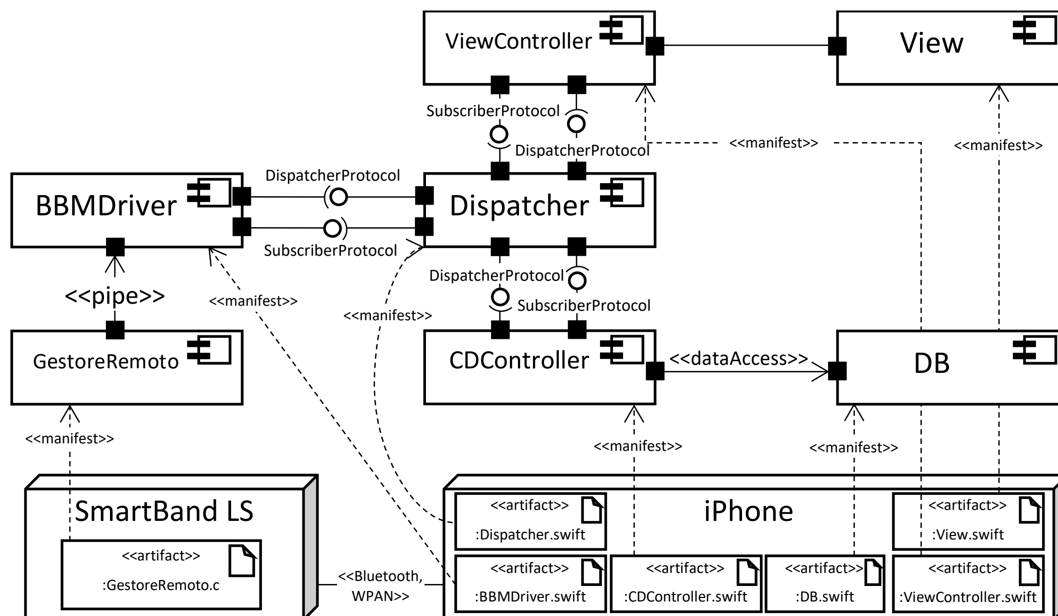


Figura 17 Vista ibrida (dislocazione e C&C) dell'architettura del sistema

Di seguito daremo un'immagine che astrae da particolari diagrammi e che riporta maggiori dettagli sui moduli che compongono l'architettura.

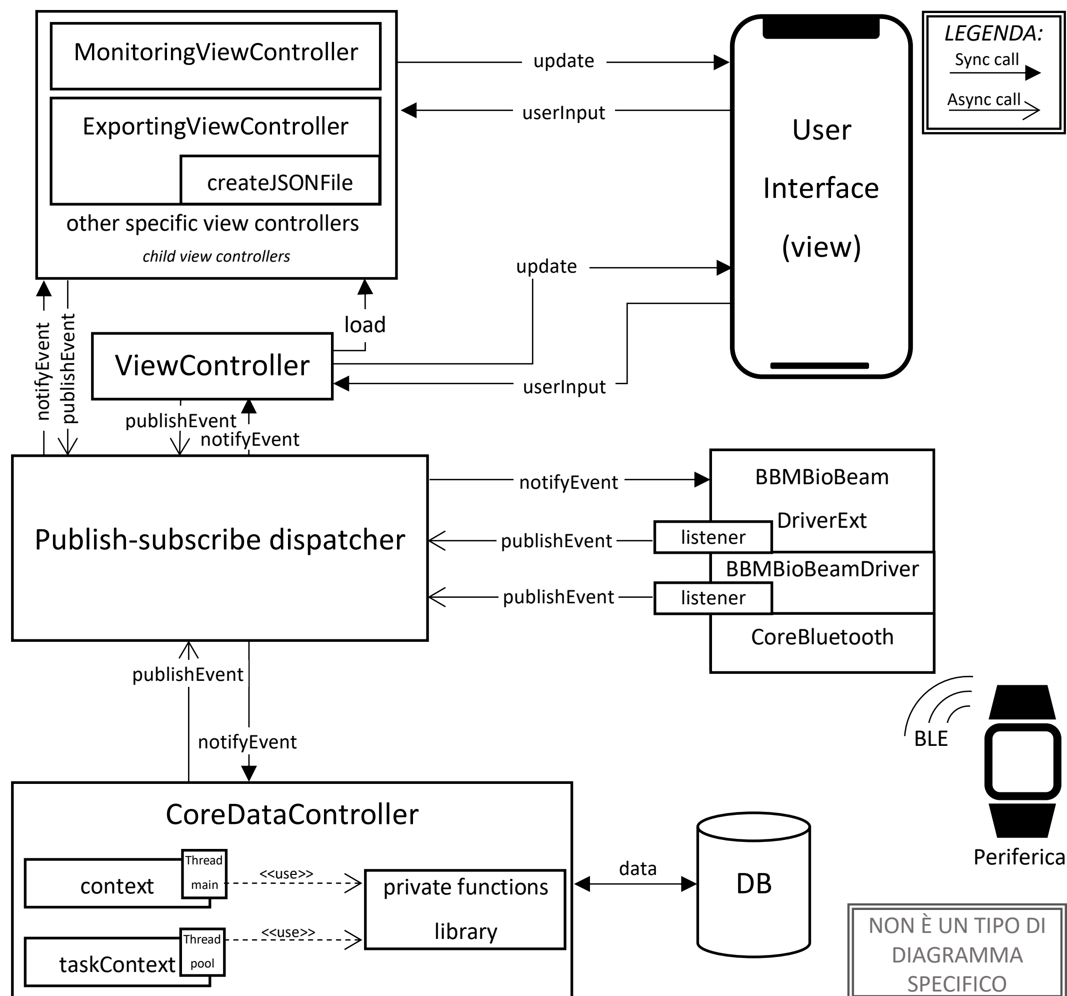


Figura 18 Rappresentazione ad alto livello dell'architettura dell'applicazione

La figura mostra i principali moduli che realizzano l'applicazione. In particolare la periferica comunica con il driver al fine di inviare dati al dispositivo. Tale componente, che è stato fornito dall'azienda presso la quale si è svolto il tirocinio, si basa sul framework CoreBluetooth. Il modulo è stato sviluppato in Objective-C ed includeva dipendenze verso altri framework che causavano problemi con il linguaggio Swift. Di conseguenza è stato fondamentale comprendere il codice fornito ed eliminare le dipendenze che causavano gli errori. Tutto ciò, non solo ha permesso di svolgere un'attività di revisione del codice[13] favorevole all'azienda, ma

[13] Generalmente si utilizza il termine in lingua inglese code review. Indica un'attività di controllo di un programma leggendo parti del suo codice.

principalmente è stato utile per incrementale le personali abilità nella comprensione di sorgenti scritti da altri programmatori. In secondo luogo, una volta predisposto il corretto funzionamento del driver, è stata costruita un'estensione che dotasse il modulo di caratteristiche necessarie alla realizzazione dell'applicazione. In particolare l'espansione è basata sulla stessa logica del driver così da facilitare la revisione agli sviluppatori di quest'ultimo. Il lettore interessato alle funzionalità offerte da questo modulo può proseguire la lettura al paragrafo 6.6.

Per quanto riguarda la memorizzazione dei dati, essa viene fatta dal modulo CoreDataController che comunica direttamente con il framework CoreData. Oltre a ciò tale componente fornisce le operazioni necessarie al recupero dei dati. Per fare questo impiega un insieme di metodi privati, eseguiti su notifiche provenienti dal dispatcher. Il lettore può approfondire questo componente nel CAPITOLO 7, nel quale viene trattata anche la progettazione del database.

Per quanto riguarda la parte superiore della figura è possibile notare i vari controllori. Lo scopo di questi moduli è ricevere i comandi dall'utente per trasformarli in messaggi da inviare al modello. Oltre a ciò devono predisporre le informazioni sull'interfaccia in modo tale che l'utente possa visualizzarle. Il lettore può ottenere maggiori dettagli nel paragrafo 6.7 del capitolo corrente.

Infine, come si può intuire dalla figura, ciò che l'utilizzatore finale realmente vede ed utilizza sono le varie viste dettagliate nel CAPITOLO 8.

Nei paragrafi successivi verranno descritti i pattern Model-View-Controller e publish-subscribe.

6.3 Pattern Model-View-Controller

L'applicazione è stata sviluppata secondo il pattern MVC (Model-View-Controller), al fine di realizzare un software basato sui componenti e sulle responsabilità.

Come specificato in [12] [13], il modello consiste in un insieme di dati e dei metodi necessari per processarli. Ad ogni modello sono associate una o più viste, ciascuna delle quali mostra una o più rappresentazioni grafiche sullo schermo. La vista è

inoltre in grado di eseguire le operazioni esposte dal modello. Il livello di collegamento tra le due parti è rappresentato dal controllore[14] che deve offrire all'utente i mezzi per fornire comandi e dati. Quando quest'ultimo riceve un output per l'utente, lo traduce in un messaggio e lo invia ad una o più viste. Equivalentemente, se riceve un input dall'utente che deve aggiornare lo stato, lo trasforma in un messaggio e lo invia al modello. In particolare nell'applicazione è stata usata la versione Cocoa[15] del pattern MVC riportata nella seguente figura che riassume i concetti precedentemente espressi.

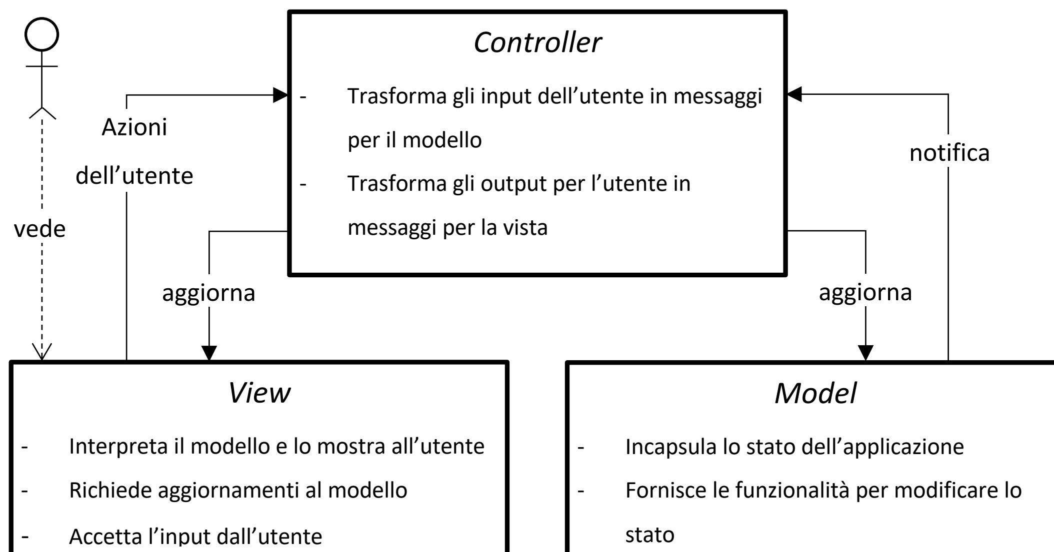


Figura 19 Responsabilità MVC

In riferimento alla Figura 18, nell'applicazione il modello è rappresentato dal driver e dal gestore del database i quali sono adibiti alla ricezione dei dati ed al loro salvataggio e recupero. Invece il controllore è l'insieme dei moduli che comunicano direttamente con la vista al fine di interpretare l'input dell'utente e formattare l'output. Infine, per quanto riguarda l'interfaccia utente, è costituita dall'insieme degli elementi grafici utilizzati per mostrare i dati all'utente. In particolare sulla base delle scelte dell'utente il view controller[16] mostrerà viste diverse.

[14] Tipicamente indicato con il suo termine in lingua inglese controller.
[15] Ambiente nativo di programmazione ad oggetti sviluppato da Apple per i sistemi operativi, come macOS e iOS.
[16] Controllore adibito a gestire una parte dell'interfaccia grafica.

Nel paragrafo successivo viene descritto il pattern publish-subscribe utilizzato per la comunicazione tra i vari attori, al fine di mantenere il sistema il più possibile disaccoppiato.

6.4 Pattern publish-subscribe

Per quanto riguarda la comunicazione tra i vari attori del MVC abbiamo adottato lo schema di messaggistica publish-subscribe. Come riportato in [14], i componenti interagiscono annunciando eventi. In particolare mittenti e destinatari dialogano attraverso un intermediario, chiamato dispatcher o broker. Ciascun componente deve sottoscriversi al sottoinsieme degli eventi ai quali è interessato, sarà poi compito dell'infrastruttura publish-subscribe assicurarsi che ogni messaggio pubblicato sia consegnato a tutti gli abbonati. La principale forma di dispatcher in questi stili può essere vista come una sorta di bus di eventi annunciati dai vari mittenti. Questo schema permette di creare componenti disaccoppiati in modo tale che la correttezza di un modulo non dipenda da quella degli altri. Questo perché il mittente pubblica il messaggio, ma non si preoccupa di quanti e quali destinatari lo riceveranno. Equivalentemente l'abbonato a cui viene notificato un evento non conosce il mittente del messaggio. Questa proprietà, riducendo il numero di dipendenze fra i moduli, permette di facilitare le modifiche ad una parte del sistema senza intaccare altre parti di esso. In particolare nel progetto è stato utilizzato uno stile denominato event-only. Ovvero il dispatcher è un semplice oggetto che si occupa di instradare i messaggi ai componenti appropriati. È compito del ricevitore gestire l'evento. Questo sistema permette di avere un insieme di componenti più eterogeneo. Nel dettaglio il sistema utilizza un filtraggio basato su argomenti, cioè i messaggi vengono pubblicati su canali logici con nomi. In questo modo gli abbonati riceveranno tutti e solo i messaggi degli eventi ai quali si sono registrati.

Nel paragrafo successivo verrà descritta l'infrastruttura publish-subscribe utilizzata nell'applicazione.

6.4.1 Infrastruttura publish-subscribe dell’applicazione

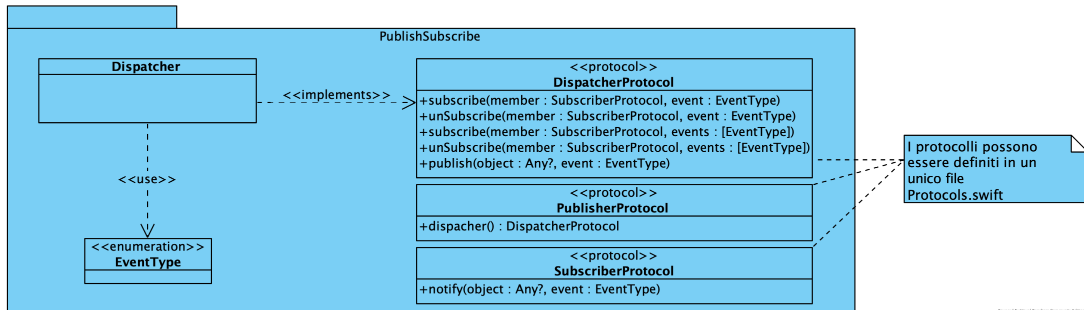


Figura 20 Contenuto pacchetto PublishSubscribe

Il pacchetto “PublishSubscribe” contiene solamente tre file swift che non hanno dipendenze verso altri moduli. Di conseguenza può essere liberamente copiato in altri progetti per riutilizzare lo stesso meccanismo di comunicazione tra componenti. In particolare il file “EventType.swift” definisce un’enumerazione per identificare gli eventi, così da rendere il codice più leggibile. Tale struttura non è fondamentale e può essere tranquillamente sostituita con un identificativo evento mediante un tipo primitivo. Per quanto riguarda la classe “Dispatcher” definisce l’implementazione e la logica per ricevere messaggi dai mittenti e comunicarli ai vari destinatari. Infine il file “Protocols.swift” specifica i protocolli che devono essere implementati dai vari attori e che saranno descritti nel paragrafo successivo.

6.4.1.1 *Definizione dei protocolli*

In questo paragrafo vengono riportati e descritti i protocolli utilizzati nella comunicazione. Infine daremo una vista di tipo comportamentale (aka componenti e connettori) per delineare una visione ad alto livello della dinamica.

<<protocol>> DispatcherProtocol

`subscribe(member: SubscriberProtocol, event:EventType);`
`subscribe(member: SubscriberProtocol, events:[EventType]);`
`unSubscribe(member: SubscriberProtocol, event:EventType);`
`unSubscribe(member: SubscriberProtocol, events:[EventType]);`
`publish(object: Any?, event:EventType);`

Codice 1 Protocollo del dispatcher

Il dispatcher deve fornire metodi per permettere alle varie componenti di abbonarsi ed annullare l’iscrizione a determinati eventi. In particolare sono fornite funzioni per

effettuare le operazioni anche su più eventi contemporaneamente. Oltre a ciò definisce il metodo che consente, ai vari mittenti, di inviare messaggi al dispatcher specificando l'evento.

<<protocol>> PublisherProtocol	<<protocol>> SubscriberProtocol
`dispatcher:DispatcherProtocol { get }`	`notify(object: Any?, event:EventType)`

Codice 2 Protocolli per i publisher e subscriber

I vari mittenti devono mantenere un riferimento al dispatcher per poter pubblicare gli avvisi, invece i destinatari devono implementare il metodo notify necessario a gestire i messaggi ricevuti. In particolare al destinatario deve essere comunicato l'evento che si è verificato ed opzionalmente un oggetto che riporta eventuali informazioni aggiuntive. Infine, per dare una visione di come tutti questi attori interagiscono, riportiamo una vista comportamentale.

Figura 21 Vista comportamentale Publish-Subscribe

Nel paragrafo successivo verranno descritti i dettagli implementativi del dispatcher.

6.4.1.2 Implementazione del dispatcher

Questa sezione intende riportare alcuni dettagli implementativi del dispatcher contenuti nel file "Dispatcher.swift". La seguente figura mostra le strutture principali utilizzate da questo componente.

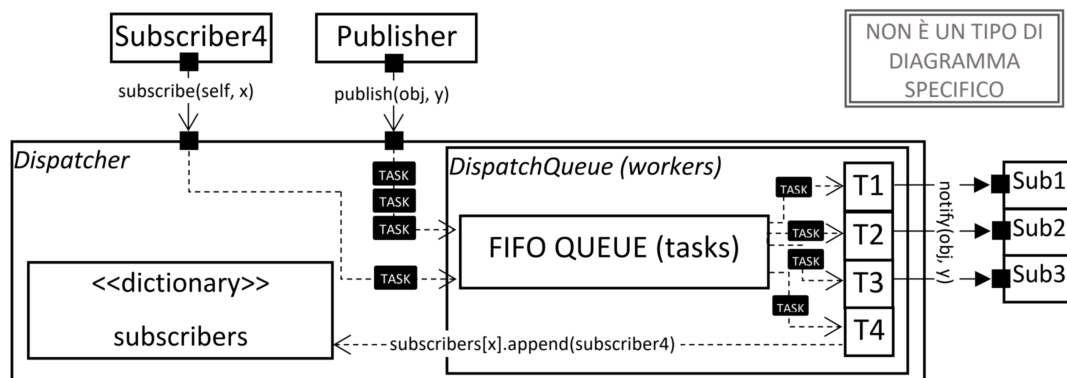


Figura 22 Struttura del dispatcher

Come si può notare dalla figura il dispatcher si compone di un dizionario e di una dispatch queue[17]. In particolare il dizionario è la struttura dati utilizzata per memorizzare i vari abbonati la quale, dato un particolare evento, restituisce l'array dei moduli interessati a quell'argomento. È facile intuire che il dispatcher debba essere thread-safe in quanto più thread potrebbero contemporaneamente pubblicare eventi mentre altri abbonano o annullano l'iscrizione di alcuni componenti. Di conseguenza tutti gli accessi alla struttura "subscribers" sono protetti da una read/write lock[18]. Inoltre, come si può vedere dalla figura, qualsiasi metodo venga invocato sul dispatcher genera un'attività sottoposta alla dispatch queue, ritornando immediatamente il controllo al chiamante. Come specificato in [15], le dispatch queue sono code FIFO che permettono di eseguire attività in modo concorrente, utilizzando un thread-pool gestito dal sistema. In particolare i compiti generati dai metodi subscribe ed unsubscribe apportano modifiche al dizionario.

```
func publish(object: Any?, event:EventType)
{
  workers.async {
    pthread_rwlock_rdlock(&self.lock)
    for subscriber in
    self.subscribers[event] ?? [] {
      self.workers.async {
        subscriber.notify(object: object,
        event: event)
      }
    }
    pthread_rwlock_unlock(&self.lock)
  }
}
```

Codice 3 Implementazione del metodo publish

Invece un'operazione di publish può generare più task, uno per ciascun abbonato all'evento in questione. Quando un thread svolge uno di questi compiti va ad eseguire il metodo notify del destinatario. Di conseguenza possono esistere più thread che eseguono lo stesso codice di gestione di un abbonato in modo concorrente. Da ciò segue che tutti i metodi notify devono essere thread-safe per evitare errori di concorrenza. Questo perché non è possibile stabilire a priori quando e chi pubblicherà determinati eventi.

[17] Sono code per eseguire attività in modo asincrono o sincrono rispetto al chiamante.
[18] Tipo di lock che permette un accesso concorrente per le operazioni di lettura ed esclusivo per quelle di scrittura.

6.5 Framework CoreBluetooth

Come specificato in [6], il framework CoreBluetooth fornisce le classi necessarie per comunicare con i dispositivi che supportano la tecnologia wireless Bluetooth a bassa energia.

Quando si lavora con questa tecnologia è necessario distinguere i due attori principali: centrali e periferiche. In generale le periferiche possiedono informazioni richieste dalla centrale per eseguire delle attività. La comunicazione tra le due parti si basa sull'architettura client-server. Come prima cosa, quando ci si avvicina a questo framework, bisogna individuare quale tra i due attori sarà remoto e quale locale. Questa scelta determinerà le classi di CoreBluetooth che dovremo utilizzare. Nel caso specifico avremo la periferica remota e la centrale locale. Ciò porta ad utilizzare principalmente le seguenti quattro classi: CBCentralManager, CBCentralManagerDelegate, CBPeripheral e CBPeripheralDelegate.

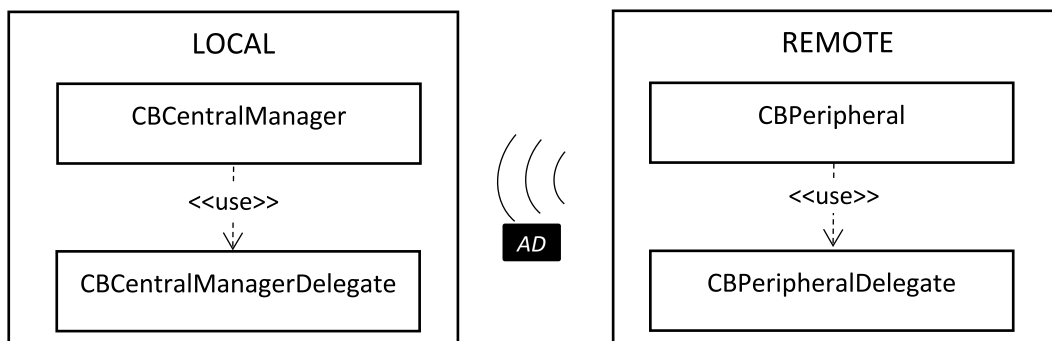


Figura 23 Rappresentazione delle classi di CoreBluetooth

Un'istanza di CBCentralManager rappresenta una centrale locale fornendo i metodi necessari per scoprire, connettersi e disconnettersi dalle periferiche remote, rappresentate dalla classe CBPeripheral. In particolare quest'ultima consente di scoprire servizi e caratteristiche dei dispositivi remoti. Generalmente quando si istanzia la classe CBCentralManager è necessario determinare il suo delegato, ovvero l'oggetto di una classe che implementa il protocollo CBCentralManagerDelegate. Tale istanza definisce i comportamenti da adottare quando si verificano determinati eventi, come ad esempio la scoperta di nuove periferiche.

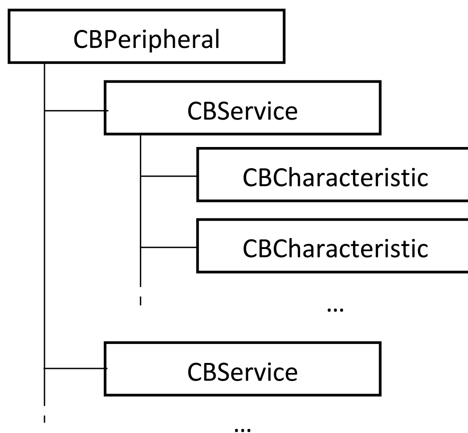


Figura 24 Organizzazione di una CBPeripheral

Equivalentemente, se CBPeripheral rappresenta le periferiche remote, la classe CBPeripheralDelegate sarà il protocollo da implementare per definire le azioni al fine di monitorare la scoperta e l'interazione dei servizi e delle proprietà del dispositivo remoto. In particolare, in CoreBluetooth, i servizi sono rappresentati dalla classe CBService e le caratteristiche dalla classe CBCharacteristic. Generalmente l'utilizzo del Bluetooth può essere scandito principalmente in quattro fasi:

1. Scoperta dei dispositivi remoti.
2. Connessione alla periferica dopo averla scoperta.
3. Scoperta dei servizi della periferica.
4. Scoperta delle caratteristiche del servizio scelto.

Determinata la caratteristica è possibile leggerne il valore o sottoscriversi per recuperare attributi dinamici. Quest'ultima azione permette di ricevere automaticamente notifiche dalla periferica quando il valore cambia.

Tali informazioni sono state fondamentali per comprendere il driver fornito dall'azienda e per estenderlo con le funzionalità mancanti. Di seguito alcuni degli aspetti, introdotti in questo paragrafo, verranno maggiormente dettagliati in relazione all'applicazione realizzata.

6.6 Driver di comunicazione con la periferica

In riferimento alla Figura 16, il pacchetto "Beam" contiene il driver per comunicare con la periferica, fornito dall'azienda presso la quale si è svolto il tirocinio. In particolare, come prima cosa, è necessario connettersi alla periferica e sottoscriversi alle caratteristiche del servizio. Dopodiché lo scopo del driver è quello di decodificare ogni pacchetto ricevuto e passarlo al livello superiore. Ciò ha permesso di lavorare ad un maggior livello di astrazione senza entrare nel protocollo specifico e nel parsing

dei dati inviati dal dispositivo Lifesense Band 2. Il modulo è stato scritto dal team di BioBeats con il linguaggio Objective-C. Di conseguenza è stato proposto di riscrivere un nuovo driver in Swift, ma a causa della mancanza di tempo il tutor aziendale ha consigliato l'utilizzo del loro componente. Tale scelta è stata presa considerando anche gli altri compiti da completare, come l'interfaccia utente e la progettazione del database. Ovviamente, prima di utilizzare questo driver, è stato necessario comprendere il codice scritto. Questa attività, non solo ha incrementato le personali abilità nel leggere i sorgenti di altri programmatori, ma ha fornito la possibilità di apprendere la sintassi del linguaggio Objective-C che risulta molto diversa da quella di Swift. Inoltre, questo compito, è stato utile anche all'azienda come attività di code review.

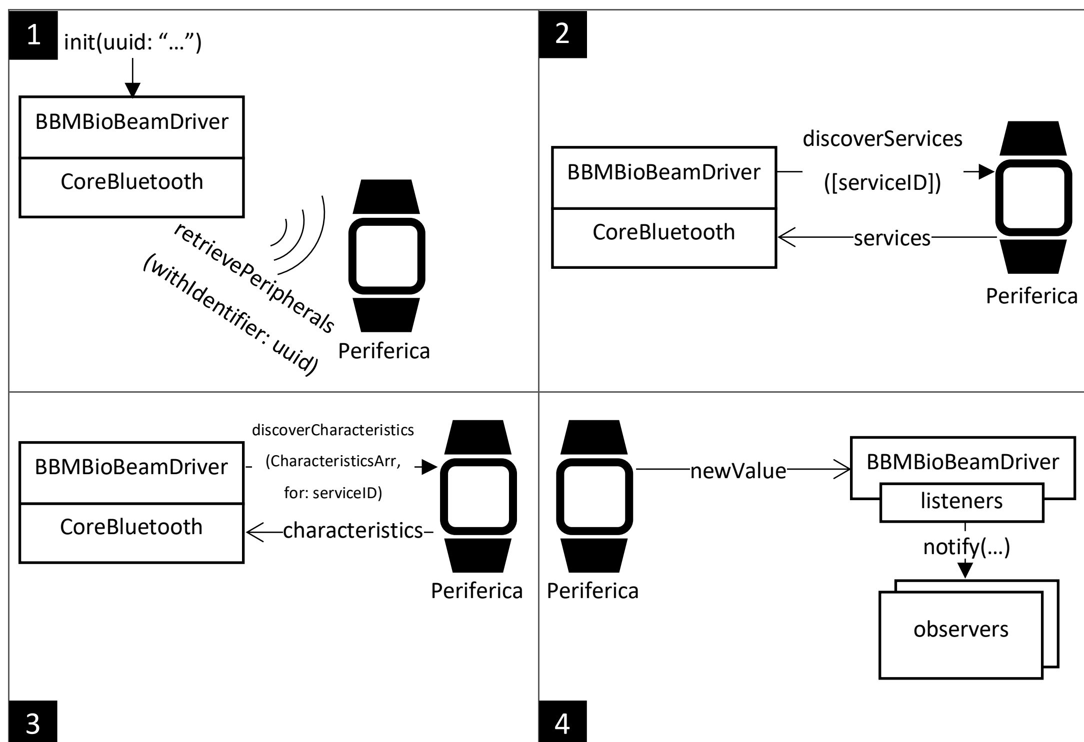


Figura 25 Inizializzazione e utilizzo del driver

In particolare il driver si appoggia sulla libreria nativa CoreBluetooth e richiede, in fase di costruzione dell'oggetto, l'identificativo univoco universale della periferica alla quale connettersi. Stabilita la connessione vengono scoperte le caratteristiche del servizio interessato. In particolare, senza entrare nei dettagli specifici del protocollo della periferica, si ricercano tre caratteristiche necessarie a cooperare ed interagire con il dispositivo remoto. Specificatamente siamo interessati alla

caratteristica che permette di ottenere notifiche relative ai cambiamenti dei valori corrispondenti alla frequenza cardiaca, numero di passi, variabilità RR e qualità del sonno. Di conseguenza, una volta individuata, il driver utilizza il metodo setNotifyValue di CoreBluetooth per comunicare la volontà di essere informato sugli annunci relativi alle misurazioni di tali valori. A questo punto, per ciascun pacchetto contenente un dato, il protocollo prevede di inviare alcuni frame. Questi ultimi devono essere interpretati al fine di ottenere le informazioni. Di conseguenza, come prima cosa, è opportuno individuare il tipo di dato inviato dalla periferica. Dopodiché bisogna effettuare il parsing, ovvero leggere segmenti di byte per comprendere il significato del pacchetto. Le istruzioni relative all'interpretazione dei byte sono riportate nel protocollo fornito dal costruttore del dispositivo. Successivamente alla ricostruzione del dato, il driver termina la sua funzionalità passandolo al modulo di livello superiore. In particolare la comunicazione tra il driver e i moduli esterni avviene mediante il pattern observer. Tale stile prevede che il soggetto, la componente che notifica gli eventi, offra dei metodi per registrare ed eliminare gli observer, ovvero i moduli interessati a ricevere le notifiche. Questo pattern risulta simile al pattern publish-subscribe con la sostanziale differenza che in quest'ultimo il destinatario del messaggio non conosce il mittente e viceversa.

In particolare il driver svolge il ruolo di soggetto al quale sottoporre gli oggetti che rispettano l'interfaccia per ricevere le informazioni. È compito della classe che implementa tale protocollo determinare le operazioni da fare quando viene notificato un nuovo dato.

Oltre a ciò il driver fornisce ulteriori funzionalità, come ad esempio l'aggiornamento del firmware della periferica, che non sono state approfondite in quanto irrilevanti ai fini della realizzazione del progetto.

Nei paragrafi successivi verranno descritte le problematiche e le soluzioni adottate per integrare il driver, scritto in Objective-C, nel progetto sviluppato in Swift. Infine verrà descritta, con riferimenti al codice implementativo, l'estensione del driver realizzata per creare ulteriori funzionalità.

6.6.1 Il driver e le dipendenze con il Protobuf di Google

Prima *Dopo*

Il driver fornito dall'azienda faceva utilizzo del protocol buffers di Google. Come specificato in [16], esso fornisce un meccanismo automatizzato per serializzare dati strutturati. Ovvero è come l'XML, ma più semplice e veloce. Per utilizzarlo è sufficiente creare il file con estensione ".proto" che definisce il formato dei messaggi. Dopodiché il compilatore del protocol buffers genera le classi per accedere ai dati nel linguaggio dell'applicazione. La versione corrente supporta molti linguaggi tra i quali anche Objective-C, ma attualmente Swift non è predisposto da Google. Nonostante ciò non significa che non sia possibile utilizzarlo, in quanto su Github è possibile trovare la libreria SwiftProtobuf sviluppata da Apple. Malgrado ciò, secondo l'esperienza del team di BioBeats, l'integrazione può risultare macchinosa e talvolta, se fatta male, non sempre completamente funzionante. Per questo motivo, considerando anche il tipo di applicazione, su consiglio del team aziendale è stato deciso di rimuovere le dipendenze verso il protobuf di Google. Per fare ciò è stato creato un nuovo progetto con l'obiettivo di far funzionare il driver con Swift per poi importarlo nell'applicazione finale. Di conseguenza, oltre alla fase di code review, sono state apportate modifiche per trasformare ogni struttura del protocol buffers in un dizionario nativo di Objective-C. Tale operazione è stata costantemente monitorata da alcuni sviluppatori dell'azienda che in qualsiasi momento potevano vedere le modifiche apportate e inviate su Github.

Dopo aver eliminato le dipendenze verso il protobuf è stato necessario far interagire il codice del driver, scritto in Objective-C, con il linguaggio Swift utilizzato per sviluppare gli altri moduli dell'applicazione. Tali aspetti saranno dettagliati nel paragrafo successivo.

6.6.2 Interazione tra Objective-C e Swift

Eliminate le dipendenze verso il protocol buffers, come riportato nel paragrafo precedente, deve comunque essere possibile chiamare il driver sviluppato in Objective-C da Swift. Come specificato in [17], per fare ciò è essenziale creare un file bridging header[19] nel quale inserire le interfacce delle classi che devono essere utilizzate in codice Swift. Dopodiché è necessario controllare che nelle impostazioni di build[20] del compilatore, all'opzione Objective-C Bridging Header, sia indicato il percorso del file creato. A questo punto, esponendo il codice del driver, è possibile creare oggetti della classe Objective-C dai sorgenti scritti in Swift.

6.6.3 Estensione del driver con nuove funzionalità

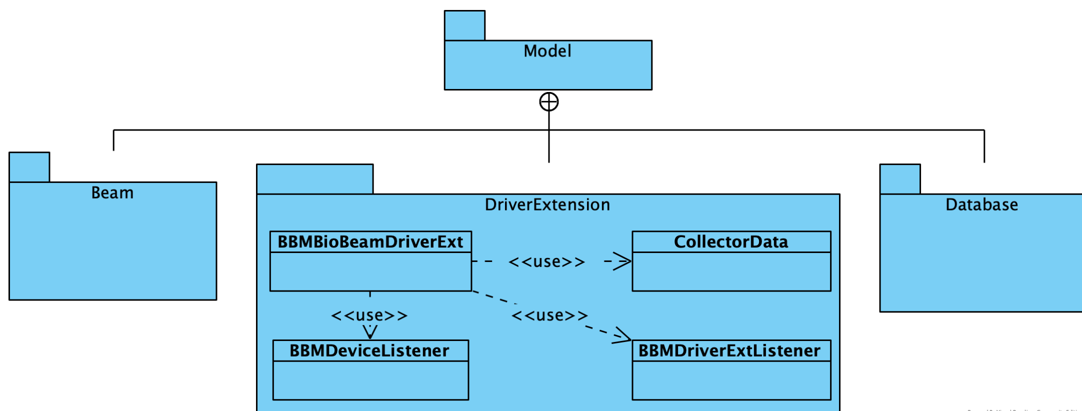


Figura 26 Vista strutturale (decomposizione ed uso) del pacchetto DriverExtension

La cartella "DriverExtension" contiene le classi necessarie ad estendere le funzionalità del driver ed a definire i vari ascoltatori del pattern observer. In particolare la classe "CollectorData" realizza l'osservatore al quale vengono notificati i nuovi dati rilevati dalla periferica. Tale componente ne verifica l'integrità ed informa il dispatcher della comunicazione al fine di notificare l'evento al gestore del database. Per quanto riguarda "BBMDriverExtListener" è un osservatore passato

[19] Particolare file che permette l'interoperabilità tra Objective-C e Swift.
[20] Contengono informazioni su aspetti particolari del processo di generazione di un'applicazione.

direttamente al livello inferiore ed utilizzato per comunicare agli altri moduli dettagli sul driver come lo stato corrente, situazioni di errore, dispositivo connesso ed altre informazioni. L'implementazione non fa altro che generare un messaggio per il dispatcher in modo tale che tutti i componenti interessati lo ricevano ed operino di conseguenza. Invece il sorgente "BBMBioBeamDriverExt" definisce funzionalità aggiuntive al driver. In particolare l'estensione rispetta lo stile di programmazione appreso leggendo il codice fornito. Infatti, mediante la sovrascrittura, alcune parti del driver sono state riscritte facendo particolare attenzione a lasciarlo sempre in uno stato consistente. Tra le varie operazioni aggiunte vi è la possibilità di richiedere lo stato del Bluetooth del dispositivo, lo stato di monitoraggio ed ottenere informazioni sulla periferica attualmente connessa. Oltre a ciò sono state definite operazioni più complesse che hanno richiesto anche di interagire direttamente con le classi di CoreBluetooth. Tra le quali viene data la possibilità di avviare e fermare la scansione di nuove periferiche. Per fare ciò è stato necessario ridefinire alcuni metodi del CBCentralManagerDelegate introducendo un nuovo osservatore per comunicare, alle altre componenti, i dispositivi trovati. Analogamente ulteriori modifiche al delegato sono state necessarie per permettere la connessione e disconnessione verso una periferica dopo aver inizializzato il driver.

```
/* ... */
case .TIME_EXPIRED: do { //Tempo scaduto per la risposta di connessione
   if self.driverStatus == BBMDriverStatus_CONNECTING {
      //Resetta lo stato del driver
      self.driverStatus = BBMDriverStatus_NO_DEVICE
      for listener in self.driverListeners ?? [] {
         listener.driverDidError?(
           DriverError.init("Time expired for connection to the device.",
           false)
         )
      }
   }
}
/* ... */
```

Codice 4 Gestione tentativo di connessione fallita

In particolare il tentativo di connessione attiva un timer di 60 secondi dopo i quali la mancata risposta viene interpretata come una connessione fallita.

```
func suspendNotifications() -> BBMDriverOperationReturnStatus {
   guard self.driverStatus == BBMDriverStatus_WAITING_NOTIFICATION else {
      return BBMDriverOperationReturnStatus_OPERATION_NOT_PERMITTED
   }
   guard notifyCharacteristic != nil &&
```

```
          notifyCharacteristic!.isNotifying &&
          indicateCharacteristic != nil &&
          indicateCharacteristic!.isNotifying else {
       return BBMDriverOperationReturnStatus_FLOW_ERROR
    }
    peripheral?.setNotifyValue(false, for: notifyCharacteristic!)
    peripheral?.setNotifyValue(false, for: indicateCharacteristic!)
    return BBMDriverOperationReturnStatus_SUCCESS
 }
```

Codice 5 Metodo del driver per sospendere la ricezione dei pacchetti

Oltre a ciò, comunicando direttamente con il framework CoreBluetooth, l'estensione permette di sospendere e riprendere l'acquisizione di pacchetti da parte del dispositivo remoto, come riportato nel Codice 5. In particolare, dopo aver effettuato i relativi controlli, si utilizza il metodo setNotifyValue per comunicare la richiesta alla periferica. Ovviamente la ridefinizione dei metodi è tale da continuare a supportare tutte le funzionalità che il driver offriva precedentemente, lasciandolo sempre in uno stato consistente. Per quanto riguarda la comunicazione, questo modulo utilizza il pattern observer. In particolare l'implementazione dell'osservatore delle periferiche, al quale vengono notificati i dispositivi trovati in fase di scansione, è realizzata dalla classe "BBMDeviceListener". Anche in questo caso il modulo si limita a costruire un opportuno messaggio da inviare al dispatcher per segnalare l'evento agli altri componenti del sistema.

Nel paragrafo successivo verranno trattati i controllori e le principali funzionalità offerte. Infine concluderemo il capitolo discutendo l'esecuzione in background dell'applicazione.

6.7 I controllori dell'applicazione

Questo paragrafo intende riportare una descrizione ad alto livello dei controllori utilizzati nell'applicazione. Poiché si ritiene che un'analisi approfondita potrebbe creare un testo di difficile interpretazione ed estremamente noioso, verranno definiti solamente gli aspetti principali.

In generale tali moduli traducono le richieste dell'utente in messaggi per il modello e viceversa. Di conseguenza sono costituiti prevalentemente da gestori per i vari componenti della vista.

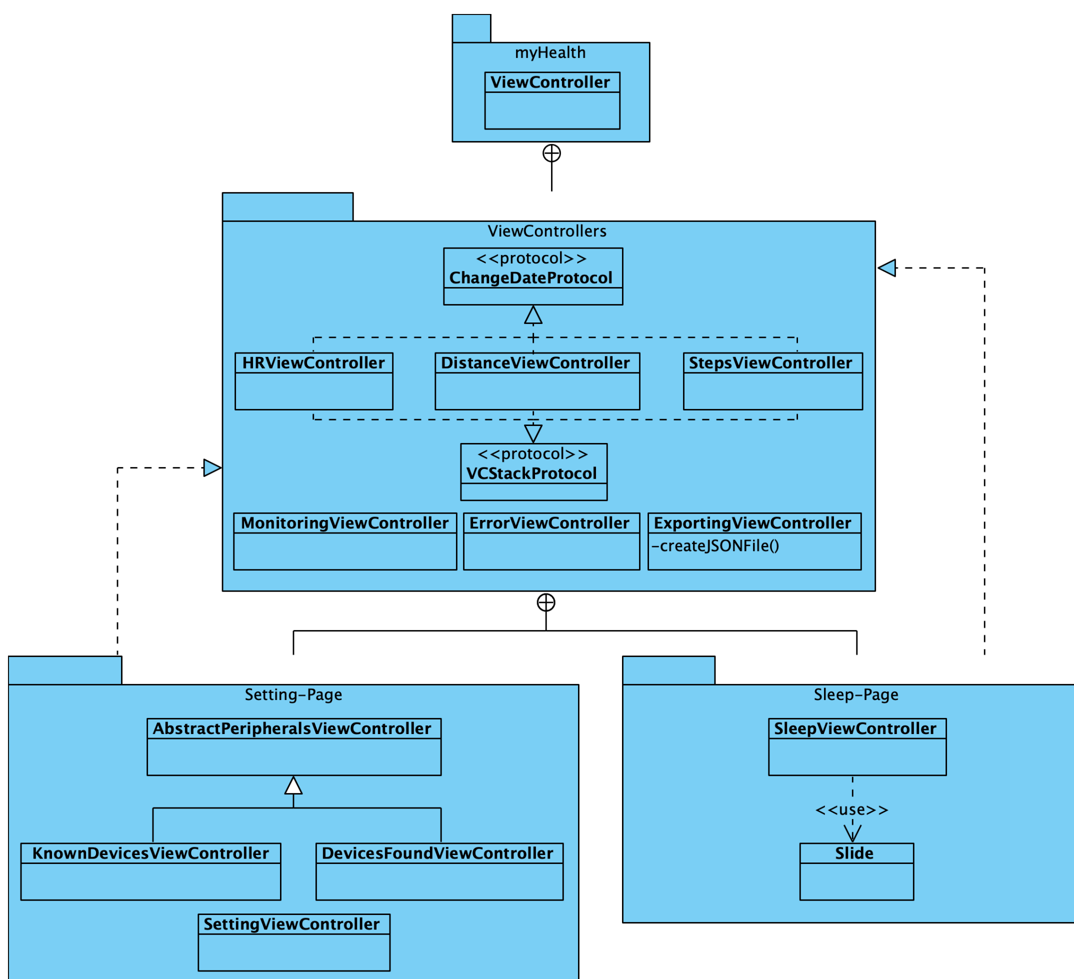


Figura 27 Vista strutturale dei controllori

La figura rappresenta una vista strutturale per mostrare dove sono archiviati i vari moduli. All’avvio dell’applicazione viene caricato e mantenuto per tutta l’esecuzione il “ViewController” principale, contenuto nella root del progetto. In particolare, tale controllore, riceve dal modello eventuali errori. In questi casi viene disposto il modulo “ErrorViewController” in modalità modale, ovvero impedendo l'interazione con qualsiasi altra interfaccia utente fino a quando quest’ultimo non viene chiuso. Oltre a ciò definisce i gestori dei bottoni del menu. Specificatamente, quando un pulsante viene premuto, il “ViewController” carica al suo interno un controllore secondario adibito a gestire la vista richiesta dall’utente. Di conseguenza si viene a creare un’associazione uno ad uno tra i controllori e le interfacce, consentendo di avere un codice più modulare e semplice da leggere. In particolare “HRViewController”, “DistanceViewController” e “StepsViewController” sono

responsabili delle richieste e degli output delle viste relative alla frequenza cardiaca, distanza percorsa ed al numero di passi. Tali controllori sono conformi ai protocolli “ChangeDataProtocol” e “VCStackProtocol”. Il primo definisce l’insieme dei metodi per gestire la richiesta di visualizzazione delle informazioni in una specifica data. Invece il secondo raggruppa le funzionalità per generare una corrispondenza biunivoca tra i dati del modello e la loro raffigurazione. Invece, per quanto riguarda la vista relativa ai dati sul sonno, essa è gestita dal modulo “SleepViewController”. Tale controllore è conforme al protocollo “ChangeDataProtocol” e utilizza la classe “Slide” per mostrare i vari intervalli di tempo in cui l’utente ha riposato nella data selezionata.

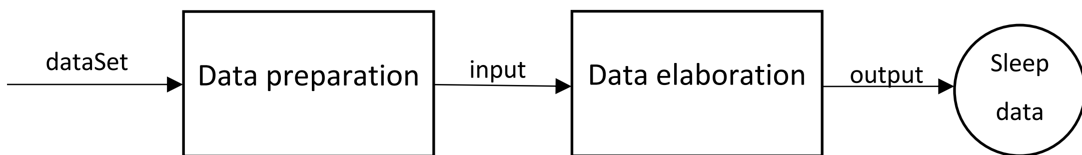


Figura 28 Riassume i passi necessari per interpretare i dati sonno

```
func setupSlideScrollView(slides : [Slide]) {
   //Pulisce la scrollView
   scrollView.subviews.forEach { $0.removeFromSuperview() }
   //Imposta la dimensione del contenuto
   scrollView.contentSize = CGSize(width: scrollView.frame.width *
        CGFloat(slides.count), height: scrollView.frame.height)
   scrollView.isPagingEnabled = true
   //Inserisce le pagine
   for i in 0 ..< slides.count {
     slides[i].frame = CGRect(x: scrollView.frame.width * CGFloat(i),
     y: 0, width: scrollView.frame.width, height:scrollView.frame.height)
     scrollView.addSubview(slides[i])
   }
   //Imposta l'indicatore delle pagine
   pageControl.numberOfPages = slides.count
   pageControl.currentPage = 0
}
```

Codice 6 Metodo adibito ad aggiungere gli intervalli di riposo all’interfaccia utente.

Si fa presente che i dati relativi al sonno non sono direttamente interpretabili, ma devono essere elaborati come riportato nella Figura 28. Per fare ciò, recuperiamo i dati, effettuiamo un passo di preparazione ed infine li elaboriamo. Tale operazione è fornita, mediante una libreria, dalla casa produttrice del dispositivo. In particolare il risultato finale permette di ottenere gli intervalli di riposo, suddivisi per sonno leggero e profondo. Infine questi ultimi verranno utilizzati per creare delle viste personalizzate, le quali saranno aggiunte all’interfaccia utente mediante il Codice 6.

Invece, per quanto riguarda “MonitoringViewController” ed “ExportingViewController”, gestiscono rispettivamente le richieste provenienti dalla vista di monitoraggio ed esportazione dati. Specificatamente quest’ultimo incapsula la logica per trasformare un insieme di dati in un file JSON il cui formato sarà approfondito nel paragrafo successivo. Per quanto concerne la pagina di impostazioni essa viene gestita da due controllori. “SettingViewController” è il gestore primario della vista che si occupa delle richieste dei bottoni principali, come quelle per avviare e terminare la scansione. Oltre ciò esso carica al suo interno un ulteriore controllore di tipo “KnownDevicesViewController” oppure “DevicesFoundViewController”. In modo specifico il primo è adibito a notificare alla vista i dispositivi conosciuti, invece il secondo effettua azioni equivalenti per le periferiche trovate in fase di scansione. Poiché questi due controllori sono molto simili è stato definito un “AbstractPeripheralsViewController” conforme al “VCStackProtocol” che definisce gli aspetti comuni, come ad esempio informare la vista di un nuovo dispositivo remoto. Oltre a ciò esso include il metodo responsabile di informare il modello della richiesta di connessione, invocato quando l’utente preme sulla rappresentazione di una periferica.

6.7.1 Struttura del file JSON generato dall’applicazione

In questo paragrafo verrà descritta la struttura del file JSON creato dall’applicazione. In particolare, come precedentemente descritto, esso viene generato dal modulo “ExportingViewController”.

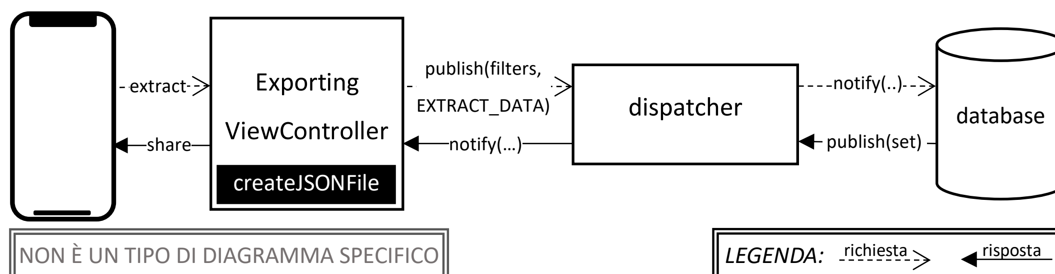


Figura 29 Mostra ad alto livello i componenti coinvolti nell'estrazione dei dati

Quando l’utente richiede di esportare i dati, mediante il controllore, viene inviato un messaggio per il modello indicando i filtri espressi dall’utente al fine di recuperare le

informazioni. Dopodiché il controllore attende di ricevere il sottoinsieme dei dati richiesti.

```
extension Dictionary {
  // dictionary to json
  func toJSON() -> String {
    let invalidJson = "{ \"errore:\" Errore durante la conversione }"
    do {
      let jsonData = try JSONSerialization.data(withJSONObject: self,
      options: .prettyPrinted)

      return String(bytes: jsonData, encoding: String.Encoding.utf8)
      ?? invalidJson
    } catch {
      return invalidJson
    }
  }
}
```

Codice 7 Metodo per generare la stringa JSON da un dizionario

Quando questi ultimi sono stati ricevuti, il modulo crea il dizionario che raggruppa le varie informazioni. Di conseguenza è possibile ottenere la stringa JSON corrispondente utilizzando il metodo riportato nel Codice 7. Si fa presente che tale funzione non è fornita direttamente dalla struttura, ma è stata aggiunta utilizzando il costrutto extension[21].

```
    /* ... */
    let fileURL = dir.appendingPathComponent(file) //Url del file

    do {
       //Scrivo sul file
       try contentFile!.toJSON().write(to: fileURL,
       atomically: false, encoding: .utf8)
    }
    catch {
       //Si è verificato un errore
       dispatcher.publish(object: ["An error occurred!",
       "An error occured during the generation of JSON file."],
       event: .ERROR)
       contentFile=nil //Rimuovo il dizionario con i dati da scrivere
       toggleSpinner() //Nascondo l'indicatore di attività
       return;
    }
    /* ... */
    //Mostro il pop-up per la condivisione
    let activity = UIActivityViewController(
       activityItems: [message, fileURL],
       applicationActivities: nil
    )
    present(activity, animated: true, completion: nil)
    /* ... */
```

Codice 8 Generazione del file JSON e condivisione

[21] Il costrutto extension di Swift permette di aggiungere nuove funzionalità a una classe esistente, struttura, enumerazione o protocollo.

Infine la stringa JSON viene scritta su un nuovo file e presentato all‘utente mediante le varie opzioni di condivisione. In particolare sarà possibile salvare il file sul cloud, inviarlo per email, stamparlo e molto altro.


```
{
  "Device" : "iPhone di Simone",
  "Filters" : ["FrequenzaCardiaca", "VariabilitaRR", "Passo", "DatoSonno" ],
  "DataInizio" : "2019-07-01",
  "DataFine" : "2019-07-01",
  "DatoSonnoCount" : 1207,
  "DatoSonno" : [
    {
      "dataRilevazione" : "2019-07-01 09:59:59 +0000",
      "level" : 99,
      "dataInterval" : 300
    },
    ...
  ],
  "VariabilitaRRCount" : 27129,
  "VariabilitaRR" : [
    {
      "dataRilevazione" : "2019-07-01 10:21:16 +0000",
      "rrInterval" : 0.46000000000000002,
      "offset" : 0.46000000000000002,
      "sensorData" : 2
    },
    ...
  ],
  "FrequenzaCardiacaCount" : 750,
  "FrequenzaCardiaca" : [
    {
      "dataRilevazione" : "2019-07-01 10:19:59 +0000",
      "level" : 66
    },
    ...
  ],
  "PassoCount" : 796,
  "Passo" : [
    {
      "dataRilevazione" : "2019-07-01 10:26:38 +0000",
      "numeroPassi" : 1810,
      "calorie" : 66.065,
      "distanzaPercorsa" : 1370
    },
    ...
  ]
}
```


Figura 30 File JSON generato dall'applicazione

La Figura 30 mostra una parte del file generato che consiste in un oggetto JSON in cui vengono memorizzate le informazioni riportate nella seguente tabella.

Chiave	*Contenuto*
Device	Stringa contenente il nome del dispositivo dal quale sono stati estratti i dati.
Filtri	Array JSON con i tipi di dati estratti.

DataInizio	Stringa contenente la data di inizio intervallo per filtrare i dati.
DataFine	Stringa contenente la data di fine intervallo per filtrare i dati.
DatoSonno, VariabilitaRR, FrequenzaCardiaca, Passo	Array di oggetti JSON corrispondenti ai dati estratti.
DatoSonnoCount, VariabilitaRRCount, FrequenzaCardiacaCount, PassoCount	Intero che determina il numero di oggetti JSON presenti nell'array corrispondente. (Es. DatoSonnoCount determina il numero di elementi presenti nell'array DatoSonno).

Tabella 12 Descrizione chiavi contenute nel file JSON

Alcune delle informazioni potrebbero mancare qualora non siano state richieste al momento dell'estrazione. Ad esempio, se l'utente decide di non estrarre dati di tipo sonno, sia l'array "DatoSonno" che il contatore "DatoSonnoCount" non saranno presenti nel file generato.

Oltre a ciò si riporta il formato dei dati JSON. Specificatamente per gli oggetti relativi ai passi saranno definite le seguenti chiavi: "dataRilevazione", "numeroPassi", "calorie" e "distanzaPercorsa". Invece per quanto riguarda i dati di tipo sonno e frequenza cardiaca abbiamo le chiavi "dataRilevazione" e "level". In aggiunta i dati sul sonno hanno anche il valore "dataInterval". Infine, per quanto concerne la variabilità RR, sono definite le chiavi "dataRilevazione", "rrInterval", "offset", e "sensorData".

Il lettore può ottenere una descrizione approfondita relativa ai dati estratti nel CAPITOLO 7.

Le informazioni, sopra riportate, saranno necessarie al programmatore che dovrà progettare e sviluppare la funzione per recuperare i valori dal file JSON.

6.8 Ciclo di vita di un’applicazione iOS

Prima di vedere gli aspetti relativi all’esecuzione dell’applicazione in background è necessario dare una panoramica generale del ciclo di vita di un’applicazione iOS. È importante determinare lo stato corrente dell’applicazione in quanto specifica cosa può o non può fare in qualsiasi momento.

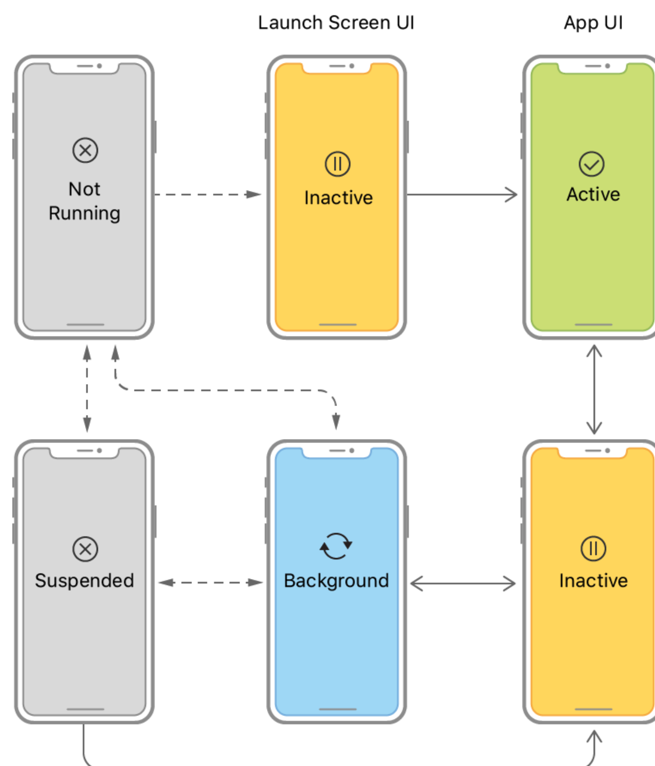


Figura 31 Ciclo di vita di un'app iOS [17]

Come riportato in [18], inizialmente l’applicazione risulta non in esecuzione e rimane in questo stato finché l’utente non avvia il programma. Dopo il lancio il sistema mette l’app in uno stato inattivo o in background, determinato dal fatto che l’interfaccia utente stia per apparire sullo schermo. Quando l’applicazione entra in primo piano[22] lo stato passa automaticamente in attivo, oscillando, negli istanti successivi, tra periodi inattivi, in background ed attivi finché l’applicazione non termina. Ogni qualvolta che lo stato cambia esso viene notificato mediante i metodi dell’oggetto UIApplicationDelegate. Tali funzioni sono utili per eseguire del codice in relazione al nuovo stato. Ad esempio può essere vantaggioso salvare i dati prima di entrare in

[22] Generalmente viene utilizzato il termine in lingua inglese foreground.

background e ripristinare lo stato quando l'app ritorna in primo piano. In particolare l'unico stato non annunciato è quello di sospeso in cui l'applicazione rimane in memoria, ma non esegue alcun codice.

Nel paragrafo successivo verranno trattati gli aspetti relativi all'esecuzione dell'applicazione in background.

6.8.1 Esecuzione dell'applicazione in background

Durante il ciclo di vita, un'applicazione passa nello stato di background per diverse ragioni. Ad esempio lo stato può cambiare a seguito di avvisi di sistema, anche se la causa più comune è scatenata dall'utilizzatore che esce da un'app in primo piano. Quindi quest'ultima passa per un breve periodo in background prima che il sistema la sospenda. In questo lasso di tempo, in cui l'applicazione è in background, tutte le attività avviate devono terminare il prima possibile e cercare di rilasciare il maggior numero di risorse. Se il tempo necessario per fare ciò è superiore a quello fornito è possibile richiedere tempo extra al sistema per completare l'azione. In particolare l'applicazione da realizzare richiede un'esecuzione in background H24, che non può essere garantita mediante le funzionalità base precedentemente descritte. Ciò che sarebbe opportuno fare è realizzare un servizio che fornisca una comunicazione continua con la periferica remota. Sfortunatamente però il sistema iOS non permette di creare dei demoni che eseguono compiti non supervisionati dal sistema. Nonostante ciò qualsiasi utilizzatore dei dispositivi iOS può facilmente verificare che l'esecuzione in background a lungo termine è permessa. Infatti basta aprire l'applicazione "Musica" ed avviare una canzone per rendersi conto che la cosa sia realizzabile. Di conseguenza, cercando tra la documentazione, è stata individuata una modalità background che consente un'esecuzione a lungo termine per un insieme ristretto di funzionalità. Specificatamente è possibile lanciare attività relative alla riproduzione di audio, servizi di localizzazione, voip[23], supporto per

[23] Acronimo di Voice over IP ed indica una tecnologia che rende possibile effettuare una conversazione sfruttando una connessione Internet.

notifiche, aggiornamenti regolari dal server e comunicazione con accessori Bluetooth Low Energy (BLE) o conversione del dispositivo in un accessorio BLE. Dichiarare una di queste attività, per utilizzare la modalità background per un uso illegittimo della specifica funzionalità, comporta il rifiuto dell'app quando quest'ultima viene inviata per essere pubblicata nell'App Store[24]. Fortunatamente, per quanto riguarda l'applicazione da realizzare, è sufficiente sfruttare legittimamente la modalità background per comunicare con accessori BLE.

Tali aspetti verranno affrontati nel paragrafo successivo.

6.8.1.1 Utilizzo di CoreBluetooth in background

Come specificato in [19], molte attività di CoreBluetooth sono disattivate quando l'applicazione non si trova in primo piano. Dichiarando la modalità background per comunicare con accessori BLE, l'applicazione viene riattivata per elaborare determinati eventi relativi al Bluetooth. Come prima cosa è necessario dichiarare tale condizione inserendo nel file "Info.plist"[25] la chiave UIBackgroundMode con valore bluetooth-central, la quale indica che l'applicazione comunica con periferiche mediante tecnologia BLE. In alternativa è possibile settare questa impostazione utilizzando l'interfaccia utente fornita da Xcode. In questo modo il sistema riattiva l'applicazione quando i metodi del CBCentralManagerDelegate e del CBPeripheralDelegate vengono invocati. Ciò consente di gestire eventi come i cambiamenti di stato della centrale, connessione e disconnessione verso le periferiche, notifiche di valori da parte dei dispositivi remoti e molti altri. Poiché un'applicazione in background non avrà mai la priorità sulle risorse rispetto a quella in primo piano, l'obiettivo del sistema è quello di ridurre al minimo l'utilizzo della radio per migliorare la durata della batteria. Di conseguenza, aumentando l'intervallo di scansione dei pacchetti pubblicitari, la ricerca di periferiche è modificata in modo tale che la scoperta di più dispositivi sia raggruppata in un singolo

[24] L'App Store è uno strumento realizzato da Apple per scaricare e acquistare applicazioni.
[25] Info.plist è un file fornito da Xcode per raccogliere informazioni sull'applicazione.

evento. In particolare, poiché irrilevante per il corretto funzionamento dell'applicazione, abbiamo deciso di fermare la scansione quando l'applicazione entra in background. Gli unici eventi che siamo interessati a trattare sono la connessione verso la periferica e la ricezione di valori. Poiché il sistema mette a disposizione circa 10 secondi, come riportato in [19], l'esecuzione dei compiti di gestione deve terminare il prima possibile.

```
func applicationWillEnterForeground(_ application: UIApplication) {
    /* … */
    (dispatcher as? Dispatcher)?.bgExec = false
    /* … */
}

func applicationDidEnterBackground(_ application: UIApplication) {
    /* … */
    (dispatcher as? Dispatcher)?.bgExec = true
    dispatcher.publish(object: nil, event: .STOP_SCAN)
    /* … */
}
```

Codice 9 Aggiornamento variabile esecuzione in background

Per fare ciò abbiamo dotato il dispatcher, utilizzato per la comunicazione publish-subscribe, di una variabile che determina se l'esecuzione avviene in background oppure in foreground. Tale attributo sarà aggiornato dai metodi menzionati nel paragrafo 6.8.

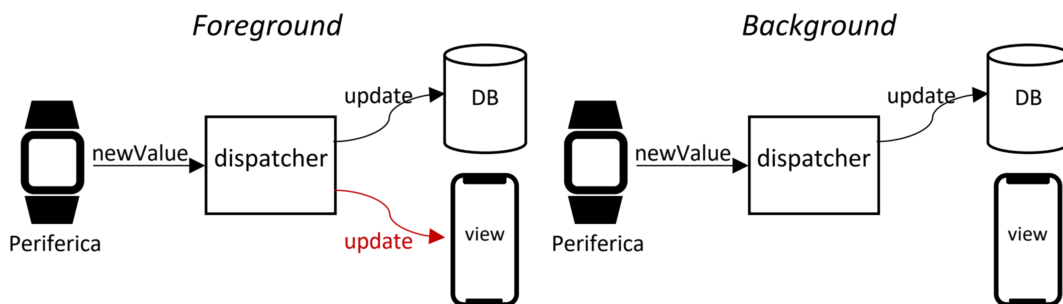


Figura 32 Dispatcher degli eventi in foreground e background

In questo modo ciascun componente può evitare, controllando la proprietà, di propagare messaggi relativi all'aggiornamento della vista quando l'applicazione non risulta in primo piano. Così facendo, nell'istante in cui la periferica invia un nuovo valore, l'attività è ridotta al controllo dell'integrità del dato ed al salvataggio nel database. Conseguentemente al fatto di aver ridotto il carico di esecuzione, non aggiornando la vista, si potrebbe verificare un problema di inconsistenza tra ciò che viene mostrato all'utente ed il reale stato del modello.

Supponiamo infatti che si verifichi il seguente scenario:

1. L’utente sta visualizzando il numero totale di pacchetti ricevuti e sia tale valore pari a 7000.
2. L’applicazione entra in background a seguito di un’azione dell’utente.
3. La periferica invia 10 nuovi valori.
4. L’applicazione controlla l’integrità dei dati e li salva nel database senza inviare aggiornamenti alla vista.
5. L’utente rientra nell’applicazione e visualizza come numero totale di pacchetti 7000 invece di 7010.

Al fine di risolvere problemi derivanti da scenari simili, il view controller principale[26] dichiara di voler essere informato quando i metodi applicationWillEnterForeground e applicationDidEnterBackground vengono invocati.

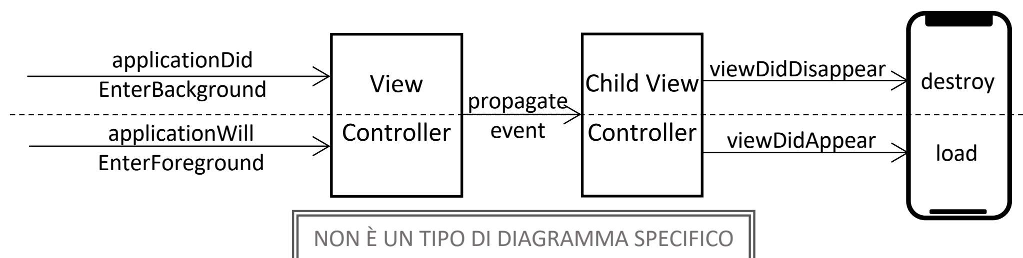


Figura 33 Garantire la consistenza tra la vista ed il modello

In questo modo, tale modulo, invoca le funzioni viewDidAppear e viewDidDisappear del controllore secondario, solitamente chiamate rispettivamente quando la corrispondente vista viene mostrata e nascosta. Così facendo nell’istante in cui l’applicazione entra in background il controllore secondario elimina i componenti grafici che possono essere facilmente ricreati, riducendo in questo modo la quantità di memoria utilizzata. Invece, quando l’app entra in foreground, il controllore ricostruisce gli elementi grafici al fine di mostrare sempre lo stato corrente del modello. D’altra parte è evidente il leggero picco di esecuzione quando l’app passa in primo piano, dovuto alla ricostruzione dell’intera vista corrente. Considerando però il tipo di applicazione tale aspetto può essere tranquillamente trascurato, in

[26] Riferisce al controllore principale sempre presente menzionato nel paragrafo 6.7.

quanto il carico è equivalente a quello generato quando si transita da una vista ad un'altra.

A questo punto non ci resta che predisporre il supporto per la conservazione ed il restauro dello stato al fine di creare un'applicazione in background a lungo termine. In questo modo il sistema salva lo stato del CBCentralManager quando sta per chiudere l'applicazione per liberare memoria. Nel dettaglio lo stato mantiene i servizi per i quali il gestore stava facendo la scansione, le periferiche connesse, quelle alle quali stava tentando di connettersi e le caratteristiche alle quali è stato abbonato il responsabile centrale. Conseguentemente, quando il sistema rilancia l'applicazione in background, lo stato del gestore può essere ripristinato.

In particolare, per dichiarare a CoreBluetooth di conservare lo stato, è necessario, al momento della creazione del gestore centrale, specificare l'opzione CBCentralManagerOptionRestoreIdentifierKey fornendo una stringa di identificazione univoca per il ripristino. Dopodiché, come riportato in [19], è richiesto di fornire l'implementazione del metodo willRestoreState del CBCentralManagerDelegate al fine di sincronizzare lo stato dell'applicazione con lo stato del sistema Bluetooth. Tale funzione, che è la prima ad essere eseguita al riavvio del software, ha come parametro un dizionario che contiene informazioni sul gestore centrale conservato dal sistema al momento della chiusura dell'app.

Ricapitolando abbiamo realizzato un'applicazione che permette un'interazione con CoreBluetooth a lungo termine. Quando l'applicazione passa in background si cerca di occupare meno memoria possibile eliminando tutti gli oggetti facilmente ricostruibili. Inoltre, al fine di creare task veloci durante l'esecuzione in background, ciascun modulo evita di inviare messaggi relativi all'aggiornamento della vista. Quando l'applicazione passa in primo piano opportune funzioni ricostruiranno le interfacce utente considerando lo stato corrente del modello. Oltre a ciò, qualora il sistema debba terminare l'applicazione, mediante il meccanismo della conservazione dello stato di CoreBluetooth, esso provvede a salvare il gestore centrale ed a ripristinarlo quando quest'ultima viene rilanciata in background.

CAPITOLO 7
PROGETTAZIONE E REALIZZAZIONE DATABASE

L'applicazione richiede di memorizzare diversi dati fisiologici permanentemente nel dispositivo. Di conseguenza è stato necessario realizzare un piccolo database locale basato sul framework CoreData. In questo capitolo, come prima cosa, verrà riportata la progettazione concettuale e logica della base di dati. Dopodiché sarà introdotto e descritto il framework utilizzato. In secondo luogo verrà delineata la realizzazione del database ed infine saranno mostrate alcune delle principali query utilizzate.

7.1 Analisi dei dati

Analizzando il driver del Lifesense Band 2 sono stati individuati principalmente quattro tipi di dati che vengono trasmessi all'applicazione. In particolare abbiamo informazioni relative al sonno, ai passi, alla frequenza cardiaca e alla variabilità di quest'ultima. Nel dettaglio i dati sonno forniscono sostanzialmente, oltre alla data di rilevazione, l'intervallo temporale dalla misurazione precedente ed un valore che deve essere interpretato mediante una libreria sviluppata dal produttore dell'orologio. Una collezione di questi dati può fornire un quadro sulle abitudini dell'individuo, monitorando il numero di ore e la qualità del sonno. Contrariamente più informazioni vengono rilevate da un dato di tipo passo in cui, oltre alla data in cui è stata effettuata la rilevazione ed il numero di passi, fornisce valori sul numero di calorie consumate e sulla distanza percorsa. Invece, per quanto riguarda la

frequenza cardiaca, il dispositivo fornisce la data di rilevazione ed il numero di battiti che il cuore compie in un minuto. Infine si ricevono dalla periferica informazioni riguardanti la variabilità della frequenza cardiaca, ovvero l'intervallo temporale che intercorre tra due battiti successivi, anche detta variabilità RR. Specificatamente vengono restituiti valori relativi alla misurazione, alla data di rilevazione, all'offset dell'intervallo RR[27] ed al valore di alcuni sensori per determinare se il dato è buono. Oltre a ciò si richiede di raggruppare i dati all'interno di sessioni che determinano la data di inizio e fine monitoraggio. Si fa notare che non necessariamente la data di rilevazione è strettamente inclusa nel periodo di sessione, in quanto il dispositivo continua ad effettuare misurazioni anche quando è disconnesso. Tali informazioni saranno poi trasmesse non appena viene avviata una nuova sessione di monitoraggio. Per concludere questo paragrafo si ricorda che l'applicazione è interessata a memorizzare le periferiche alle quali è stata effettuata almeno una connessione. In particolare sarebbe opportuno ricordare quali sono le sessioni che memorizzano dati inviati da un determinato dispositivo remoto.
Di seguito vengono riportate tutte le entità individuate con i loro attributi.

7.1.1 Collezione periferiche

La collezione descritta in questo paragrafo memorizza le periferiche che sono state connesse all'applicazione. In particolare, al fine di recuperare una periferica, è necessario mantenere il suo UUID (Universally Unique Identifier).

Attributo	*Descrizione*	*Vincoli*
UUID	Identificativo della periferica.	Primary Key
Nome	Nome della periferica.	Nome is not null

Tabella 13 Attributi delle periferiche derivati dall'analisi

7.1.2 Collezione sessioni

La sessione determina il periodo di inizio e fine monitoraggio. Specificatamente la data inizio deve essere sempre presente, contrariamente quella di fine sessione

[27] Determina la fine dell'intervallo RR rispetto alla data di rilevazione.

viene immessa solamente al termine. In particolare, in ogni istante di tempo, può essere presente una sola sessione con una data di fine non specificata e che rappresenta la sessione corrente.

Attributo	*Descrizione*	*Vincoli*
InizioSessione	Data di inizio sessione.	InizioSessione<=FineSessione && InizioSessione is not null
FineSessione	Data di fine sessione.	

Tabella 14 Attributi delle sessioni derivati dall'analisi

Vincoli aggiuntivi:

1. "Non è possibile avere due sessioni con fineSessione = null"
   $\forall x. (\forall y.\ (x \neq y\ \wedge x.fineSessione = null) \Rightarrow y.fineSessione \neq null)$

7.1.3 Collezione dati sonno

Questa collezione memorizza i pacchetti spediti dal dispositivo e che devono essere interpretati mediante la libreria fornita dal produttore della periferica per ricavare informazioni sulla qualità del sonno.

Attributo	*Descrizione*	*Vincoli*
DataRilevazione	Data in cui è stato ottenuto il valore.	DataRilevazione is not null
Level	Valore ricevuto dalla periferica.	Level is not null
DataInterval	Intervallo in ms[28] dal dato precedente.	DataInterval is not null

Tabella 15 Attributi del sonno derivati dall'analisi

7.1.4 Collezione passi

La collezione memorizza i pacchetti forniti dal pedometro[29]. Include informazioni sul numero di passi, sulla distanza percorsa e sulle calorie bruciate.

Attributo	*Descrizione*	*Vincoli*
DataRilevazione	Data in cui è stato ottenuto il valore.	DataRilevazione is not null

[28] Intervallo temporale espresso in millisecondi.
[29] Il pedometro è un dispositivo usato per contare i passi, anche detto contapassi.

NumeroPassi	Numero di passi effettuati.	NumeroPassi is not null
DistanzaPercorsa	Distanza percorsa in chilometri.	DistanzaPercorsa is not null
Calorie	Calorie bruciate in chilocalorie.	Calorie is not null.

Tabella 16 Attributi dei passi derivati dall'analisi

7.1.5 Collezione frequenze cardiache

Le istanze presenti in questa collezione memorizzano solamente il numero di battiti cardiaci.

Attributo	*Descrizione*	*Vincoli*
DataRilevazione	Data in cui è stato ottenuto il valore.	DataRilevazione is not null
Level	Numero di battiti che il cuore compie in un minuto (bpm).	Level is not null

Tabella 17 Attributi delle frequenze cardiache derivati dall'analisi

7.1.6 Collezione variabilità RR

Questa collezione mantiene i pacchetti relativi alla variabilità della frequenza cardiaca. In particolare viene memorizzata la qualità della misurazione, l'intervallo RR ed il relativo offset.

Attributo	*Descrizione*	*Vincoli*
DataRilevazione	Data in cui è stato ottenuto il valore.	DataRilevazione is not null
RRInterval	Memorizza il valore dell'intervallo RR misurato.	RRInterval is not null
Offset	Offset dalla data di rilevazione, indica la fine dell'intervallo RR.	Offset is not null
SensorData	Valore ricavato dai sensori della periferica che indica se il dato è buono.	SensorData is not null

Tabella 18 Attributi della variabilità RR derivati dall'analisi

7.2 Analisi delle operazioni

Prima di procedere con la progettazione concettuale e logica della base di dati è opportuno definire le principali operazioni che verranno effettuate su di essa.
In particolare, per ciascuna, verrà descritta la funzionalità offerta e la frequenza di utilizzo giornaliera prevista. Quest'ultima sarà espressa con un valore compreso tra 0 ed 1, che indicano rispettivamente frequenza nulla e frequenza massima.
Tali informazioni saranno utili per giustificare determinate scelte di progettazione.

7.2.1 Operazioni periferiche

Le principali operazioni sulle periferiche richiedono di inserire un dispositivo nel momento in cui viene stabilita la prima connessione. Inoltre è necessario ottenere tutte le periferiche memorizzate al fine di mostrarle come dispositivi conosciuti. Oltre a ciò è opportuno recuperare l'ultima periferica connessa all'applicazione. In questo modo, successivamente al primo avvio, è possibile effettuare l'abbinamento all'ultimo dispositivo utilizzato senza l'intervento dell'utente.

Operazione	*Descrizione*	*Frequenza*
Inserimento	Inserisce una nuova periferica nel database.	0.1
Visualizzazione	Recupera tutte le periferiche memorizzate nel database.	0.5
Ultima periferica	Recupera l'ultima periferica connessa all'applicazione.	1

Tabella 19 Operazioni sulla collezione periferiche

7.2.2 Operazioni sessioni

Fondamentalmente è richiesta la possibilità di inserire una sessione quando l'utente avvia il monitoraggio e di modificarla per inserire la data di fine. Inoltre, per eventuali scopi futuri, deve essere possibile determinare il numero di dati ricevuti in una determinata sessione.

Operazione	*Descrizione*	*Frequenza*
Inserimento	Inserisce una nuova sessione nel database.	1

Modifica	Modifica una sessione già presente nel database al fine di definire la data di fine sessione.	1
Numero di dati	Determina il numero di dati fisiologici ricevuti in una determinata sessione.	0 (per scopi futuri)

Tabella 20 Operazioni sulla collezione sessioni

7.2.3 Operazioni sui dati

Questo paragrafo descrive le operazioni comuni a tutti i tipi di dati fisiologici[30].

In particolare esse sono richieste nella fase di visualizzazione ed estrazione dei dati.

Operazione	*Descrizione*	*Frequenza*
Inserimento	Inserisce un nuovo dato nel database.	1
Visualizzazione	Recupera i dati ricevuti in un determinato periodo.	1
Ultimi dati	Recupera gli ultimi dati ricevuti.	0.8
Numero dati	Determina il numero totale di dati.	1
Numero dati periodo	Determina il numero totale di dati ricevuti in un determinato periodo.	1

Tabella 21 Operazioni sulla collezione dati

7.2.3.1 Operazioni sulla frequenza cardiaca

La tabella riporta le operazioni specifiche che devono essere effettuate per i dati relativi alla frequenza cardiaca.

Operazione	*Descrizione*	*Frequenza*
Include le operazioni del paragrafo 7.2.3.		
Massimo/ Minimo/ Media	Calcola il valore massimo/minimo/media delle rilevazioni, eventualmente specificando un determinato periodo.	1
Statistiche HR	Restituisce il numero di oggetti, filtrati per un periodo, che memorizzano lo stesso valore.	1

Tabella 22 Operazioni sui dati relativi alla frequenza cardiaca

[30] Si intende che ciascun tipo di dato specifico deve permettere le operazioni elencate.

7.2.3.2 Operazioni sui passi

In questo paragrafo vengono riportate le operazioni da effettuare sui dati rilevati dal pedometro.

Operazione	*Descrizione*	*Frequenza*
Include le operazioni del paragrafo 7.2.3.		
Numero massimo di passi	Calcola il numero massimo di passi effettuati, eventualmente specificando un determinato periodo.	1
Distanza percorsa	Determina la distanza percorsa, eventualmente specificando un determinato periodo.	1

Tabella 23 Operazioni sui dati rilevati dal pedometro

7.3 Progettazione concettuale

Dall'analisi dei dati è stato derivato il seguente schema concettuale.

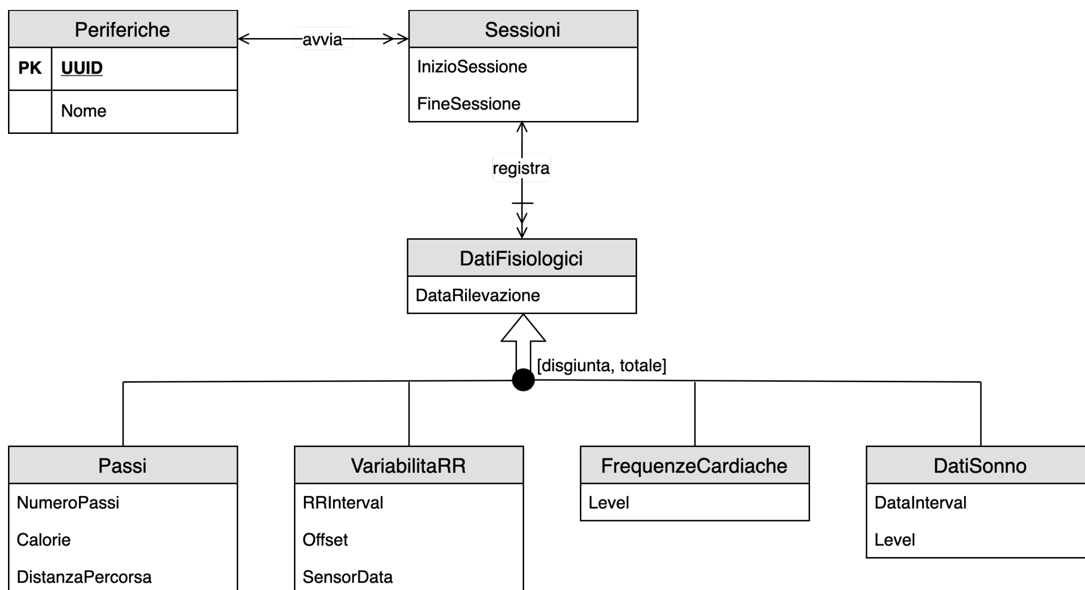


Figura 34 Schema Concettuale

La collezione periferiche mantiene tutti i dispositivi connessi almeno una volta all'applicazione. Quando viene effettuato l'accoppiamento immediatamente il sistema attiva una sessione che l'utente può successivamente interrompere e riavviare. Tutte le volte che viene avviato il monitoraggio si crea una nuova istanza nella collezione sessioni. Di conseguenza abbiamo che una periferica avvia una o più

sessioni ed una sessione è avviata da un'unica periferica. Durante l'esecuzione dell'applicazione esiste al più una sessione attiva nella quale vengono memorizzati i dati. In particolare una sessione memorizza zero o più dati fisiologici ed un dato appartiene ad un'unica sessione. Per quanto concerne la collezione dei dati fisiologici, essa raccoglie gli attributi comuni ai dati specifici. Nel dettaglio le specializzazioni di quest'ultima collezione sono i passi, la variabilità RR, le frequenze cardiache ed i dati sonno. Come si evince dallo schema concettuale la generalizzazione è disgiunta con una copertura totale. Ovvero ogni istanza di dati fisiologici è anche un'occorrenza di una entità figlia.
Lo schema concettuale derivato dalla fase di analisi è l'input della fase successiva: la progettazione logica, a seguito della quale viene determinato lo schema logico. Quest'ultimo rappresenta la base per la realizzazione del database.

7.4 Progettazione logica

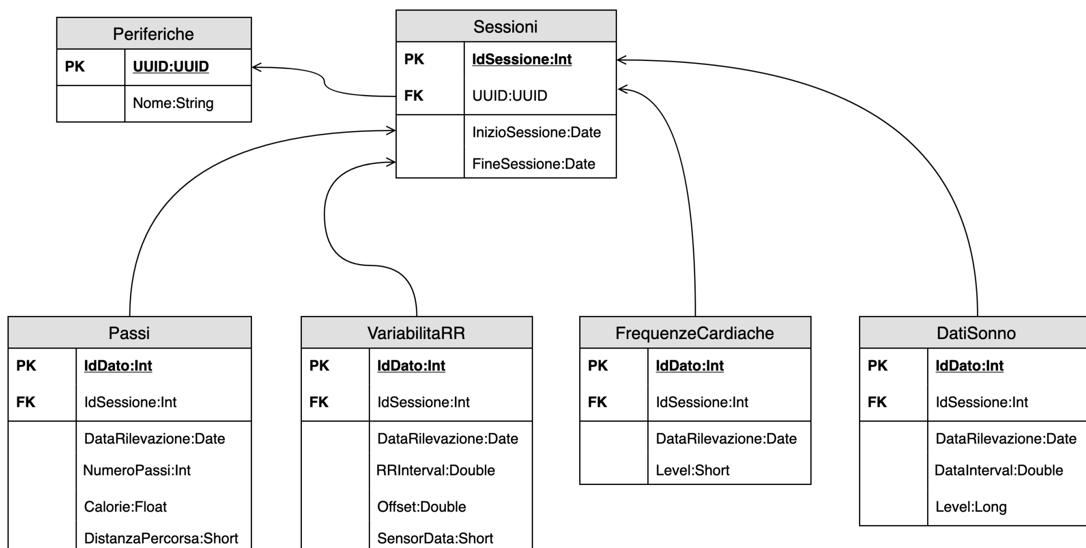


Figura 35 Schema logico

Collezione	*Vincolo*
Periferiche	Nome is not null
Sessioni	UUID is not null, InizioSessione is not null
Sessioni	InizioSessione $\leq$ FineSessione
Sessioni	InizioSessione $\leq$ CurrentDate, FineSessione $\leq$ CurrentDate

Sessioni	$\forall x.(\forall y.\ (x \neq y\ \wedge x.fineSessione = null) \Rightarrow y.fineSessione \neq null)$
Passi	IdSessione is not null, NumeroPassi is not null, Calorie is not null, DistanzaPercorsa is not null
Passi	DataRilevazione ≤ CurrentDate
VariabilitaRR	IdSessione is not null, RRInterval is not null, Offset is not null, SensorData is not null
VariabilitaRR	DataRilevazione ≤ CurrentDate
FrequenzeCardiache	IdSessione is not null, Level is not null
FrequenzeCardiache	DataRilevazione ≤ CurrentDate
DatiSonno	IdSessione is not null, DataInterval not null, Level is not null
DatiSonno	DataRilevazione ≤ CurrentDate
Sessioni	Foreign key(UUID) references Periferiche(UUID) on delete no action on update cascade
Passi	Foreign key(IdSessione) references Sessioni(IdSessione) on delete set null on update cascade
VariabilitaRR	Foreign key(IdSessione) references Sessioni(IdSessione) on delete set null on update cascade
DatiSonno	Foreign key(IdSessione) references Sessioni(IdSessione) on delete set null on update cascade
FrequenzeCardiache	Foreign key(IdSessione) references Sessioni(IdSessione) on delete set null on update cascade
Nello schema le chiavi primarie sono evidenziate in grassetto e sottolineate.	

Tabella 24 Vincoli non esprimibili con lo schema logico

La figura soprastante mostra lo schema logico derivato da quello concettuale. In particolare sono state definite chiavi primarie artificiali per ciascuna collezione, eccetto che per le periferiche in cui è stato utilizzato l’UUID per il ruolo di chiave. Inoltre, considerando che la relazione tra periferiche e sessioni è una a molti, è stata definita un chiave esterna in quest’ultima che punta alle periferiche.

Questa parte di traduzione in schema logico è abbastanza standard in quanto non vi sono valide alternative che richiedono di giustificare le scelte prese. Contrariamente, quando si traduce una generalizzazione, si devono sempre valutare le varie tecniche che si possono applicare al caso. Come prima cosa bisogna decidere se accorpare o separare le entità della generalizzazione. Se si decide di mantenere una separazione allora la traduzione avviene inserendo relazioni "isA" dai figli verso il padre. Contrariamente è opportuno decidere se unificare il padre ai figli o viceversa. Nel nostro caso specifico la generalizzazione è disgiunta con una copertura totale, di conseguenza è ragionevole pensare di fondere i concetti. A sostegno di questa decisione sono anche le operazioni da effettuare sulla base di dati che non evidenziano un particolare interesse per un'entità generale. A questo punto dobbiamo però decidere se unire il padre ai figli oppure viceversa. Considerando che la quantità di informazioni fornite dall'entità generale sono relativamente poche è sembrato opportuno trasferire le proprietà del padre ai figli. Diversamente avremmo creato un'unica entità che avrebbe memorizzato tutti gli attributi delle varie sottoclassi. Di conseguenza, essendo la generalizzazione disgiunta, ciò avrebbe generato istanze con molti attributi nulli, evidenziando così uno spreco di memoria. Scegliendo di accorpare il padre ai dati di tipo specifico si sono create quattro entità distinte. Dopodiché per ciascuna è stata definita una relazione verso l'entità sessione ti tipo uno a molti, con chiave esterna presente nei dati.

7.5 Framework CoreData

Prima di procedere con la descrizione della realizzazione del database è opportuno introdurre e descrivere il framework CoreData. Tali informazioni saranno necessarie per comprendere quanto scritto nei paragrafi successivi.

CoreData permette di salvare i dati permanentemente nel dispositivo fornendo anche la possibilità di disfare e rifare un insieme di operazioni. Lo stack del framework è definito sostanzialmente da tre classi principali: NSManagedObjectModel, NSManagedObjectContext,

NSPersistentStoreCoordinator. Al fine di semplificare la creazione e la gestione di quest'ultimo, esso viene incapsulato in un oggetto della classe NSPersistentContainer. Di seguito verranno introdotti i vari moduli e le loro funzionalità principali.

In primo luogo, come definito in [7], quando scegliamo di utilizzare questo framework dobbiamo definire il modello mediante l'editor di CoreData. Ovvero si devono specificare i tipi di dati, le proprietà e le loro relazioni. Il software permette anche di visualizzare graficamente il risultato finale. A tempo di esecuzione lo schema sarà rappresentato da un'istanza di NSManagedObjectModel, la quale contiene uno o più oggetti di NSEntityDescription che descrivono i dati. Questi ultimi mantengono istanze di NSPropertyDescription che definiscono i campi dell'entità. CoreData utilizza l'oggetto NSManagedObjectModel, fra le altre cose, per validare gli attributi e le relazioni a runtime. Inoltre mantiene una corrispondenza tra gli elementi dell'entità ed i corrispettivi oggetti della classe NSManagedObject, che rappresentano i reali dati memorizzati. Tale modulo agisce come un contenitore generico fornendo un'archiviazione efficiente per le proprietà della NSEntityDescription associata. In particolare è possibile definire una sottoclasse di NSManagedObject per determinare una logica personalizzata quando richiesto. Nonostante ciò, come definito in [20], è esplicitamente raccomandato dal framework non ridefinire una serie di proprietà e metodi. Ciascun NSManagedObject è sempre associato ad un NSManagedObjectContext che rappresenta il contesto in cui l'oggetto esiste. Specificatamente quest'ultimo fornisce metodi per salvare le modifiche e recuperare i dati dal database, creando l'astrazione di un grafo ad oggetti. Quindi risulta essere il responsabile della gestione del ciclo di vita degli oggetti NSManagedObject, poiché mantiene i cambiamenti fino a che non vengono salvati in un archivio persistente.

Un contesto isolato non è completamente funzionale in quanto non può accedere ad un modello se non attraverso il suo coordinatore, rappresentato da un'istanza della classe NSPersistentStoreCoordinator. Tale modulo è anche responsabile di gestire l'accesso all'archivio persistente, serializzando le operazioni.

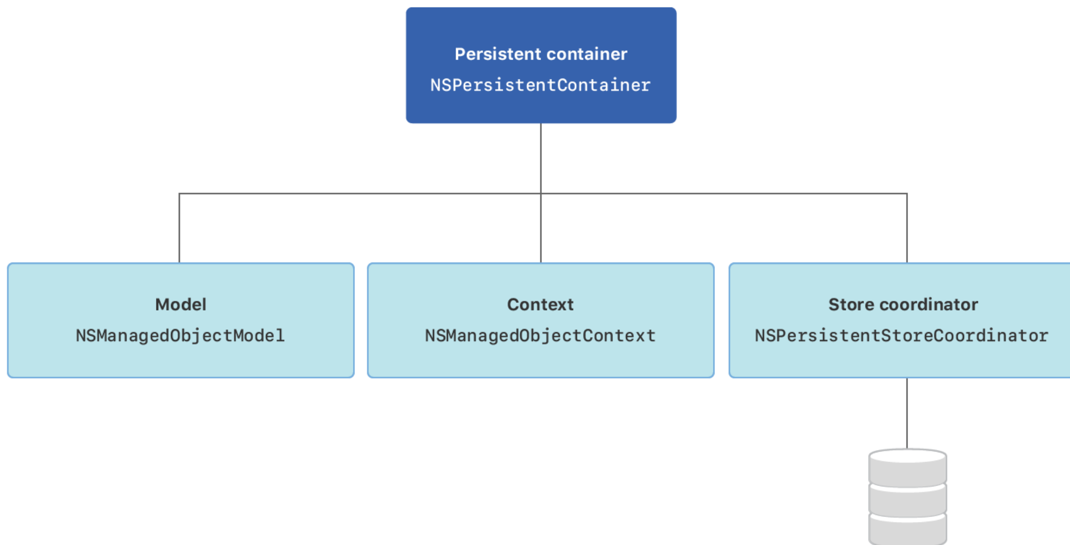


Figura 36 CoreDataStack [21]

Ricapitolando, come descritto in [21], NSManagedObjectModel definisce le entità, le proprietà e le relazioni. I dati sono rappresentati da istanze NSManagedObject e manipolati nei contesti, rappresentati da NSManagedObjectContext, che realizzano l'astrazione di grafo ad oggetti. Per salvare e recuperare i dati i contesti devono comunicare con un NSPersistentStoreCoordinator che gestisce l'accesso al database. Infine NPPersistentContainer incapsula il CoreData stack al fine di semplificarne la creazione e la gestione.

7.5.1 I contesti di CoreData ed esecuzione in background

In questo paragrafo verranno approfondite alcune nozioni sui contesti di CoreData, descrivendo anche le problematiche relative all'uso concorrente delle risorse.

Come precedentemente specificato gli oggetti della classe NSManagedObjectContext permettono di recuperare, modificare, aggiungere ed eliminare dati dal database. I cambiamenti vengono mantenuti in memoria finché esplicitamente il contesto non viene salvato. Nel caso in cui quest'ultima operazione non venga eseguita tutte le modifiche sarebbero perse, poiché non riportate nella base di dati.

Per quanto riguarda la concorrenza i contesti non sono thread-safe, quindi è opportuno adottare delle tecniche quando utilizziamo CoreData in un ambiente multi-threaded. In primo luogo la soluzione più semplice è quella di creare un unico

NSManagedObjectContext, associato alla coda principale, alla quale vengono inviate tutte le attività che richiedono l'interazione con CoreData. Questa soluzione, seppur veloce da realizzare, può portare ad un'app poco reattiva. Infatti, supponendo di dover elaborare una grande quantità di dati, ciò comporta dei ritardi alle risposte degli input dell'utente perché la coda principale è impegnata nelle interazioni con il database. Di conseguenza l'utilizzo degli indicatori di attività[31], che mostrano all'utente lo stato di elaborazione, permettono di realizzare un'interfaccia più amichevole. Questi ultimi verranno poi nascosti una volta terminata l'elaborazione.
Un'altra soluzione a cui possiamo pensare è quella di creare un NSManagedObjectContext per ciascun thread, così facendo ogni flusso di esecuzione utilizzerà il contesto privato risolvendo i problemi di concorrenza. Questa soluzione permette di creare un'app più reattiva in quanto la coda principale risulta libera da task di elaborazione, che possono essere eseguiti da thread secondari. Contrariamente, nella realtà descritta, è facile immaginare che spesso si verificano situazioni in cui due contesti contemporaneamente recuperano e modificano uno stesso dato, apportando cambiamenti che potrebbero essere incoerenti tra di loro. Di conseguenza, come riportato in [22], quando le informazioni vengono salvate le differenze devono essere rilevate e riconciliate mediante la merge policy[32] specificata.

In alternativa è possibile utilizzare una relazione padre-figlio tra i contesti utilizzati in thread diversi. Questo scenario implica la realizzazione di un unico contesto che riporta i cambiamenti nel database. Tutti gli altri contesti saranno discendenti nella catena di parentela. In questo modo, quando un figlio intende salvare le modifiche, queste non vengono riportate nella base di dati, ma inviate al suo contesto padre. Le informazioni verranno così propagate fino al contesto radice il quale le può riportare permanentemente in memoria. Questa tecnica molto semplice può non risultare ottimale in determinate situazioni. In particolare supponiamo di dover ricevere e

[31] Generalmente viene utilizzato il termine in lingua inglese activity indicator. Indica che un compito è in fase di elaborazione.
[32] È la politica da adottare per risolvere i conflitti durante un'operazione di salvataggio. Di default viene restituito un errore.

salvare una grande quantità di dati in un ambiente multi-threaded, in cui ciascun thread ha il proprio contesto figlio. Nell'ipotesi in cui l'applicazione venga terminata dopo aver elaborato buona parte dei dati, ciò comporta la perdita dei vari cambiamenti. Questo perché i vari thread inviavano le modifiche al contesto principale e non alla base di dati. Di conseguenza era necessario un commit da parte del padre per apportare permanentemente i cambiamenti. È possibile sperimentare diverse tecniche al fine di individuare quella più inerente allo scopo dell'applicazione. Di seguito riportiamo un'ultima tecnica, specificata in [23], ed utilizzata nella realizzazione dell'applicazione.

In particolare si creano due contesti: il primo, di tipo mainQueueConcurrencyType[33], associato alla coda principale[34] e legato al ciclo degli eventi dell'applicazione. Di conseguenza verrà utilizzato per le operazioni riguardanti la generazione dell'interfaccia utente. Contrariamente il secondo contesto, di tipo privateQueueConcurrencyType[33], crea e gestisce una coda privata al fine di manipolare in modo sicuro oggetti NSManagedObject. A questo punto è possibile inserire le operazioni nella coda del contesto mediante i metodi perform e performAndWait[35] che risultano essere thread-safe. Mentre perform ritorna immediatamente dopo aver sottoposto l'attività al thread del contesto, performAndWait ritorna il controllo al thread chiamante solamente dopo l'esecuzione del blocco sottoposto. In generale i blocchi vengono eseguiti come lavori distinti, quindi non appena il blocco termina chiunque può accodarne uno nuovo per ripristinare il contesto, annullare le modifiche e così via. Tra le varie tecniche abbiamo scelto quest'ultima in quanto risulta molto semplice da realizzare ed efficace per gestire la concorrenza. In particolare utilizziamo il contesto principale per soddisfare le richieste dell'utente ed il recupero delle informazioni necessarie alla generazione dell'interfaccia. Invece tutte le operazioni di salvataggio dei dati inviati dalla periferica vengono soddisfatte dal contesto privato nascosto all'utente.

---

[33] Determina il tipo di coda che sarà associata al contesto. La coda serve a mantenere le attività che devono essere eseguite.

[34] Si riferisce alla coda associata al thread principale dell'applicazione.

[35] Sono metodi offerti dalla classe NSManagedObjectContext.

Poiché la coda del contesto principale ha una priorità maggiore rispetto a quella del contesto privato, l'utente può continuare ad usufruire di un'applicazione fluida e reattiva senza interruzioni dovute da attività secondarie.

7.6 Esecuzione delle query e recupero dei dati

In questo paragrafo verrà descritta la procedura per recuperare, modificare, cancellare ed inserire oggetti nel database. Quando avviamo l'applicazione per la prima volta avremo un database vuoto, quindi sarà opportuno effettuare operazioni di inserimento per immettere dei dati. Con il linguaggio Swift tale operazione risulta molto semplice e riflette ciò che è stato detto nella parte teorica dei paragrafi precedenti. In particolare, dopo aver ottenuto un contesto, sarà necessario recuperare dallo schema l'istanza NSEntityDescription dell'occorrenza che vogliamo creare. A questo punto è possibile creare un nuovo oggetto NSManagedObject passando come parametro l'istanza precedentemente creata. Durante questa fase è opportuno indicare, come parametro dei costruttori, il contesto nel quale agiamo. Ottenuto l'NSManagedObject possiamo definire i valori che attribuiamo alle proprietà dell'oggetto ed infine salvarlo permanentemente mediante l'appropriata operazione fornita dal contesto[36]. Contrariamente per eliminare un oggetto è opportuno prima recuperarlo e successivamente, mediante il metodo delete, eliminarlo del contesto. Si ricorda che la cancellazione sarà effettiva solamente dopo aver riportato le modifiche nel database.
Invece più interessante è l'operazione di recupero in quanto, utilizzando i predicati, permette di filtrare i dati efficientemente. Per prima cosa è necessario preparare la richiesta, ovvero creare un oggetto di tipo NSFetchRequest. Alla richiesta possono essere agganciati una serie di predicati al fine di filtrare i dati o richiederli ordinati. Definito tale oggetto l'operazione deve essere avviata mediante il metodo fetch fornito dal contesto. Tale funzione ritorna un array di NSManagedObject che soddisfano la richiesta. A questo punto è possibile iterare sugli oggetti ritornati per modificarli, cancellarli oppure visualizzarli.

[36] Il salvataggio avviene mediante il metodo save del contesto.

Di seguito viene mostrato il codice che permette di leggere dal database la periferica con un particolare UUID passato come parametro, se il dispositivo non viene trovato si restituisce il valore nullo.

```
internal func ricercaPeriferica(uuid:UUID) -> Periferica? {
  var periferica:Periferica? = nil
  context.performAndWait {
    //Preparo la richiesta
    let request = NSFetchRequest<NSFetchRequestResult>.init(entityName:
    "Periferica")
    request.predicate = NSPredicate(format: "uuid = %@", uuid as CVarArg)
    request.returnsObjectsAsFaults = false
    request.fetchLimit = 1

    do {
      //Eseguo la richiesta
      let result:[Any] = try self.context.fetch(request)
      switch result.count {
        case 1: //Restituisce la periferica trovata
          periferica = result[0] as? Periferica;
          break;
        default: break;
      }
    } catch let error {
      print("Si è verificato un errore:\n"+error.localizedDescription)
    }
  }
  return periferica;
}
```

Codice 10 Recuperare una periferica da CoreData

7.7 Realizzazione del database dell'applicazione

In questo paragrafo verrà descritta la realizzazione del database utilizzato nell'applicazione. Come prima cosa riporteremo l'organizzazione dei file, con una breve descrizione, così da facilitare la comprensione ai programmatori che si troveranno a confrontarsi con il codice. In secondo luogo delineeremo la creazione del modello, definendo le varie scelte effettuate in relazione alla gerarchia delle classi e delle entità.

Dopodiché mostreremo i contesti impiegati ed i thread che li utilizzano. Infine riporteremo alcune delle principali query utilizzate nell'applicazione con i codici che le realizzano.

7.7.1 Organizzazione dei file

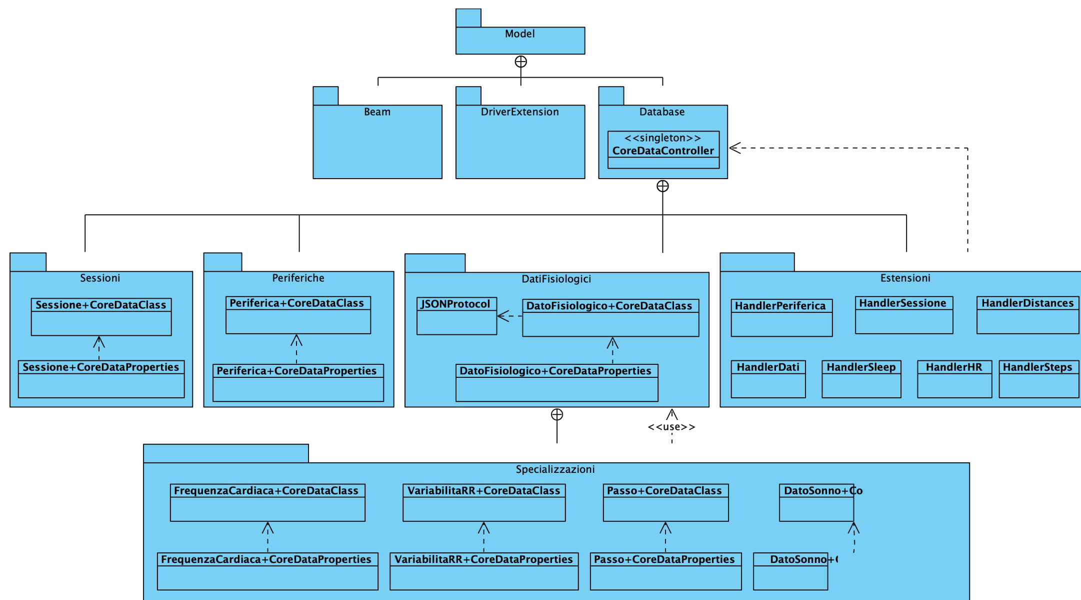


Figura 37 Vista strutturale (decomposizione ed uso) del pacchetto Database

La figura mostra una vista strutturale al fine di definire le posizioni dei moduli che realizzano il database. In particolare tutti i pacchetti sono contenuti nella directory “Database”, la quale contiene solamente il modulo che realizza il gestore del database. Tutti i componenti esterni comunicheranno con quest’ultimo per accedere ai dati. Per quanto riguarda le funzionalità del gestore queste sono contenute nel pacchetto “Estensioni” in cui i vari file, utilizzando il costrutto extension[37] di Swift, arricchiscono il modulo con metodi che realizzano le query. Per quanto riguarda le classi di CoreData utilizzate per creare gli oggetti che rappresentano le sessioni e le periferiche, esse sono archiviate in omonime cartelle. Invece il pacchetto “DatiFisiologici” contiene una cartella interna “Specializzazioni” che va a definire i moduli per i vari tipi di dati. Questi ultimi sono specializzazioni della classe “DatoFisiologico” la quale implementa il protocollo “JSONProtocol” e raggruppa le caratteristiche comuni dei vari dati. In questo modo qualsiasi oggetto memorizzato nel database dovrà fornire la possibilità di ottenere la sua rappresentazione JSON. Questa funzionalità servirà a facilitare l’esportazione dei dati e sarà dettagliata nel paragrafo 7.7.3. Si anticipa al lettore che in tale sezione verrà introdotta anche la

[37] Si ricorda che il costrutto extension permette di aggiungere metodi ad una classe già esistente.

separazione tra gerarchia di classi e di entità, utilizzata per aggirare un aspetto implementativo del framework CoreData.

7.7.2 Data Model dell'applicazione

Prima di descrivere il modello dei dati è stato indispensabile capire la differenza tra CoreData e SQLite. Fondamentalmente SQLite[38] è un motore di database relazionale mentre CoreData è un gestore di grafi ad oggetti, ovvero una collezione di istanze interconnesse tra di loro fornendo la possibilità di effettuare ricerche in modo efficiente. La relazione che lega i due elementi si basa sul fatto che CoreData utilizza SQLite per il suo archivio persistente. Questa nozione è stata fondamentale per comprendere se descrivere i dati in termini di collezioni, ovvero parlare di tabelle, oppure se trattarli come semplici oggetti legati tra di loro. Alla fine siamo giunti alla conclusione di dover descrivere gli elementi in termini di singole istanze. Questo perché sarebbe concettualmente sbagliato dichiarare a CoreData il formato di determinate tabelle quando il framework opera in termini di grafo di oggetti. Il fatto che CoreData utilizza SQLite deve essere considerato, perché lo è, un dettaglio implementativo del framework che non deve suscitare particolare interesse all'utilizzatore. Quindi è opportuno comunicare con CoreData in termini dell'astrazione che fornisce. Per questo motivo in fase di realizzazione non abbiamo ragionato in termini di tabelle, ma di classi. Ad esempio nella progettazione logica abbiamo individuato l'entità periferiche, ciò avrebbe comportato la creazione di una tabella "Periferiche" in un database relazionale. Invece quello che è stato fatto è realizzare la classe "Periferica" che permette di creare gli oggetti nel grafo gestito da CoreData.

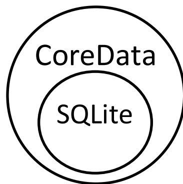


Figura 38 Relazione tra CoreData e SQLite

Prima di procedere con la descrizione del modello è importante fornire al lettore i vincoli imposti da questo framework. Innanzitutto CoreData stabilisce e gestisce automaticamente le chiavi primarie, quindi nella definizione delle classi non andremo a stabilire l'identificativo in quanto verrà assegnato automaticamente. In

[38] SQLite Consortium, SQLite.org [31]

fase di stesura del codice, dato un oggetto del grafo, che ricordiamo essere sempre un'istanza di NSManagedObject, è sempre possibile ottenere la sua chiave mediante il metodo objectID. L'altro vincolo riguarda le relazioni, ovvero se abbiamo un legame tra l'entità A e l'entità B è sempre opportuno descrivere anche la relazione inversa, cioè quella che va da B ad A. Come riportato in [24], le relazioni inverse vengono utilizzate da CoreData per assicurare la coerenza del grafo ad oggetti quando vengono apportate modifiche. A questo punto sono stati introdotti tutti i concetti necessari a comprendere la realizzazione del modello dei dati. Quindi nel progetto è stato creato un file di tipo data model in cui abbiamo riportato lo schema logico derivato nel paragrafo 7.4, ovvero abbiamo creato sei entità: "Periferica", "Sessione", "DatoSonno", "FrequenzaCardiaca", "VariabilitaRR" e "Passo". Nell'entità "Periferica" abbiamo dichiarato i due attributi richiesti: "uuid" e "nome", entrambi non opzionali. Inoltre è stata dichiarata una relazione uno a molti verso l'entità "Sessione". Di conseguenza è stata specificata la relazione inversa di tipo uno ad uno. Ciascuna sessione ha gli attributi di data inizio e fine, in cui quest'ultima può essere eventualmente nulla se la sessione è aperta. Infine si è creato un legame verso ciascuna entità di dato specifico di tipo uno a molti ed uno ad uno non opzionale per quanto riguarda quella inversa. Per quanto concerne i dati specifici tutte le entità possiedono l'attributo non opzionale "dataRilevazione", ereditato dal padre in fase di progettazione logica. In merito alle proprietà specifiche abbiamo che "Passo" determina gli attributi "numeroPassi", "distanzaPercorsa" e "calorie". Invece "VariabilitaRR" dichiara le proprietà "sensorData", "offset" e "rrInterval" con il significato riportato nei paragrafi precedenti. A proposito degli oggetti "DatoSonno" memorizzano l'attributo "level" e "dataInterval", invece "FrequenzaCardiaca" possiede solamente "level". Tutte le specifiche proprietà di queste entità sono non opzionali. Per quanto riguarda la gestione delle relazioni, in caso di cancellazione di istanze, vengono settate con il valore nullo poiché si preferisce non perdere dati. In particolare, questo dettaglio, non è rilevante per l'applicazione realizzata perché si prevede solo di aggiungere dati e mai eliminarli. Comunque sia è stato un aspetto da

considerare in quanto nuove funzionalità potrebbero ampliare la base di dati ed includere operazioni di cancellazione.

Oltre a ciò ricordiamo che le chiavi primarie e secondarie non sono state stabilite visto che sono gestite automaticamente da CoreData. Per quanto riguarda il vincolo di unicità della sessione aperta, esso viene garantito dall'applicazione che, prima di inserire una nuova istanza, provvede a chiudere quella eventualmente aperta.

Infine, tra le funzionalità di Xcode, è possibile visualizzare il risultato dalle entità dichiarate. Ciò permette di verificare velocemente che non si siano commessi errori nella descrizione dei dati.

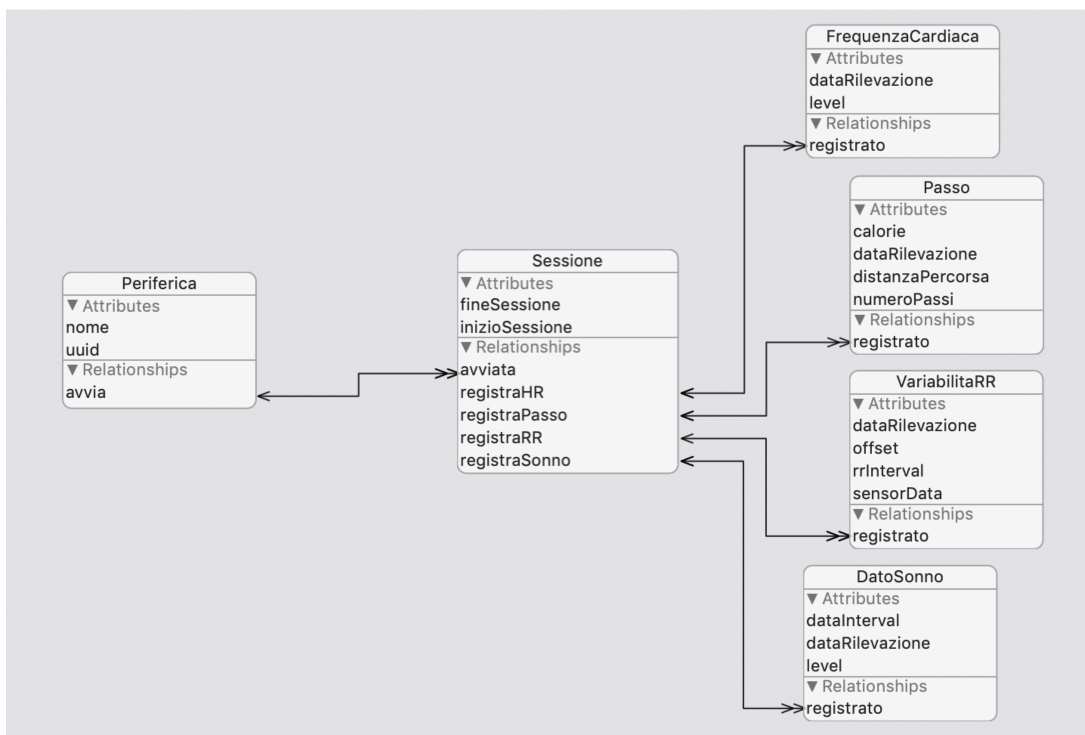


Figura 39 Modello dei dati in CoreData

Una volta definito il modello, Xcode permette di generare automaticamente le classi associate alle entità. Per ciascuna di esse vengono generati due file: "CoreDataClass" e "CoreDataProperties". Il primo definisce la classe nella quale è possibile aggiungere proprietà calcolate. Invece, per quanto riguarda il secondo file, determina un'estensione della classe in cui vengono definiti gli attributi e le relazioni specificate nel modello. Tali aspetti saranno trattati in maniera più approfondita nel paragrafo successivo.

7.7.3 Gerarchia di classi e gerarchia di entità

Gli strumenti di Xcode permettono di creare nel modello delle gerarchie di entità, le quali apparentemente sembrano molto vantaggiose. Ad esempio supponiamo che, in fase di progettazione logica, invece di accorpare l'entità padre ai figli avessimo optato per una soluzione che mantenesse i vari concetti separati. Di conseguenza, molto probabilmente, avremmo utilizzato la gerarchia fornita da CoreData per esprimere questa situazione derivando uno schema come riportato in figura.

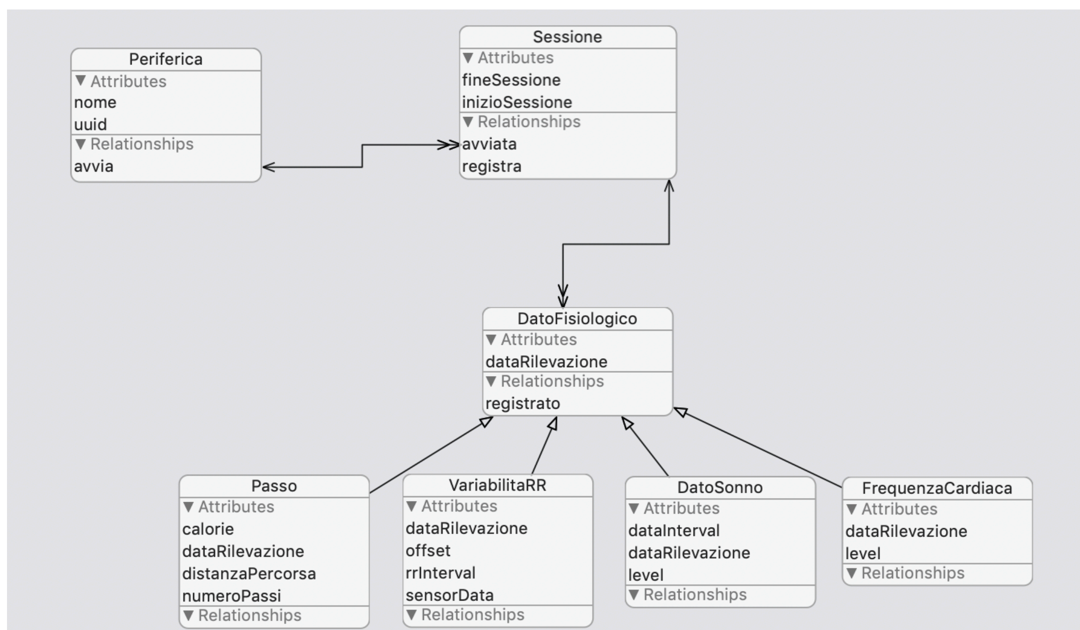


Figura 40 Modello dei dati alternativo

A questo punto, provando ad eseguire una richiesta su un qualsiasi dato specifico e visualizzando la query generata da CoreData per SQLite, notiamo che quest'ultima viene espressa in termini della tabella dei dati fisiologici.

```
CoreData: sql: INSERT INTO ZDATOFISIOLOGICO(Z_PK, Z_ENT, Z_OPT,
ZREGISTRATO, ZDATARILEVAZIONE, ZCALORIE, ZDISTANZAPERCORSA, ZNUMEROPASSI)
VALUES(?, ?, ?, ?, ?, ?, ?, ?)
```

Codice 11 Query generata da CoreData con entità DatoFisiologico

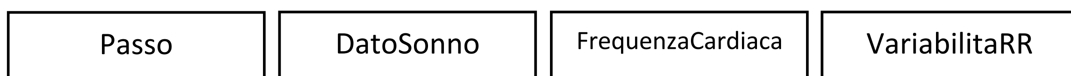
Infatti, come riportato in [25], tutte le entità che ereditano un'altra entità esistono all'interno della stessa tabella SQLite[39]. Questa scelta implementativa di CoreData può creare un problema di prestazioni, dovuto alla quantità di attributi con valore

[39] Ad esempio se abbiamo 100 specializzazioni di un'entità vuota, ciascuna con 10 attributi, CoreData creerà un'unica tabella di 1000 attributi.

nullo. Nonostante ciò la traduzione a schema logico che abbiamo proposto risulta più efficiente in quanto crea una tabella per ciascun dato specifico.

D'altro canto, nonostante non esista un'entità generica che riferisca tutti i dati, a livello implementativo potrebbe essere utile una classe "DatoFisiologico" che raggruppa gli aspetti comuni dei dati specifici. Poiché la gerarchia delle entità è indipendente da quella delle classi abbiamo immaginato di dichiarare a CoreData l'entità generica creando i due file "CoreDataClass" e "CoreDataProperties". Nel primo abbiamo definito "DatoFisiologico" sottoclasse di NSManagedObject e nel secondo file abbiamo raggruppato le proprietà comuni a tutti i dati specifici. A questo punto abbiamo dichiarato ciascun dato specifico sottoclasse di "DatoFisiologico" invece che di NSManagedObject. Di conseguenza quello che abbiamo realizzato è che a livello di classi è possibile manipolare un oggetto generico che rappresenta un dato, ma quando comunichiamo con il database è opportuno specificare sempre a quale entità appartiene. Ovvero se cerchiamo di comunicare a SQLite l'entità "DatoFisiologico" verrà sollevato un errore poiché non esiste nessuna tabella corrispondente. Di seguito si riporta una figura che riassume la gerarchia proposta.

Gerarchia delle entità



Gerarchia delle classi

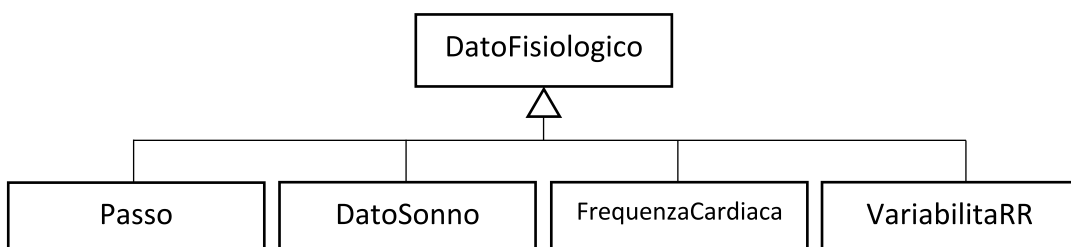


Figura 41 Gerarchia delle classi e delle entità

Questa relazione ci ha permesso di definire molto semplicemente gli aspetti comuni a tutti i dati. In particolare abbiamo reso la classe "DatoFisiologico" coerente con il protocollo JSON, che richiede di restituire le informazioni con un dizionario, influenzando cosi tutte le sue sottoclassi. Ottenuta tale struttura è possibile recuperare l'oggetto JSON corrispondente mediante il metodo toJSON. Come

riportato nel paragrafo 6.7.1, tale metodo non è fornito direttamente come operazione base ma è stato creato estendendo le funzionalità dei dizionari.

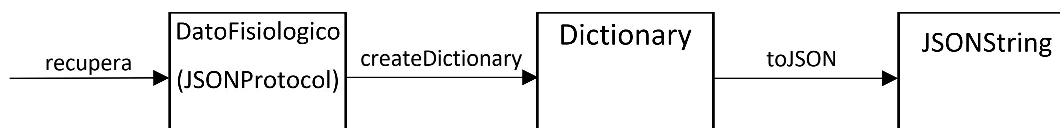


Figura 42 Passi per ottenere l'oggetto JSON di un dato

Questo meccanismo ha permesso, in modo molto semplice, di estrarre i dati nel formato richiesto. Questo perché, una volta recuperate le informazioni, è stato sufficiente chiamare il metodo per generare la stringa coerente con lo standard.
Con questo paragrafo abbiamo terminato la realizzazione del database. A questo punto è necessario un gestore che si faccia carico ed esegua le richieste provenienti dagli altri moduli. Tale componente verrà descritto nella sezione successiva.

7.7.4 Gestore del database e contesti

In questo paragrafo verrà descritto il gestore del database che fornisce i mezzi per interagire con CoreData. Tale componente racchiude tutta la logica al fine di fornire metodi per la manipolazione dei dati. Poiché nell'applicazione il gestore del database è unico, la classe che lo implementa è conforme al pattern singleton[40]. Ovvero il costruttore è privato ed una variabile di classe privata si occupa della creazione dell'istanza che viene restituita mediante il metodo getSingleton.

```
class CoreDataController: SubscriberProtocol, PublisherProtocol {
    /// Istanza del singleton
    private static let singleton = CoreDataController();
    ...
    private init() {
        ...
    }
    ...
    static func getSingleton() -> CoreDataController {
        return singleton;
    }
    ...
}
```

Codice 12 CoreDataController è conforme al pattern singleton

[40] Il pattern singleton ha lo scopo di garantire che di una classe sia creata una ed una sola istanza.

Come si può notare dal codice la classe è conforme anche ai protocolli "SubscriberProtocol" e "PublisherProtocol", riguardanti la comunicazione verso gli altri moduli.

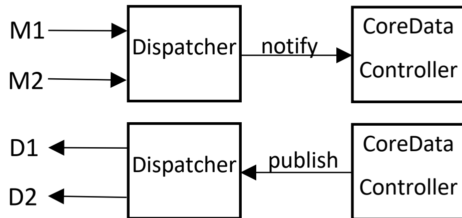


Figura 43 Ruoli del gestore del database

In particolare CoreDataController agirà come mittente quando dovrà inviare il risultato di una query e come destinatario per ottenere le richieste dalle altre componenti. Di seguito verrà riportata l'architettura complessiva del gestore del database. Successivamente verrà descritta facendo particolare attenzione all'utilizzo dei contesti di CoreData ed alla concorrenza.

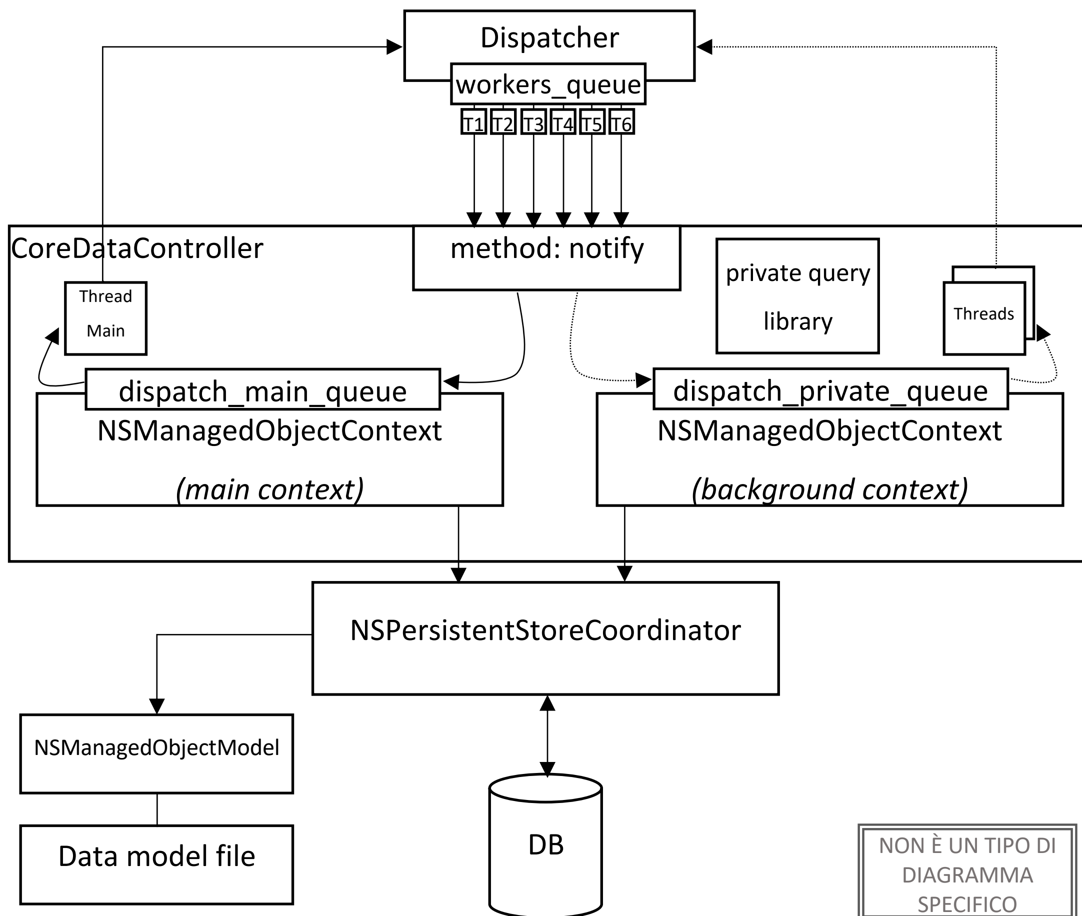


Figura 44 Architettura del gestore del database (CoreDataController)

Nella figura vengono mostrati i componenti principali che compongono il gestore del database. In particolare, cosi come tutti gli altri moduli, alla creazione indica al

dispatcher quali messaggi è interessato a ricevere. Tutte le volte che un determinato evento si verifica, il dispatcher delega un worker[41] ad eseguire le operazioni del CoreDataController. Di conseguenza, in un determinato istante, potrebbero esistere più thread che invocano funzioni del gestore, quindi è opportuno garantire un accesso concorrente. Poiché gli oggetti della classe NSManagerObjectContext non sono thread-safe, abbiamo adottato una delle tecniche presentate nel paragrafo 7.5.1 al fine di gestire la concorrenza. In particolare vengono creati due contesti. Quello principale, sul quale vengono eseguite le query in risposta alle azioni dell'utente ed il contesto in background, sul quale vengono eseguiti le attività che richiedono tempi più lunghi e che risultano trasparenti all'utilizzatore dell'applicazione. Di conseguenza, quando un thread esegue le azioni del gestore, avrà il compito di predisporre gli input e poi chiamare il corrispondente metodo della libreria privata che fornisce la query richiesta. L'utilizzo di metodi privati assicurano che la comunicazione avvenga mediante lo scambio di messaggi, senza la possibilità di eseguire direttamente le richieste avendo un riferimento al gestore del database. Tutti i metodi che eseguono le query sono racchiusi in blocchi e sottomessi, mediante le operazioni perform e performAndWait, nelle code dei contesti. Ciò garantisce che le attività siano eseguite dai thread specifici del contesto, risultando cosi sicure rispetto all'accesso concorrente. In particolare viene utilizzato il metodo perform quando il thread non deve eseguire nessuna istruzione dopo la query. In questo modo il worker[41] termina preventivamente il suo task per essere riutilizzato. Contrariamente, se dopo l'interazione con il database si devono eseguire ulteriori comandi, è necessario utilizzare il metodo performAndWait che ritorna il controllo al thread dopo aver eseguito la query. Quest'ultimo è utile quando bisogna comunicare il risultato mediante un messaggio inviato al dispatcher, che provvederà ad informare tutti gli abbonati. A questo punto ci resta solo da capire quale contesto utilizzare per soddisfare le richieste. In particolare l'inserimento di tutti i dati fisiologici avviene sul contesto in background, in quanto risultano essere eventi

[41] Termine inglese generalmente utilizzato per indicare un thread che esegue del lavoro in parallelo rispetto al thread principale.

secondari nascosti all'utente. Inoltre anche le attività che richiedono molto tempo per essere eseguite, come ad esempio l'estrazione dei dati, vengono sottoposte su tale contesto. Contrariamente per le query necessarie al recupero dei dati, al fine di generare l'interfaccia, o per le richieste provenienti dall'utente si utilizza il contesto che schedula i compiti per il thread principale. In particolare la coda utilizzata da quest'ultimo ha una priorità maggiore rispetto a quella utilizzata dal contesto in background. Questo permette all'applicazione di essere più reattiva rispondendo prontamente all'utente senza dover attendere la terminazione di task secondari.

Nel prossimo paragrafo verranno riportate alcune query e la loro implementazione.

7.7.5 Alcune query con la relativa implementazione

Una volta definito il modello ed il gestore per permettere l'accesso al database è necessario creare le query per inserire, modificare, cancellare e recuperare i dati. In questa sezione verranno riportate, a titolo esemplificativo, alcune delle implementazioni con la relativa query generata da CoreData. Per ottenere tale informazione è stato sufficiente abilitare l'output di debug SQL[42].

7.7.5.1 Recupero ultima periferica connessa

All'avvio dell'applicazione la prima cosa che viene fatta è recuperare l'ultima periferica utilizzata, per tentare una connessione automaticamente senza l'intervento dell'utente. Per ottenere tale informazione è stato sufficiente ordinare in modo decrescente, per data inizio, le istanze di sessione e prendere la prima al fine di ottenere l'ultima periferica connessa.

```
internal func ultimaPerifericaConnessa() -> Periferica?
    var periferica:Periferica? = nil
    context.performAndWait { /* contesto principale */
        var sessione:Sessione?
        let request =
        NSFetchRequest<NSFetchRequestResult>.init(entityName: "Sessione")
        request.sortDescriptors = [NSSortDescriptor(key:
        "inizioSessione", ascending: false)]
        request.returnsObjectsAsFaults = false
        request.fetchLimit = 1

        do {
```

[42] È sufficiente passare come argomenti -com.apple.CoreData.SQLDebug 3 e -com.apple.CoreData.Logging.stderr 1

```
            let result:[Any] = try self.context.fetch(request)
            switch result.count {
                case 1:
                    sessione = result[0] as? Sessione;
                    periferica = sessione?.avviata ?? nil;
                default: break;
            }
        } catch let error {
            print("Si è verificato un
            errore:\n"+error.localizedDescription)
        }
    }
    return periferica;
}
```

Codice 13 Implementazione query recupero ultima periferica connessa

Come si può notare dal codice la richiesta viene effettuata sul contesto principale in quanto è un evento che deve essere gestito il prima possibile, dato che l'utilizzatore sta attendendo la connessione.

Infine riportiamo la query generata da CoreData per recuperare la periferica.

```
CoreData: sql: SELECT 0, t0.Z_PK, t0.Z_OPT, t0.ZFINESESSIONE,
t0.ZINIZIOSESSIONE, t0.ZAVVIATA FROM ZSESSIONE t0 ORDER BY
t0.ZINIZIOSESSIONE DESC LIMIT 1
```

Codice 14 Query generata da CoreData per recuperare l'ultima periferica connessa

Il lettore con competenze sul linguaggio SQL può facilmente verificare che la query sta facendo quello che è stato scritto all'inizio del paragrafo. Inoltre si nota chiaramente che CoreData utilizza una convenzione sul nome degli attributi e delle tabelle: tutti i termini in maiuscoli con lettera iniziale Z[43].

7.7.5.2 Inserimento nuovo passo

Quando l'applicazione riceve un nuovo dato dall'orologio esso viene inserito nel database. Tutta l'operazione avviene in background senza che l'utente se ne accorga. A titolo di esempio riportiamo l'implementazione del metodo adibito all'inserimento di un nuovo passo. Tutti gli altri dati saranno inseriti analogamente.

```
internal func aggiungiPassi(dataSet:[NSMutableDictionary]) {
    func aggiungiPasso(numeroPassi:Int32, calorie: Float,
    distanzaPercorsa:Int16, date:NSDate, sessioneAttiva:Sessione) {
        let entityPasso = NSEntityDescription.entity(forEntityName:
        "Passo", in: self.taskContext)
        let nuovoPasso = Passo.init(entity: entityPasso!, insertInto:
```

[43] Poiché la Z è una delle lettere meno utilizzate si può supporre che CoreData la utilizzi per ridurre la possibilità di conflitti con i nomi. Inoltre in un eventuale ordinamento tutti i termini sarebbero mostrati come ultimi della lista.

```
        self.taskContext)

        nuovoPasso.dataRilevazione = date
        nuovoPasso.registrato = sessioneAttiva
        nuovoPasso.numeroPassi = numeroPassi
        nuovoPasso.calorie = calorie
        nuovoPasso.distanzaPercorsa = distanzaPercorsa
        nuovoPasso.registrato = sessioneAttiva
        sessioneAttiva.addToRegistra(nuovoPasso)
    }
    taskContext.perform { /* contesto privato in bg */
        guard let sessioneAttiva = self.ultimaSessioneAperta() else {
            print("Impossibile memorizzare i dati")
            return
        }
        for pedometer in dataSet {
            aggiungiPasso(numeroPassi: pedometer.value(forKey: "steps")
            as! Int32, calorie: pedometer.value(forKey: "calories") as!
            Float, distanzaPercorsa: pedometer.value(forKey: "distance")
            as! Int16, date: NSDate(timeIntervalSince1970:
            pedometer.value(forKey: "utc") as!
            TimeInterval).toLocalTime(), sessioneAttiva: sessioneAttiva)
        }
        self.saveContext(self.taskContext)
    }
}
```

Codice 15 Implementazione inserimento dati di tipo passo

Il metodo prende in input un insieme di dati e dichiara una funzione annidata[44] designata ad inserire un passo alla volta. Come è possibile vedere dal codice, viene avviato un task sul contesto in background che, per ciascun dato, istanzia un nuovo passo. Quando tutti le istanze sono state create il contesto riporta le modifiche nel database chiamando il metodo adibito al salvataggio. Di seguito viene riportata la query, compresa di dettagli, eseguita da CoreData per inserire un singolo passo.

```
CoreData: sql: INSERT INTO ZPASSO(Z_PK, Z_ENT, Z_OPT, ZREGISTRATO,
ZCALORIE, ZDATARILEVAZIONE, ZDISTANZAPERCORSA, ZNUMEROPASSI) VALUES(?, ?,
?, ?, ?, ?, ?, ?)
CoreData: details: SQLite bind[0] = (int64)28
CoreData: details: SQLite bind[1] = (int64)3
CoreData: details: SQLite bind[2] = (int64)1
CoreData: details: SQLite bind[3] = (int64)6
CoreData: details: SQLite bind[4] = 0
CoreData: details: SQLite bind[5] = (timestamp)583608735.000000
CoreData: details: SQLite bind[6] = 2600
CoreData: details: SQLite bind[7] = 3671
CoreData: sql: UPDATE OR FAIL ZSESSIONE SET Z_OPT = ?  WHERE Z_PK = ? AND
Z_OPT = ?
CoreData: details: SQLite bind[0] = (int64)2
CoreData: details: SQLite bind[1] = (int64)6
CoreData: details: SQLite bind[2] = (int64)1
```

Codice 16 Query generata da CoreData per inserire un nuovo passo

[44] Le funzioni annidate hanno una visibilità ristretta solo all'interno della funzione di chiusura (funzione esterna).

Dal codice riportato è possibile notare che l’implementazione di CoreData fa utilizzo di query parametriche.

7.7.5.3 Recupero numero massimo di passi

In questa query faremo vedere l’utilizzo delle funzioni di aggregazione, in particolare della funzione di massimo. Lo scopo è quello di recuperare il massimo numero di passi che rispettino un predicato passato come parametro della funzione.

```
internal func _recuperaMaxPassi(predicate:NSPredicate? = nil) -> Int {

    var maxSteps = 0

    context.performAndWait { /* contesto principale */
        // Campo per contare i valore massimo
        let passiExpr = NSExpression(forKeyPath: "numeroPassi")
        let maxExpr = NSExpression(forFunction: "max:", arguments:
        [passiExpr])
        let maxDescr = NSExpressionDescription()
        maxDescr.expression = maxExpr
        maxDescr.name = "maxOfSteps"
        maxDescr.expressionResultType = .integer32AttributeType

        //Preparo la richiesta
        let request = NSFetchRequest<NSFetchRequestResult>(entityName:
        "Passo")
        request.sortDescriptors = [NSSortDescriptor(key: "numeroPassi",
        ascending: false)]
        request.propertiesToFetch = [maxDescr]
        request.resultType = .dictionaryResultType
        request.returnsObjectsAsFaults = false
        request.fetchLimit = 1
        request.predicate = predicate

        //Eseguo la richiesta
        do {
            let result = try context.fetch(request) as! [NSDictionary]
            maxSteps = result.first?.value(forKey: "maxOfSteps") as? Int
            ?? 0
        } catch let error {
            print("Si è verificato un
            errore:\n"+error.localizedDescription)
        }
    }
    return maxSteps
}
```

Codice 17 Implementazione numero massimo di passi

Inoltre, a titolo esemplificativo, si riporta anche la query generata da CoreData.

```
CoreData: sql: SELECT max( t0.ZNUMEROPASSI) FROM ZPASSO t0 ORDER BY
t0.ZNUMEROPASSI DESC LIMIT 1
```

Codice 18 Query generata da CoreData per recuperare il numero massimo di passi

7.7.5.4 Statistiche frequenza cardiaca

Questa query intende mostrare l'utilizzo della clausola "Group by". Lo scopo della richiesta è quello di ottenere, per ciascun valore relativo al battito cardiaco, il numero di volte che è stato rilevato. In particolare saranno considerate solo le misurazioni effettuate nella data passata come parametro.

```
internal func statHR(date:Date) -> [NSDictionary]? {

    var result:[NSDictionary]? = nil

    context.performAndWait { /* contesto principale */
        //Campo per contare i record che hanno lo stesso valore
        let levelExpr = NSExpression(forKeyPath: "level")
        let expr = NSExpression(forFunction: "count:",
            arguments:[levelExpr])
        let descr = NSExpressionDescription()
        descr.expression = expr
        descr.name = "count"
        descr.expressionResultType = .integer32AttributeType

        //Preparo la richiesta
        let request = NSFetchRequest<NSFetchRequestResult>(entityName:
                  "FrequenzaCardiaca")
        request.propertiesToGroupBy = ["level"]
        request.propertiesToFetch = ["level",descr]
        request.resultType = .dictionaryResultType
        request.returnsObjectsAsFaults = false
        request.predicate = self.datePredicate(
            dateFrom: date,
            dateTo: nextDay(date))

        //Eseguo la richiesta
        do {
            result = try context.fetch(request) as? [NSDictionary]
        } catch let error {
            print("Si è verificato un errore:\n" +
                error.localizedDescription)
        }
    }
    return result;
}
```

Codice 19 Implementazione statistiche HR

Inoltre si riporta la corrispondente query generata dal framework CoreData.

```
CoreData: sql: SELECT t0.ZLEVEL, COUNT( t0.ZLEVEL) FROM ZFREQUENZACARDIACA t0 WHERE ( t0.ZDATARILEVAZIONE >= ? AND  t0.ZDATARILEVAZIONE < ?) GROUP BY t0.ZLEVEL
CoreData: details: SQLite bind[0] = (timestamp)587952000.000000
CoreData: details: SQLite bind[1] = (timestamp)588038400.000000
```

Codice 20 Query generata da CoreData per le statistiche HR

CAPITOLO 8
PROGETTAZIONE E REALIZZAZIONE INTERFACCIA UTENTE

In questo capitolo verrà descritto il framework CoreGraphics utilizzato per creare alcuni elementi grafici dell'interfaccia utente. Inoltre sarà raffigurata e descritta la storyboard, la progettazione delle icone e l'organizzazione dei file, definendo per ciascuno il proprio contenuto. Infine verranno riportate delle figure che mostrano la grafica completa dell'applicazione. Purtroppo, per ovvi motivi, non sarà possibile raffigurare in questo documento le varie animazioni realizzate, ma cercheremo di dare al lettore sufficienti dettagli affinché possa immaginarle.

8.1 Framework CoreGraphics

Prima di procedere con la descrizione dell'interfaccia grafica introduciamo i concetti principali, che sono stati utilizzati nell'applicazione, del framework CoreGraphics al fine di aiutare un lettore poco esperto a comprendere il codice e le nozioni scritte nei paragrafi successivi.

Come specificato in [4], CoreGraphics sfrutta la potenza della tecnologia Quartz per eseguire rendering 2D. Quest'ultimo è un motore di disegno bidimensionale accessibile nell'ambiente iOS e può essere usato per usufruire di molte funzionalità, tra le quali il disegno su traccia ed ombreggiatura. Come riportato in [26], Quartz 2D usa il modello del pittore per creare le immagini. Ovvero i disegni sulla pagina possono essere sovrapposti, attraverso ulteriori operazioni di disegno, consentendo

di costruire immagini estremamente sofisticate da un piccolo numero di potenti primitive.

8.1.1 Il contesto grafico

Quando dobbiamo creare un'immagine, con primitive di basso livello, è ovvio cimentarsi in un ambiente di disegno in grado di mantenere lo stato grafico corrente che può essere salvato su uno stack e successivamente ripristinato. Tale elemento prende il nome di contesto grafico ed incapsula le informazioni impiegate da Quartz per disegnare una figura su un dispositivo di output. Lo stato influenza tutte le primitive di disegno utilizzate, al fine di confutare qualsiasi dubbio si veda la seguente figura.

```
//Linea nera
context.setStrokeColor(UIColor.black.cgColor)
context.beginPath()
context.move(to: .zero)
context.addLine(to: .init(x: 50, y: 50))
context.strokePath()

//Linea rossa
context.setStrokeColor(UIColor.red.cgColor)
context.beginPath()
context.move(to: .init(x: 50, y: 50))
context.addLine(to: .init(x: 100, y: 100))
context.strokePath()
```

Codice 21 Utilizzo di CoreGraphics per disegnare una linea

Figura 45 Il contesto grafico è modale

Infatti si può facilmente notare che per tracciare una linea viene usato lo stesso codice, ma lo stato influenza il colore utilizzato per disegnare. In particolare il contesto grafico va a definire, la larghezza della linea, il tipo della linea, la posizione corrente, il colore della linea, il colore di riempimento, l'area di clipping, la trasparenza, la matrice di trasformazione ed altri parametri consultabili in [26].

La maggior parte delle proprietà assumono un ovvio significato anche per il lettore meno esperto, quindi andremo a definire principalmente il funzionamento dell'area di ritaglio e della matrice di trasformazione.

Il clipping è il processo mediante il quale si determina quali parti di un'immagine appartengono al volume di disegno. Tutto ciò che non è incluso in tale area viene buttato. Al fine di mostrare come agisce si riporta l'esempio precedente in cui comunichiamo al contesto grafico una nuova area di clipping.

```
//Clipping
context.clip(to: .init(x: 0, y: 0, width: 50,
height: 50))

//Linea nera
context.setStrokeColor(UIColor.black.cgColor)
context.beginPath()
context.move(to: .zero)
context.addLine(to: .init(x: 50, y: 50))
context.strokePath()

//Linea rossa
context.setStrokeColor(UIColor.red.cgColor)
context.beginPath()
context.move(to: .init(x: 50, y: 50))
context.addLine(to: .init(x: 100, y: 100))
context.strokePath()
```

Codice 22 Utilizzo di CoreGraphics per realizzare il clipping

Figura 46 Risultato del clipping

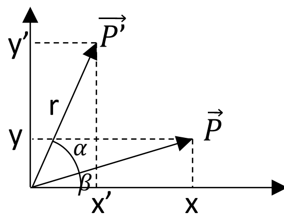
Come si può notare il codice del disegno della linea rossa è ancora presente, ma essendo fuori dal volume di visualizzazione essa non viene disegnata. Il problema del clipping non è cosi banale come può sembrare in quanto, l'algoritmo utilizzato, può influire sulle prestazioni dell'applicazione. Uno tra gli algoritmi più semplici è quello di Cohen-Sutherland [27]. Comunque sia si ritiene sufficiente quanto detto per comprendere ciò che il lettore si troverà a leggere nei paragrafi successivi.

Più interessante, al fine di comprendere maggiormente il contenuto di questo capitolo, è la matrice di trasformazione a cui dedichiamo il prossimo paragrafo.

8.1.1.1 *Matrice di trasformazione e coordinate omogenee*

La matrice di trasformazione è utilizzata per traslare, scalare e ruotare un oggetto che deve essere disegnato dal contesto grafico. In particolare supponiamo di voler traslare un punto $\vec{P}=\begin{bmatrix} x \\ y \end{bmatrix}$ in un piano bidimensionale di una quantità $\vec{T}=\begin{bmatrix} tx \\ ty \end{bmatrix}$, ciò equivale a determinare il punto $\vec{P'}=\vec{P}+\vec{T}=\begin{bmatrix} x+tx \\ y+ty \end{bmatrix}$. Se invece vogliamo scalare lo stesso punto per una quantità $\vec{\sigma}=\begin{bmatrix} \sigma_x \\ \sigma_y \end{bmatrix}$, equivale a moltiplicare a sinistra per la matrice corrispondente, ovvero $\vec{P'}=\begin{bmatrix} \sigma_x & 0 \\ 0 & \sigma_y \end{bmatrix}\vec{P}=\begin{bmatrix} \sigma_x * x \\ \sigma_y * y \end{bmatrix}$.

Infine se intendiamo ruotare, sempre lo stesso punto $\vec{P}$ di una quantità α, corrisponde a svolgere i seguenti calcoli applicando la trigonometria:



$$x' = r * \cos(\alpha + \beta)$$
$$= r * \cos\beta\cos\alpha - r * \sin\beta\sin\alpha$$
$$= x * \cos\alpha - y * \sin\alpha$$
$$y' = r * \sin(\alpha + \beta)$$
$$= r * \cos\beta\sin\alpha + r * \sin\beta\cos\alpha$$
$$= x * \sin\alpha + y * \cos\alpha$$

Ovvero $\vec{P'} = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix} \vec{P}$.

Quindi si vede facilmente che se abbiamo una successione di trasformazioni che scalano o ruotano è sempre possibile ottenere una matrice risultante da applicare a ciascun punto. Invece, se nella catena di trasformazioni compare una traslazione, ciò comporta una scomoda combinazione di prodotti tra matrici e somme di vettori, rendendo il problema computazionalmente significativo. Tale complicanza può essere superata utilizzando le coordinate omogenee in cui la concatenazione di trasformazioni affini si riduce al prodotto di opportune matrici 3x3. In particolare tutti i punti saranno rappresentati da un vettore a tre componenti $\vec{P} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$, la matrice per effettuare la traslazione è la seguente $\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix}$, quella per scalare è $\begin{bmatrix} \sigma_x & 0 & 0 \\ 0 & \sigma_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$ ed infine per ruotare utilizziamo la matrice $\begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$.

Tale studio è stato fondamentale al fine di comprendere come creare la matrice di trasformazione per il contesto grafico utilizzando l'oggetto CGAffineTransform specificato in [28].

8.1.2 Sistemi di coordinate di Quartz 2D

In questo paragrafo affrontiamo il sistema di coordinate di Quartz 2D fondamentale per comprendere come disegnare un componente grafico.

Come definito in [26], il sistema di coordinate permette di determinare la posizione e la dimensione di un oggetto in coordinate dello spazio utente. Tali coordinate devono essere viste come un'astrazione realizzata da Quartz delle coordinate del

dispositivo, poiché gli oggetti disegnati in quest'ultimo spazio non possono essere riprodotti su altri dispositivi senza una distorsione visibile. La componente che permette di passare da uno spazio all'altro è la matrice di trasformazione precedentemente discussa. Ogni punto nello spazio utente è rappresentato da due coordinate (x, y) che rappresentano rispettivamente la posizione sull'asse delle ascisse e sull'asse delle ordinate.

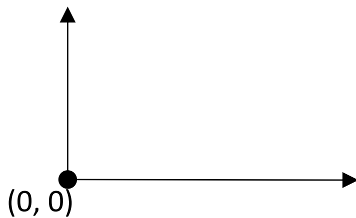


Figura 47 Sistema di coordinate di Quartz

Nel sistema Quartz l'asse x aumenta da sinistra verso destra e quello delle y dal basso verso l'alto con origine degli assi nel punto di coordinate (0, 0) che si trova in basso a sinistra.

Alcune tecnologie utilizzano un sistema di coordinate modificato che sposta l'origine nel punto in alto a sinistra e l'asse delle y aumenta di valori mentre si sposta verso il basso.

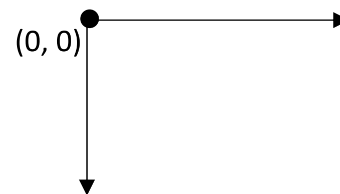


Figura 48 Sistema di coordinate di UIKIt

In particolare il framework UIKit, che fornisce l'infrastruttura per creare l'interfaccia, restituisce i contesti grafici con i sistemi di coordinate modificati.

8.2 Storyboard dell'applicazione

Nei paragrafi precedenti è stato chiarito che cos'è il contesto grafico, come funziona ed a cosa serve. Prima di vedere come è stato utilizzato è opportuno partire dalla descrizione della storyboard dell'applicazione.

La storyboard permette di risparmiare tempo nella creazione delle interfacce grafiche consentendo di progettare più viste di controllo in un unico file. Tutto ciò consente di visualizzare le scene e le varie transizioni. Inoltre semplifica l'utilizzo del layout automatico permettendo di definire equazioni matematiche che determinano la posizione e la dimensione degli elementi. Di conseguenza una scena verrà riprodotta correttamente su dispositivi con dimensioni diverse dello schermo.

Di seguito si riporta una figura che mostra la storyboard complessiva dell'applicazione.

Figura 49 Storyboard dell'applicazione

Sulla sinistra abbiamo il view controller principale che definisce l'intestazione dell'applicazione ed il menu. Inoltre gran parte dell'area è ricoperta da un contenitore utilizzato per caricare all'interno le pagine. In particolare, all'avvio dell'applicazione, viene caricata la vista di monitoraggio. Mediante il menu è poi possibile disporre le schede di visualizzazione dati, esportazione e di impostazioni. Quest'ultima pagina si suddivide a sua volta in due viste: la prima contiene informazioni sulla periferica connessa e su eventuali dispositivi conosciuti in precedenza, la seconda visualizza le periferiche che sono state trovate mediante una ricerca.

Successivamente tutte le schede verranno descritte singolarmente riportando maggiori dettagli.

8.2.1 Elementi UIKit utilizzati nell’applicazione

In questo paragrafo si riporta una descrizione generale delle viste e controlli utilizzati per creare l’interfaccia grafica al fine di produrre, nei paragrafi successivi, una descrizione più leggera e scorrevole senza ripetere ogni volta la funzionalità di ciascun elemento.

Componente UI	***Descrizione***
UIView	Oggetto che gestisce il contenuto per un’area rettangolare sullo schermo.
UILabel	Una vista che visualizza una o più righe di testo di sola lettura.
UIButton	Un controllo che esegue del codice in risposta alle interazioni dell’utente.
UIScrollView	Un oggetto che consente lo scorrimento delle viste in esso contenute.
UIStackView	Un’interfaccia semplificata per predisporre una raccolta di viste in una colonna o in una riga.
UISwitch	Un controllo che permette una scelta binaria.
UIDataPicker	Un controllo utilizzato per immettere una data.
UITabBarController	Un controllo di selezione che permette di decidere quale controllore figlio caricare.
UIGestureRecognizer	Definisce la classe base per il riconoscimento delle gesture.
UIActivityIndicatorView	Mostra all’utente che un task è in esecuzione.
UIPageControl	Un controllo che visualizza una serie orizzontale di punti ciascuno corrispondente ad una pagina.

Tabella 25 Elementi UIKit utilizzati nella storyboard

Tali nozioni sono tutte specificate nella documentazione UIKit [29] nella quale è possibile reperire ulteriori informazioni.

8.3 Organizzazione dei file

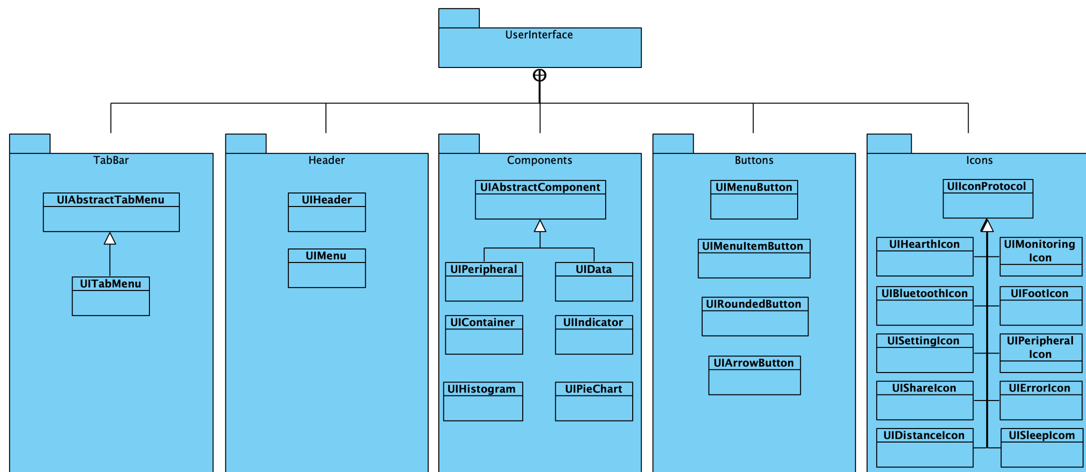


Figura 50 Vista strutturale del pacchetto UserInterface

La figura mostra una vista strutturale del pacchetto "UserInterface". Tutti i nomi dei moduli aderiscono alla convenzione di cominciare con le lettere UI, indicando che la classe definisce una vista personalizzata. Tale pacchetto può essere facilmente prelevato ed importato in altri progetti al fine di riutilizzare le componenti grafiche realizzate. La cartella di minor importanza è quella denominata "Header" che contiene due moduli disgiunti "UIMenu" ed "UIHeader", i quali si limitano a definire delle ombre ed a disegnare sullo sfondo dei quadrati in maniera pseudo-casuale. Il lettore interessato a vedere il risultato grafico può recarsi al paragrafo 8.4. Contrariamente tutti gli altri pacchetti sono di maggiore interesse, per questo motivo dedichiamo un paragrafo per ciascuno di essi.

8.3.1 Pacchetto TabBar

Il modulo "UIAbstractTabMenu" fornisce la logica base per realizzare un menu di tabulazione animato. In particolare recupera tutti i bottoni presenti al suo interno e per ciascun elemento installa un ascoltatore sull'evento di tocco. Quando il bottone viene premuto il modulo non fa altro che portare dietro il pulsante un rettangolo in cui è possibile disegnare. Tale rettangolo viene spostato da un bottone all'altro

utilizzando una funzione di interpolazione lineare. A questo punto è sufficiente definire il metodo che specifica il disegno da mostrare sullo sfondo. In questo modo l'utente, che preme il pulsante, avrà un feedback su quale scheda sta visualizzando.

Figura 51 Mostra il disegno realizzato da UITabMenu

In particolare il modulo "UITabMenu" non fa altro che disegnare un bordo sull'estremità inferiore del rettangolo, come mostrato in Figura 51.

Di seguito si riporta il relativo frammento di codice.

```
//Disegno il sublayer che consiste in un bordo inferiore che si sposta
//in relazione al bottone cliccato
context.setLineWidth(8)

context.setStrokeColor(UIPalette.greenColor.withAlphaComponent(0.65).cgColor)
context.setShadow(offset: CGSize(width: 0, height: -1), blur: 5, color:
        UIPalette.greenColor.withAlphaComponent(0.65).cgColor);
context.beginPath()
context.move(to: .init(x: (self.subLayerButton)!.minX, y:
         (self.subLayerButton)!.maxY))
context.addLine(to: .init(x: (self.subLayerButton)!.maxX, y:
         (self.subLayerButton)!.maxY))
context.strokePath()
```

Codice 23 Frammento di codice che realizza UITabMenu

8.3.2 Pacchetto Buttons

Nell'applicazione vengono utilizzati sostanzialmente quattro bottoni diversi.

Figura 52 Bottone di tipo UIRoundedButton

Quello più semplice è realizzato dal modulo "UIRoundedButton" che utilizza il contesto grafico per definire lo sfondo con bordi arrotondati ed effetto ombra. In particolare tale bottone viene usato per le azioni principali.

Più interessante è la classe "UIMenuButtom" che definisce lo stile e l'animazione per il pulsante del menu.

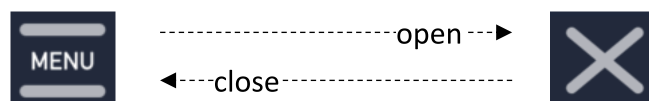


Figura 53 Animazione del bottone menu

Quando il bottone viene premuto le due barre ruotano fino a formare una croce. Cliccando nuovamente viene eseguita l'animazione inversa al fine di ripristinare lo stato iniziale del pulsante.

Per quanto riguarda il modulo “UIMenuItemButton” definisce lo stile dei pulsanti presenti nel menu.

Figura 54 Bottoni navigazione del menu

In particolare, utilizzando il contesto grafico, ciascun bottone viene suddiviso in due aree. La prima area contiene l’icona del bottone che deve essere indicata al momento della creazione insieme alla descrizione ed al titolo che vengono stampati nell’area a fianco.

Figura 55 Bottoni UIArrowButton per selezionata la data

Infine, l’ultimo bottone, “UIArrowButton” disegna al suo interno una freccia ed è utilizzato per cambiare la data selezionata nelle pagine di visualizzazione.

8.3.3 Pacchetto Components

Questo pacchetto contiene componenti grafici di carattere generale per mostrare i dati all’utente. In particolare “UIContainer” è una vista personalizzata che definisce un contenitore con bordi arrotondati ed effetto ombra. Di maggior interesse sono le classi “UIIndicator”, “UIPieChart” ed “UIHistogram”. La prima consente di mostrare graficamente il rapporto tra due valori mediante un arco.

Figura 56 Indicatore realizzato da UIIndicator

```
//Arco che rappresenta il 100% dell'indicatore
//Caratteristiche linea:
context.setLineWidth(gapWidth) //Dimensione gapWidth
context.setLineCap(.round) //Fine linea arrotondata
//Colore linea
context.setStrokeColor(UIColor.black.withAlphaComponent(0.3).cgCo
lor)

//Disegno l'arco totale
context.beginPath()
context.addArc(center: .zero, radius: radius, startAngle: .pi,
endAngle: 0, clockwise: true)
context.strokePath()

//Calcolo l'angolo finale in relazione al valore di parziale e
totale
let endAngle = (1-max(0,min(1,parziale/totale)))*CGFloat.pi
//Caratteristiche linea:
context.setLineWidth(gapWidth-4) //Dimensione gapWidth-4
context.setStrokeColor(UIPalette.greenColor.cgColor) //Colore
linea
context.setShadow(offset: .zero, blur: gapWidth/2, color:
UIPalette.greenColor.cgColor) //Ombra
//Disegno l'arco parziale
context.beginPath()
context.addArc(center: .zero, radius: radius, startAngle: .pi,
endAngle: endAngle, clockwise: true)
context.strokePath()
```

Codice 24 Frammento di codice che realizza UIIndicator

La figura mostra il risultato finale dell'indicatore ed è affiancata dal frammento di codice impiegato per la realizzazione di quest'ultimo.

Invece "UIPieChart" realizza un areogramma richiedendo all'utilizzatore tre vettori paralleli. Il primo determina, in percentuale, la quantità di torta assegnata ad un'etichetta. Invece il secondo ed il terzo determinano le etichette ed i colori da usare per le varie partizioni. Di seguito si riporta l'elemento grafico ed il frammento di codice adibito alla realizzazione del grafico a torta.

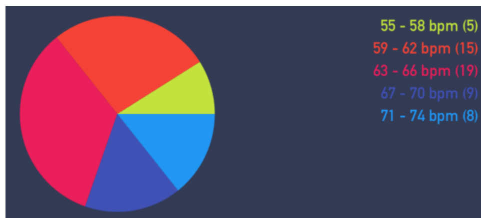


Figura 57 Grafico a torta realizzato da UIPieChart

```
//Disegno il grafico a torta
for (index,value) in portion.enumerated() {
      if(colors.count > index) {
context.setFillColor(colors[index].cgColor)}
      else {
context.setFillColor(UIColor.black.cgColor)}
      context.beginPath()
      context.move(to: .zero)
      context.addArc(center: .zero,
           radius: radius,
           startAngle: sum,
           endAngle: sum+2*CGFloat.pi*value,
                       clockwise: false)
      context.fillPath()
      sum += 2*CGFloat.pi*value
      percentSum += value
      guard percentSum < 1 else {return}
}
```

Codice 25 Frammento di UIPieChart che realizza il grafico a torta

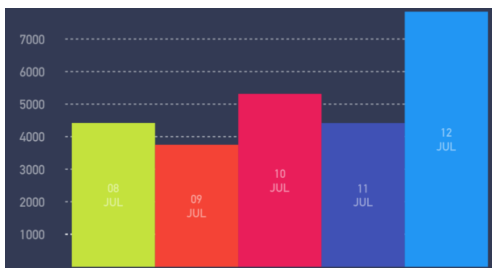


Figura 58 Istogramma realizzato da UIHistogram

Per quanto concerne il componente grafico "UIHistogram" realizza un istogramma richiedendo, come parametri di input, le altezze dei rettangoli con le relative etichette e l'array degli indicatori da posizionare sull'asse delle ordinate.

Infine la visualizzazione dei dati fisiologici e delle periferiche si basa sul modulo astratto "UIAbstractComponent", che suddivide l'area di disegno in tre campi. Quello centrale è riempito direttamente dal sorgente in esame che inserisce una descrizione ed il relativo valore. Inoltre fornisce due metodi che permettono di personalizzare il contenuto posto sulla destra e sulla sinistra dell'area centrale. In particolare "UIData" estende "UIAbstractComponent" definendo sulla sinistra l'icona relativa al dato da visualizzare e sulla destra la data in cui è stato rilevato. Invece per quanto riguarda "UIPeripheral" definisce, nell'area posta sulla sinistra,

l'icona della periferica e su quella a destra lo stato della periferica che può essere: in connessione, connesso o fallito in relazione agli eventi che si verificano.

Figura 59 Relazione tra UIAbstractComponent, UIData e UIPeripheral

8.3.4 Pacchetto Icons

Tutte le icone presenti nell'applicazione sono state progettate e create utilizzando il contesto grafico. Per ciascuna figura è stata realizzata una classe che agisce come una factory alla quale chiedere l'immagine. Ogni volta che si ha necessita di utilizzare un'icona è opportuno passare al metodo, che restituisce il disegno, la dimensione. In questo modo è possibile usare lo stesso oggetto per ottenere la stessa immagine in dimensioni diverse. Al fine di semplificare e fare chiarezza sulle primitive del sistema grafico, invocate per realizzare il disegno, è stata necessaria una progettazione delle icone. In primo luogo il piano di disegno è stato suddiviso mediante una griglia al fine di definire le linee guida per realizzare l'icona. In secondo luogo si sono individuati i punti di intersezione cruciali per la buona riuscita del disegno. A questo punto è stato sufficiente unire i vari punti con linee e curve per ottenere l'immagine desiderata. Di seguito vengono riportati i principali schemi di progettazione affiancando per alcuni il frammento di codice che realizza il disegno. Si fa presente che, nelle figure sottostanti, i punti neri individuano i parametri delle primitive grafiche, invece il punto rosso determina l'origine degli assi opportunamente spostato utilizzando la matrice di trasformazione.

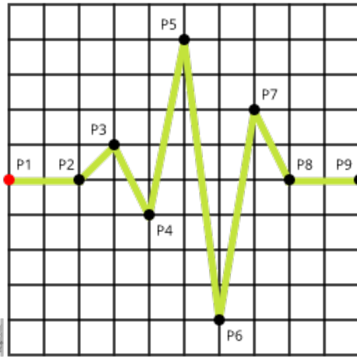


Figura 60 Progettazione icona UIMonitoringIcon

```
//Disegno l'icona
context.beginPath()
context.move(to: .zero) //P1
context.addLine(to: .init(x: 2*widthUnit, y: 0)) //P2
context.addLine(to: .init(
      x: context.currentPointOfPath.x+widthUnit,
      y: context.currentPointOfPath.y+heightUnit)) //P3
context.addLine(to: .init(
      x: context.currentPointOfPath.x+widthUnit,
      y: context.currentPointOfPath.y-2*heightUnit)) //P4
context.addLine(to: .init(
      x: context.currentPointOfPath.x+widthUnit,
      y: context.currentPointOfPath.y+5*heightUnit)) //P5
context.addLine(to: .init(
      x: context.currentPointOfPath.x+widthUnit,
      y: context.currentPointOfPath.y-8*heightUnit)) //P6
context.addLine(to: .init(
      x: context.currentPointOfPath.x+widthUnit,
      y: context.currentPointOfPath.y+6*heightUnit)) //P7
context.addLine(to: .init(
      x: context.currentPointOfPath.x+widthUnit,
      y: context.currentPointOfPath.y-2*heightUnit)) //P8
context.addLine(to: .init(
      x: context.currentPointOfPath.x+2*widthUnit,
      y: context.currentPointOfPath.y)) //P9
context.strokePath()
```

Codice 26 Realizzazione icona UIMonitoringIcon

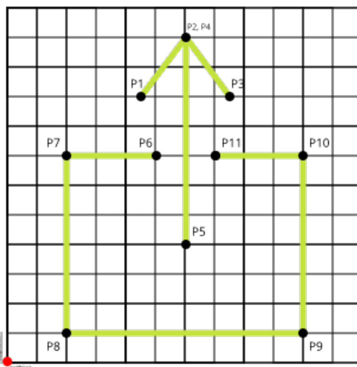
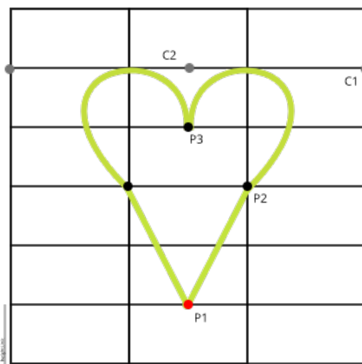


Figura 61 Progettazione icona UIShareIcon

```
//Disegno l'icona
context.beginPath()
//P1
context.move(to: .init(x: 9*widthUnit/2, y: 9*heightUnit))
//P2
context.addLine(to: .init(x: 6*widthUnit, y: 11*heightUnit))
//P3
context.addLine(to: .init(x: 15*widthUnit/2, y:
9*heightUnit))
//P4
context.move(to: .init(x: 6*widthUnit, y: 11*heightUnit))
//P5
context.addLine(to: .init(x: 6*widthUnit, y: 4*heightUnit))
//P6
context.move(to: .init(x: 5*widthUnit, y: 7*heightUnit))
//P7
context.addLine(to: .init(x: 2*widthUnit, y: 7*heightUnit))
//P8
context.addLine(to: .init(x: 2*widthUnit, y: heightUnit))
//P9
context.addLine(to: .init(x: 10*widthUnit, y: heightUnit))
//P10
context.addLine(to: .init(x: 10*widthUnit, y: 7*heightUnit))
//P11
context.addLine(to: .init(x: 7*widthUnit, y: 7*heightUnit))
context.strokePath()
```

Codice 27 Realizzazione icona UIShareIcon



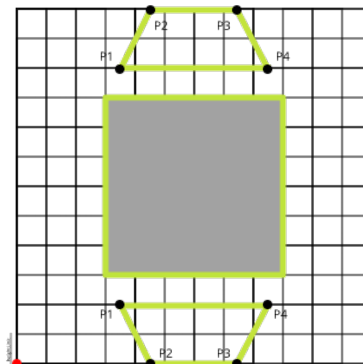


Figura 62 Progettazione icone UIHearthIcon e UIPeripheralIcon

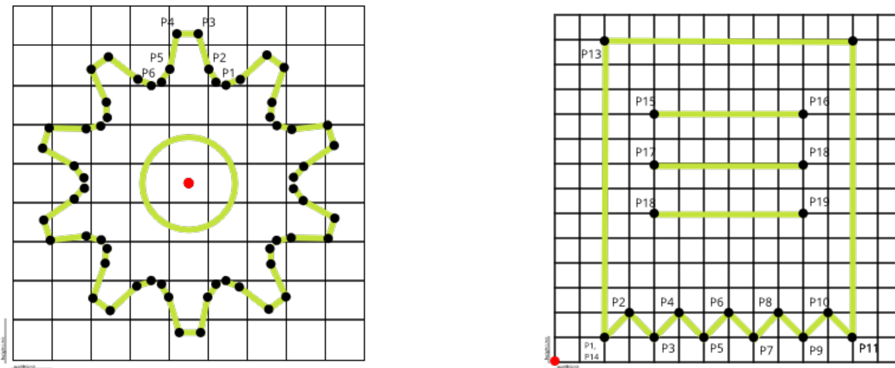


Figura 63 Progettazione icone UISettingIcon ed UIErrorIcon.

L'icona "UISettingIcon" è un pattern ripetuto, ovvero è stato sufficiente disegnarne un piccolo frammento e successivamente replicarlo mediante una trasformazione di rotazione per ottenere l'intero disegno.

8.4 Panoramica dell'interfaccia grafica

In questo paragrafo verranno mostrate tutte le viste, utilizzate nell'applicazione, indicando quali classi compongono l'interfaccia e come l'utente può interagire con esse. Volendo il lettore può interpretare questa sezione come una guida, leggermente più tecnica, all'utilizzo dell'applicazione.

Per prima cosa si fa presente che ciascuna pagina include l'intestazione ed il menu dell'applicazione.

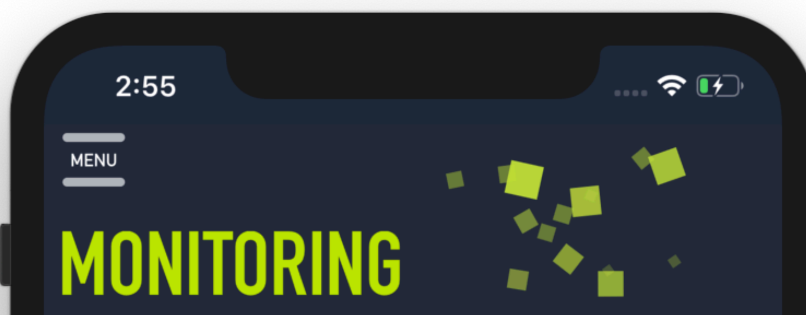


Figura 64 Intestazione dell'applicazione

L'intestazione contiene il titolo della pagina ed il bottone per aprire il menu. Cliccando quest'ultimo l'intero contenitore si sposta verso destra, mediante un'animazione che utilizza una funzione di interpolazione lineare, al fine di mostrare

la scheda di navigazione che rappresenta l'elemento principale per raggiungere le altre pagine.

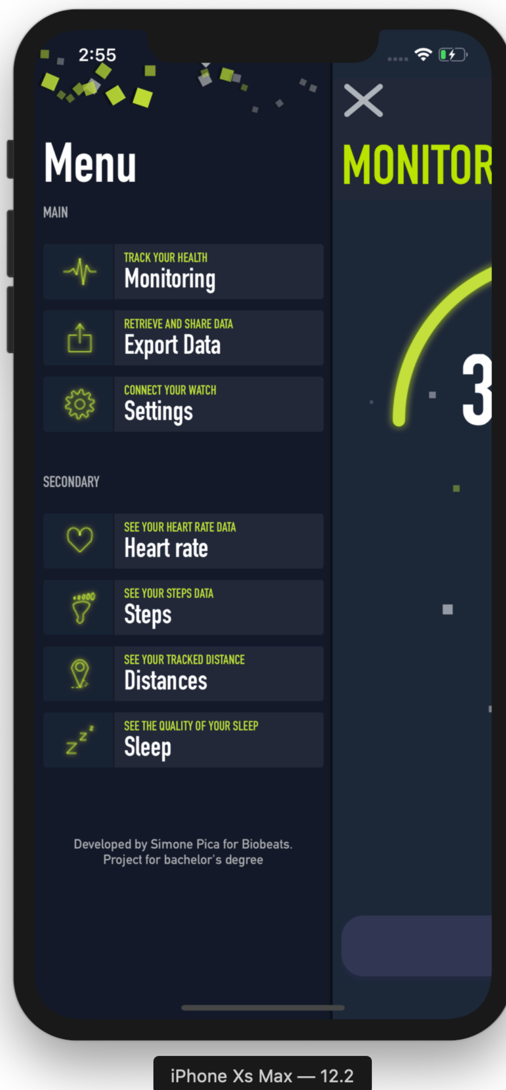


Figura 65 Menu dell'applicazione

Il menu contiene una serie di bottoni suddivisi in due categorie: principali e secondari. Quelli principali contengono riferimenti alle pagine che realizzano le funzionalità fondamentali dell'applicazione come il monitoraggio, l'esportazione dei dati e le impostazioni per connettere la periferica. I bottoni secondari permettono di raggiungere le pagine che consentono di visualizzare le informazioni.

Cliccando su uno dei bottoni, presenti nel menu, viene caricata la pagina richiesta e nell'intestazione parte un'animazione che cambia gradualmente il titolo in essa

contenuto. In particolare viene realizzato un effetto typewriter. Di seguito ne riportiamo un'esecuzione in cui vengono simulati i vari frame.

0	**Monitoring**	12	**Moni|**	24	**Set**
1	**Monitorin**	13	**Mon**	25	**Set|**
2	**Monitorin|**	14	**Mon|**	26	**Sett**
3	**Monitori**	15	**Mo**	27	**Sett|**
4	**Monitori|**	16	**Mo|**	28	**Setti**
5	**Monitor**	17	**M**	29	**Setti|**
6	**Monitor|**	18	**M|**	30	**Settin**
7	**Monito**	19	**|**	31	**Settin|**
8	**Monito|**	20	**S**	32	**Setting**
9	**Monit**	21	**S|**	33	**Setting|**
10	**Monit|**	22	**Se**	34	**Settings**
11	**Moni**	23	**Se|**		

Tabella 26 Mostra l'animazione dalla scritta Monitoring a Settings.

La tabella deve essere interpretata come 35 frame mostrati all'utente uno dopo l'altro. Questa animazione, ma anche quella del bottone menu, sono state realizzate manualmente mediante l'utilizzo dei timer. Infine, nel caso in cui l'utente decida di chiudere il menu, è sufficiente ripremere sul medesimo pulsante utilizzato per aprirlo. In alternativa è possibile avviare la procedura di chiusura scorrendo il dito da destra verso sinistra[45].

[45] Generalmente viene utilizzato il termine in lingua inglese swipe.

8.4.1 Pagina di monitoraggio

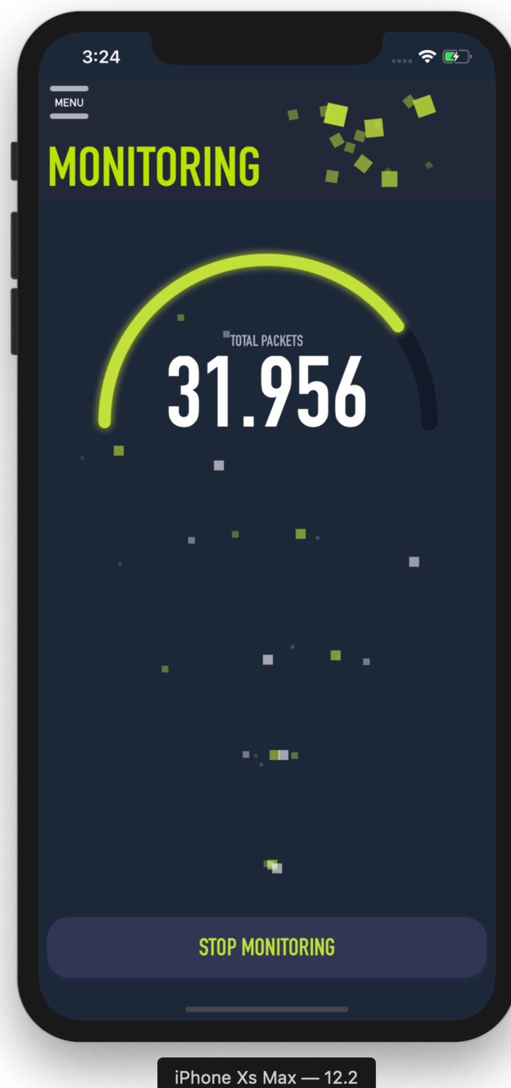


Figura 66 Vista principale dell'applicazione

Questa pagina si compone principalmente di tre componenti. In basso abbiamo un unico bottone che permette di avviare e sospendere la ricezione dei dati dalla periferica. Sopra il bottone è posizionato un emettitore, ovvero un oggetto del framework CoreAnimation che genera un insieme di particelle. In particolare quando l’applicazione è in uno stato in cui non riceve pacchetti l’emettitore è in pausa e non visualizza nessun elemento. Non appena l’utente preme il bottone per prelevare informazioni dalla periferica, l’emettitore simula la ricezione dei dati animando un insieme di quadrati verdi e bianchi che si muovono verso l’indicatore dei pacchetti.

Quest'ultimo rappresenta l'ultimo componente presente nella pagina che mostra il rapporto tra il numero di pacchetti ricevuti in data odierna e quelli del giorno precedente. Invece al suo interno è presente il numero totale di dati ricevuti.

8.4.2 Pagina per l'esportazione dei dati

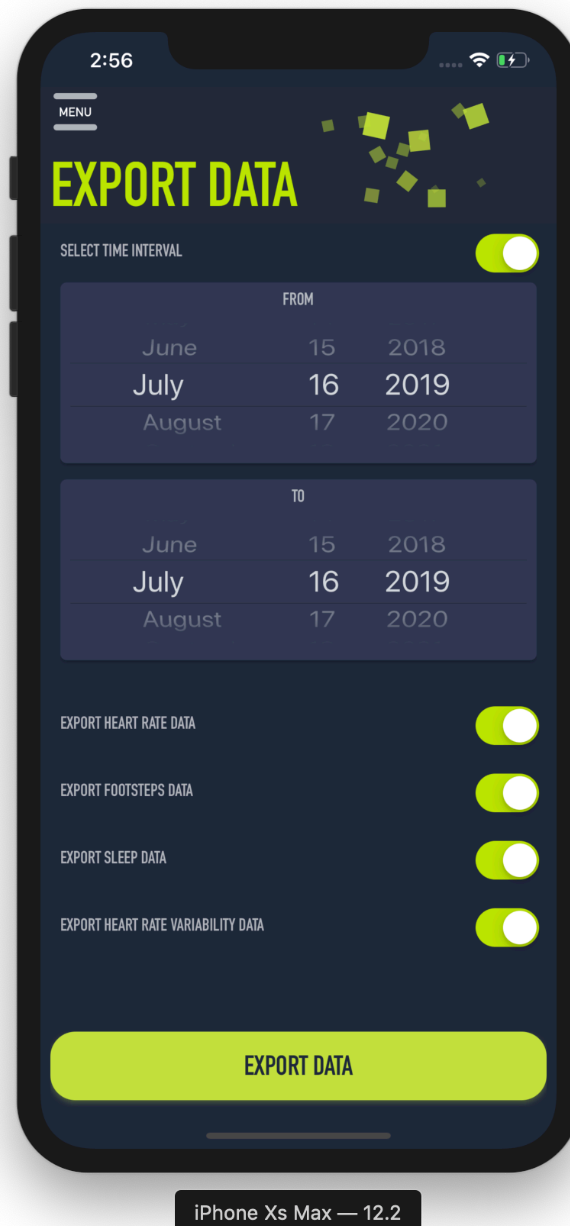


Figura 67 Vista per esportare i dati nel formato JSON

Questa scheda espone i filtri che vengono utilizzati per recuperare un sottoinsieme di dati dal database.

Analizzando la schermata dall'alto verso il basso, il primo "UISwitch" permette di attivare o disattivare la possibilità di selezionare un intervallo di tempo. Quando l'interruttore è impostato su attivo gli "UIDataPicker" sono abilitati ed è possibile selezionare una data. Gli ultimi "UISwitch" riferiscono invece ai tipi di dati che vogliamo recuperare, se attivati tali informazioni vengono incluse nel file generato. L'ultimo componente della pagina è un "UIButton" che permette di avviare l'operazione per creare il file contenente i dati esportati nel formato JSON.

8.4.3 Pagina impostazioni

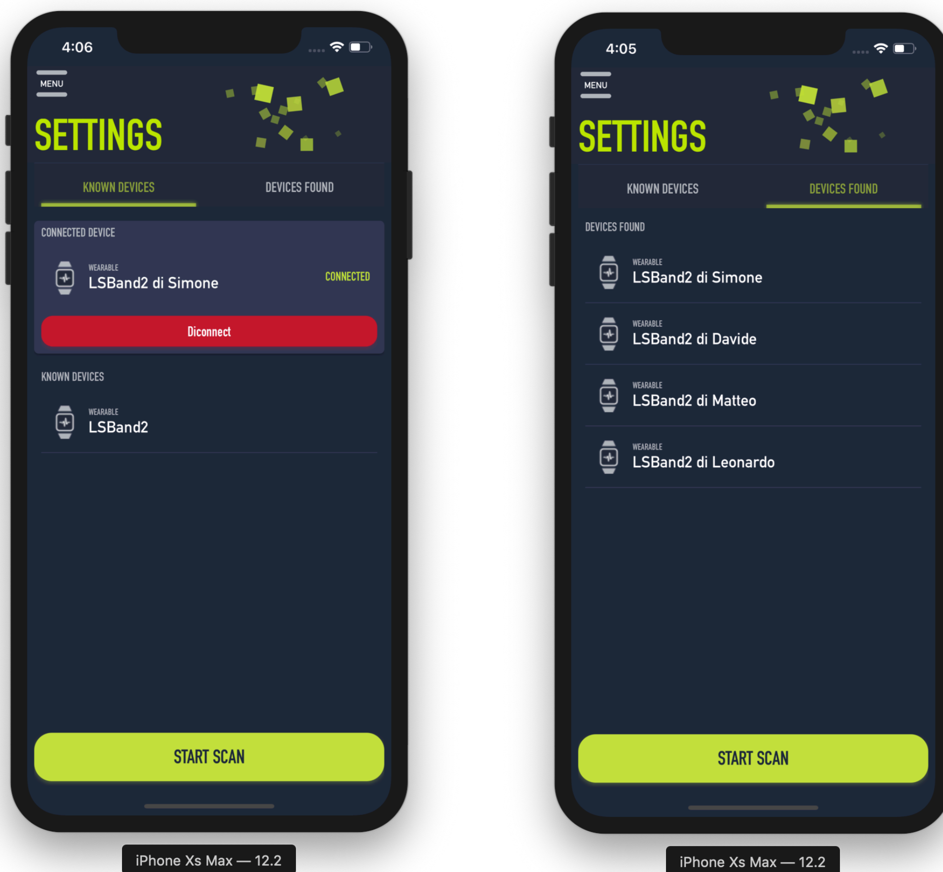


Figura 68 Vista impostazioni di connessione

La pagina di impostazioni permette di ricercare e connettersi ad una periferica. La prima figura mostra la scheda dei dispositivi conosciuti. Tale schermata è divisa in due aree, quella in alto mostra la periferica attualmente connessa dando eventualmente la possibilità di disconnettersi. La seconda area, posizionata sotto la

prima, consente di visualizzare le periferiche alle quali è stata effettuata almeno una connessione. Dato che il numero di dispositivi da visualizzare potrebbe crescere nel tempo, l'intera schermata è stata inserita in una "UIScrollView" per consentire un'espansione maggiore rispetto alla dimensione dello schermo del dispositivo.
Oltre a ciò cliccando sul bottone per avviare la scansione, posizionato nella parte inferiore dell'applicazione, si avvia una transizione smooth scroll[46] verso la schermata presentata nella seconda figura. A questo punto quando nuove periferiche vengono scoperte sono visualizzate direttamente in questa area. Inoltre è possibile avviare un tentativo di connessione cliccando su una di esse. Per lo stesso motivo, precedentemente esposto, anche questa vista è racchiusa in una "UIScrollView". In aggiunta, al fine di organizzare nel modo migliore le periferiche, esse vengono inserite all'interno di una "UIStackView". Infine facciamo notare che un altro modo per passare da una schermata all'altra è quello di utilizzare il tab menu[47] presente sotto l'intestazione dell'applicazione.

[46] È un effetto di scorrimento fluido.
[47] Tipo di menu a linguette.

8.4.4 Pagine per la visualizzazione dei dati

Figura 69 Vista visualizzazione dati sulla frequenza cardiaca del 13 luglio

Figura 70 Vista visualizzazione dati sonno del 11 luglio

La Figura 69 mostra la pagina di visualizzazione dati sulla frequenza cardiaca. Subito dopo l’intestazione dell’applicazione è posizionato l’elemento che permette di cambiare la data al fine di visualizzare le informazioni acquisite nei giorni precedenti. Dopodiché, procedendo dall’alto verso il basso, abbiamo due indicatori: il primo mostra il rapporto tra il valore minimo e massimo rilevato nella data selezionata. Invece il secondo mostra il rapporto tra la media dei valori ed il massimo. Successivamente è possibile notare il grafico a torta che suddivide le informazioni in intervalli di valori. Infine sono riportate le ultime misurazioni effettuate.

Per quanto concerne la Figura 70 mostra i dati relativi al sonno. In particolare essa si compone di due grafici. L’istogramma determina, in ordine temporale, gli intervalli di sonno leggero e profondo che si sono alternati durante il riposo. Invece, per quanto riguarda l’aerogramma, visualizza la percentuale di sonno leggero e profondo

in relazione al numero totale di ore di riposo. Infine, poiché in una determinata data una persona può alternare periodi di sonno a quelli di attività, i diagrammi sono inseriti in una scheda. In questo modo l'utente scorrendo il dito verso destra e sinistra può visualizzare tutti gli intervalli di riposo del giorno selezionato.

Figura 71 Vista visualizzazione numero passi del 12 luglio

Figura 72 Vista visualizzazione distanza percorsa del 12 luglio

La Figura 71 rappresenta la visualizzazione dei dati sui passi in una specifica data. Sotto l'intestazione, oltre ai bottoni per cambiare la data selezionata, è presente l'indicatore che mostra il rapporto tra il numero di passi effettuati nella data indicata ed il numero massimo di passi salvati nell'applicazione. Oltre a ciò è possibile notare un grafico a torta che esibisce la percentuale di passi effettuati nelle prime dodici ore della giornata e quelli compiuti nelle ultime dodici ore. Infine, per quanto riguarda l'istogramma, mette a confronto i passi totali rilevati nei cinque giorni precedenti. Per quanto concerne la Figura 72 mostra dati analoghi, ma relativi alle distanze. Ovvero abbiamo l'indicatore che esprime il rapporto tra la distanza

percorsa nella data selezionata e la distanza massima. Dopodiché è presente l'istogramma che permette di comparare le rilevazioni dei giorni precedenti. Infine viene riportata la lista delle distanze percorse durante la giornata, suddivise per intervalli di tempo.

8.4.5 Visualizzazione degli errori

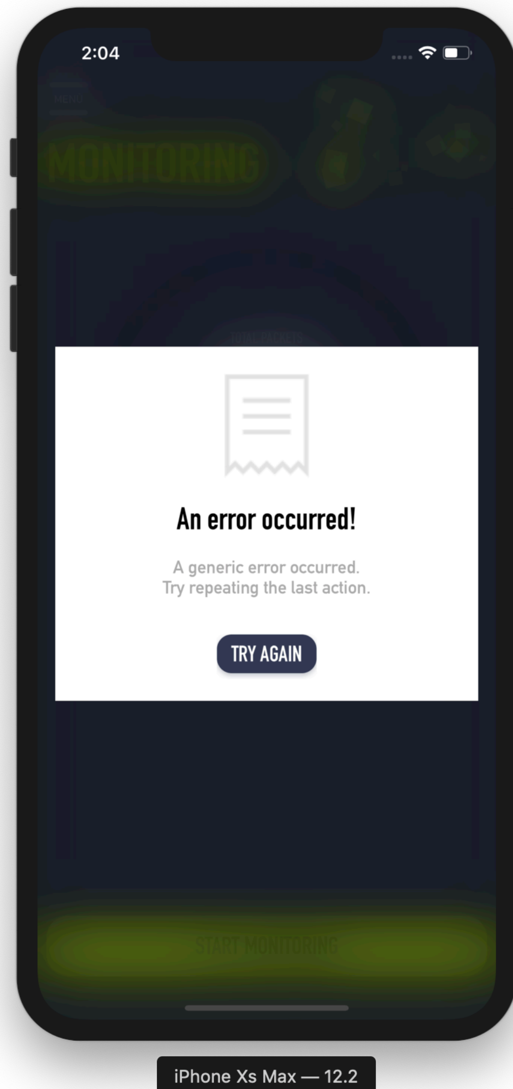


Figura 73 Vista per mostrare all'utente gli errori

Per la visualizzazione degli errori vengono utilizzati dei pop-up aperti sulla schermata corrente. Tali avvisi possono essere personalizzati a seconda dell'errore andando a definire il titolo, la descrizione e l'icona da mostrare. Per chiudere il messaggio è

sufficiente premere sul bottone presente nel riquadro oppure effettuare un tocco al di fuori dell’area dell’avviso.

CAPITOLO 9
TEST DELL'APPLICAZIONE

Questo capitolo intende riportare le tecniche utilizzate per testare l'applicazione realizzata.
In primo luogo, durante la fase di implementazione e nella prima fase di testing, sono stati condotti dei test di unità che mirano a valutare il corretto funzionamento di un metodo o di una classe. In particolare per funzionalità semplici e limitate ho ritenuto sufficiente effettuare verifiche statiche mediante tecniche come Walkthrough[48] e Inspection[49]. Invece per le componenti più estese sono state effettuate anche verifiche dinamiche progettando batterie di prova che garantissero una copertura totale dei comandi. Durante questa fase, al fine di concentrarsi unicamente su una singola componente, sono stati realizzati dei driver[50] e degli stub[51]. In secondo luogo sono stati effettuati dei test d'integrazione per valutare la corretta comunicazione tra i vari moduli. Poiché l'intera interazione avviene mediante il dispatcher è stato sufficiente garantire la correttezza di quest'ultimo al fine di assicurare la corretta comunicazione tra i moduli. Ovviamente ulteriori verifiche sono state necessarie per controllare il formato dei messaggi inviati. Dopodiché, mediante un test di sistema, è stato verificato il funzionamento dell'intero software e la presenza di tutti i requisiti

[48] Tecnica utilizzata per eseguire una lettura critica del codice simulandone l'esecuzione.
[49] Tecnica utilizzata per eseguire una lettura mirata del codice.
[50] Componente fittizia per pilotare un modulo.
[51] Componente fittizia per simulare un modulo.

stabiliti in fase di analisi. Infine, tramite il collaudo del software, tali aspetti sono stati verificati ed accettati dal tutor aziendale.

Oltre a ciò, dopo la fine del tirocinio, ho continuato ad utilizzare l’applicazione al fine di testare aspetti relativi all’utilizzo a lungo termine.

Nei paragrafi successivi di questo capitolo riporteremo, a titolo esemplificativo, alcuni esempi di test di semplici componenti, senza entrare nei dettagli specifici.

9.1 Recuperare la rappresentazione di una periferica

Questo paragrafo riporta la batteria di test al fine di verificare il corretto funzionamento del metodo adibito a recuperare la rappresentazione di una periferica.

```
  func getUIPheripheral(id:String) -> UIPeripheral? {
1    guard let stack = getStackView() else
2       {return nil}
3    for view in stack.arrangedSubviews {
4       guard let peripheral = view as? UIPeripheral else
5          {continue}
6       if (peripheral.blePeripheral?.deviceId == id)
7          {return peripheral;}
     }
8    return nil;
  }
```

Codice 28 Metodo per recuperare la rappresentazione di una periferica

Il codice mostra la funzione che, dato l’identificativo di una periferica, restituisce la sua rappresentazione. In particolare se l’elemento grafico non esiste restituisce il valore nullo.

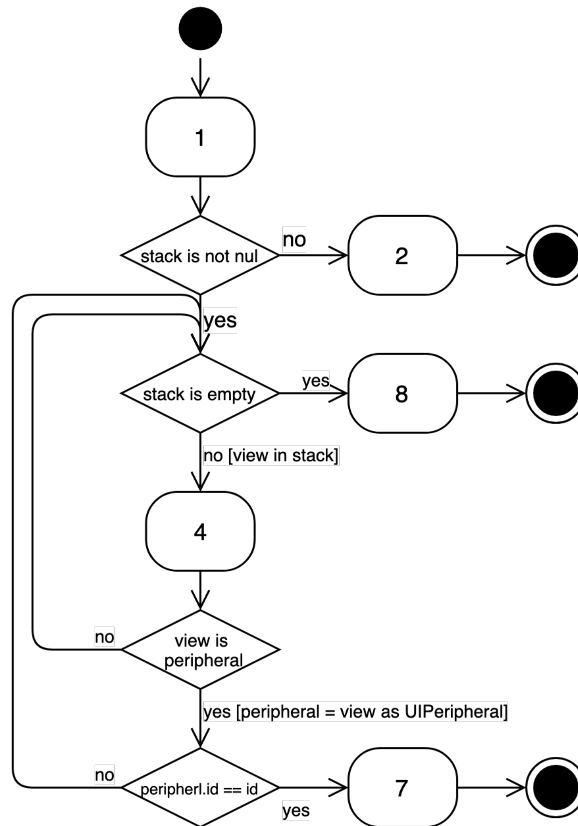


Figura 74 Diagramma di flusso per recuperare la rappresentazione di una periferica

La Figura 74 rappresenta il grafo di flusso, ottenuto a partire dal Codice 28, al fine di definire la struttura identificandone le parti. Di seguito viene presentata una batteria di test.

Batteria di test			
Input	Output	Ambiente	Copertura
Id:"123"	nil	stack:nil	1,2
Id:"234"	nil	stack:[UID(234)]	1,3,4,5,3,8
Id:"123"	UIP(123)	stack:[UIP(567),UIP(123)]	1,3,4,6,3,4,6,7

Tabella 27 Batteria di test per il metodo getUIPeripheral

Come si può notare dalla tabella i test proposti coprono tutti i comandi del metodo. In particolare il secondo test ha come output atteso il valore nullo in quanto lo stack, seppur contenendo la rappresentazione di qualcosa con identificativo "234", l'oggetto non risulta essere una periferica.

9.2 Funzione di aggregazione sulla frequenza cardiaca

In questo paragrafo si riportano i test per verificare la correttezza del gestore che si fa carico delle richieste relative alle funzioni di aggregazione della frequenza cardiaca. In particolare si analizza il flusso delle istruzioni che preparano l'input per invocare il metodo che esegue la query.

```
   func notify(object: Any?, event: EventType) {
1     switch event {
         /* ... */
2        case .GET_FUN_HR: do { //Richiesta fun. agg. su HR
3           guard var dictionary = object as? Dictionary<String, Any>
4           else {return}
5           guard let funs = dictionary["fun"] as? [String]
6           else {return}
7           for fun in funs {
8              guard fun == "min" ||
                     fun == "max" ||
                     fun == "average"
9              else {continue}
10             dictionary[fun] = self.funHR(
                  predicate: datePredicate(
                     dateFrom: dictionary["dateFrom"] as? Date,
                     dateTo: dictionary["dateTo"] as? Date),
                  function: fun)
            }
11          dispatcher.publish(object: dictionary, event: .FUN_HR)
         }
         /* ... */
12       default: break;
      }
   }
```

Codice 29 Gestore richiesta funzioni di aggregazione per la frequenza cardiaca

Come si evince dal codice, il gestore del database riceve la richiesta dal dispatcher il quale comunica il tipo di evento e delle informazioni aggiuntive. Nel caso delle funzioni di aggregazione per la frequenza cardiaca, l'oggetto object viene interpretato come un dizionario. Tale struttura dovrà specificare una serie di chiavi tra le quali "fun", che riporta l'array delle funzioni da calcolare, "dateFrom" e "dateTo" specificano eventualmente un determinato periodo nel quale limitare il calcolo. Predisposto l'input il gestore invoca il metodo che esegue la query corrispondente. In particolare si riporta solo la batteria dei test relativi alla richiesta in esame tralasciando gli altri aspetti.

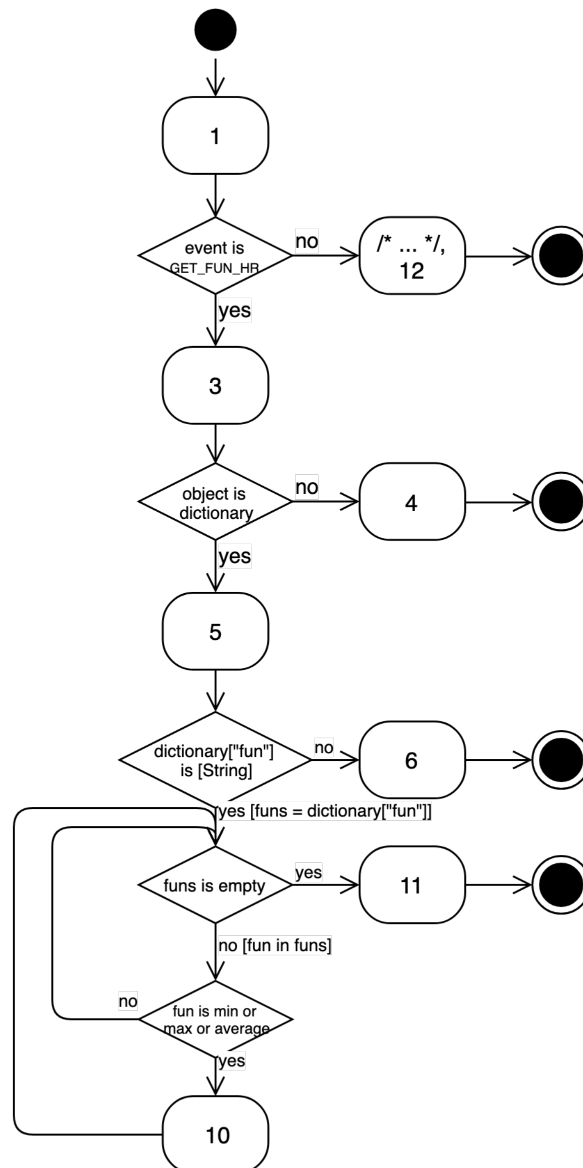


Figura 75 Diagramma di flusso per le funzioni di aggregazione per i dati HR

Il diagramma di flusso mostra la struttura del Codice 29 identificandone le parti. Di seguito si riporta la batteria di test che garantiscono una copertura totale dei comandi relativi al frammento di codice mostrato.

Batteria di test		
Input	Output	Copertura
object: nil, event: GET_FUN_HR	nil	1,2,3,4
object: nil, event: OTHER	nil	1,12

object: ["fun":["min", "max", "average"]], event: GET_FUN_HR	publish([… "min":61, "max":65, "average": 63], FUN_HR)	1,2,3,5,7,8,10, 7,8,10,7,8,10,7,11
object: ["fun":["sum", "min"], "dateTo": 07/08/19], event: GET_FUN_HR	publish([… "min":65], FUN_HR)	1,2,3,5,7,8,9, 7,8,10,7,11
object: ["fun":10] event: GET_FUN_HR	nil	1,2,3,5,6

Tabella 28 Batteria di test per il metodo notify per l'evento GET_FUN_HR

Tutti i test devono essere eseguiti in un ambiente che utilizza uno stub per il database contenente i seguenti due dati relativi alla frequenza cardiaca: 61 (08/08/19), 65 (07/08/19).

9.3 Generazione predicato sul periodo per le query

In questa sezione si riportano i test che verificano la correttezza del metodo del gestore del database, che crea il predicato, al fine di restringere la ricerca ad un determinato periodo.

```
   internal func datePredicate(dateFrom:Date?, dateTo:Date?) ->
   NSCompoundPredicate? {
1     var datePredicate:NSCompoundPredicate? = nil
2     var fromPredicate:NSPredicate? = nil
3     if (dateFrom != nil) {
4        fromPredicate = NSPredicate(
            format: "dataRilevazione >= %@", dateFrom! as CVarArg)
5        datePredicate = NSCompoundPredicate(
            andPredicateWithSubpredicates: [fromPredicate!])
      }
6     if (dateTo != nil) {
7        let toPredicate = NSPredicate(
            format: "dataRilevazione < %@", dateTo! as CVarArg)
8        datePredicate = datePredicate != nil ?
               NSCompoundPredicate(andPredicateWithSubpredicates:
                  [fromPredicate!, toPredicate]) :
               NSCompoundPredicate(andPredicateWithSubpredicates:
                  [toPredicate]);
      }
9     return datePredicate
   }
```

Codice 30 Metodo per generare il predicato sul periodo

Lo scopo del codice in esame è di realizzare una restrizione sui dati da recuperare. Per fare ciò prende in input la data di inizio e fine periodo e restituisce il predicato da usare nelle query.

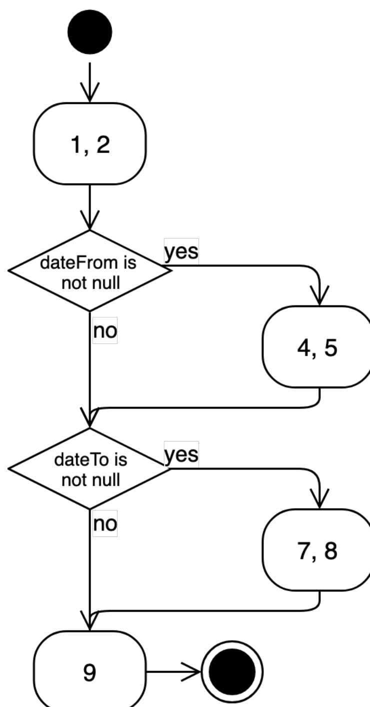


Figura 76 Diagramma di flusso del metodo per generare il predicato sul periodo

Il diagramma di flusso mostra la struttura del Codice 30 identificandone le varie parti.

Batteria di test		
Input	Output	Copertura
dateFrom: nil, dateTo: nil	nil	1,2,3,6,9
dateFrom: 8/20/2019, dateTo: nil	NSCompoundPredicate(from: 8/20/2019)	1,2,3,4,5,6,9
dateFrom: nil, dateTo: 9/20/2019	NSCompoundPredicate(to: 9/20/2019)	1,2,3,6,7,8,9
dateFrom: 8/20/2019, dateTo: 9/20/2019	NSCompoundPredicate(from: 8/20/2019, to: 9/20/2019)	1,2,3,4,5,6,7,8,9

Tabella 29 Batteria di test per il metodo datePredicate

La batteria di test garantisce una copertura totale dei comandi. Inoltre l’ambiente in cui vengono eseguiti i test è irrilevante ai fini dell’output atteso.

CAPITOLO 10
ANALISI DEI DATI ACQUISITI

In questo capitolo mostreremo dei grafici generati a partire dai dati acquisiti nei mesi di luglio ed agosto. In particolare le informazioni sono state rilevate ed estratte utilizzando l'applicazione realizzata. Poiché quest'ultima si limita a mostrare statistiche giornaliere, è stato cercato un software che permettesse di generare grafici su larga scala partendo da una notevole quantità di dati. Considerando che l'applicazione esporta le informazioni utilizzando lo standard JSON è fondamentale il supporto di tale formato da parte del software esterno. Questa attività ha permesso anche di verificare la correttezza del file JSON generato. In particolare, tra i molti programmi che permettono di generare grafici, è stato scelto Microsoft Excel in quanto risulta molto semplice da utilizzare. Infatti, per importare i dati, è stato sufficiente selezionare il file testuale esportato. A questo punto il programma, sfruttando la tecnologia Power Query[52], automaticamente ne analizza la correttezza e trasforma il contenuto in una rappresentazione tabellare. Dopodiché è stato opportuno selezionare le righe, corrispondenti alle informazioni rilevanti, al fine di inserirle nel foglio di calcolo di Excel. Infine, utilizzando gli appositi strumenti del programma, è stato possibile generare il grafico corrispondente.

A partire da questi schemi cercheremo di dedurre informazioni sulle abitudini e sulla salute della persona.

[52] Come riportato in [32], Power Query è una tecnologia di collegamento dei dati che consente di elaborarli in base a specifiche esigenze di analisi.

10.1 Analisi del numero di passi e distanze percorse

Come riportato nell'introduzione di questo capitolo, nei mesi di luglio ed agosto è stata costantemente utilizzata l'applicazione al fine di testarne il corretto funzionamento ed acquisire dati. Di seguito riportiamo i grafici a linee relativi al numero di passi giornalieri ed alla distanza percorsa.

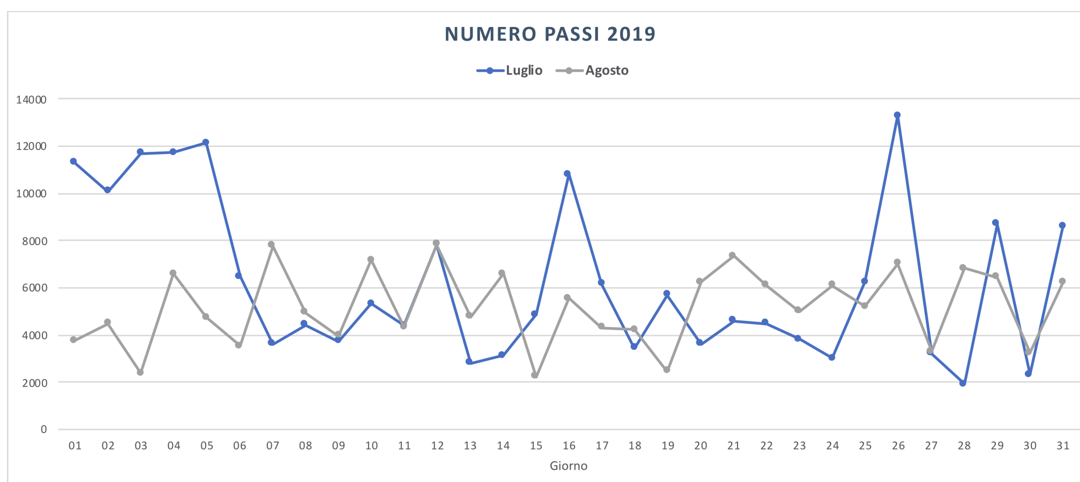


Figura 77 Grafico a linee relativo al numero di passi nei mesi di luglio ed agosto

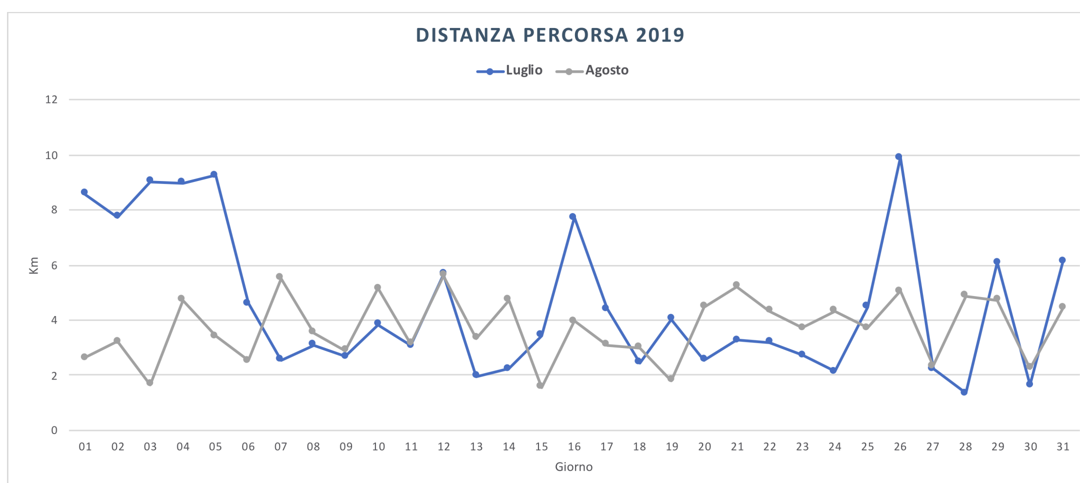


Figura 78 Grafico a linee relativo alle distanze percorse nei mesi di luglio ed agosto

Tali immagini sono state generate mediante Microsoft Excel elaborando il file JSON estratto dall'applicazione. In particolare la Figura 77 mostra, per ciascun giorno del mese, il numero di passi rilevati. La linea blu riferisce alle rilevazioni effettuate nel mese di luglio, invece la linea grigia fa riferimento ai dati di agosto. Per quanto riguarda la Figura 78 riporta informazioni analoghe, ma relative alla distanza percorsa. In primo luogo si può notare che i giorni di massima attività sono stati il 26

luglio ed il giorno 12 per il mese di agosto. Invece il 28 luglio ed il 15 agosto vi è stata un'attività minima. In secondo luogo notiamo chiaramente che i grafici in Figura 77 e Figura 78 sono simili, poiché raffigurano uno stesso andamento, anche se derivati da insiemi diversi di dati. Questo ci fa dedurre che la distanza percorsa sia calcolata direttamente dal numero di passi. A sostegno di questa ipotesi sono le specifiche tecniche del dispositivo remoto che non riportano nessun sistema di posizionamento, infatti gli unici sensori presenti sono il pedometro e quello cardio. Di conseguenza tale informazione dovrà essere derivata da altre ed è logico pensare che alla base del calcolo ci sia il numero di passi rilevati.

10.2 Analisi delle frequenze cardiache

Questo paragrafo riporta i grafici delle frequenze cardiache rilevate nei mesi di luglio ed agosto.

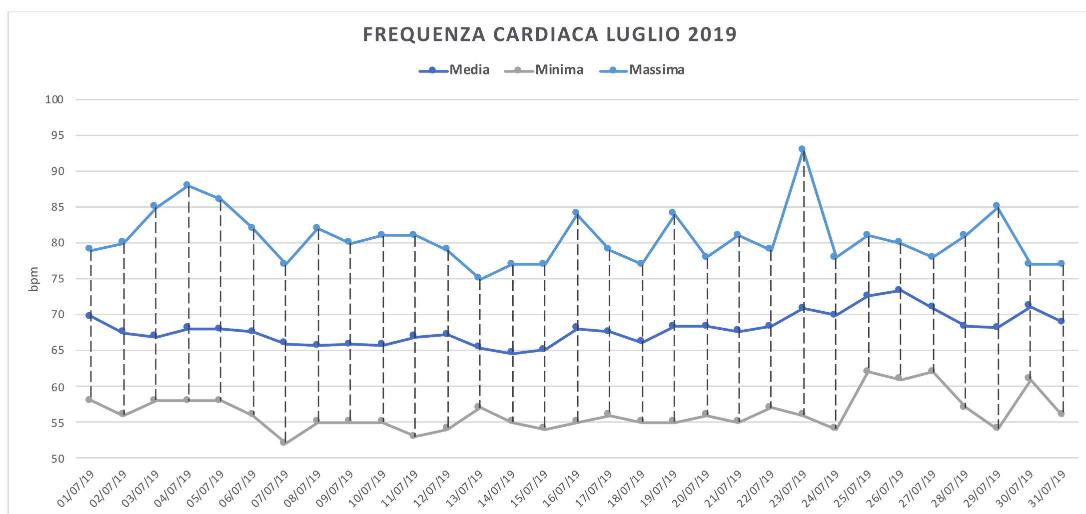


Figura 79 Grafico a linee relativo alla frequenza cardiaca nel mese di luglio

Come è possibile notare dalla figura vi sono tre linee. Il tratto di colore celeste indica il valore massimo rilevato nella giornata, mentre quello grigio determina la frequenza cardiaca minima. Invece la linea blu rappresenta la media delle rilevazioni giornaliere. In particolare si può notare che la massima rilevazione è stata effettuata il 23 luglio di oltre 90bpm, mentre la minima, di circa 52bpm, il 7 luglio. In generale invece è stata rilevata una media giornaliera di circa 68-69bpm.

Analogamente si riporta il grafico delle rilevazioni nel mese di agosto, adottando lo stesso schema dei colori.

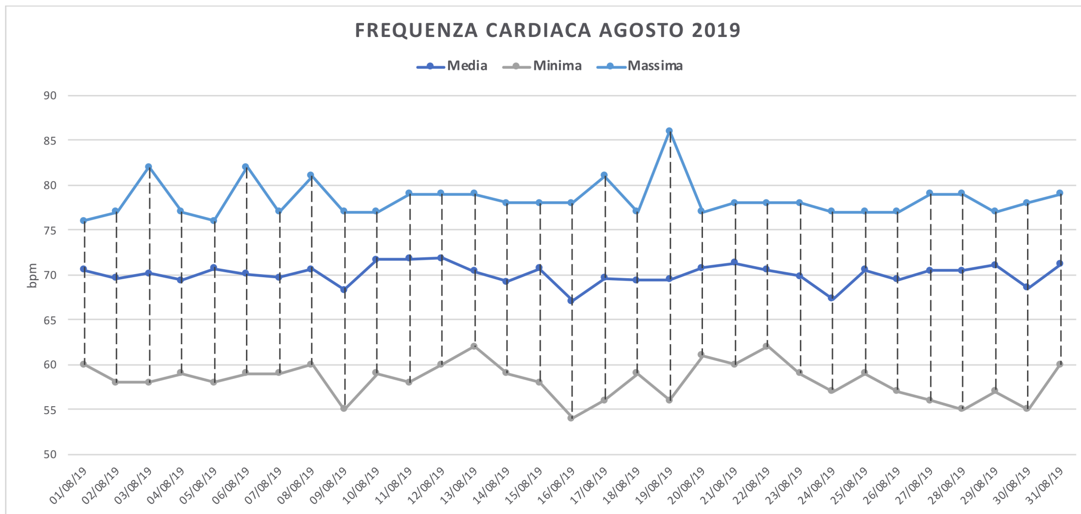


Figura 80 Grafico a linee relativo alla frequenza cardiaca nel mese di agosto

Dalla Figura 80 si può notare che nel mese di agosto la frequenza cardiaca massima è stata rilevata il giorno 19 di oltre 85bpm, invece la minima, di circa 54bpm, si è verificata il 16 agosto. Anche in questo caso, come è logico che sia, la media delle frequenze cardiache giornaliere risulta essere intorno ai 69-70bpm.

10.3 Analisi della quantità e qualità del sonno

Questo paragrafo intende riportare una visione generale della qualità del sonno ricavata dai dati rilevati nei mesi di luglio ed agosto.

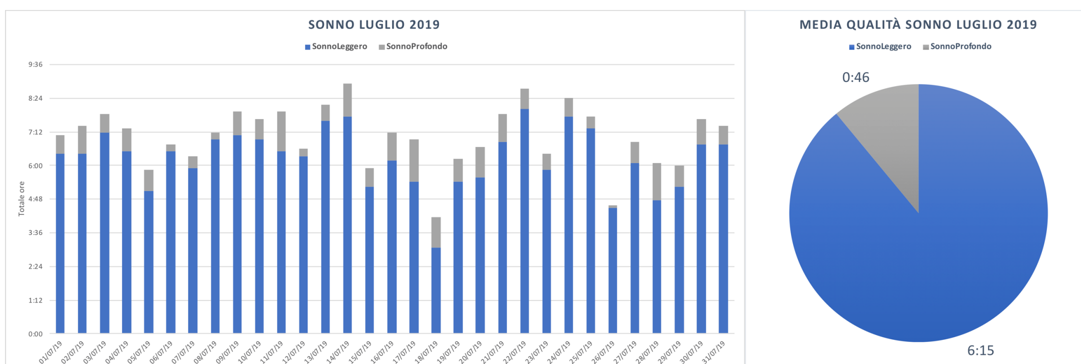


Figura 81 Istogramma ed areogramma relativi al sonno nel mese di luglio

La Figura 81 mostra due grafici in cui vengono mostrati i dati del sonno rilevati durante il mese di luglio. In particolare il colore blu è utilizzato per rappresentare il sonno leggero, mentre il colore grigio per definire il sonno profondo. Per quanto

riguarda l'istogramma determina il numero totale di ore di riposo suddivise per tipo di sonno. Il massimo tempo di riposo si è verificato il 14 luglio con circa 9 ore, invece il minimo, di circa 4 ore e 20 minuti, si è riscontrato il 18 luglio.

Per quanto concerne l'areogramma mostra il rapporto tra la media giornaliera del tempo passato in un sonno leggero e profondo rispetto al totale. In particolare si evidenzia che mediamente si ha un sonno leggero lievemente superiore alle 6 ore, mentre quello profondo è inferiore ad 1 ora. La somma dei due valori determina il tempo medio trascorso a dormire, ovvero 7 ore ed 1 minuto.

Analogamente si riportano i grafici derivati dai dati acquisiti durante il mese di agosto.

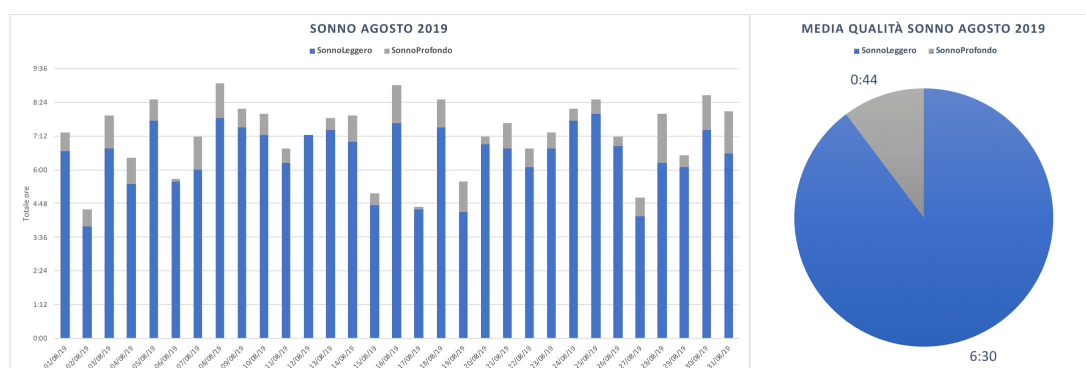


Figura 82 Istogramma ed areogramma relativi al sonno nel mese di agosto

Dalla Figura 82 si può notare una media giornaliera di 7 ore e 14 minuti totali di riposo, di cui 6 ore e 30 minuti in sonno leggero e 44 minuti in sonno profondo. In particolare il massimo tempo di riposo è avvenuto l'8 agosto di circa 9 ore, invece il minimo, di 4 ore e 35 minuti, il 2 agosto.

CAPITOLO 11
CONCLUSIONI

Al termine del tirocinio l'intera applicazione è stata completamente sviluppata e testata soddisfacendo le specifiche richieste. Come riportato nell'introduzione, quest'ultima verrà utilizzata per condurre esperimenti scientifici con la collaborazione dell'ospedale Santa Chiara. Inoltre non si esclude la possibilità che il software venga ampliato inviando i dati direttamente ad un database online.
In particolare questa esperienza mi ha permesso di incrementare le conoscenze relative allo sviluppo di applicazioni per dispositivi iOS. Inoltre mi ha consentito di applicare le tecniche, apprese nel corso di laurea, ad un progetto reale al fine di formalizzare i requisiti e progettare il software. In particolare si sono affrontate tutte le fasi del ciclo di sviluppo. Ovviamente, per motivi di mancanza di tempo, alcune sono state approfondite più di altre. Sicuramente la fase implementativa è una di quelle che ha richiesto molto lavoro a causa della quantità di concetti da affrontare. In primo luogo sono state apprese nozioni sulla tecnologia Bluetooth Low Energy e sul framework CoreBluetooth, comprendendo il ruolo degli attori coinvolti e le loro interazioni. In secondo luogo ulteriori concetti sono stati approfonditi sul framework CoreData per memorizzare le informazioni permanentemente nel dispositivo locale. Nel dettaglio sono state analizzate e risolte le problematiche relative all'accesso concorrente. Oltre a ciò, utilizzando il framework CoreGraphics, è stata disegnata l'intera interfaccia utente approfondendo le primitive grafiche e le conoscenze relative al layout ed alle animazioni.

Infine è stata affrontata la questione pertinente al ciclo di vita di un'applicazione. In particolare sono state individuate le problematiche e le soluzioni per un'esecuzione a lungo termine in background. Oltre a ciò il tirocinio mi ha permesso di migliorare le relazioni interpersonali al fine di cooperare per raggiungere obiettivi comuni. Inoltre è stato possibile apprendere le metodologie utilizzate in un ambiente lavorativo e la coordinazione tra i vari team di sviluppo.

In conclusione questa esperienza è stata del tutto positiva grazie anche al supporto del team di BioBeats che mi ha messo a mio agio fin dal principio. Inoltre, il fatto di aver lavorato su svariati aspetti, mi ha fornito diversi spunti sugli argomenti da approfondire e che ritengo interessanti per accedere, nel migliore dei modi, al mondo del lavoro.

BIBLIOGRAFIA

[1] Apple, «Xcode, Ambiente di sviluppo.,» [Online]. Available: https://developer.apple.com/xcode/ide/.

[2] Apple, «E-book: The Swift Programming Language (Swift 5.0),» [Online]. Available: https://books.apple.com/it/book/the-swift-programming-language-swift-5-0/id881256329.

[3] Apple, «Linguaggio di programmazione Swift,» [Online]. Available: https://www.apple.com/swift/.

[4] Apple, «Framework CoreGraphics,» [Online]. Available: https://developer.apple.com/documentation/coregraphics.

[5] Apple, «Framework CoreAnimation,» [Online]. Available: https://developer.apple.com/documentation/quartzcore.

[6] Apple, «Framework CoreBluetooth,» [Online]. Available: https://developer.apple.com/documentation/corebluetooth.

[7] Apple, «Framework CoreData,» [Online]. Available: https://developer.apple.com/documentation/coredata.

[8] «Slack,» [Online]. Available: https://slack.com/intl/en-it/.

[9] Apple, «Source Control,» [Online]. Available: https://developer.apple.com/library/archive/documentation/ToolsLanguages/Conceptual/Xcode_Overview/UsingSourceCodeControl.html.

[10] Github, «Github,» [Online]. Available: https://github.com/.

[11] V. Driessen, «GitFlow,» [Online]. Available: https://it.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow.

[12] T. Reenskaug, «The original MVC report,» [Online]. Available: http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf.

[13] T. Reenskaug, «Defines the MVC terms,» [Online]. Available: http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf.

[14] P. e. a. Clemens, Documenting Software Architectures., Addison Wesley, 2003.

[15] Apple, «DispatchQueue,» [Online]. Available: https://developer.apple.com/documentation/dispatch/dispatchqueue.

[16] Google, «Protocol Buffer,» [Online]. Available: https://developers.google.com/protocol-buffers/docs/overview.

[17] Apple, «Importing Objective-C into Swift,» [Online]. Available: https://developer.apple.com/documentation/swift /imported_c_and_objective-c_apis/importing_objective-c_into_swift.

[18] Apple, «Managing Your App's Life Cycle,» [Online]. Available: https://developer.apple.com/documentation/uikit /app_and_scenes/managing_your_app_s_life_cycle.

[19] Apple, «Core Bluetooth Background Processing for iOS Apps,» [Online]. Available: https://developer.apple.com/library/archive/documentation/ NetworkingInternetWeb/Conceptual/CoreBluetooth_concepts/ CoreBluetoothBackgroundProcessingForIOSApps/ PerformingTasksWhileYourAppIsInTheBackground.html.

[20] Apple, «NSManagedObject,» [Online]. Available: https://developer.apple.com/documentation/coredata/nsmanagedobject.

[21] Apple, «Core Data Stack,» [Online]. Available: https://developer.apple.com/documentation/coredata /setting_up_a_core_data_stack.

[22] Apple, [Online]. Available: https://developer.apple.com/library/archive/documentation /Cocoa/Conceptual/CoreData/ChangeManagement.html.

[23] Apple, «NSManagedObjectContext,» [Online]. Available: https://developer.apple.com/documentation/coredata /nsmanagedobjectcontext.

[24] Apple, «How Managed Objects are related,» [Online]. Available: https://developer.apple.com/library/archive/documentation/Cocoa /Conceptual/CoreData/HowManagedObjectsarerelated.html.

[25] Apple, «Entity Inheritance,» [Online]. Available: https://developer.apple.com/library/archive/documentation /Cocoa/Conceptual/CoreData/KeyConcepts.html# //apple_ref/doc/uid/TP40001075-CH30-SW16.

[26] Apple, «Quartz 2D Programming Guide,» [Online]. Available: https://developer.apple.com/library/archive/documentation /GraphicsImaging/Conceptual/drawingwithquartz2d/dq_overview /dq_overview.html#//apple_ref/doc/uid/TP30001066-CH202-TPXREF101.

[27] «Algoritmo Cohen-Sutherland,» [Online]. Available: https://it.wikipedia.org/wiki/Algoritmo_Cohen-Sutherland.

[28] Apple, «CGAffineTransform,» [Online]. Available: https://developer.apple.com/documentation/coregraphics /cgaffinetransform.

[29] Apple, «UIKit,» [Online]. Available: https://developer.apple.com/documentation/uikit.

[30] Apple, «Source Control,» [Online]. Available: https://developer.apple.com/library/archive/documentation/ToolsLanguages/Conceptual/Xcode_Overview/UsingSourceCodeControl.html.

[31] SQLite Consortium, «SQLite,» [Online]. Available: https://www.sqlite.org/index.html.

[32] Microsoft, «Panoramica e formazione su Power Query,» [Online]. Available: https://support.office.com/it-it/article/ panoramica-e-formazione-su-power-query -ed614c81-4b00-4291-bd3a-55d80767f81d.