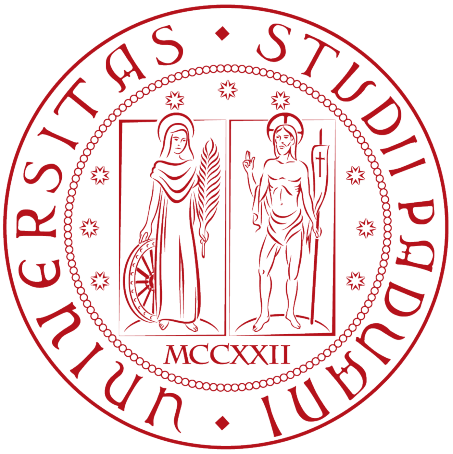
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Clean Architecture applicata allo sviluppo di un'app per smartwatch

Tesi di laurea

Relatore

Prof. Francesco Ranzato

Laureando

Enrico Zangrando

Anno Accademico 2022-2023

"If you only do what you can do, you will never be more than you are now."

— Master Oogway

Dedicato a ...

Tutti coloro che hanno illuminato il mio cammino verso la conoscenza e la crescita personale.

Alla mia famiglia, per il loro costante sostegno, amore e fiducia in me. Il vostro impegno nell'educazione mi ha ispirato a perseguire il mio percorso accademico con passione.

Ai miei insegnanti e professori, per avermi guidato e ispirato attraverso le sfide dell'apprendimento. Le vostre lezioni e il vostro impegno hanno plasmato la mia mente e il mio spirito in modo indelebile.

Agli amici, al sostegno reciproco ed ai momenti di leggerezza che hanno reso questo viaggio così memorabile.

A tutti coloro che hanno condiviso le loro conoscenze e competenze, anche coloro che ho incontrato solo attraverso i libri e le risorse online. La vostra generosità nell'aprire nuove porte alla conoscenza è stata un dono inestimabile.

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di stage, della durata di circa trecento ore, presso l'azienda Vimar S.p.A. L'obiettivo era di creare un applicazione per *smartwatch* che fosse in grado di interfacciarsi con il sistema di domotica Vimar. Durante lo sviluppo dell'applicazione mi sono concentrato sull'utilizzo di un'architettura software solida che fosse in grado di assicurare una facile manutenzione del codice e che lo rendesse facilmente estendibile. Ho quindi scelto di utilizzare la *Clean Architecture*, ideata da Robert C. Martin. Basandosi sui principi di progettazione Single responsibility, Open-closed, Liskov substitution, Interface segregation, Dependency inversion (SOLID) l'utilizzo di quest'architettura permette una separazione delle logiche di *business* da quelle applicative. Il *software* diventa quindi indipendente da framework, dall'interfaccia grafica, da fonti di dati e in generale da qualsiasi agente esterno. L'applicazione sviluppata durante lo stage in ambiente Android con il linguaggio di programmazione Kotlin mi ha permesso di applicare la *Clean Architecture* ad un esempio concreto sfruttando il pattern architetturale Model View ViewModel (MVVM).

Ringraziamenti

Innanzitutto, vorrei esprimere la mia gratitudine al Prof. Francesco Ranzato, relatore della mia tesi, per l'aiuto e il sostegno fornitomi durante la stesura del lavoro.

Desidero ringraziare con affetto i miei genitori per il sostegno, il grande aiuto e per essermi stati vicini in ogni momento durante gli anni di studio.

Ringrazio tutta la mia famiglia, zii, nonni, cugini che nel corso degli anni mi sono sempre stati vicini.

Ho desiderio di ringraziare poi i miei amici per tutte le bellissime avventure vissute.

Padova, Dicembre 2023 Enrico Zangrando

Indice

Elenco delle figure

Elenco delle tabelle

[1] *Refactoring Guru*. URL: https://refactoring.guru.

Capitolo 1

Introduzione

1.1 Norme tipografiche

Per la scrittura del documento sono state adottate le seguenti convenzioni tipografiche:

- gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune menzionati vengono definiti nel glossario, situato alla fine del presente documento;
- i termini in lingua straniera o facenti parti del gergo tecnico sono evidenziati con il carattere *corsivo*.

1.2 L'azienda

Vimar è stata fondata nel 1945 a Marostica e ha iniziato la sua attività producendo interruttori e prese elettriche. Nel corso degli anni l'azienda ha ampliato la sua gamma di prodotti. Oggi Vimar è presente in tutto il mondo ed offre una vasta gamma di soluzioni per l'automazione residenziale e industriale come sensori di movimento, sistemi di controllo dell'illuminazione, sistemi di automazione per tende e persiane, impianti audio, videocitofoni e sistemi di sicurezza. I prodotti Vimar sono caratterizzati da un'elevata qualità estetica, affidabilità tecnologica e facilità d'installazione.

Vimar è impegnata nell'innovazione tecnologica e nella sostenibilità. Ha sviluppato soluzioni intelligenti per la gestione dell'energia e la domotica, consentendo agli utenti di controllare e automatizzare vari aspetti delle loro abitazioni o edifici.

1.3 L'idea

L'idea dello stage è stata quella di creare un'applicazione per il sistema operativo WearOS che permettesse di interfacciarsi con i più importanti dispositivi controllabili

Figura 1.1: Logo Vimar S.p.A

dall'applicazione *smartphone*.

1.3.1 L'app View

L'app View ha un ruolo centrale all'interno dell'ecosistema Vimar poiché permette agli utenti di interagire con il mondo domotico e videocitofonico. L'applicazione permette di accendere e spegnere le luci, regolare luminosità o colore della luce, rispondere a chiamate provenienti dai sistemi di citofonia, accendere e regolare i termostati, regolare l'apertura di persiane o tapparelle, controllare attuatori o sensori, programmare scene cioè una configurazione o un insieme predefinito di azioni o comportamenti che uno o più dispositivi possono eseguire in risposta a determinate condizioni o *trigger*.

1.4 Organizzazione del testo

Il secondo capitolo descrive gli strumenti e le metodologie utilizzate dal *team* di sviluppo.

Il terzo capitolo approfondisce la proposta di stage che mi è stata presentata illustrandone obiettivi e pianificazione delle attività.

Il quarto capitolo illustra le esigenze che mi sono state presentate e i requisiti che ho ricavato dopo un'attenta analisi.

Il quinto capitolo approfondisce il concetto di architettura software illustrando poi come è strutturata l'applicazione ed infine le interfacce di cui è composta.

Il sesto capitolo illustra i test che sono stati pensati ed implementati nell'applicazione.

Nel settimo capitolo viene descritta la retrospettiva fatta al termine dello stage

Capitolo 2

Processi e metodologie

Il capitolo illustra i processi di sviluppo, le metodologie e gli strumenti adottati dai membri dell'azienda.

2.1 Il framework SCRUM

In Vimar per lo sviluppo di software viene adottato un approccio Agile sfruttando il framework Scrum.

Il framework Scrum ha l'obiettivo di portare a termine un progetto *software* sviluppato in gruppo in maniera iterativa. Il framework incoraggia i gruppi ad imparare attraverso l'esperienza e ad organizzarsi in modo autonomo mentre si lavora su un problema.

La realizzazione del progetto viene suddivisa in bervi periodi di tempo definiti *sprint*. Prima di questi periodi di tempo vengono definiti gli obiettivi da conseguire entro la fine dello *sprint*, al termine di esso si effettua una retrospettiva per determinare cosa ha funzionato e cosa no. In questo modo è possibile sfruttare l'esperienza acquisita per migliorare gli *sprint* futuri.

2.1.1 Il team

Le dimensioni del *team* Scrum sono in genere ridotte, circa 10 persone, ma sufficienti per portare a termine una notevole quantità di lavoro in un solo *sprint*. All'interno di un team Scrum è necessario che vengano ricoperti tre ruoli specifici:

- *Product owner*: ha lo scopo di definire le priorità del lavoro che verrà svolto dal gruppo. Si occupa di creare e gestire il *backlog* di prodotto, fornire al gruppo indicazioni chiare su quali funzionalità fornire in futuro ed inoltre decide quando rilasciare il prodotto. Nel caso di Vimar il prodotto è interno perciò il *product owner* coincide con il responsabile di progetto.
- Gli *Scrum Master*: hanno il compito di promuovere i valori ed i principi illustrati dal framework all'interno del *team*. Pianificano le risorse necessarie per l'esecuzione delle cerimonie Scrum: pianificazione dello *sprint*, *stand-up meeting*, revisione e retrospettiva dello *sprint*.

- Membri del team di sviluppo: i *team* più efficaci sono quelli più affiliati e in genere includono da cinque a sette membri. Per stabilire le dimensioni ottimali del team Jeff Bezos, fondatore di Amazon, ha coniato la regola delle due pizze secondo la quale per sfamare un qualsiasi gruppo non dovrebbero essere necessarie più di due pizze. I membri del *team* condividono le loro conoscenze in modo che nessuno possa ostacolare la consegna del lavoro.

2.1.2 Artefatti Scrum

Gli artefatti Scrum sono informazioni utilizzate dal gruppo di lavoro che aiutano a definire il prodotto e il lavoro da svolgere per crearlo.

Backlog di prodotto

Il *backlog* di prodotto è un elenco contenente tutti i lavori che devono essere eseguiti, viene redatto dal *product owner* o dal responsabile di progetto. Si tratta di un elenco dinamico di funzionalità, requisiti e miglioramenti o correzioni dal quale si può strutturare un *backlog* per lo *sprint*. Viene costantemente rivisto e modificato in termini di priorità da parte del responsabile. Questo perché man mano che si acquisiscono conoscenze o avvengono cambiamenti di mercato, alcuni elementi del backlog potrebbero diventare non più pertinenti.

Backlog dello *sprint*

Il *backlog* dello *sprint* è l'elenco di elementi selezionati dal *team* per l'implementazione nello *sprint* corrente. L'elenco viene costruito dal gruppo di lavoro durante la riunione di pianificazione dello *sprint* scegliendo alcuni degli elementi facenti parte del *backlog* di prodotto.

2.1.3 Cerimonie Scrum

Le cerimonie del framework Scrum rappresentano le riunioni che i *team* eseguono regolarmente.

Pianificazione dello *sprint*

Durante questa cerimonia viene pianificato il lavoro da svolgere durante l'imminente ciclo di *sprint*, l'evento è gestito dallo *scrum master*. Durante la riunione viene definito lo *sprint backlog* in maniera collaborativa, definendo le attività da svolgere ed una stima del tempo richiesto per completare ognuna di esse.

Daily scrum meeting o stand up meeting

Incontro giornaliero di circa quindici minuti, viene definito *stand up meeting* perché andrebbe svolto in piedi in modo da terminare velocemente. L'obiettivo dell'incontro è sincronizzarsi capendo lo stato delle attività di tutti. Durante il *meeting* ognuno comunica, in genere, cosa ha fatto il giorno precedente, se ha incontrato qualche difficoltà e aggiorna gli altri su cosa ha intenzione di portare a termine nel corso della giornata.

Revisione dello sprint

Riunione durante la quale il lavoro realizzato durante lo *sprint* viene presentato da parte del *team* e validato. Per l'incontro vengono definiti dei limiti di tempo e solitamente non si usano *slide*.

Retrospettiva dello sprint

Questa riunione viene effettuata dopo la revisione, durante l'incontro si valuta ciò che ha funzionato e ciò che si potrebbe migliorare nel prossimo sprint.

Planning poker

Il *planning poker* è un metodo utilizzato per stimare quanto tempo potrebbe richiedere un certa attività per essere portata a termine. Si gioca in gruppo ed ogni membro del *team* ha in mano un mazzo di carte con una sequenza di numeri, la sequenza raccomandata è stata ricavata dalla sequenza di Fibonacci:

$$0, 0.5, 1, 2, 3, 5, 8, 13, 20, 40, 100$$

Si comincia prendendo un'attività dal *backlog*, basandosi sull'attività scelta ogni membro del *team* sceglierà una carta da scoprire il cui valore rappresenta una sua stima temporale per portare a termine l'attività. Quando tutti hanno scelto si scoprono contemporaneamente le carte. Se tutti sono d'accordo si assegna all'attività la stima scelta altrimenti si discute su divergenze tra i valori scelti, di solito la discussione coinvolge chi ha scelto i valori più bassi e chi ha scelto quelli più alti. Terminata la discussione segue una nuova stima con nuova rilevazione delle carte. Se la stima continua ad oscillare tra diversi valori interviene lo *scrum master* scegliendo il valore più adeguato.

2.2 Strumenti utilizzati

Di seguito vengono elencati gli strumenti utilizzati dal *team* di sviluppo per comunicare ed organizzare il lavoro.

Microsoft Teams

Microsoft Teams è un applicativo utilizzato per videoconferenze e *chat*, in Vimar il team ne usufruisce per le comunicazioni interne.

Jira

Jira è uno strumento sviluppato da Atlassian utilizzato per tenere traccia dello stato delle attività. All'interno del programma è possibile creare diverse attività, assegnarne ognuna ad uno o più membri del *team*, dare alle attività una priorità ed una stima di tempo per il completamento. Si possono creare *board* in cui monitorare lo stato delle attività.

Confluence

Confluence è una piattaforma sviluppata da Atlassian in cui è possibile scrivere e pubblicare documentazione. In Vimar viene utilizzata per documentare decisioni prese

durante lo sviluppo dei progetti, le norme di progetto, per documentare requisiti ed esigenze dei progetti, per la documentazione tecnica di progetti e dell'ecosistema Vimar.

Capitolo 3

Descrizione dello stage

Verrà descritta la proposta di stage e la pianificazione delle attività necessarie al raggiungimento degli obiettivi prestabiliti.

3.1 Introduzione al progetto

Vimar realizza impianti di domotica controllabili tramite l'applicazione per *smartphone* Vimar View. L'applicazione consente di eseguire diverse operazioni, come accendere e spegnere le luci, regolare la luminosità e il colore della luce, rispondere alle chiamate provenienti dai sistemi di citofonia, controllare e regolare i termostati, nonché gestire l'apertura di persiane o tapparelle. È possibile programmare delle scene, ovvero configurazioni predefinite di azioni o comportamenti che uno o più dispositivi possono eseguire in risposta a specifiche condizioni o *trigger*. Inoltre, l'applicazione, offre la possibilità di controllare attuatori e sensori.

Per gestire le interazioni l'app sfrutta diversi moduli: il modulo utilizzato per la gestione dei dispositivi di citofonia si chiama Vimar Session initiation protocol (SIP) Software development kit (SDK) e viene utilizzato per permettere ad un client SIP di comunicare con un SIP server.

Il modulo utilizzato per il mondo domotico si chiama IPConnector ed è un SDK scritto in Kotlin Multiplatform che sfrutta l'omonimo protocollo di comunicazione proprietario di Vimar descritto successivamente.

IPConnector

Il protocollo IP Connector è finalizzato alla comunicazione bidirezionale tra un *server* e uno o più *client* attraverso un collegamento di rete (sia locale sia remoto) per svolgere le seguenti funzioni:

- Identificare automaticamente la presenza di *server* in rete (solo locale);
- Registrare il *client* come interlocutore autenticato nel *server*;
- Ricevere informazioni sulla struttura e risorse esportate dal *server*;
- Inviare comandi e richieste di stato in modo sincrono;
- Ricevere informazioni di cambio stato in modo asincrono;

Il protocollo prevede inoltre meccanismi di protezione della comunicazione e di gestione di particolari situazioni (cambio configurazione di rete, interruzione e ripristino della comunicazione, *timeout* etc...) che richiedano un corretto ripristino del canale di comunicazione.

Elementi necessari affinché un *client* di terze parti possa registrarsi come interlocutore del *server* sono l'identificativo univoco assegnato da Vimar alla terza parte, il cosiddetto *Third-Party Tag*, e la chiave di cifratura Rivest-Shamir-Adleman (RSA) privata concordata tra Vimar e la terza parte.

La comunicazione avviene attraverso un canale di tipo WebSocket, in connessione locale, su una porta stabilita tra *server* e *client* che deve essere protetta tramite cifratura Secure Sockets Layer (SSL); per garantire la sicurezza della comunicazione contro possibili repliche di pacchetto. In questo modo il *server* può dialogare con più *client* contemporaneamente, ognuno su una porta ad esso dedicata. Il protocollo di comunicazione prevede l'incapsulamento delle informazioni in pacchetti dati in formato JavaScript Object Notation (JSON).

Le informazioni relative ai sottosistemi esportati dai *gateway* sono strutturate secondo la specifica delle System function (SF) Vimar. Le SF sono definite da:

- Un tipo che identifica la tipologia di funzione implementata (es. Luce, Persiana, Sensore, etc...);
- Un nome definito durante la configurazione;
- Una lista di System function element (SFE);

Le SFE sono definite da:

- Un tipo che indica la modalità d'accesso;
- Una lista di valori che può assumere;

Ad esempio una luce che non prevede la funzione di regolazione della luminosità sarà rappresentata dalla SF *Light* e la lista di SFE conterrà un SFE per leggere la proprietà On/Off ed un SFE per modificarla. Una luce che prevede la regolazione della luminosità è rappresentata dalla medesima SF però la lista di SFE avrà, in aggiunta alle precedenti, un SFE per leggere la luminosità della luce ed uno per modificarla.

Dall'inizio dello sviluppo dell'applicazione View si è subito reso evidente come il componente IPConnector fosse ideale da gestire tramite un SDK comune per tutte le piattaforme realizzato utilizzando *Kotlin Multiplatform*.

3.1.1 La proposta

Vista la continua diffusione dei dispositivi indossabili, risulta comodo avere un'applicazione per controllare i dispositivi di casa dal polso. La mia proposta di stage richiedeva di sviluppare da zero un'applicazione per il sistema operativo WearOS che permettesse all'utente di effettuare semplici ma efficaci interazioni con il sistema Internet of Things (IoT) Vimar.

3.2 Obiettivi

3.2.1 Classificazione

Gli obiettivi del progetto verranno classificati utilizzando la seguente notazione:

O[TIPO]-[NUMERO]

Dove la O sta per obiettivo, il tipo sarà uno tra i seguenti:

- O per gli obiettivi obbligatori, quindi vincolanti poiché richiesti dal committente;
- D per dei traguardi desiderabili cioè non vincolanti o strettamente necessari, ma dal riconoscibile valore aggiunto;
- F per quelli facoltativi, obiettivi non vincolanti il cui valore aggiunto non è particolarmente significativo;

Il numero serve come identificativo per l'obiettivo.

3.2.2 Definizione degli obiettivi

Gli obiettivi della proposta di stage erano i seguenti:

- OO-1: Competenza nello sviluppo di un'applicazione per WearOS tramite framework Android;
- OO-2: Capacità di analisi al fine di assegnare a determinate esigenze i corrispettivi requisiti;
- OO-3: Progettazione di un'applicazione mobile a partire dai requisiti;
- OO-4: Comprensione del sistema IoT Vimar;
- OD-1: Implementazione dei test per il software prodotto;
- OD-2: Scrittura della documentazione per il prodotto realizzato;
- OF-1: Implementazione di *Google assistant* con l'applicazione;

3.3 Modalità di svolgimento dello stage

Lo stage è stato svolto in presenza, dal 4 settembre 2023 al 27 ottobre 2023 negli uffici ricerca e sviluppo di Vimar a Padova. L'orario di lavoro era flessibile, potevo cominciare dalle 8:00 alle 8:30, la pausa pranzo iniziava alle 12:30 e aveva una durata variabile, minimo un'ora fino ad un massimo di due. Il pomeriggio si continuava a lavorare fino al raggiungimento delle 8 ore lavorative.

3.4 Pianificazione delle attività

3.4.1 Pianificazione iniziale

Prima dell'inizio dello stage sono state pianificate le attività da svolgere con cadenza settimanale:

- **Prima settimana (40 ore) - dal 04/09 al 08/09**
  - Verifica delle credenziali e degli strumenti di lavoro assegnati;
  - Studio dell'infrastruttura Vimar esistente;
  - Formazione sulle tecnologie adottate;

  - Analisi dei requisiti;
  - Discussione dei requisiti raccolti

- **Seconda settimana (40 ore) - dal 11/09 al 15/09**
  - Ricerca sul protocollo utilizzato per la comunicazione tra *smartphone* e *smartwatch*;

- **Terza settimana (40 ore) - dal 18/09 al 22/09**
  - Progettazione dell'app per *smartwatch* in modo che possa interfacciarsi con l'app esistente;

- **Settimane dal 25/09 al 21/10**
  - Implementazione *software* dell'app progettata;
  - *Unit testing* e *bug fixing*;

- **Ottava settimana (40 ore) - dal 23/10 al 27/10**
  - Scrittura della documentazione;
  - Presentazione del progetto svolto;

Capitolo 4

Analisi dei requisiti

Descrizione del lavoro svolto per determinare i requisiti da implementare nell'applicazione partendo dalle esigenze

4.1 Esigenze

Un'esigenza rappresenta la necessità espressa dagli *Stakeholder* e motivata dal business. Una necessità può avere anche una forma non funzionale (*performance*, *design*, vincolo) qualora sia importante al fine della caratterizzazione di quanto richiesto ed è a tutti gli effetti una visione del *business*[1]. All'inizio del progetto mi sono state illustrate le esigenze da soddisfare consultabili nella tabella 4.1

4.2 Requisiti

Un requisito può essere definito come segue[2]:

- Una condizione necessaria ad un utente per risolvere un problema o raggiungere un obiettivo;
- Condizione che deve essere soddisfatta o posseduta da un sistema per adempiere ad un obbligo;
- Descrizione documentata di una condizione di una delle due tipologie precedenti;

I requisiti verranno identificati come segue:

R[Importanza][Tipologia][Codice]

Dove:

- **Importanza**:
  - **1**: Rappresenta un requisito obbligatorio in quanto primario e fondamentale;

[1] *System Engineering Body of Knowledge*. URL: https://sebokwiki.org.
[2] «IEEE Standard Glossary of Software Engineering Terminology». In: *IEEE Std 610.12-1990* (1990), pp. 1–84. DOI: 10.1109/IEEESTD.1990.101064.

Tabella 4.1: Esigenze

ID Esigenza	Descrizione
ESG01-010	Si necessita la creazione di un'app per *smartwatch* come *companion* dell'app View
ESG01-020	Si richiede che possa comandare tutti i dispositivi comunemente controllabili dallo *smartphone*
ESG01-030	Si richiede che da *smartwatch* si possano comandare gli oggetti tramite comandi vocali
ESG01-040	Si richiede di poter ricevere le notifiche di stato
ESG01-050	Si richiede che gli oggetti abbiano uno stato *"live"*
ESG01-060	Si richiede di poter usare lo *smartwatch* senza la necessità di comunicare con lo *smartphone*
ESG01-070	Si richiede di poter rispondere alle video-citofonate
ESG01-080	L'app per *smartwatch* deve potersi configurare tramite l'app per *smartphone*

  - **2**: Identifica un requisito desiderabile, non necessario ma con un valore aggiunto riconoscibile;
  - **3**: Un requisito opzionale, il cui valore aggiunto non è significativo;

- **Tipologia**:
  - **F**: Rappresenta un requisito funzionale quindi che definisce una funzione di un sistema o di alcuni suoi componenti;
  - **Q**: Un requisito qualitativo che deve garantire la qualità di certi aspetti del prodotto;
  - **V**: Vincolo, significa che il requisito deve assicurare che venga rispettato un determinato vincolo;
- **Codice**: identificativo di ogni requisito;

Dopo aver analizzato le esigenze richieste ho stilato una tabella (4.2) contenete i requisiti per soddisfarle.

4.2.1 Analisi delle esigenze

Inizialmente era previsto lo sviluppo di un'applicazione che sfruttasse l'app View installata su *smartphone* per ottenere dati ed eseguire operazioni. Questa soluzione avrebbe richiesto la creazione di servizi sullo *smartphone* che esponessero i dati ed i comandi da eseguire al dispositivo indossabile. Inoltre sarebbe stata necessaria la connessione tra *smartphone* e *smartwatch* per poter utilizzare l'app. Per ovviare a questi svantaggi ho deciso di sviluppare l'applicazione in modo che sfruttasse il *cloud* per comunicare con i dispositivi remoti e non IP Connector come fa l'app View. In questo modo l'applicazione per *smartwatch* può funzionare in maniera autonoma, senza la necessità che telefono ed orologio siano connessi.

Vantaggi dell'app standalone

Utilizzando un'applicazione *standalone* non è necessario che l'utente abbia l'app View installata sul telefono. Inoltre essendo lo *smartwatch* un dispositivo con capacità energetiche limitate andavano valutati anche i consumi energetici; per le chiamate di rete utilizzate per ricevere dati ed eseguire comandi sui dispositivi remoti:

- **se il telefono e lo *smartwatch* sono connessi** grazie ad una funzionalità offerta dal sistema operativo, qualsiasi richiesta di rete eseguita dall'orologio verrà inoltrata allo *smartphone* che dopo l'esecuzione inoltrerà la risposta allo *smartwatch*; consentendo quindi di ridurre il consumo energetico del dispositivo indossabile.
- **Se lo *smartphone* non è connesso all'orologio** allora risulta necessario che lo *smartwatch* sia provvisto di una connessione alla rete.

Autenticazione

Per quanto riguarda il processo di autenticazione il *logout* può essere effettuato direttamente dall'applicazione. Il *login* invece richiede l'inserimento di dati da parte dell'utente, operazione che per molti utenti risulterebbe scomoda se effettuata su *smartwatch*. Dato che l'operazione, nella maggior parte dei casi, andrà eseguita sporadicamente ho deciso che il *login* si effettua tramite lo *smartphone*, quindi è necessario che quando l'utente effettua l'accesso i dispositivi siano connessi.

Dispositivi controllabili

I dispositivi indossabili, date le dimensioni, non permettono di eseguire comodamente operazioni come fanno ad esempio i telefoni, non è quindi corretto trasferire tutte le funzionalità dell'applicazione *smartphone* nell'app per *smartwatch* altrimenti si rovinerebbe l'esperienza utente. Tramite il dispositivo indossabile devono essere esposte solamente le funzionalità più utili e veloci, per questi motivi non tutti i dispositivi controllabili tramite app View sono controllabili tramite *smartwatch* e le operazioni eseguibili su di essi sono state filtrate in modo da mantenere solamente quelle più utili e veloci.

Tabella 4.2: Requisiti

ID Requisito	Descrizione	Fonte
R1V1	L'applicazione deve essere sviluppata per il sistema operativo WearOS	ESG01-010
R1F1	L'applicazione deve apparire come *companion* dell'app View negli *store*	ESG01-010
R1F2	L'utente deve poter accendere le luci	ESG01-020
R1F3	L'utente deve poter spegnere le luci	ESG01-020
R1F4	L'utente deve poter aumentare la luminosità di una lampadina, se il dispositivo risulta compatibile	ESG01-020

R1F5	L'utente deve poter ridurre la luminosità di una luce, se risulta compatibile	ESG01-020
R1F6	L'utente deve poter aprire le tapparelle	ESG01-020
R1F7	L'utente deve poter chiudere le tapparelle	ESG01-020
R1F8	L'utente deve poter aumentare il livello di apertura di una tapparella	ESG01-020
R1F9	L'utente deve poter diminuire il livello di apertura di una tapparella	ESG01-020
R2F10	L'utente deve poter eseguire una scena	ESG01-020
R1F11	L'utente deve poter accendere un dispositivo di climatizzazione	ESG01-020
R1F12	L'utente deve poter spegnere un dispositivo di climatizzazione	ESG01-020
R2F13	L'utente deve poter selezionare la modalità di un dispositivo di climatizzazione	ESG01-020
R2F14	L'utente deve poter aumentare la temperatura obiettivo di un dispositivo di climatizzazione	ESG01-020
R2F15	L'utente deve poter diminuire la temperatura obiettivo di un dispositivo di climatizzazione	ESG01-020
R1F16	L'utente deve poter visualizzare lo stato di accensione delle luci in tempo reale	ESG01-050
R1F17	L'utente deve poter visualizzare la luminosità impostata su una lampadina compatibile in tempo reale	ESG01-050
R1F18	L'utente deve poter visualizzare il livello di apertura di una tapparella in tempo reale	ESG01-050
R1F19	L'applicazione deve mostrare la temperatura misurata da un dispositivo di climatizzazione in tempo reale	ESG01-050
R1F20	L'applicazione deve mostrare la temperatura obiettivo di un dispositivo di climatizzazione in tempo reale	ESG01-050
R1F21	L'utente deve poter vedere la modalità di un dispositivo di climatizzazione in tempo reale	ESG01-050
R1F22	L'applicazione deve essere utilizzabile anche se lo *smartwatch* non è connesso allo *smartphone*	ESG01-060
R1V2	Per ricevere gli stati dei dispositivi l'applicazione deve sfruttare la libreria kiot	ESG01-060

R1V3	Per eseguire comandi sui dispositivi l'applicazione deve sfruttare la libreria kiot	ESG01-060
R1V4	L'autenticazione dell'utente deve essere effettuata su *smartphone*	ESG01-080
R3F23	Eventuali notifiche provenienti dai dispositivi devono essere mostrate all'utente	ESG01-040
R1F24	L'utente deve poter visualizzare gli impianti disponibili per l'account collegato	Interno
R1F25	L'utente deve poter disconnettere l'account dall'applicazione	Interno

Capitolo 5

Progettazione e codifica

Descrizione dell'architettura e dei design patterns utilizzati per lo svliuppo dell'applicazione

5.1 Tecnologie e strumenti

Di seguito vengono illustrate le tecnologie e gli strumenti utilizzati per lo sviluppo.

Android Studio

Android studio è un Integrated development environment (IDE) gratuito sviluppato da JetBrains ottimizzato per la creazione di applicazioni Android native.

Bitbucket

Bitbucket permette di caricare in *cloud* progetti basati su Version control system (VCS) mercurial o git.

Git

VCS gratuito ed *open source* che permette di mantenere traccia delle modifiche apportate ad un insieme di file e facilita il ripristino di vecchie versioni o la risoluzione di conflitti.

Kotlin

Kotlin è un linguaggio di programmazione *open-source* sviluppato dall'azienda di *software* JetBrains. Kotlin si basa sulla Java Virtual Machine (JVM) e viene utilizzato principalmente per lo sviluppo di applicazioni Android.

Android SDK

L'Android SDK è un insieme di strumenti e risorse forniti da Google per gli sviluppatori al fine di creare applicazioni Android. Questo kit fornisce una solida base per sviluppare, testare e distribuire applicazioni Android per una vasta gamma di dispositivi, inclusi smartphone, tablet, smartwatch, televisori e altro ancora. L'SDK offre un insieme completo di librerie e Application Programming Interface (API) per l'accesso alle

funzionalità del sistema, come la fotocamera, i sensori, la connettività di rete, la geo-localizzazione e molto altro. Gli sviluppatori possono utilizzare queste API per creare applicazioni ricche di funzionalità. Inoltre l'SDK fornisce un insieme di risorse grafiche per progettare l'interfaccia utente delle applicazione Android.

Material Design 3.0

Il Material Design è un design sviluppato da Google che fornisce delle linee guida da seguire per sviluppare interfacce che siano accattivanti, accessibili e che assicurino una piacevole esperienza utente.

5.2 Architettura di un sistema

L'architettura di un sistema è la suddivisione del sistema in componenti, la disposizione di essi e la loro comunicazione. Una buona architettura ha lo scopo di facilitare la realizzazione, distribuzione, estensione, manutenzione e validazione di un sistema.

5.2.1 Qualità di un'architettura

Le qualità di un'architettura *software* dovrebbero essere misurabili in modo da poterne evidenziare eventuali cambiamenti nel tempo, esse sono:

- **Sufficienza**: cioè un sufficiente grado di soddisfacimento dei requisiti
- **Comprensibilità**: l'architettura dovrebbe essere compresa dagli *stakeholders*
- **Modularità**: dovrebbe essere presente una buona scomposizione in componenti, cioè parti chiare e distinte tra loro che non svolgono compiti sovrapposti
- **Robustezza**: l'architettura dovrebbe essere indipendente dal formato dei dati in input, devono essere i dati ad adattarsi al sistema e non viceversa
- **Flessibilità**: cioè permettere una facile manutenzione adattiva ed evolutiva
- **Riusabilità**: che permette il riuso di componenti

Per raggiungere questi risultati una buona architettura deve quindi promuovere i concetti di incapsulamento e basso accoppiamento ma elevata coesione tra le componenti.

Basso accoppiamento

L'accoppiamento viene definito come il grado in cui un componente *software* è legato ad altri, quindi nel caso di due classi è quanto una delle due classi dipende dall'altra. Se una classe possiede un elevato grado di accoppiamento con altre significa che se si vuole utilizzare la classe sarà necessario utilizzare anche tutte le classi accoppiate ad essa.

Elevata coesione

Con coesione ci si riferisce a come le parti di un sistema sono legate ma nonostante ciò, ognuna di esse ha un singolo e ben definito scopo. Si ha molta coesione se tutte le parti di un sistema collaborano per raggiungere un obiettivo comune ma ognuna di esse si limita a svolgere un singolo scopo senza l'implementazione di funzionalità non legate ad esso.

Incapsulamento

L'incapsulamento è la capacità di nascondere dati, processi e logiche utilizzate dal *software*. Un alto incapsulamento permette di poter utilizzare una classe conoscendo solamente la sua interfaccia, senza bisogno di sapere quali siano i dettagli implementativi della classe.

5.3 Principi di un'architettura pulita

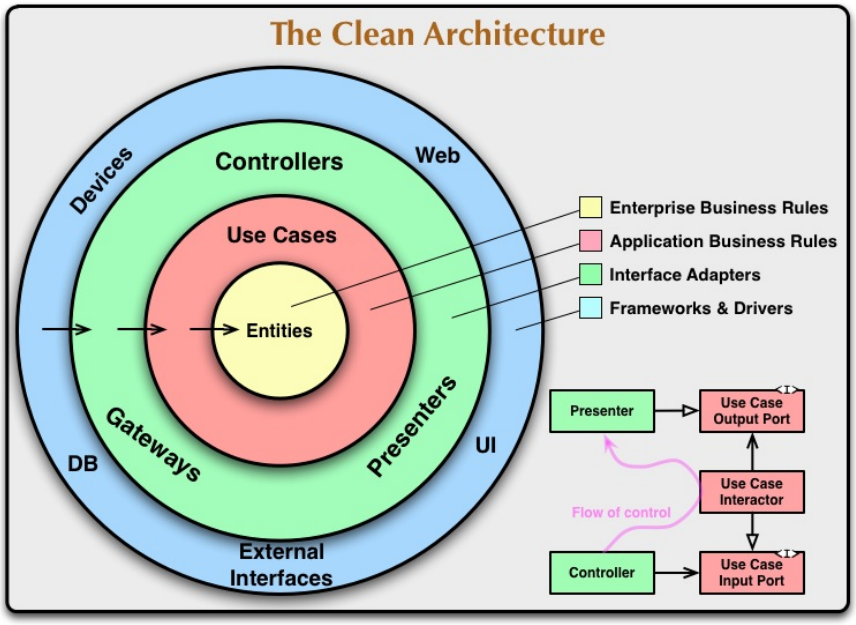


Figura 5.1: Diagramma delle dipendenze per un'architettura pulita

Una *Clean Architecture* come illustrato da Robert C. Martin[1] si prospetta di creare sistemi che siano:

- Indipendenti da framework: l'architettura non deve essere dipendente dall'esistenza o meno di alcune librerie ma quest'ultime andrebbero usate come strumenti che si adattano al nostro *software* e non il contrario;
- Testabili: la logica di business dovrebbe essere testabile senza UI, database o alcun elemento esterno;
- Indipendenti da UI: l'interfaccia utente dovrebbe cambiare facilmente senza che sia richiesto il cambiamento della *business logic*;
- Indipendenti dalle fonti di dati: si dovrebbe essere in grado di cambiare le fonti dei dati a piacimento, senza dover modificare le logiche di business. Una buona architettura adatta i dati ottenuti alle proprie strutture di *business logic* e non viceversa.

[1] Robert C. Martin. *The Clean Code Blog*. URL: https://blog.cleancoder.com.

5.3.1 Regola delle dipendenze

La regola delle dipendenze viene illustrata nell'immagine 5.1, dice che le dipendenze nel codice dovrebbero puntare solo verso l'interno, più in particolare nulla di ciò che viene dichiarato all'interno dei cerchi più esterni dovrebbe essere nominato in quelli più interni. Analizziamo quindi l'immagine 5.1:

- *Entities*: il cerchio più interno, rappresenta le regole di business ovvero regole che definiscono o limitano alcuni aspetti di un determinato business. Queste regole non cambiano molto dopo essere state definite e non dovrebbero cambiare se cambia qualcosa nei cerchi più esterni. Le entità rappresentano la codifica di queste regole, possono essere classi che definiscono solamente metodi, solamente dati o entrambi. Nel caso di sistemi molto grandi queste classi dovrebbero poter essere utilizzate da più applicazioni.
- *Use Cases* o casi d'uso rappresentano le regole di business utilizzate dall'applicazione che si sta sviluppando, essi dirigono il flusso di dati da e verso le entità. Questa parte del *software* non è influenzata da cambiamenti nelle fonti di dati e cambiamenti nei casi d'uso non costringono a cambiamenti nelle entità. Gli *use case* dovrebbero cambiare se cambiano le operazioni effettuate dall'applicazione.
- *Interface adapters*: componenti software che si occupano di convertire i dati ottenuti dall'esterno nella forma più conveniente per entità e casi d'uso. Vale anche il contrario cioè convertono i dati utilizzati all'interno dell'applicazione nel formato più conveniente per fonti di dati esterne o per eventuali interfacce utente.
- *Frameworks and Drivers*: il cerchio più esterno in cui vengono tenuti framework e sorgenti di dati. Generalmente dovrebbe contenere solo il codice necessario per comunicare con i cerchi più interni. Tenere questo genere di cose nel cerchio più esterno ci permette di poterle cambiare a nostro piacimento senza dover modificare gran parte dell'applicazione.

5.4 Principi SOLID

L'applicazione della *Clean Architecture* aiuta ad applicare i principi SOLID, principi che determinano delle linee guida da seguire per la realizzazione di *software* che sia facile da estendere, mantenere e comprendere, obiettivi condivisi da un'architettura pulita.

- **S**: *Single Responsibility Principle*
- **O**: *Open/Closed Principle*
- **L**: *Liskov Substitution Principle*
- **I**: *Interface Segregation Principle*
- **D**: *Dependency Inversion Principle*

Single Responsibility Principle

Il *Single Responsibility Principle* afferma che un modulo dovrebbe avere una sola ragione per cambiare, la modifica di un modulo dovrebbe avere origine da una singola e ben definita regola di business e non molteplici. Applicare questo principio permette la realizzazione di *software* modulare, comprensibile e flessibile poiché assegnando singole responsabilità alle componenti queste risulteranno distinte e senza compiti sovrapposti, inoltre in caso di cambiamenti o aggiustamenti sarà necessario modificare un singolo componente.

Open/Closed Principle

Il principio di Apertura/Chiusura afferma che le componenti *software* dovrebbero essere aperte all'estensione ma chiuse alla modifica. Significa che una componente deve permettere l'aggiunta di funzionalità senza la necessità di modificare il codice esistente. Questo implica che i moduli devono essere progettati per essere immutabili e quando un requisito cambia il modulo responsabile deve essere esteso e non deve essere modificato del codice funzionante. Questo principio rende il codice maggiormente estendibile, basta estendere le classi esistenti con le nuove funzionalità senza dover modificare e rischiare di rendere il codice esistente non funzionante.

Liskov Substitution Principle

Il principio di sostituzione di Liskov sancisce che un oggetto dovrebbe poter essere sostituito da un suo sotto-tipo senza rompere il programma. Più formalmente, definita $\sigma(x)$ una proprietà di un oggetto x di tipo T. Allora $\sigma(y)$ dovrebbe essere vera per ogni oggetto y di tipo S con S sottotipo di T. In simboli:

$$S \leq T \implies \forall x : T.\sigma(x) \implies \forall y : S.\sigma(y)$$

Il principio è importante poiché permette di aspettarsi un comportamento simile lungo tutta la gerarchia di derivazione da parte di funzionalità definite nella classe base. In particolare una classe figlia deve sempre essere in grado di processare le stesse richieste e produrre lo stesso tipo di risultato che ci si aspetta dalla classe padre. Alcuni recenti linguaggi orientati agli oggetti hanno introdotto delle regole basate sul principio di sostituzione per consentire l'utilizzo del polimorfismo in maniera sicura. Applicando il principio si facilita la collaborazione poiché ogni sviluppatore può creare nuove classi conformi alle stesse interfacce senza influenzare la corretta esecuzione delle funzionalità di base.

Interface Segregation Principle

Il principio di segregazione delle interfacce sancisce che nessuna componente *software* dovrebbe dipendere da codice che non usa. Una classe dovrebbe quindi implementare solo i metodi di cui ha bisogno per cui andrebbero definite tante interfacce specifiche che richiedono l'implementazione di pochi metodi. Questo principio aumenta la coesione poiché ogni classe implementa solamente le funzionalità di cui ha bisogno, inoltre rende il codice più facile da comprendere poiché le interfacce si concentrano sull'esecuzione di singoli comportamenti.

Dependency Inversion Principle

Il principio di inversione delle dipendenze afferma che nessun modulo di alto livello dovrebbe dipendere da moduli di basso livello e che, ad entrambi i livelli, non ci devono essere dipendenze dirette ma andrebbero usate delle astrazioni come le interfacce. L'astrazione dovrebbe appartenere al modulo di livello più alto e permette in questo modo di scrivere del codice meno dipendente dall'implementazione. Il principio ha lo scopo di ridurre l'accoppiamento tra le classi. Inoltre applicando questo principio è possibile sostituire le classi di livello più basso senza influenzare i moduli di livello più alto.

5.5 Architettura dell'app

Per sviluppare l'applicazione ho deciso di adottare un pattern architetturale frequentemente utilizzato per lo sviluppo Android, il pattern MVVM. L'obiettivo era mantenere separati i concetti di *business logic* e presentazione.

Il pattern organizza il codice in 3 componenti:

- ***Model***: il modello rappresenta i dati e la *business logic* dell'applicazione, in questo caso viene rappresentato dagli *use cases*;
- ***View***: l'interfaccia con cui interagisce l'utente, la sua unica responsabilità dovrebbe essere quella di mostrare i dati e gestire gli input dell'utente. La *view* non dovrebbe contenere *business logic* o gestire cambi di stato dell'applicazione;
- ***ViewModel***: è un intermediario tra l'interfaccia e il modello, espone stati che la *view* è in grado di osservare e, in caso questi cambino, avere determinate reazioni. Gestisce inoltre le interazioni dell'utente con l'interfaccia;

Il pattern MVVM al contrario di quelli usati in precedenza per lo sviluppo di applicazioni Android mi ha permesso di riutilizzare il *ViewModel* in diverse parti dell'applicazione poiché non c'è un rapporto uno ad uno tra interfaccia e modello ma un rapporto uno a molti.

5.5.1 Moduli e pacchetti dell'applicazione

Durante lo sviluppo dell'applicazione ho applicato i principi della *Clean Architecture* descritta precedentemente al fine di ottenere un'applicazione che sarà facile da mantenere, estendere e testare. Per ottenere un risultato accettabile l'applicazione si compone di tre moduli principali:

- ***core***: all'interno di questo modulo vengono definite le classi che rappresentano le entità ed i casi d'uso e le interfacce per le fonti di dati usate dai moduli di livello più basso;
- **python**: il modulo contente tutta la logica per ottenere e salvare i dati utilizzando la libreria Python;
- **app**: il modulo contente la logica per il funzionamento dell'app e tutto ciò che riguarda il *presentation layer*;

Grazie a questa suddivisione è possibile decidere quali componenti sono utilizzabili all'interno di un singolo modulo, non integrando quindi le classi definite nei moduli app e python all'interno del modulo *core* si può avere la certezza che le entità e i casi d'uso non dipendano da componenti di livello più basso.

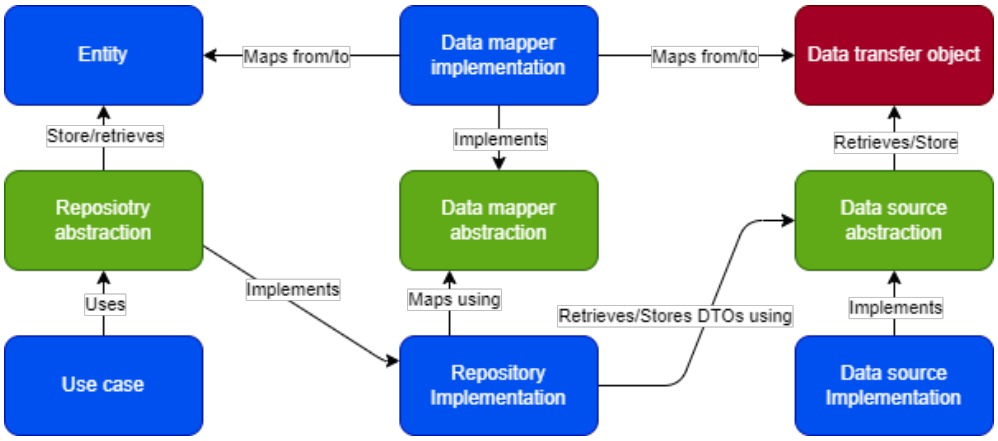


Figura 5.2: Rappresentazione di come le componenti dei moduli *core* e *python* interagiscono tra loro

Modulo core

Il modulo *core* contiene i seguenti pacchetti:

- ***data***: all'interno di questo pacchetto vengono definite le regole per ottenere e salvare dati, esso contiene altri 4 pacchetti:
  - ***repository***: al suo interno vengono implementate le interfacce dei repository definite nel pacchetto *com.vimar.core.domain.repository*;
  - ***source***: contiene le interfacce da che andranno implementate da ogni fonte di dati;
  - ***dto***: al suo interno vengono definiti i *Data Transfer Objects* utilizzati per salvare le informazioni ottenute dalle sorgenti di dati;
  - ***mapper***: contiene i *mapper* che hanno il compito di trasformare i *dto* in entità che potranno essere utilizzate dall'applicazione;
- ***domain***: all'interno di questo pacchetto viene definito tutto ciò che è inerente alla logica di business, al suo interno si trovano altri 3 pacchetti:
  - ***entity***: contiene le entità che rappresentano le regole di business;
  - ***repository***: vengono definite le interfacce dei repository che hanno il compito di salvare e ottenere dati come entità;
  - ***useCase***: vi si trovano i casi d'uso che verranno utilizzati dall'applicazione;

Modulo python

All'interno del modulo python è contenuto il pacchetto *data* nel quale vengono implementate le interfacce per ottenere e salvare dati definite all'interno del modulo *core*.

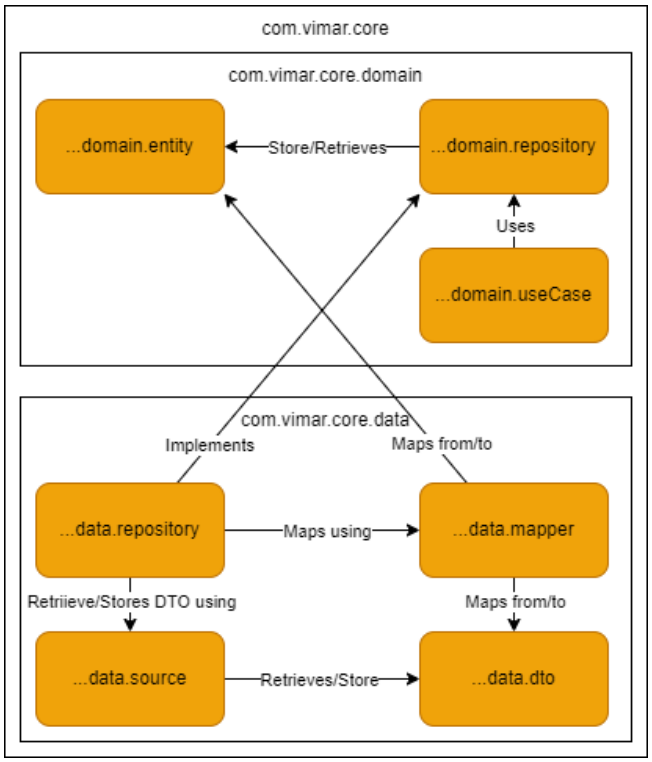


Figura 5.3: Rappresentazione grafica dei pacchetti all'interno del modulo *core*

Modulo app

All'interno del modulo app sono presenti i seguenti pacchetti:

- **framework**: contiene alcune classi create per il corretto funzionamento dell'applicazione e per eseguire la *dependency injection*;
- ***activity***: contiene le attività utilizzate dall'applicazione;
- ***presentation***: al suo interno vengono implementate le componenti utilizzate per gestire le User interface (UI) e le interazioni da parte dell'utente;
- ***tiles***: vi si possono trovare le implementazioni dei servizi utilizzati per aggiornare o gestire eventuali interazioni con le *tile*;
- ***worker***: contiene i lavori che l'app esegue in *background*;

5.6 Design Pattern utilizzati

5.6.1 Dependency Injection

La *dependency injection* è un *design pattern* che prevede il passaggio delle dipendenze a un oggetto anziché consentirgli di costruirle autonomamente. L'obiettivo è separare i concetti di costruzione ed utilizzo di un oggetto, creando così programmi con un basso livello di accoppiamento. I vantaggi nell'utilizzare questa tecnica sono i seguenti:

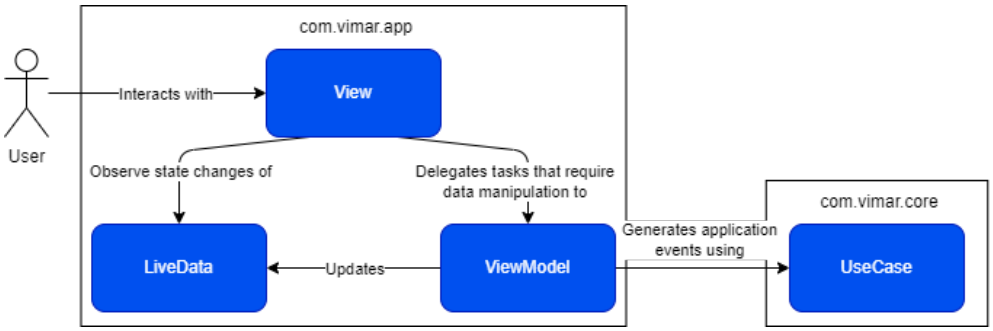


Figura 5.4: Rappresentazione di come il modulo app è collegato al modulo core e di conseguenza a quello Python

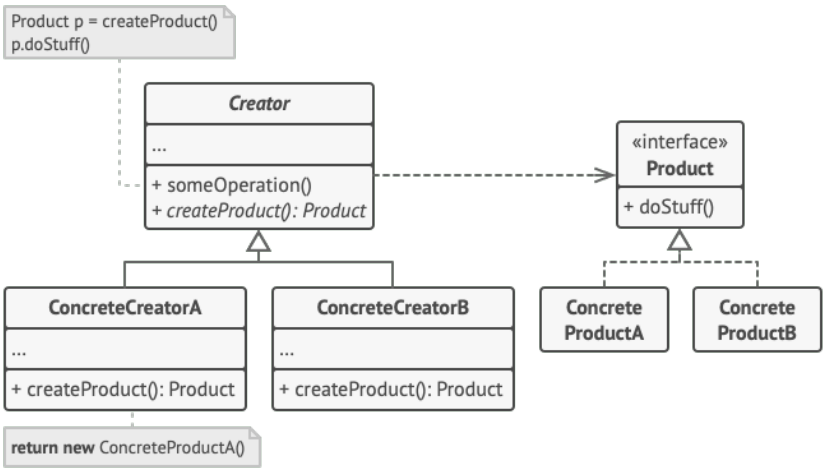


Figura 5.5: Schema logico del *Factory Method*[2]

- La classe risulta più indipendente poiché non è tenuta a conoscere il modo in cui costruire gli oggetti di cui ha bisogno.
- È possibile dichiarare l'interfaccia implementata dalla dipendenza e non la classe di implementazione. Questo ci consente di cambiare il servizio utilizzato durante l'esecuzione, senza la necessità di ricompilare.
- Le classi diventano più facili da testare. Infatti si possono simulare i comportamenti attesi dai servizi iniettati utilizzando dei *mock*.

All'interno dell'applicazione, la *dependency injection* è stata impiegata per la costruzione delle sorgenti di dati. Queste vengono inizializzate all'avvio, salvate in un modello comune e successivamente fornite alle classi che ne fanno uso. Allo stesso modo gli *use case* vengono creati e successivamente passati ai *ViewModel*.

5.6.2 Factory Method

Il *Factory Method* è un *design pattern* creazionale che fornisce un'interfaccia per la creazione di un oggetto in una classe madre, ma permette alle sottoclassi di alterare il

tipo di oggetto che verrà creato.[3] Come raffigurato dal grafico in figura 5.5 il *pattern* si compone di 3 classi principali:

- l'interfaccia *Product* che è comune a tutti gli oggetti che possono essere prodotti dal *Creator* e dalle sue sottoclassi;
- le classi *ConcreteProduct*, differenti implementazioni dell'interfaccia *Product*;
- la classe astratta *Creator* che dichiara il metodo di costruzione, cioè una funzione che restituisce un oggetto di tipo *Product*;
- le classi *ConcreteCreator* le quali eseguono l'*override* del metodo di costruzione e restituiscono il tipo di oggetto desiderato.

Utilizzando il *Factory Method* si ottengono i seguenti vantaggi:

- Diminuzione dell'accoppiamento poiché la classe madre non sa con esattezza che oggetto verrà creato, il codice risulterà più modulare e facile da mantenere.
- Maggiore riusabilità del codice, la logica di creazione è isolata in una classe separata e consente di creare nuovi oggetti utilizzando codice già scritto;
- Maggiore estensibilità, perché è possibile aggiungere nuovi tipi di oggetti senza modificare il codice esistente.
- Essendo la logica di creazione degli oggetti confinata in classi specifiche, le modifiche a tale logica possono essere apportate senza influire sul resto del sistema. La manutenzione del codice risulta quindi più semplice poiché viene ridotta la probabilità di introdurre errori in altre parti dell'applicazione.

Il *Factory Method* è ampiamente utilizzato in molti framework e librerie. Ad esempio, in Android, viene utilizzato per la costruzione dei *ViewModel*. Nell'applicazione, ho creato nuovi *ConcreteCreator*, uno per ogni tipo di *ViewModel* implementato, che mi consentono di sfruttare le funzionalità del framework Android utilizzando oggetti creati da me.

Facendo un esempio più concreto all'interno di una UI è possibile dichiarare una variabile per ogni *ViewModel* e inizializzarla utilizzando una funzionalità della libreria *Android KTX*. Questa funzione costruirà un nuovo *ViewModel* se non è mai stato creato altrimenti restituirà il *ViewModel* esistente. Per eseguire l'*injection* delle dipendenze nel *ViewModel* è necessario fornire un'implementazione dell'interfaccia *ViewModelProvider.Factory*, corrispondente alla classe astratta *Creator* descritta in precedenza. L'implementazione ha il compito di costruire il *ViewModel* desiderato passando al suo costruttore tutte le dipendenze necessarie, infine deve ritornare l'oggetto appena creato al chiamante; nel mio caso viene fatta l'*injection* degli *UseCase* utilizzati dal modello.

5.6.3 Pattern Observer

Il pattern *observer* definisce una dipendenza uno a molti permettendo ad un oggetto chiamato *subject* di notificare eventuali cambiamenti di stato ad altri oggetti che lo stanno osservando.

Il pattern è stato utilizzato all'interno dell'applicazione per gestire i cambiamenti di stato delle UI. Ogni interfaccia infatti possiede un *ViewModel* che ha il compito di

[3] *Refactoring Guru*. URL: https://refactoring.guru.

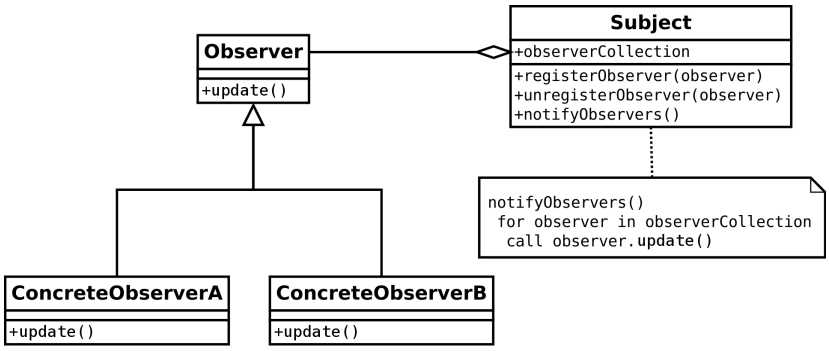


Figura 5.6: Grafico UML rappresentante il funzionamento del pattern *Observer*

gestire le interazioni dell'utente. Dopo aver gestito la richiesta il *ViewModel* aggiorna un oggetto di tipo *LiveData* il quale notifica il cambiamento alla *View* che di conseguenza si aggiorna, come raffigurato dal diagramma 5.4. In questo caso il *subject* è l'oggetto di tipo *LiveData* mentre l'*observer* è la *View* quindi l'interfaccia con cui interagisce l'utente.

5.7 Codifica

Per l'applicazione sono state realizzate quattro schermate principali: accesso, selezione di un impianto, selezione della categoria di dispositivi che si vogliono controllare ed infine una lista con i dispositivi disponibili e con i quali è possibile interagire.

5.7.1 Struttura di un'applicazione WearOS

Prima di vedere come sono composte le schermate è necessario capire le principali componenti di un'applicazione Android.

Activity

L'*activity* è il componente principale di un'applicazione Android e può essere vista come una schermata con cui l'utente può interagire. Un'attività fornisce la finestra sulla quale l'applicazione può disegnare l'interfaccia utente, inoltre permette al sistema di eseguire determinate parti di codice basandosi su specifici stadi del ciclo di vita dell'attività.

Fragment

Un *fragment* rappresenta un'interfaccia e il suo comportamento all'interno di un'attività. Un *fragment* può essere utilizzato da più attività differenti. Inoltre mentre il ciclo di vita di un'attività è gestito dal sistema il ciclo di vita di un *fragment* viene gestito dall'*activity* che lo contiene.

Recycler View

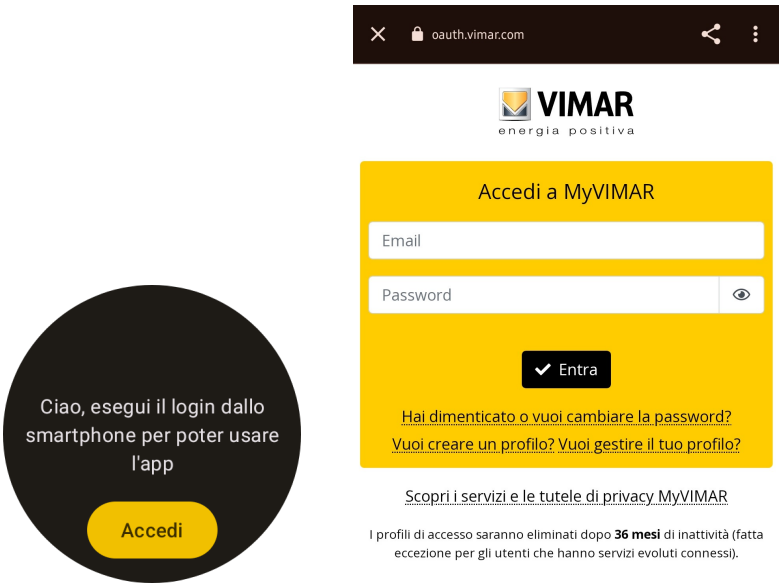
Una *RecyclerView* è un tipo di *View* utilizzato per mostrare liste o griglie di dati. Ogni componente della lista si chiama *ViewHolder*.

Risulta essere molto efficiente poiché ogni *ViewHolder* della lista viene costruito quando risulta visualizzabile da parte dell'utente e distrutto quando non lo è più. Questo consente di non tenere in memoria ogni componente della lista ed il suo layout ma solamente i dati.

Tile

Le *tiles* offrono agli utenti la possibilità di accedere in modo rapido e comodo ad informazioni e funzionalità importanti direttamente dalla schermata principale dello *smartwatch*.

5.7.2 Schermata di login



(a) Interfaccia di *login* dello *smartwatch*

(b) *Web-view* visualizzata sullo *smartphone*

Figura 5.7: Le due interfacce utilizzate per effettuare l'accesso

Al primo avvio dell'applicazione verrà chiesto all'utente di effettuare il login. Sullo *smartwatch* comparirà una schermata contenente un bottone che, se cliccato, farà comparire sullo *smartphone* connesso una *web-view* che consentirà all'utente di inserire le proprie credenziali ed accedere.

Agli avvii successivi dell'app il *login* viene fatto automaticamente e andrà rifatto dall'utente solo dopo il *logout* o quando scade il *refresh token*, la cui durata è variabile e viene decisa dal *provider* del *token*. Nel mio caso scade ogni sei mesi, salvo cambiamenti futuri.

Per la funzionalità di *login* viene utilizzato il protocollo Open Authorization 2.0 (OAuth 2.0) con l'estensione Proof Key for Code Exchange (PKCE). Per prima cosa è necessario ottenere un *authorization code* dal proprio provider OAuth 2.0, in questo caso si trova all'indirizzo *oauth.vimar.com*, inserendo nella richiesta anche il *code challenge* richiesto dall'estensione PKCE per verificare che non ci siano intercettazioni

durante lo scambio dei *tokens*. Dopo aver eseguito con successo il *login* l'utente dovrà dare il consenso per il trasferimento del codice appena ottenuto sullo *smartwatch*. Ora che l'*authorization code* è sull'orologio si possono richiedere *access* e *refresh token* al provider eseguendo una richiesta *HTTP* di tipo POST specificando come parametri nel *body* l'*authorization code* e il codice da verificare che è stato usato in precedenza dall'estensione PKCE. Fatto ciò è possibile salvare in locale i *tokens* ottenuti utilizzando una versione criptata delle *SharedPreferences* del framework Android, così facendo è possibile effettuare in automatico l'accesso senza salvare le credenziali dell'utente.

5.7.3 Schermata per la selezione dell'impianto

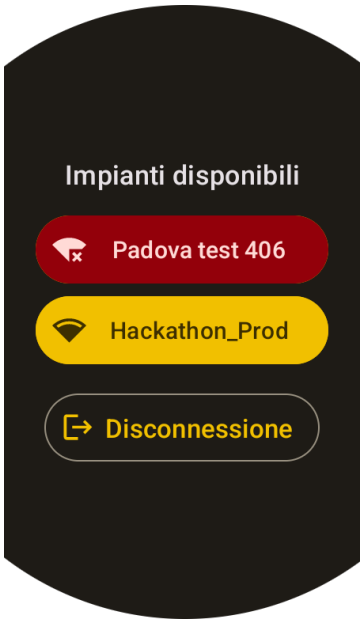


Figura 5.8: Schermata di selezione dell'impianto

Dopo aver effettuato il login si passa al *fragment* che mostra la lista con gli impianti disponibili. In questa schermata l'utente potrà decidere di visualizzare i dispositivi di un impianto online quindi indicato dal colore giallo; gli impianti indicati con il colore rosso sono *offline* oppure non compatibili. Nel caso in cui ci sia un solo impianto disponibile si passa direttamente all'interfaccia successiva. Se la scelta dell'impianto è stata effettuata in precedenza, viene caricato direttamente l'ultimo impianto selezionato, se *online*, di conseguenza si passa direttamente alla schermata delle categorie. Dato che questo *fragment* si trova alla radice dell'app è presente anche il bottone per la disconnessione.

Quando il *fragment* viene creato il *ViewModel* corrispondente comincia a caricare gli impianti disponibili e verifica che siano *online*. Se era stato selezionato un impianto in precedenza e questo risulta *online* si passa direttamente alla schermata delle categorie; altrimenti il *ViewModel* notifica il *fragment* che il caricamento è terminato con successo e quest'ultimo mostra all'utente la lista ottenuta dal *ViewModel*.

Se il caricamento dovesse terminare con uno stato di errore l'utente viene indirizzato ad una schermata che gli illustra come mai il caricamento non è andato a buon fine.

5.7.4 Schermata per la selezione della categoria

Figura 5.9: Schermata di selezione della categoria

Per una migliore organizzazione dei contenuti non vengono mostrati subito tutti i dispositivi disponibili ma viene chiesto all'utente di selezionare quale categoria di *device* desidera visualizzare. La schermata presenta come titolo principale il nome dell'impianto selezionato e sotto di esso il titolo della lista. Ogni voce nell'elenco delle categorie disponibili mostra:

- un'icona rappresentante la categoria stessa;
- il nome della categoria;
- il numero di dispositivi controllabili appartenenti ad essa;

Quando il *fragment* viene creato il *ViewModel* comincia a verificare se ci sono dispositivi disponibili per ogni categoria e aggiorna lo stato della *View* in modo che mostri uno *spinner* per indicare all'utente il caricamento. Dopo aver verificato il numero di dispositivi per ogni categoria il *ViewModel* notifica il successo alla *View* che si aggiorna mostrando la lista delle categorie; verranno mostrate solamente quelle per le quali è stato trovato almeno un dispositivo. L'utente può selezionare una delle categorie tappandoci sopra per passare al *fragment* in cui è possibile controllare i dispositivi.

5.7.5 Schermata per il controllo dei dispositivi

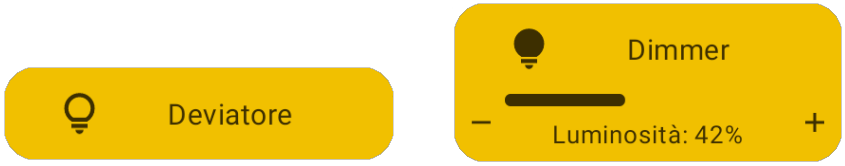
La schermata finale dell'app è quella in cui si possono controllare i dispositivi. Appena viene creato il *fragment* il *ViewModel*, dati l'impianto e la categoria scelta, comincia a caricare i dispositivi disponibili notificando alla *View* lo stato di caricamento. Quando i dispositivi sono pronti il *fragment* viene notificato e mostra all'utente i dispositivi disponibili. La schermata è composta da:

Figura 5.10: Schermata per il controllo dei dispositivi

- il nome dell'impianto scelto;
- il titolo della lista visualizzata;
- se presenti vengono mostrate le *bulk actions* per la categoria di dispositivi scelta;
- la lista con i dispositivi controllabili;

Per mostrare gli elementi della lista viene utilizzata una *recycler view*, ogni tipo di *device* ha un proprio *view holder* e il *binding* viene eseguito basandosi sul tipo di dato prelevato dalla lista.

ViewHolder per le luci



(a) Luce spenta **(b)** Luce *dimmerabile* accesa

Figura 5.11: I *layout* utilizzati per mostrare le luci

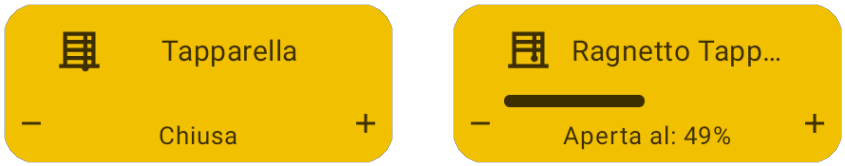
Il *ViewHolder* utilizzato per le luci presenta:

- un'icona che rappresenta lo stato in cui si trova la luce;
- il nome della luce;
- se è possibile variare la luminosità della luce viene visualizzata la luminosità attualmente impostata tramite un testo ed una barra di progresso;

Le operazioni eseguibili sul componente sono le seguenti:

- tappando sul componente è possibile alternare lo stato della luce, se è accesa si spegne e se è spenta si accende;
- se la luce ha luminosità variabile:
  - tenendo premuto il lato destro del componente è possibile aumentare la luminosità;
  - tenendo premuto il lato sinistro del componente è possibile ridurre la luminosità;

ViewHolder per le tapparelle



(a) Tapparella chiusa **(b)** Tapparella aperta

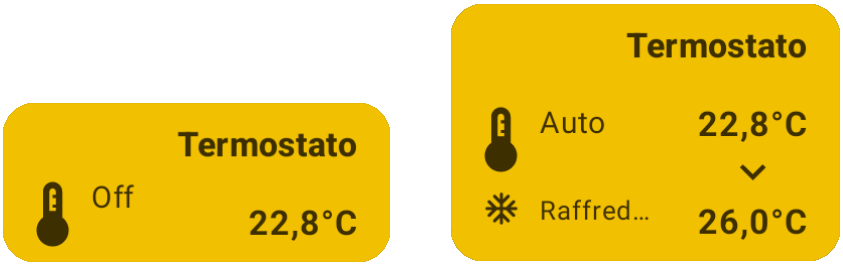
Figura 5.12: I *layout* utilizzati per mostrare le tapparelle

Il componente per mostrare le tapparelle presenta:

- un'icona che indica lo stato della tapparella;
- il nome della tapparella;
- il livello di apertura della tapparella indicato da un testo ("Chiusa" se il livello è uguale a 0) e da una barra di progresso;

Le possibili interazioni sono le seguenti:

- tappando sulla tapparella è possibile invertirne lo stato, se è chiusa si apre se è aperta si chiude;
- tenendo premuto il lato destro del componente è possibile aumentare il livello di apertura;
- tenendo premuto il lato sinistro del componente è possibile ridurre il livello di apertura;
- se la tapparella è in movimento è possibile tappare sul componente per fermarla, lo stato di movimento viene segnalato all'utente da uno spinner al posto dell'icona;



(a) Dispositivo di climatizzazione spento **(b)** Dispositivo di climatizzazione acceso

Figura 5.13: I *layout* utilizzati per mostrare i dispositivi di climatizzazione

ViewHolder per i dispositivi di climatizzazione

Tramite il componente per i dispositivi di climatizzazione è possibile visualizzare:

- un'icona inserita a scopo estetico;
- il nome del dispositivo;
- la temperatura rilevata dal dispositivo;
- se acceso è possibile vedere:
  - la modalità di funzionamento;
  - la modalità di commutazione;
  - la temperatura obiettivo;

È possibile invertire lo stato di un dispositivo tappando sul componente, se è acceso si spegne se è spento si accende.

5.7.6 Tile per le bulk actions

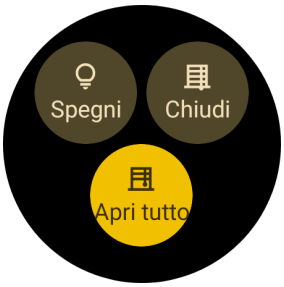


Figura 5.14: Scheda per l'esecuzione delle *bulk actions*

Le *tiles* o schede permettono all'utente di eseguire azioni molto velocemente direttamente dalla schermata principale dell'orologio. Una funzionalità utile da implementare in una di queste schede è l'esecuzione delle *bulk actions* in modo che l'utente possa chiudere o aprire tutte le tapparelle oppure spegnere tutte le luci senza dover aprire l'applicazione.

La scheda presenta tre bottoni ognuno distinto da un'icona e da un'etichetta per indicare l'azione da compiere. Dopo aver premuto su uno dei tre bottoni si avvia un'attività che esegue la *bulk action* corrispondente.

5.7.7 Tile per la temperatura di un ambiente

(a) Stato iniziale della scheda

(b) Layout principale della *tile*

Figura 5.15: Scheda per la visualizzazione della temperatura rilevata da un dispositivo di climatizzazione

Un informazione utile da avere nella schermata principale dello *smartwatch* è la temperatura di una stanza. Quando l'utente aggiunge la scheda gli viene mostrato un bottone per aggiungere il dispositivo dal quale ottenere la temperatura. Dopo che il bottone viene tappato si apre un'attività che mostra l'elenco dei dispositivi di climatizzazione, dopo il tap sul dispositivo desiderato la *tile* si aggiorna mostrando:

- il nome del impianto a cui appartiene il dispositivo;
- il nome del dispositivo;
- la temperatura rilevata;

Per aggiornare la temperatura viene utilizzata una classe del framework Android chiamata *Worker* che ogni 15 minuti si avvia in *background* e ottiene la nuova temperatura, la salva in locale ed infine richiede l'aggiornamento della *tile*. Per limitare i consumi energetici il *Worker* si avvia solamente se il dispositivo ha almeno il 15% della batteria rimanente.

Capitolo 6

Verifica e validazione

In questo capitolo vengono illustrati i test implementati durante lo sviluppo

6.1 Unit test

Un test di unità verifica il corretto funzionamento di una singola unità di codice, ci si riferisce quindi ad una parte del codice specifica ed isolata che può essere testata in maniera indipendente. Se dopo delle modifiche i test non dovessero andare a buon fine verrà segnalata allo sviluppatore la parte di codice non funzionante. Per gli *unit test* è stata utilizzata la libreria JUnit 4.

Tabella 6.1: Test d'unità

Codice	Descrizione	Esito
TU-1	Verificare che un valore superiore a 100 per la luminosità delle luci venga approssimato a 100	Superato
TU-2	Verificare che un valore inferiore a 0 per la luminosità delle luci venga approssimato a 0	Superato
TU-3	Verificare che se viene passato un valore superiore a 100 per la posizione delle tapparelle questo venga approssimato a 100	Superato
TU-4	Verificare che se viene passato un valore inferiore a 0 per la posizione delle tapparelle questo venga approssimato a 0	Superato

6.2 Test dell'interfaccia utente

I test dell'interfaccia utente servono a verificare che l'interfaccia utente funzioni correttamente e assicuri un'esperienza ottimale per l'utente. Per testare l'interfaccia utente dell'applicazione è stato utilizzato il framework ufficiale Android: Espresso. Purtroppo per mancanza di tempo non è stato possibile completare l'implementazione di tutti i test pensati.

Tabella 6.2: Test dell'interfaccia utente

Codice	Descrizione	Esito
TI-1	Verificare che se l'utente tappa un impianto *online* si passi alla schermata delle categorie	Superato
TI-2	Verificare che se l'utente tappa su un impianto *offline* venga mostrato il *toast* di errore	Superato
TI-3	Verificare che se c'è un solo impianto disponibile si passi automaticamente alla schermata delle categorie	Superato
TI-4	Verificare che se c'è un impianto salvato viene mostrata direttamente la schermata delle categorie	Superato
TI-5	Verificare che se non c'è alcun impianto disponibile venga mostrato il relativo messaggio	Non implementato
TI-6	Verificare che se una categoria non contiene dispositivi disponibili non viene mostrata nella lista	Non implementato
TI-7	Verificare che se non ci sono categorie disponibili venga mostrato il corrispondente messaggio	Non implementato
TI-8	Verificare che in caso vengano ottenuti i *token* corretti per l'autenticazione l'utente passi alla schermata degli impianti	Non implementato
TI-9	Verificare che se lo *smartphone* e l'orologio non sono connessi viene mostrato il relativo errore durante l'operazione di *login*	Non implementato
TI-10	Verificare che dopo aver tappato il pulsante *logout* l'utente venga riportato alla schermata di *login*	Non implementato
TI-11	Verificare che dopo il tap sul bottone di *logout* vengono cancellati tutti i dati salvati sull'applicazione relativi all'utente disconnesso	Non implementato

Capitolo 7

Conclusioni

In questo capitolo viene fatta una retrospettiva confrontando il piano di lavoro preventivato con ciò che è stato fatto, vengono esposti gli obiettivi conseguiti e le consocenza acquisite durante lo stage.

7.1 Consuntivo finale

La pianificazione delle attività fatta assieme al tutor aziendale e presentata nella sezione 3.4.1 ha subito alcune variazioni. Innanzitutto mi è stato proposto di poter realizzare un'applicazione che comunicasse con i dispositivi tramite chiamate *HTTP* invece di dover comunicare con l'applicazione per lo *smartphone* perciò in aggiunta allo studio della comunicazione tra telefono ed orologio ho condotto un'analisi sulle due modalità di comunicazione e sono giunto alla conclusione che è meglio un'applicazione *standalone*. Perciò ho dovuto studiare la libreria da utilizzare e come integrare una libreria Python all'interno di un progetto Android.

La codifica ha richiesto più tempo del previsto, di conseguenza non è stato possibile integrare tutti i test previsti poiché si è preferito scrivere la documentazione per il futuro mantenimento ed estensione dell'app.

Tabella 7.1: Consuntivo finale

Attività svolta	Ore Pianificate	Ore effettive	Scostamento
Comprensione delle tecnologie da utilizzare	30	30	0
Verifica delle credenziali e degli strumenti di lavoro assegnati	5	4	-1
Studio dell'infrastruttura Vimar	6	4	-2
Analisi dei requisiti	20	14	-6
Comprensione della libreria vimar per il controllo dei dispositivi via cloud	7	6	-1

Integrazione della libreria Python in Android	2	4	+2
Progettazione dell'app per smartwatch	40	35	-5
Codifica dell'app	160	180	+20
Scrittura della documentazione	33	24	-9

7.2 Raggiungimento degli obiettivi

Degli obiettivi prefissati ad inizio stage purtroppo non è stato possibile portare a termine l'implementazione di tutti i test previsti e l'integrazione di *Google assistant* per mancanza di tempo.

Tabella 7.2: Resoconto obiettivi

Codice	Obiettivo	Esito
OO-1	Competenza nello sviluppo di un'applicazione per WearOS tramite framework Android	Raggiunto
OO-2	Capacità di analisi al fine di assegnare a determinate esigenze i corrispettivi requisiti	Raggiunto
OO-3	Progettazione di un'applicazione mobile a partire dai requisiti	Raggiunto
OO-4	Comprensione del sistema IoT Vimar	Raggiunto
OD-1	Implementazione dei test per il software prodotto	Non raggiunto
OD-2	Scrittura della documentazione per il prodotto realizzato	Raggiunto
OF-1	Implementazione di *Google assistant* con l'applicazione	Non raggiunto

7.3 Retrospettiva sullo stage

Dopo aver terminato lo stage ho eseguito una retrospettiva personale per capire cosa avevo guadagnato da quest'esperienza.

7.3.1 Conoscenze acquisite

Durante lo stage ho avuto modo di apprendere nuove conoscenze e di approfondire alcune di quelle pregresse. In particolare:

- **Android**: in passato mi ero già cimentato nello sviluppo di applicazione native Android, tuttavia grazie allo stage è stato possibile ampliare enormemente le mie conoscenze a riguardo. Ho compreso meglio il funzionamento dell'ecosistema Android, sopratutto di come vengono gestite le operazioni in background.

- **Kotlin**: le mie conoscenze riguardanti il linguaggio sono diventate più solide, ho avuto modo di utilizzare alcune peculiarità del linguaggio che non avevo mai provato come ad esempio le *coroutines* e più in generale il *multi-threading*.
- **Espresso**: non avevo mai utilizzato la libreria per il test UI di Android, ora so come testare le interfacce di un'applicazione in modo da aumentare la qualità del *software* prodotto.

7.3.2 Competenze acquisite

Durante lo stage ho avuto modo di migliorare dal punto di vista professionale, nello specifico:

- **Sviluppo di applicazione mobile**: in passato mi ero già cimentato personalmente nello sviluppo di applicazioni *mobile* native e multi-piattaforma. Durante lo stage è migliorata la mia capacità di progettazione di UI e User experience (UX). Ho inoltre migliorato la mia capacità di progettazione software.
- **Metodologie di lavoro SCRUM**: ho avuto modo di applicare ed entrare in contatto con i metodi di lavoro AGILE ed in particolare il framework SCRUM.
- **Collaborazione**: ho migliorato la mia capacità di lavorare in gruppo, in particolare l'espressione e condivisione delle mie idee in modo che siano facilmente comprensibili dagli altri.
- **Documentazione**: ho imparato a scrivere della buona documentazione in modo che chi lavorerà sull'applicazione in futuro sarà in grado di capire il perché sono state fatte determinate scelte.

7.3.3 Valutazione personale

Mi ritengo soddisfatto dei miglioramenti ottenuti durante il percorso di stage. L'azienda mi ha lasciato lavorare in autonomia sul progetto, concedendomi la libertà di scegliere come strutturare le interfacce, quali funzionalità implementare e come strutturare l'applicazione, il tutto sempre guidato da un collega che ascoltava le mie idee e mi dava suggerimenti su come migliorarle. Ho trovato un ambiente di lavoro molto accogliente pieno di persone pronte ad aiutarmi in caso di difficoltà.

Sono estremamente soddisfatto delle conoscenze e competenze che ho acquisito durante il mio stage. Ritengo quindi che il progetto di stage sia stato un'esperienza straordinariamente positiva e istruttiva, che ha contribuito in modo significativo alla mia crescita professionale.

Acronimi e abbreviazioni

API Application Programming Interface. 16, 17, 40

IDE Integrated development environment. 16, 40

IoT Internet of Things. 8, 9, 40

JSON JavaScript Object Notation. 8, 40

JVM Java Virtual Machine. 16, 40

MVVM Model View ViewModel. iii, 21

OAuth 2.0 Open Authorization 2.0. 27, 41

PKCE Proof Key for Code Exchange. 27, 28, 41

RSA Rivest-Shamir-Adleman. 8, 41

SDK Software Development Kit. 7, 8, 16, 17, 41

SF System function. 8, 41

SFE System function element. 8, 41

SIP Session initiation protocol. 7, 41

SOLID Single responsibility, Open-closed, Liskov substitution, Interface segregation, Dependency inversion. iii, 19, 41

SSL Secure Sockets Layer. 8, 41

UI User interface. 23, 25, 38, 42

UX User experience. 38, 42

VCS Version control system. 16, 42

Glossario

API in informatica con il termine *Application Programming Interface API* (ing. interfaccia di programmazione di un'applicazione) si indica ogni insieme di procedure disponibili al programmatore, di solito raggruppate a formare un set di strumenti specifici per l'espletamento di un determinato compito all'interno di un certo programma. La finalità è ottenere un'astrazione, di solito tra l'hardware e il programmatore o tra software a basso e quello ad alto livello semplificando così il lavoro di programmazione. 39

Framework In informatica e specificamente nello sviluppo software, un framework è un'architettura logica di supporto sulla quale un software può essere progettato e realizzato, spesso facilitandone lo sviluppo da parte del programmatore. La sua funzione è quella di creare una infrastruttura generale, lasciando al programmatore il contenuto vero e proprio dell'applicazione. Lo scopo di un framework è infatti quello di risparmiare allo sviluppatore la riscrittura di codice già scritto in precedenza per compiti simili. iii, 3, 4, 18, 19, 23, 25, 28, 33, 34, 38, 40

IDE Un ambiente di sviluppo integrato è un applicativo software che fornisce funzionalità complete per lo sviluppo software. 39

IoT L'Internet of Things (IoT), in italiano "Internet delle Cose," è un concetto tecnologico che si riferisce a una rete di dispositivi fisici, oggetti e altri oggetti incorporati con sensori, software e altre tecnologie che consentono loro di raccogliere e scambiare dati con altri dispositivi e sistemi attraverso Internet. L'obiettivo principale dell'IoT è quello di consentire la comunicazione e la condivisione di informazioni tra oggetti fisici, rendendoli "intelligenti" e in grado di interagire con l'ambiente circostante e con altri dispositivi, spesso senza richiedere l'intervento umano. 39

JSON JSON è un formato per lo scambio di dati basato sul linguaggio di programmazione JavaScript. Il formato è facile da leggere e scrivere per gli umani, facile da analizzare e generare per le macchine, sfrutta convenzioni familiari ai programmatori ma indipendente dal linguaggio utilizzato; queste premesse lo rendono un formato ideale per lo scambio di dati. 39

JVM La Java Virtual Machine è una macchina virtuale che fornisce un ambiente di esecuzione per i programmi scritti in linguaggio di programmazione Java. La JVM consente di eseguire codice Java su una varietà di piattaforme hardware e software senza doverlo adattare specificamente a ciascuna piattaforma. La JVM interpreta il bytecode Java, che è un linguaggio intermedio compilato a partire dal codice sorgente Java, e lo esegue in un ambiente sandbox sicuro. 39

OAuth 2.0 OAuth 2.0 (Open Authorization 2.0) è un protocollo per l'autorizzazione e l'autenticazione ampiamente utilizzato per garantire l'accesso a servizi web, API e applicazioni. Consente agli utenti di concedere a terze parti un accesso limitato alle loro risorse senza condividere le proprie credenziali (come nomi utente e password). 39

PKCE l'estensione PKCE (Proof Key for Code Exchange) è un componente importante del protocollo OAuth 2.0, specificamente progettato per migliorarne la sicurezza. L'obiettivo principale di PKCE è proteggere da attacchi di tipo "authorization code interception" quando l'applicazione client è pubblica (cioè non può mantenere segretamente il proprio client secret), il che è spesso il caso nelle app mobili. 39

RSA In crittografia la sigla RSA indica un algoritmo di crittografia asimmetrica, inventato nel 1977 da Ronald Rivest, Adi Shamir e Leonard Adleman utilizzabile per cifrare o firmare informazioni. Il sistema di crittografia si basa sull'esistenza di due chiavi distinte, che vengono usate per cifrare e decifrare. Se la prima chiave viene usata per la cifratura, la seconda deve necessariamente essere utilizzata per la decifratura e viceversa. La questione fondamentale è che, nonostante le due chiavi siano fra loro dipendenti, non è possibile risalire dall'una all'altra, in modo che se anche si è a conoscenza di una delle due chiavi, non si possa risalire all'altra, garantendo in questo modo l'integrità della crittografia. 39

SDK Un software development kit, in informatica, indica genericamente un insieme di strumenti per lo sviluppo e la documentazione di software. 39

SF Una system function indica la funzionalità a cui adempie un dispositivo Vimar. Ogni implementazione di uan system function può avere una diversa lista di SFE che dipende dalla possibilità o meno di poter eseguire determinate operazioni. 39

SFE Un system function element è il componente base di una SF e rappresenta le informazioni riguardanti lo stato, i comandi eseguibili ed i parametri modificabili di una SF. 39

SIP SIP (Session Initiation Protocol) è un protocollo di comunicazione ampiamente utilizzato per l'iniziazione, la gestione e la terminazione delle sessioni di comunicazione in reti IP (Internet Protocol). Queste sessioni di comunicazione possono includere chiamate vocali, videochiamate, messaggistica istantanea e altre forme di comunicazione in tempo reale su Internet. 39

SOLID In ingegneria del software, SOLID è un acrostico riferito a cinque principi dello sviluppo del software orientato agli oggetti descritti da Robert C. Martin in diverse pubblicazioni dei primi anni 2000. Tali principi vengono detti SOLID principles. La parola è un acronimo che serve a ricordare tali principi (Single responsibility, Open-closed, Liskov substitution, Interface segregation, Dependency inversion). 39

SSL SSL, o Secure Sockets Layer, è un protocollo di sicurezza Internet basato sulla crittografia. Fu sviluppato da Netscape nel 1995 allo scopo di garantire la privacy, l'autenticazione e l'integrità dei dati nelle comunicazioni Internet. Per riuscire a fornire un elevato grado di privacy, SSL crittografa i dati che vengono

trasmessi attraverso il Web. Ciò significa che chiunque provi a intercettare tali dati vedrebbe solo un miscuglio ingarbugliato di caratteri quasi impossibile da decifrare. 39

UI L'Interfaccia Utente, abbreviata come UI, è la parte di un'applicazione, un sistema o un dispositivo che permette agli utenti di interagire con esso. Si tratta di un punto di contatto tra l'utente e il sistema, attraverso il quale gli utenti possono inserire comandi, ricevere informazioni e navigare tra le funzionalità e i contenuti. 18, 39

UX L'Esperienza Utente (UX) rappresenta il complesso delle emozioni, delle percezioni e delle reazioni che un utente sperimenta interagendo con un prodotto, un servizio, un sito web, un'applicazione o qualsiasi sistema o dispositivo. L'obiettivo principale del design dell'esperienza utente è creare un'interazione positiva e significativa tra l'utente e il sistema. 39

VCS In ingegneria del software il controllo di versione è si occupa di tenere traccia e gestire le modifiche che vengono apportate al codice sorgente del software. 39

WebSocket WebSocket è un protocollo di rete basato su TCP, che definisce le modalità di scambio dei dati tra le reti. Il TCP stabilisce una connessione tra due punti finali di comunicazione, che sono chiamati socket. In questo modo è possibile collegare i dati in due direzioni. Con una connessione bidirezionale come in WebSocket (o Web Socket), i dati vengono scambiati simultaneamente in entrambe le direzioni. Il vantaggio è che i dati vengono visualizzati rapidamente. 8, 42

Bibliografia

Articoli consultati

«IEEE Standard Glossary of Software Engineering Terminology». In: *IEEE Std 610.12-1990* (1990), pp. 1–84. DOI: 10.1109/IEEESTD.1990.101064 (cit. a p. 11).

Siti web consultati

Atlassian. URL: https://www.atlassian.com.

Documentazione Android. URL: https://developer.android.com.

Martin, Robert C. *The Clean Code Blog*. URL: https://blog.cleancoder.com (cit. a p. 18).

Refactoring Guru. URL: https://refactoring.guru (cit. a p. 25).

System Engineering Body of Knowledge. URL: https://sebokwiki.org (cit. a p. 11).