POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria del Cinema e dei Mezzi di Comunicazione

Tesi di Laurea Magistrale

La Clean Architecture applicata allo sviluppo mobile in ambiente Android

Relatrice:
Prof. Laura Farinetti

Candidata:
Aurora Vassallo

Dicembre 2020

Ringraziamenti

Vorrei ringraziare tutte le persone che mi hanno supportata nella stesura di questa tesi durante questi mesi.
Prima di tutto, vorrei ringraziare Synesthesia, uno spazio sereno e ricco di persone brillanti, che mi ha accolta dandomi questa speciale opportunità. In particolare, ringrazio Gianmarco, che mi ha consigliato l'argomento e Marco, per le sue spiegazioni a distanza.
Un grazie speciale va alla professoressa Laura Farinetti, per avermi guidata, consigliata e supportata durante questo anno.
E poi ringrazio mamma e papà, per amarmi e non avermi mai fatto mancare niente, mio fratello, per essere il mio mentore e miglior insegnante.
Gabriele, l'amore della mia vita, per farmi sentire speciale ogni giorno e Spadino, per aver riempito le mie giornate di gioia.
Greta, per esserci ogni istante e per darmi la certezza di esserci per sempre. Laura e Betta, per le loro risate, il loro affetto e la loro preziosa amicizia lunga una vita. Fabio, per essere stato il miglior compagno e amico in questi anni universitari.
Per finire, vorrei ringraziare Alice per aver letto questa tesi.

Prefazione

L'organizzazione di un sistema software è fondamentale e purtroppo viene spesso tralasciata dagli sviluppatori alle prime armi. In questa tesi si vogliono tracciare le linee guida per costruire un'architettura che permetta di pianificare un progetto che poggi su fondamenta solide.
In particolare, la tipologia di architettura di cui viene discusso è la Clean Architecture, ideata da Robert C. Martin, la quale permette di produrre un software che sia facile da sviluppare, distribuire e manutenere. Basandosi sui principi di progettazione SOLID e sui principi di coesione e accoppiamento dei componenti, l'applicazione della Clean Architecture consente di separare le logiche di business dalle logiche dell'applicazione rendendo il software indipendente dal framework, dalla UI, dal database e da qualsiasi altro agente esterno.
Inoltre, vengono descritti i pattern architetturali MVC (Model-View-Controller), MVP (Model-View-Presenter) e MVVM (Model-View-ViewModel), grazie ai quali è possibile avere delle soluzioni specifiche per implementare un certo stile architettonico a livello di moduli.
Dal momento che la tesi è stata sviluppata in collaborazione con l'azienda Synesthesia srl, è stato possibile studiare come i concetti esposti vengano applicati concretamente.
Attraverso una Sample App sviluppata in ambiente Android con il linguaggio di programmazione Kotlin, viene illustrato come organizzare un'applicazione mobile aderendo alle regole della Clean Architecture sfruttando il pattern architetturale MVVM.
In aggiunta, la tesi prevede la conversione a livello teorico dell'architettura di un'app sviluppata per il corso di Digital Interaction Design, attenendosi alle specifiche delineate dall'azienda.
Infine, l'ultimo capitolo si concentra sui test e sul perché l'utilizzo della Clean Architecture permetta di implementare un codice facilmente testabile.

È stato reso possibile partecipare ad un progetto in corso di Synesthesia riguardante lo sviluppo di app con l'utilizzo dei Flavor e delle Build Variant, perciò è stata aggiunta un'appendice in merito a questo argomento.

Obiettivo

L'obiettivo finale della tesi è quello di comprendere il significato di architettura software, in particolare della Clean Architecture, e di come l'adozione di un buon piano progettuale permetta di realizzare un'applicazione mobile che sia allo stesso tempo funzionante e facilmente mantenibile.

Per raggiungere questo obiettivo si è proposto un percorso a tappe che parte dallo studio dello stato dell'arte della Clean Architecture fino ad arrivare alla conversione dell'architettura di un'app già esistente, passando per i principi di progettazione SOLID, i principi dei componenti e i pattern architetturali.

L'obiettivo secondario di questa tesi è quello di accompagnare il lettore in questo cammino con chiarezza e semplicità. Poiché il mondo dell'architettura software e della programmazione in generale è molto articolato, si è cercato di adottare un linguaggio il meno complesso possibile mantenendo una struttura del discorso lineare.

Così facendo, si offre la possibilità di avere un quadro generale sull'architettura software e un esempio di implementazione da poter applicare ai propri progetti.

Indice

Lista delle Figure

Legenda colori

Nel corso della tesi si incontreranno diversi grafici per rappresentare le relazioni che intercorrono tra i vari elementi di un sistema software.
Per chiarezza, ogni tipologia di elemento verrà rappresentato sempre con lo stesso colore:
componente: #d81a38
package: #344649
file sorgente: #5cbbd1

Nel capitolo 5 si incontrerà un'ulteriore distinzione: tra elementi già esistenti ed elementi da creare. Per questo motivo si sono utilizzati i rettangoli con la traccia colorata per rappresentare i primi e i rettangoli pieni per i secondi.

Capitolo 1

L'architettura Software e il Compito dell'Architetto

Non c'è dubbio che oggi il mondo dipenda dal software. È un elemento essenziale di tutti i dispositivi digitali ed elettronici che ormai accompagnano quotidianamente la nostra vita aiutandoci a svolgere compiti di ogni genere. Un elenco in ordine sparso di oggetti che fino a qualche anno fa non pensavamo minimamente di poter associare al software comprende spazzolini da denti, scarpe, occhiali, canne da pesca, bici, irrigatori e molto altro ancora. Vedendola in modo più ampio, il software è presente in tutte le aziende tradizionali che non si occupano di vendere o produrre software.

Ma che cos'è effettivamente questo software? Di che cosa è fatto? Quali parti lo compongono? Come lo si può rappresentare?
La maggior parte delle persone, esclusi gli sviluppatori e chi lavora nel settore, quando sentono parlare di software immaginano qualcosa di astratto che permette il funzionamento della tecnologia per magia. Qualcuno potrebbe essere a conoscenza del lavoro complesso che si cela dietro la realizzazione di un software, qualcun altro è pratico di qualche componente, ma difficilmente si ha un'idea esauriente di cosa esso sia.
Il software (dall'inglese *ware*, prodotto, e *soft*, leggero), è stato elaborato per modificare il comportamento delle macchine in modo leggero, ovvero in modo che sia facile da modificare. Rappresenta, all'atto pratico, l'insieme di istruzioni che dicono ad un calcolatore cosa fare e comprende tutta la serie di algoritmi, procedure, routine che permetteranno all'hardware, la componente materiale, di funzionare.
Pertanto, il software materialmente, non è altro che linee e linee di codice. Ma come sono organizzate queste linee di codice? Da dove si parte a scriverle? Chi decide chi si occupa di quale aspetto? Qui entrano in gioco l'architettura e il ruolo dell'architetto software.
Su internet non mancano le definizioni quando si parla di architettura, esistono persino

siti web che contengono raccolte di definizioni di architettura. Tuttavia, sfogliandole si può notare che tutte utilizzano all'incirca gli stessi termini o sinonimi, quali: organizzazione, sistema, componenti, relazioni, ambiente.
Dato che questa tesi si occupa principalmente della Clean Architecture, la definizione più coerente che sembra giusto citare è quella data dall'ideatore dell'architettura pulita, Robert C. Martin:

> *"L'architettura di un sistema software è la "forma" data a quel sistema da coloro che la costruiscono. Per "forma" si intende la divisione di tale sistema in componenti, nella disposizione di essi e nei modi in cui tali componenti comunicano tra loro. Lo scopo è quello di facilitare lo sviluppo, la distribuzione, il funzionamento e la manutenzione del sistema software in esso contenuto".*

Questa definizione evidenza in modo chiaro e conciso tutti gli elementi chiave che servono per comprendere il ruolo dell'architettura in un software.
In primo luogo, la parola su cui concentrarsi è "forma".
Come descritto nel libro *Clean Architecture* di Robert C. Martin, spesso si fa confusione tra architettura e struttura: "*la parola "architettura" viene spesso usata nel contesto di qualcosa che si trova a un livello superiore, distinto dai dettagli di basso livello, mentre "struttura" sembra implicare strutture e decisioni a un livello più basso. Ma questa distinzione non ha senso considerando quello che fa davvero un architetto.*
[…] I dettagli di basso livello e la struttura di alto livello fanno parte dello stesso insieme. Essi costituiscono un unico insieme che definisce la forma del sistema. Non potete avere gli uni senza l'altra; in effetti, non esiste una linea che li separi. Vi è semplicemente un continuum di decisioni dai livelli più elevati a quelli più bassi"[1].
Ovviamente, il paragone immediato da applicare per comprendere il concetto di "forma" del software è con l'architettura di un edificio: la conformazione, l'aspetto esteriore, l'elevazione e la disposizione degli spazi e delle stanze rappresentano gli elementi di alto livello, la posizione delle prese di corrente, degli interruttori e delle luci rappresentano gli elementi di basso livello. Entrambi gli aspetti fanno parte della struttura della casa e sono il risultato delle decisioni prese dall'architetto prima della realizzazione effettiva dell'edificio. Allo stesso modo l'architetto del software prende decisioni su come organizzare i vari componenti partendo prima dai requisiti di alto livello, per poi andare a organizzare il lavoro nel dettaglio, e tutto questo andrebbe fatto, così come per la costruzione di edifici e ponti, prima che gli sviluppatori inizino a scrivere righe di codice.

L'utilizzo del condizionale è voluto, come sottolinea Robert C. Martin (chiamato anche *Uncle Bob*), capita spesso che per questioni di tempistiche richieste dal cliente, la realiz-

1 Robert C. Martin, *Clean Architecture,* Apogeo, 2018

zazione di una buona architettura di un sistema venga messa da parte, per cui si inizia a scrivere codice senza avere basi solide, "basta che il software funzioni".
Però, come evidenziato dalla definizione di architettura sopra citata, “lo scopo è quello di facilitare lo sviluppo, la distribuzione, il funzionamento e *la manutenzione del sistema*”. Certo, un programma deve funzionare, ma deve essere anche facile da manutenere altrimenti, se è impossibile da modificare allora non funzionerà più quando cambieranno i requisiti. Mentre, un programma con un’architettura ben pensata e organizzata sarà facile da far funzionare anche quando verranno richiesti altri cambiamenti.
Questo discorso fa risaltare i due valori del software: il comportamento e l’architettura. Il primo ha una priorità urgente, ma non sempre particolarmente importante. Il secondo è importante, ma mai particolarmente urgente.
I manager di un’azienda di sviluppo software spesso non sono in grado di valutare l’importanza dell’architettura software, perciò ritengono più importante “*skippare*” la fase di realizzazione di quest’ultima per concentrarsi direttamente sul funzionamento: scrivere codice al più presto distribuibile che soddisfi i requisiti momentanei del cliente. Invece, è sempre importante difendere l’architettura, che si tratti di un sistema piccolo o di un progetto ampio: “*Se l’architettura verrà per ultima, lo sviluppo del sistema diverrà sempre più costoso e alla fine ogni modifica diverrà praticamente impossibile anche se riguarda solo una parte del sistema*” (Robert C. Martin).

Oltre al principale problema di scarsa manutenibilità, una cattiva architettura software presenta le seguenti complicazioni.

- *Difficile comprensione*: nel momento in cui si lavora ad un sistema software, bisogna far in modo che chiunque entri nel team riesca a comprendere ciò che è stato scritto in precedenza dagli altri sviluppatori. Capita spesso di dover mettere mano a codice scritto da altri, se l’organizzazione del sistema viene a mancare, la comprensione risulterà molto difficile e il tempo perso sarà notevole. Uno sviluppatore, quando guarda per la prima volta un nuovo software, deve capire subito di che tipologia di sistema si sta parlando, se di un applicativo che gestisce le prenotazioni di un coworking o se di un software per elaborare le bustepaga. L’architettura deve “urlare” il tema del software;,deve urlare “coworking” o “bustepaga” così come una piantina che rappresenta una sala, una cucina, un bagno urla “casa”.
- *Scarsa riusabilità*: quando l’architettura viene a mancare o risulta poco curata, nel progetto si ritroveranno inevitabilmente grossi blocchi di codice e funzionalità duplicate. Nel momento in cui si dovrà effettuare una modifica su uno di questi blocchi, bisognerà riportare tale cambiamento su tutti gli altri blocchi connessi. Quindi significa che bisognerà effetturare lo stesso lavoro tante volte tante quante sono le duplicazioni e implica la perdita di tempo e di risorse spendibili, invece, per lavoro utile.
- *Difficilmente testabile*: bisogna sempre tener conto che qualsiasi funzionalità,

qualsiasi pezzo di codice scritto in azienda deve essere testato, è la parte più importante. Una cattiva progettazione dell'architettura rende complicata la fase di scrittura dei test automatici, poiché componenti troppo grandi con dipendenze complesse rendono impossibile testare in modo isolato il componente o il mocking delle dipendenze. Questo argomento verrà trattato nell'ultimo capitolo in maniera più approfondita.

Si può quindi osservare che tralasciare la parte di progettazione significa andare a condizionare gli attributi che determinano la *Quality of Service* di un software, ovvero i requisiti non funzionali quali[2]:

- *disponibilità*: è il grado in cui un sistema, un sottosistema o un'apparecchiatura è accessibile ed è in grado di svolgere una funzione richiesta in determinate condizioni ad un dato istante (es. fornire un servizio ad un utente), o durante un dato intervallo di tempo, supponendo che siano assicurati i mezzi esterni eventualmente necessari;
- *affidabilità*: è la capacità di un oggetto di eseguire una funzione richiesta in una data condizione per un periodo di tempo specificato ed anche la capacità di garantire l'integrità e la coerenza di un'applicazione e delle sue transazioni. È anche definibile come la probabilità che un'unità funzionale esegua la sua funzione richiesta per un intervallo specificato;
- *robustezza*: è la capacità di far fronte ad errori durante l'esecuzione o input errati gestendo arresti e azioni inattese. Un esempio può essere la gestione delle eccezioni per evitare comportamenti non previsti o arresti anomali dell'esecuzione;
- *gestibilità:* è la capacità di amministrare e quindi gestire le risorse del sistema per garantire la disponibilità e le prestazioni del sistema stesso rispetto alle altre funzionalità. Si tratta dell'insieme di funzionalità di sistema utilizzate per l'avvio e l'arresto del server, un esempio sono l'installazione di nuovi componenti, la gestione delle autorizzazioni di sicurezza e l'esecuzione di altre attività;
- *flessibilità:* è la capacità di adattarsi a modifiche della configurazione hardware o dell'architettura senza provocare un grande impatto sul sistema sottostante;
- *performance*: è la capacità di eseguire le funzionalità richieste in un lasso di tempo che soddisfa gli obiettivi specificati;
- *scalabilità:* è la capacità di supportare i requisiti di disponibilità e performance all'aumentare del carico. La scalabilità può essere di due tipi: verticale, aumentando la capacità del server esistente (es: in termini di memoria o numero CPU), oppure orizzontale aggiungendo server o numero di istanze;
- *estensibilità:* è la capacità di aggiungere facilmente nuove funzionalità minimizzando l'impatto sulle funzionalità già esistenti;
- *manutenibilità:* è la capacità di un software di essere facilmente manutenuto, ad esempio in termini di correzione di bug, sostituzione di componenti o massimizzando la vita di un prodotto;

2 Alessio Fiore, *Quality of service di un software,* `italiancoders.it`, 14 Febbraio 2018

- *riusabilità:* è la capacità di usare un componente in uno o più contesti senza dover apportare modifiche allo stesso. Questo aspetto assume una notevole importanza perché permette di ottenere un risparmio in termini di tempi e costi;
- *sicurezza*: è la capacità di assicurare che le informazioni gestite da un componente non siano accessibili e modificabili da altre parti.

Lasciando ora da parte i requisiti non funzionali di un software, sarebbe opportuno focalizzarsi nuovamente sulla definizione di architettura data da Uncle Bob, in particolare sull'ultima frase: "*lo scopo è quello di facilitare lo sviluppo, la distribuzione, il funzionamento e la manutenzione del sistema*", e quindi sulle fasi che un buon architetto software deve cercare di ottimizzare in termini di costo e qualità.
Del funzionamento e della manutenzione del sistema è stato già discusso in precedenza, lo sviluppo, il deploy e, in aggiunta, la scalabilità sono altri momenti estremamente fondamentali che devono funzionare al meglio.
Nella fase di sviluppo va sempre tenuto a mente il principio *KISS* (Keep It Simple Stupid), che afferma "*la maggior parte dei sistemi funziona meglio se vengono mantenuti semplici anziché complessi; quindi, la semplicità dovrebbe essere un obbiettivo chiave nella progettazione e dovrebbe essere evitata la complessità non necessaria*" (Robert C. Martin).
Se si lavora su codice semplice e lineare da comprendere, gli sviluppatori, che sono persone e non macchine (bisogna sempre tenerlo presente), riusciranno a coordinarsi meglio tra loro nelle fasi di sviluppo, riuscendo a parallelizzare il lavoro in più risorse senza "pestarsi i piedi". Inoltre, qualsiasi aumento della complessità aumenta notevolmente il lavoro nel momento in cui bisognerà aggiustare o modificare il codice.
Inoltre, citando l'articolo di Dario Frongillo, "*per essere efficace, un sistema software deve essere distribuibile. Maggiore è il costo di deployment, meno efficace è il sistema*.[3]" Quindi, un buon architetto software deve predisporre le fondamenta per un sistema facilmente distribuibile e quindi coordinarsi al meglio con il Devops del team[4].
Sfortunatamente, le strategie di distribuzione non vengono molto esaminate durante le fasi iniziali di sviluppo. Questo porta a creare sistemi che possono anche essere semplici da sviluppare, ma complicati da fornire all'utente finale.
Infine, bisogna soddisfare il requisito di scalabilità. Come già accennato prima "la scalabilità è la capacità del sistema di gestire volumi di elaborazione crescenti nel futuro, se richiesto". I tipi di incremento di carico possono essere di diverse tipologie: il numero degli utenti, le richieste da parte degli utenti, la quantità di dati che il sistema deve gestire, ecc. Un sistema in grado di gestire questi aumenti di carico senza impattare negativamente

3 Dario Frongillo, *Architettura Software: introduzione, importanza e compiti di un SW Architect*, `italiancoders.it`, 28 Maggio 2018

4 *Devops è una metodologia di sviluppo del software utilizzata in informatica che punta alla comunicazione, collaborazione e integrazione tra sviluppatori e addetti alle operations della information technology (IT). (Wikipedia).*

altre qualità (di prestazioni e disponibilità) è un sistema che rispetta la scalabilità.
Ritornando a una delle domande poste ad inizio capitolo, come rappresentiamo questo software? Si è detto che la forma del software è l'architettura, ma si parla ancora in termini astratti. Non possiamo creare una piantina di un software per rappresentare l'architettura e non possiamo neanche consegnare un intero applicativo comprensivo di tutto il codice e dire "questa è l'architettura del software".
Purtroppo, ci si deve rassegnare all'idea che l'architettura non somigli a nulla, né tanto meno si possono definire architettura dei grafici: "*Sebbene gradevoli e visivamente parlanti, i riquadri di un grafico di PowerPoint non sono l'architettura di un sistema software. Senza dubbio essi rappresentano una determinata visione di un'architettura, ma confondere i riquadri con il loro significato, ovvero con l'architettura, significa perdersi sia il significato sia l'architettura. L'architettura non somiglia a nulla. Una determinata scelta visuale non è un dato di fatto. È solo una scelta basata su un insieme di altre scelte: che cosa includere; che cosa escludere; che cosa evidenziare con una forma o un colore; che cosa celare impiegando scelte di uniformità o vere e proprie omissioni*.[5]" (Kevlin Henney).

L'ultima domanda a cui rispondere è, quindi, chi è questa figura misteriosa denominata "architetto software"?
Non è altro che un developer che continua a programmare assumendo, però, task differenti. È un abile programmatore che continua a scrivere codice mentre guida il resto del team verso un design che massimizza la qualità e produttività del software. Se smettesse di scrivere non sarebbe a conoscenza della developer experience e delle difficoltà in fase di sviluppo.
Come si può ben capire da questa descrizione, il *software architect* difficilmente sarà una figura giovane, sarà piuttosto una persona con anni di esperienza perché le responsabilità nelle sue mani saranno significative e le conoscenze dovranno essere approfondite.
I punti di forza di un architetto dovrebbero essere i seguenti[6]:

- *comunicabilità:* durante la giornata lavorativa, l'architetto deve confrontarsi con i clienti nella lingua del business, come i manager di tutti i livelli, gli analisti aziendali e gli sviluppatori. Il carisma e la capacità di coinvolgere sarà un enorme vantaggio, poiché è fondamentale spiegare correttamente le proprie azioni. Gli architetti sono oratori laconici, eloquenti e competenti. Inoltre, queste capacità sono fondamentali perché l'architetto partecipa alla maggior parte dei processi di discussione e spesso è necessario raggiungere dei compromessi accettabili e vantaggiosi per tutte le parti coinvolte;
- *conoscenza tecnica ampia e profonda:* dovrebbe essere ovvio, l'architetto di solito ha esperienza in diversi stack tecnologici a un livello elevato. Inoltre, dovrebbe essere preparato a produrre un gran numero di documentazione tecnica, rapporti e diagrammi;

5 Kevlin Hanney, Prefazione *Clean Architecture,* Apogeo, 2018
6 Nikolay Ashanin, *The Path to Becoming a Software Architect*, `medium.com`, 1 Ottobre 2017

- *responsabilità:* le decisioni dell'architetto sono generalmente le più gravose. Pertanto, una persona in questa posizione dovrebbe essere consapevole e in grado di rispondere personalmente delle decisioni prese. Se l'errore dello sviluppatore può costare un paio di giorni di lavoro di una persona, l'errore dell'architetto può costare anni-persona di lavoro su progetti complessi;
- *resistenza allo stress:* l'architetto lavora con persone provenienti da ambiti di lavoro diversi e dovrà affrontare richieste in rapida evoluzione o persino ambienti aziendali in evoluzione. Pertanto, è necessario essere pronti ad affrontare lo stress ed essere in grado di sfuggire alle emozioni negative;
- *capacità di gestione:* sia in termini di capacità organizzative che di leadership. La capacità di guidare un team, che può essere distribuito e composto da specialisti molto diversi tra loro, è essenziale;
- *abilità analitiche:* uno dei compiti più preziosi è la capacità di rappresentare un problema astratto sotto forma di un oggetto reale e finito del sistema che gli sviluppatori stanno già valutando, progettando e sviluppando. Ottime capacità di comunicazione sono essenziali per rappresentare l'astrazione nella forma del sistema finale per i membri del team e per il cliente.

Questo paragrafo ha voluto essere un breve excursus sulle qualità e sui compiti che un architetto del software deve ricoprire per far comprendere meglio al lettore quanto sia fondamentale la presenza di un lavoratore competente in questo settore e quanto le sue decisioni incidano sulla riuscita di un progetto di qualità.

È necessario fare un'ultima riflessione sugli incarichi di un architetto software. Come scritto nell'elenco soprastante, "l'architetto software dovrebbe anche essere preparato a produrre un gran numero di documentazione tecnica, rapporti e diagrammi". Occorre tener presente che l'aspetto importante dell'architettura non è il risultato finale, ma la logica di sviluppo. Pertanto, è indispensabile documentare tutte le decisioni che hanno portato ad una specifica architettura, così come la logica di tali decisioni.
La documentazione è importante sia per l'architetto, nel momento in cui ha bisogno di rivisitare la logica a base delle decisioni prese, sia per coloro che dovranno mantenere il sistema. Le architetture documentate risultano inevitabilmente più efficaci di quelle che non lo sono, poiché quest'ultime rendono difficile dimostrare che soddisfano i requisiti dichiarati. Questo discorso mette in risalto un aspetto considerevole: l'architettura non viene assolutamente decisa nella sua interezza a priori, dal momento che è impossibile viaggiare nel tempo, non si possono conoscere oggi quali saranno gli interventi e le necessità future che li guideranno.

Per fare un buon lavoro nonostante il problema del viaggio nel tempo e tutte le considerazioni esposte in questo primo capitolo, è necessario cercare di tendere a un percorso il più clean possibile.

> "*Il percorso cui siamo più interessati è quello più pulito. Che riconosce la leggerezza del software e punta a preservarla come una delle priorità del sistema. Che riconosce che stiamo operando alla base di conoscenze incomplete, ma che comprende anche che, in quanto esseri umani, operare con conoscenze incomplete è qualcosa che facciamo normalmente, qualcosa che ci appartiene. […]*
> *"Una buona architettura deriva dalla consapevolezza che essa è più un viaggio che una destinazione, più un processo evolutivo che un artefatto congelato.*[7]" (Kevlin Henney).

7 Prefazione *Clean Architecture,* Apogeo, 2018

Capitolo 2

Clean Architecture

2.1 Introduzione

> *"Se pensate che una buona architettura sia costosa, provate a usare una cattiva architettura"* (Brian Foote e Joseph Yoder).

Come annunciato alla fine del capitolo precedente, il percorso a cui dobbiamo cercare di tendere è quello più pulito. Questo percorso porterà a fare le cose per bene, di conseguenza permetterà di procedere rapidamente e, quindi, di risparmiare sui costi.
La Clean Architecture si riferisce all'organizzazione del progetto in modo tale che sia facile da comprendere e facile da modificare nelle varie fasi di sviluppo, distribuzione e manutenzione, tenendo oltremodo conto che un buon sistema deve essere scalabile.
Sia che il progetto sia di piccole dimensioni, sia esso di grandi, è importante cercare di tendere a questo tipo di architettura. Ovviamente, quando si parla di architettura è fondamentale tenere a mente che il progetto evolve e costruire un sistema perfetto fin dall'inizio è quasi sempre un pensiero utopistico. Per questo motivo, cercare di avere un'organizzazione che sia il più possibile flessibile, oltre che funzionante, è necessario e deve essere un processo intenzionale.

Esistono un'infinità di sistemi software e se ne possono trovare di radicalmente differenti, ma le regole dell'architettura software sono sempre indipendenti da ogni altra variabile. Ciò è annunciato da Robert C. Martin nella premessa del suo libro "*Clean Architecture: guida per diventare abili progettisti di architetture software*".
Questo capitolo si concentrerà essenzialmente sul libro citato.
Robert C. Martin ("Uncle Bob") programma fin dal 1970. È cofondatore di c*leancorders.com*, che produce corsi video online per gli sviluppatori, ed è fondatore di Uncle Bob Con-

sulting LLC, che offre consulenze software, corsi e servizi di approfondimento per grandi aziende in tutto il mondo. Ha lavorato per Master Craftsman at 8th Light, Inc., una società di consulenze software di Chicago. Ha pubblicato decide di articoli in varie riviste e partecipa regolarmente a conferenze internazionali e fiere commerciali. Ha lavorato per tre anni come direttore editoriale di C++ Report ed è stato il primo presidente dell'Agile Alliance.
Martin ha scritto e curato molti libri, fra i quali *The Clean Coder, Clean Code, UML for Java Programmers, Agile Software Development, Extreme Programming in Practice, More C++ Gems, Pattern Languages of Program Design 3* e *Designing Object Oriented C++ Applications Using the Booch Method.*

Non c'è una vera e propria definizione di Clean Architecture, anche perché si basa su una gamma di altri modelli di architettura dei sistemi, quali:

- *Architettura esagonale* (detta anche "porte e adattatori"), sviluppata da Alistair Cockburb e adottata da Steve Freeman e Nat Pryce nel loro libro *Growing Object Oriented Software With Test*
- *DCI* (Data, Context, and Interaction), di James Coplien e Trygve Reenskaug
- *BCE* (Boundary-Control-Entity), introdotta da Ivar Jacobson nel suo libro *Object Oriented Software Engineering: A Use-Case Driven Approach*

Benché queste architetture possano variare nei dettagli, si assomigliano e tutte, inclusa la Clean Architecture, hanno lo stesso obiettivo: la separazione degli ambiti. Tutte conseguono questa separazione suddividendo il software in livelli. La divisione che hanno è quella tra regole operative e interfacce del sistema.
Le caratteristiche dei sistemi prodotte da queste architetture sono le seguenti[1]:

- *indipendenza dai framework:* l'architettura non dipende dall'esistenza di una determinata libreria di software. Questo consente di impiegare tali framework come strumenti, invece di costringere i developers a sviluppare il sistema all'interno dei vincoli da essi imposti;
- *collaudabilità:* le regole operative possono essere collaudate senza la UI, il database, il server web o qualsiasi altro elemento esterno;
- *indipendenza dalla UI:* la UI può cambiare con facilità, senza costringere a modificare il resto del sistema. Una UI web potrebbe essere sostituita con una UI a console, per esempio, senza che ciò obblighi a modificare le regole operative;
- *indipendenza dal database:* si può usare qualsiasi tipologia di database (Oracle, SQL server for Mongo, BigTable, CouchDb). Le regole operative non sono legate al database;
- *indipendenza da qualsiasi agente esterno:* le regole operative non sanno nulla delle interfacce con il mondo esterno;

Un obiettivo rilevante della Clean Architecture è fornire ai *developers* un metodo per organizzare il codice in modo tale da incapsulare la logica aziendale e mantenerla separata

1 Robert C. Martin, *Clean Architecture,* Apogeo, 2018

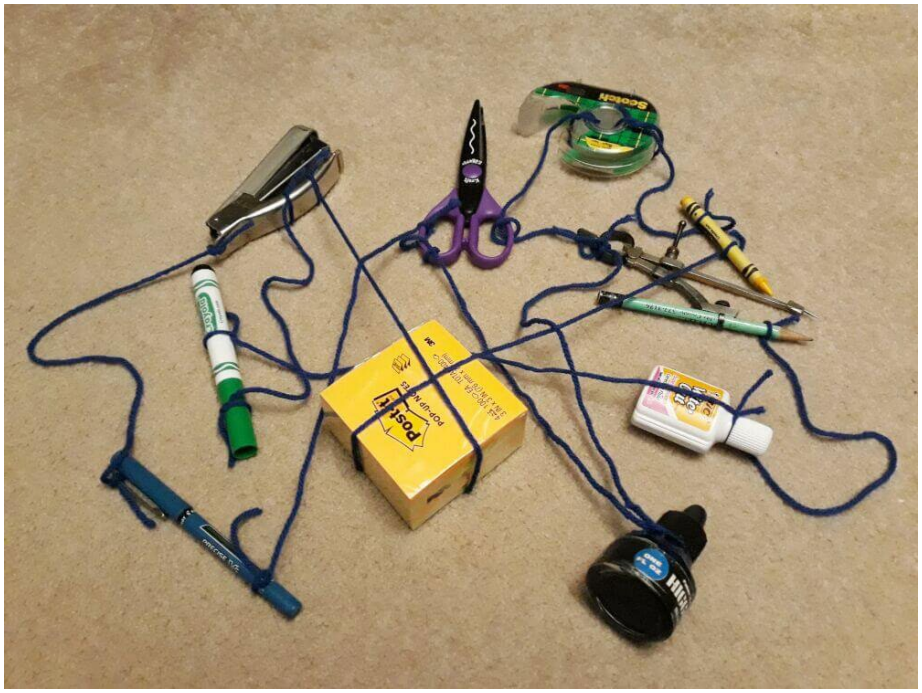
dal processo di distribuzione. Per fare questo bisogna separare gli elementi di un progetto in livelli. Ogni livello è rappresentato visivamente da un cerchio concentrico. I livelli esterni sono meccanismi di livello inferiore, mentre quelli interni raffigurano i livelli superiori, cioè le politiche.

La regola base che fa funzionare questa architettura è la regola della dipendenza: "*le dipendenze presenti nel codice sorgente devono puntare solo all'interno, verso le politiche di alto livello*".

Il codice degli strati interni non può avere nessuna conoscenza di quello che succede negli strati esterni, qualsiasi entità (funzioni, classi, variabili) presente nei livelli esterni non può essere menzionata nei livelli più interni.

Il principio è quello di evitare di avere "*Spaghetti Code*", ovvero evitare un programma che contenga pochissima struttura software e che, indipendentemente dalla buona volontà dei programmatori, con il passare del tempo sarà impossibile da modificare e comprendere, poiché si verranno a creare dipendenze astruse e implementazioni molto grandi che invocheranno un unico flusso di processo multistadio.

Per comprendere meglio questo concetto si possono utilizzare un paio di immagini[2].



2.1 - Metafora di Spaghetti Code

Nella foto 2.1 se si vogliono sostituire le forbici con un coltello bisogna sciogliere i fili che vanno alla penna, alla boccetta di inchiostro, al nastro e al compasso. Quindi bisogna ricollegare quegli oggetti al coltello. Forse funziona per il coltello, ma cosa succede se la penna e il nastro dicono: "Aspetta, avevamo bisogno delle forbici". La penna e il nastro non funzionano più e devono essere cambiati, il che a sua volta influisce sugli oggetti ad essi legati. È un disastro.

2 Suragch, *Clean Architecture for the rest of us*, pusher.com, 4 Gennaio 2019

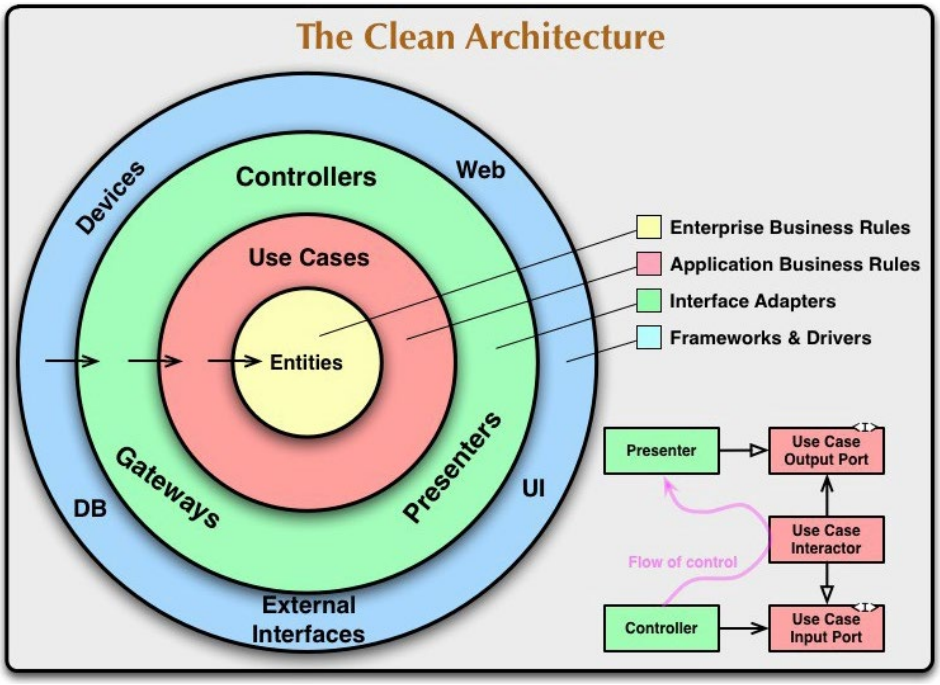
Si confronti ora la situazione precedente con quella rappresentata nella foto 2.2.

2.2 - Metafora di Clean Code

Come si sostituiscono le forbici? Bisogna solo tirare fuori il filo delle forbici da sotto i post-it e aggiungere un nuovo filo legato a un coltello. Molto più facile. I post-it non si preoccupano perché la corda non era nemmeno legata ad essa.
L'architettura rappresentata dalla seconda immagine è ovviamente più facile da modificare. Finché i post-it non devono essere cambiati spesso, questo sistema sarà molto facile da mantenere. Questo stesso concetto sottende l'architettura che rende il software facile da mantenere e modificare. Mentre la prima immagine rende molto bene il concetto di "*Spaghetti Code*".

Lo schema 2.3 raffigura l'immagine più comune del concetto di Clean Architecture:
I cerchi concentrici rappresentano le varie aree del software. Di seguito vengono riportate le descrizioni dei quattro livelli principali[3].

3 *The Clean Architecture,* The Clean Code Blog by Robert C. Martin, 13 Agosto 2012



2.3 - The Clean Architecture

2.1.1 Enterprise Business Rules (Entities)

Le entità incapsulano le regole operative critiche dell'azienda. Un'entità può essere un oggetto con metodi o può essere un insieme di strutture e funzioni di dati.
Se non si ha un'azienda e si sta solo scrivendo una singola applicazione, queste entità sono gli oggetti operativi dell'applicazione. Incapsulano le regole più generali e di alto livello. È meno probabile che cambino quando qualcosa di esterno cambia. Nessuna modifica operativa a una particolare applicazione dovrebbe influire sul livello entità.

2.1.2 Application Business Rules (Use Cases)

Il software in questo livello contiene le regole operative specifiche dell'applicazione. Incapsula e implementa tutti i casi d'uso del sistema. Questi casi d'uso orchestrano il flusso di dati da e verso le entità e chiedono a tali entità di utilizzare le proprie regole operative per raggiungere gli obiettivi del caso d'uso.
Non ci aspettiamo che i cambiamenti in questo livello influenzino le entità. Inoltre, non ci aspettiamo che questo livello venga influenzato da modifiche alle esternalità come il database, l'interfaccia utente o qualsiasi framework comune. Questo strato è isolato da tali preoccupazioni.
Tuttavia, ci aspettiamo che le modifiche al funzionamento dell'applicazione influenzeranno i casi d'uso e quindi il software in questo livello. Se i dettagli di un caso d'uso cambiano, parte del codice in questo livello sarà sicuramente influenzato.

2.1.3 Interface Adapters (Controllers, Gateways, Presenters)

Il software in questo livello è un insieme di adattatori che convertono i dati dal formato più conveniente per i casi d'uso e le entità al formato più conveniente per qualche attore esterno come il Database o il Web. È questo livello, ad esempio, che conterrà interamente l'architettura MVC di una GUI. I presenter, le viste e i controller appartengono tutti a questo livello.

Analogamente, in questo livello i dati vengono convertiti dalla forma più conveniente per entità e casi d'uso nella forma più conveniente per qualsiasi framework di persistenza utilizzato, ovvero il database. Nessun codice all'interno di questo cerchio dovrebbe sapere nulla del database.

Inoltre, in questo livello c'è qualsiasi altro adattatore necessario per convertire i dati da una forma esterna, come ad esempio un servizio esterno, alla forma interna utilizzata dai casi d'uso e dalle entità.

2.1.4 Frameworks and Drivers (Devices, DB, Web, UI, External Interfaces)

Il livello più esterno è generalmente composto da framework e strumenti come Database, Web Framework, ecc. In genere non si scrive molto codice in questo livello se non il codice collante che comunica al cerchio successivo verso l'interno.

Questo livello è quello in cui convergono tutti i dettagli. Il Web è un dettaglio. Il database è un dettaglio. Bisogna mantenerli all'esterno dove possono fare poco danno.

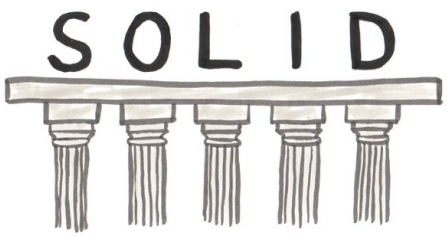
In seguito, le varie aree verranno trattate maggiormente nel dettaglio.

2.2 I principi di progettazione SOLID

I principi SOLID riguardano lo sviluppo software e sono linee guida che sarebbe opportuno seguire se si vuole costruire un software che sia facile da scalare, da manutenere e da comprendere. L'utilizzo di queste regole riduce la complessità del codice separando le responsabilità e definendo le relazioni tra le varie funzioni e le strutture nelle classi.

Una classe è un raggruppamento di funzioni e dati, per cui questi principi non sono validi solo per la programmazione a oggetti, ma per qualsiasi tipo di paradigma. Ogni sistema software ha dei raggruppamenti di questo tipo, indipendentemente dal fatto che essi si chiamino classi.

I principi SOLID sono cinque e sono stati introdotti da Uncle Bob nel suo libro *Design Princples and Design Patterns* (pubblicato da *objectmentor.com*, 2000). L'autore Michael Feathers è stato responsabile di creare l'acronimo:



- **S**ingle Responsibility Princple- SRP
- **O**pen/Closed Principle- OCP
- **L**iskov Substitution Principle- LSP
- **I**nterface Segregation Principle- ISP
- **D**ependency Inversion Principle- DIP

Prima di descrivere i principi nel dettaglio e le loro implicazioni in termini di architettura del software, è opportuno fare mente locale sui concetti di *Low Coupling* (basso accoppiamento), *High Cohesion* (alta coesione) ed *Encapsulation* (Incapsulamento)[4].

2.2.1 Low Coupling

L'accoppiamento nel contesto dello sviluppo di software è definito come il grado in cui un modulo, una classe, o altro costrutto, è legato direttamente ad altri. Per esempio, il grado di accoppiamento tra due classi può essere visto come la misura di quanto una classe dipende da un'altra. Se una classe è strettamente accoppiata (tight o high coupling) con una o più classi, vuol dire che è necessario utilizzare tutte le classi che sono accoppiate quando si desidera utilizzare anche solo una di loro.

2.2.2 High Cohesion

La coesione è la misura in cui due o più parti di un sistema sono correlate e come queste lavorano insieme per creare qualcosa di maggior valore rispetto a quello che fa ogni singola parte. In un sistema software siffatto, è ovvio che non è possibile creare un'alta coesione se si usano classi che fanno troppe cose e che hanno quindi molte responsabilità. Questo tipo di classi tendono ad essere autoreferenziali e la coesione con l'esterno diventa difficile da realizzare.

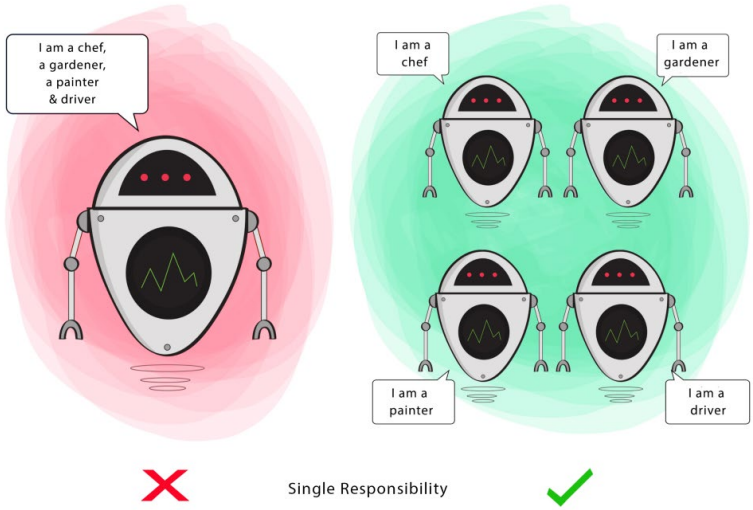
2.2.3 Encapsulation

La maggior parte degli sviluppatori definisce l'incapsulamento come un modo di nascondere informazioni. Come, ad esempio, l'azione di definire metodi pubblici per leggere membri privati di una classe. Questa definizione, tuttavia, spesso porta ad una incompleta comprensione e implementazione dell'incapsulamento, in quanto quest'ultimo non serve solo per nascondere dati, ma anche processi e logica. Un forte incapsulamento è evidenziato dalla possibilità da parte degli sviluppatori di utilizzare una determinata classe o un modulo conoscendo solo la sua interfaccia; lo sviluppatore non conosce, e non deve conoscere, i dettagli implementativi della classe o del modulo.

4 Giuseppe Capodieci, *SOLID Design Principles*, `losviluppatore.it`, 2019

2.2.1 S – Single Responsibility

Un modulo deve avere una sola responsabilità, un solo motivo per cambiare.



2.4 - Single Responsibility

Se si sfogliano gli articoli sul web, la descrizione di questo principio è esattamente quella definita nell'immagine. Ogni oggetto, modulo, classe deve eseguire un solo compito, avere una singola responsabilità. Ma Uncle Bob nel suo libro ci tiene a specificare che non è questo il vero significato di questo principio: "I sistemi software vengono modificati per soddisfare utenti e commitenti; tali utenti e committenti sono il "motivo per cambiare" di cui parla questo principio. […] Probabilmente vi saranno più utenti o committenti che vogliono che il sistema sia modificato nello stesso modo. Qui, in realtà facciamo referimento a un gruppo: una o più persone che richiedono tale modifica. Chiameremo tale gruppo un attore. Pertanto, la versione finale del principio SRP è:

"Un modulo dovrebbe essere responsabile di uno e un solo attore".

Un modulo è un insieme coeso di funzioni e strutture, la definizione più semplice è "un file di codice sorgente".

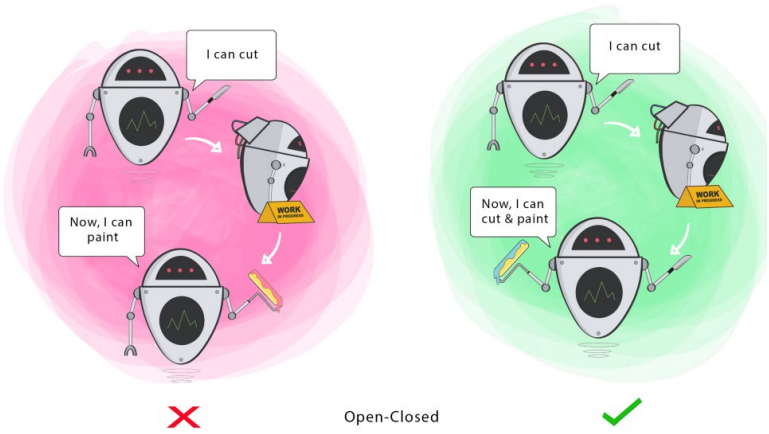
Quello che vuole dire Uncle Bob è che, certo, un modulo deve avere una sola responsabilità perché si evitano classi enormi e monolitiche, perché si testa e si legge meglio il codice e la manutenzione è più semplice, ma questa responsabilità deve essere scelta in relazione agli attori.

Ciò significa, per esempio, che una classe Employee di un'applicazione di paghe non può sia calcolare le paghe che riportare le ore di lavoro e salvare il tutto in qualche database. Non perché sono tre responsabilità diverse ma perché di calcolare le paghe se ne occupa il settore contabilità, di riportare le ore se ne occupano le risorse umane e di salvare il tutto se ne occupano gli amministratori del database. Tre attori differenti. Questi andrebbero a modificare per motivi differenti lo stesso codice sorgente e ciò porta inevitabilmente a duplicazioni, fusioni e molti altri problemi.

Quindi, secondo Uncle Bob, per dividere in modo appropriato le responsabilità è importante fare il ragionamento sugli attori.

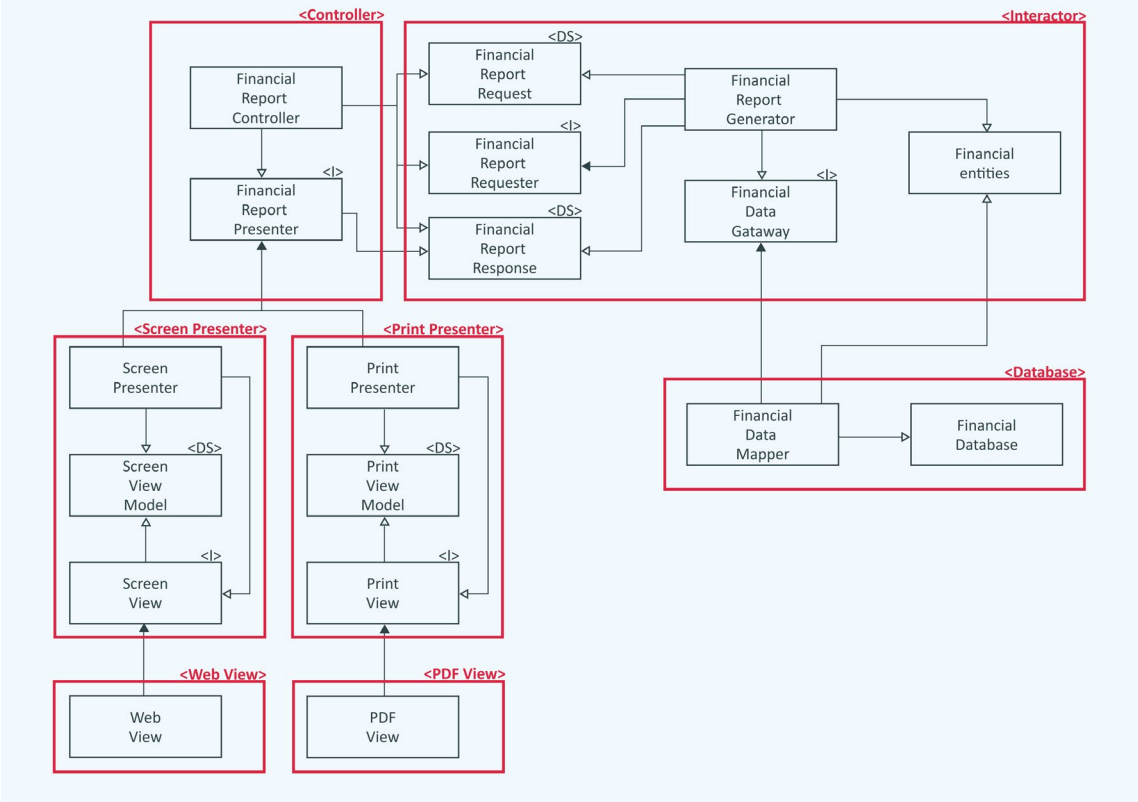
2.2.2 O – Open-Closed Princple

Un'opera software dovrebbe essere aperta alle estensioni ma chiusa alle modifiche.



2.5 - Open-Closed

In altre parole, si dovrebbe essere sempre in grado di aggiungere nuove funzionalità e di estendere una classe senza modificare il comportamento interno. Questo vale non solo per le classi, ma assume un significato anche superiore quando si considera il livello dei componenti dell'architettura.
Questo principio è una delle anime dell'architettura software.
Di seguito si riporta l'esperimento di Robert C. Martin, contenuto nel libro *Clean Architecture: guida per diventare abili progettisti di architettura software.*
Un sistema ispeziona e produce i dati finanziari per un report. Questo report deve essere mostrato sia su un'interfaccia web e sia deve essere stampato su una stampante monocromatica. Le richieste di output sono chiaramente differenti.
Per fare un buon lavoro bisogna applicare, innanzi tutto, il principio SRP per suddividere le due responsabilità: il calcolo dei dati per il report e la presentazione di tali dati in un formato web o da stampa.
Dopo aver curato tale divisione, bisogna organizzare le dipendenze per garantire che le modifiche a una di queste responsabilità non causino modifiche nell'altra. Inoltre, l'organizzazione dovrebbe garantire che il comportamento possa essere esteso senza annullare le modifiche. Questo risultato si ottiene separando i processi in classi e suddividendo tali classi in componenti. Come illustrato nella figura i componenti comprendono: Controller, Interactor, Database, i Presenter e le View.

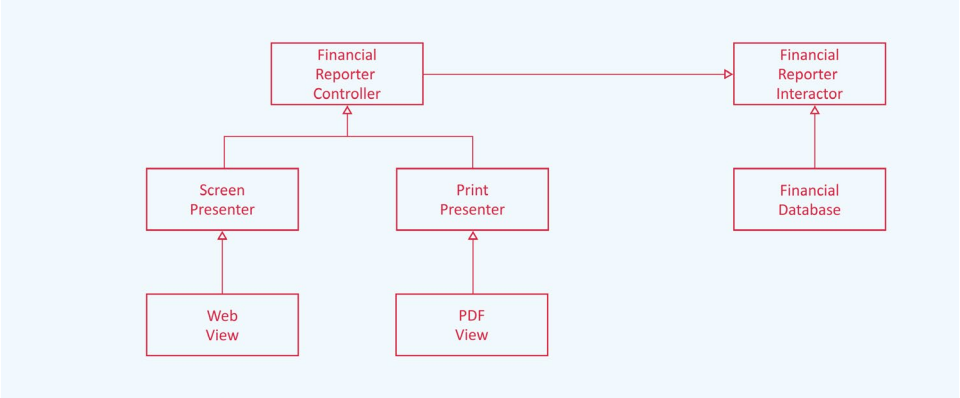


2.6 - Suddivisione dei processi in classi e separazione delle classi in componenti

Le classi contrassegnate con <I> sono interfacce, quelle contrassegnate con <DS> sono strutture dati. Le punte di freccia aperte sono relazioni di uso, mentre quelle chiuse sono relazioni di implementazione o ereditarietà.

È da notare che le linee che demarcano i componenti vengono attraversate sempre in un'unica direzione, questo significa che tutte le relazioni fra i componenti sono unidirezionali. Queste frecce si dirigono verso i componenti che vogliamo proteggere dalle modifiche.

L'interactor ha una posizione privilegiata perché contiene le regole operative, le politiche di livello più elevato dell'intera applicazione.



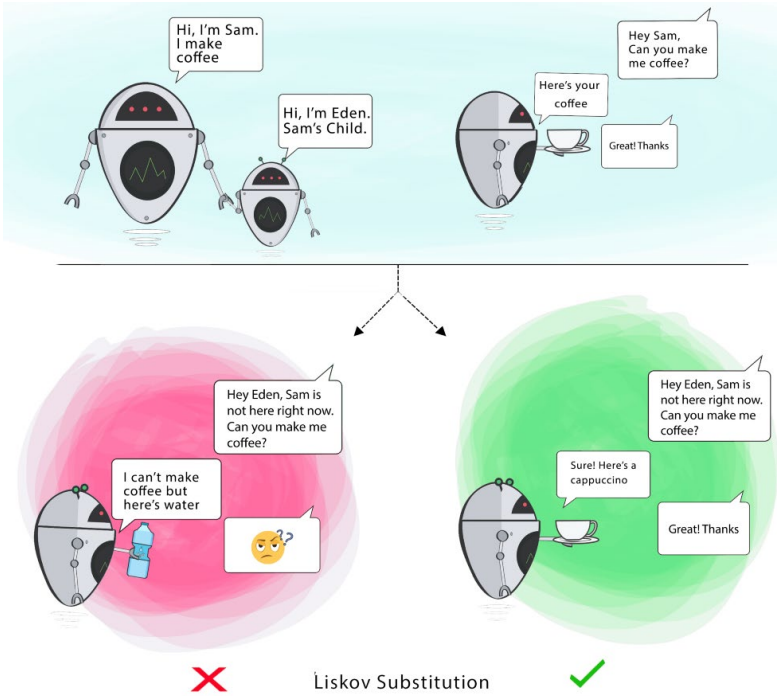
2.7 - Le relazioni tra i componenti sono unidirezionali

Questo è il modo in cui opera il principio OCP a livello dell'architettura software.

Per difendere i componenti di alto livello dalle modifiche apportate dai componenti di basso livello, le funzionalità vengono organizzate seguendo una gerarchia di componenti sulla base di come, perchè e quando cambiano.

2.2.3 L – Liskov Subsitution Principle

Se S è un sottotipo di T, allora gli oggetto di tipo T in un programma possono essere sostituiti con oggetti di tipo S senza alterare nessuna delle proprietà auspicate del programma.



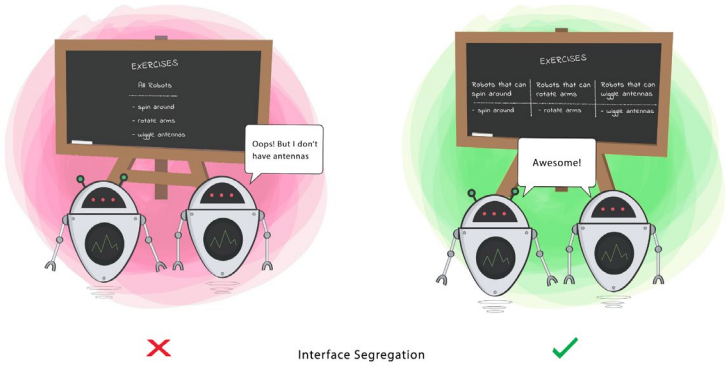
2.8 - Liskov Substitution

A livello di codice, questo concetto afferma che si dovrebbe essere sempre in grado di impiegare una sottoclasse al posto di una classe padre, senza causare alcuna modifica. Il principio assicura, inoltre, che il comportamento della classe derivata non influenza quello della classe padre. Perciò, la classe che eredita dovrebbe essere sempre in grado di processare le stesse richieste e produrre lo stesso tipo di risultato della classe padre. Se la classe figlia non rispetta questi requisiti significa che è completamente cambiata dal padre e viola il principio LSP.

Nel corso degli anni questo principio si è trasformato in un principio di più alta progettazione del software, che non riguarda solo il codice, ma anche le interfacce e le implementazioni. Le interfacce possono assumere molte forme. Si potrebbe avere a che fare con un'interfaccia Java implementata da più classi oppure si potrebbe avere un insieme di servizi i quali rispondo tutti alla stessa interfaccia REST. In queste situazioni è applicabile il principio LSP ed è importante attuarlo.

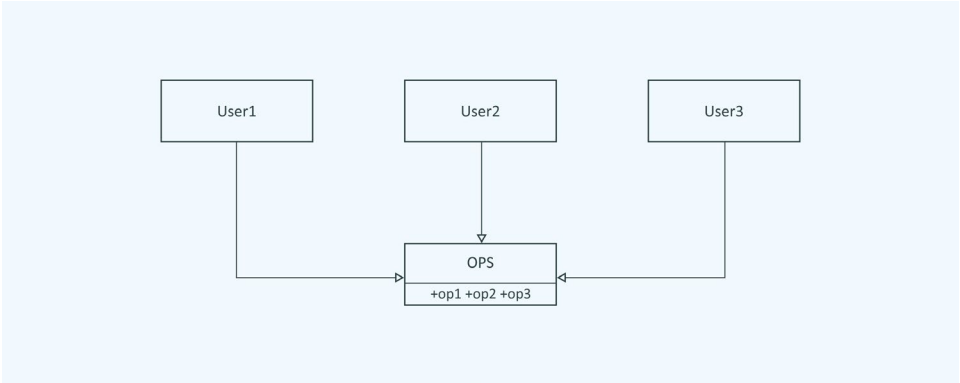
2.2.4 I – Interface Segregation Principle

Un modulo non dovrebbe essere forzato a dipendere da metodi che non usa.



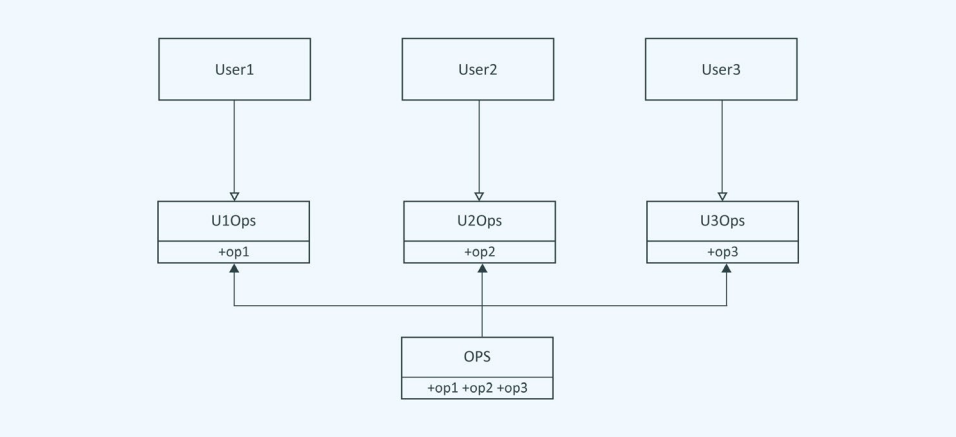
2.9 - Interface Segregation

Questo principio afferma che è preferibile che una classe implementi solo le interfacce di cui ha bisogno, quindi queste ultime devono essere molte, specifiche e di piccole dimensioni (composte da pochi metodi).
Nella situazione illustrata dalla figura 2.10 ci sono tre utenti che dipendono dalla classe OPS. Ma User1 necessita solo del metodo op1, User2 impiega solo op2 e User3 ha bisogno solo di op3.



2.10 - Il principio ISP

Questa implementazione provoca problemi poiché, in un linguaggio come Java, le modifiche effettuate al metodo op1 obbligheranno la ricompilazione e il ri-deploy di User2 e User3, anche se in realtà nulla è cambiato.
Il problema può essere risolto applicando il principio ISP, come nella figura 2.11.

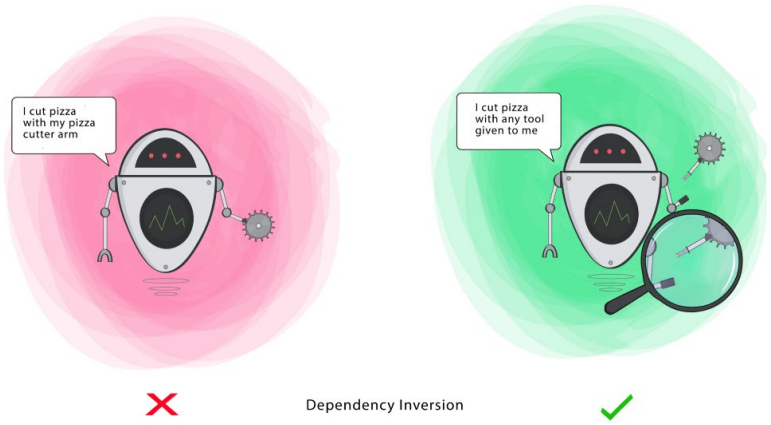


2.11 - Operazioni segregate

Osservando le frecce con attenzione, si può notare che User1 dipenderà solo dall'interfaccia U1ops che contiene il metodo op1 e non più dalla classe OPS. Pertanto, una modifica a OPS che non coinvolge User1 non richiederà la compilazione e il ri-deploy di User1.
Come gli altri principi, anche il principio ISP vale a livello di architettura.
Si prenda in esame l'esempio trattato da Robert C. Martin: un architetto deve realizzare un sistema S e intende includere in tale sistema uno specifico framework F, legato dai suoi autori a un determinato database D. Dunque, S dipenderà da F, il quale dipenderà da D.
Si ipotizzi che vi siano delle funzionalità contenute in D che, però, non vengono utilizzate da F: di conseguenza non interassano a S. Nel caso in cui queste funzionalità di D vengano modificate, potrebbe essere necessario un ri-deploy di F e di S. Nel peggiore dei casi, un difetto di funzionalità di D potrebbe provocare problemi a F e S.
Dipendere da qualcosa che contiene più del necessario è pericoloso e può provocare problemi imprevedibili.

2.2.5 D – Dependency Inversion Principle

Moduli di alto livello non dovrebbero dipendere da moduli di basso livello. Entrambi dovrebbero dipendere dalle astrazioni. Le astrazioni non dovrebbero dipendere dai dettagli. I dettagli dovrebbero dipendere dalle astrazioni.



2.12 - Dependency Inversion

Questo principio aiuta a disaccoppiare il codice in modo da ottenere la massima flessibilità dei sistemi.
In un linguaggio a tipi statici come Java, ciò significa che l'uso, l'importazione e l'inclusione di istruzioni dovrebbero solo far riferimento a moduli di codice sorgente contenenti interfacce, classi astratte o qualche altro genere di dichiarazione astratta. Non dovrebbero dipendere da nulla di concreto.
In altri termini, il principio DIP dice che la classe A non dovrebbe fondersi con la classe B per eseguire il metodo b1 contenuto in quest'ultima. Piuttosto, dovrebbe dipendere da un'interfaccia I che permetterà di connettere il metodo b1 alla classe. Inoltre, entrambe la classe A e l'interfaccia I non dovrebbero conoscere come il metodo funziona. Ma il metodo deve soddisfare le specifiche dell'interfaccia.

Questo principio verrà maggiormente discusso e spiegato a livello applicativo nei capitoli successivi.

2.3 I Componenti

I principi SOLID sono principi da applicare ai singoli moduli (classi). Molto spesso questi principi vengono associati direttamente alla programmazione a oggetti, ma Robert C. Martin ha precisato che sono applicabili a qualsiasi tipo di classe, perché essa non è che un raggruppamento di funzioni e dati. Inoltre, si è visto come questi principi possano venire impiegati anche ad un livello più elevato, quello dell'architettura.
Ora diviene opportuno parlare dei componenti: le più piccole unità che possono essere fornite nell'ambito di un sistema e sono soggette a deployment.
In parole più semplici, un componente è un raggruppamento di classi. Per esempio, una libreria è un componente, l'insieme delle *view* raggruppate in un package è un componente. In Java si tratta di file .jar.
I componenti dovrebbero essere progettati in modo da poter essere forniti e sviluppati in modo indipendente. Per ottenere questo risultato, bisogna affidarsi ad altri principi, quelli di coesione dei componenti e quelli di accoppiamento dei componenti.
Come visto in precedenza la coesione è la misura in cui due o più parti di un sistema sono correlate e come queste lavorano insieme per creare qualcosa di maggior valore rispetto a quello che fa ogni singola parte. Mentre, l'accoppiamento nel contesto dello sviluppo di software è definito come il grado in cui un modulo, una classe, o altro costrutto, è legato direttamente ad altri.

2.3.1 Coesione dei componenti

I principi riguardanti la coesione sono tre e servono per decidere quali classi appartengono a quali componenti. Le classi e i moduli che prendono corpo in un componente devono appartenere a un gruppo coeso. Il componente non può consistere semplicemente di un ammasso incontrollato di classi e moduli; al contrario, tutti i moduli devono condividere uno specifico tema o scopo.

2.3.1.1. REP – *Reuse/Release Equivalence Principle*

Una granularità di riutilizzo è una granularità di release.

Le classi e moduli che si trovano raggruppati insieme in un componente dovrebbero essere rilasciati insieme. Dovrebbe avere senso che questi condividano lo stesso numero di versione e lo stesso codice di release e siano inclusi nella stessa documentazione di release.

2.3.1.2. CCP – *Common Clouser Principle*

Raccogliete nei componenti quelle classi che cambiano per gli stessi motivi e allo stesso tempo. Separate in componenti differenti quelle classi che cambiano in momenti differenti e per motivi differenti.

Si può, dunque, evidenziare un'equivalenza tra questo principio e il *Single Responsibility Principle*.

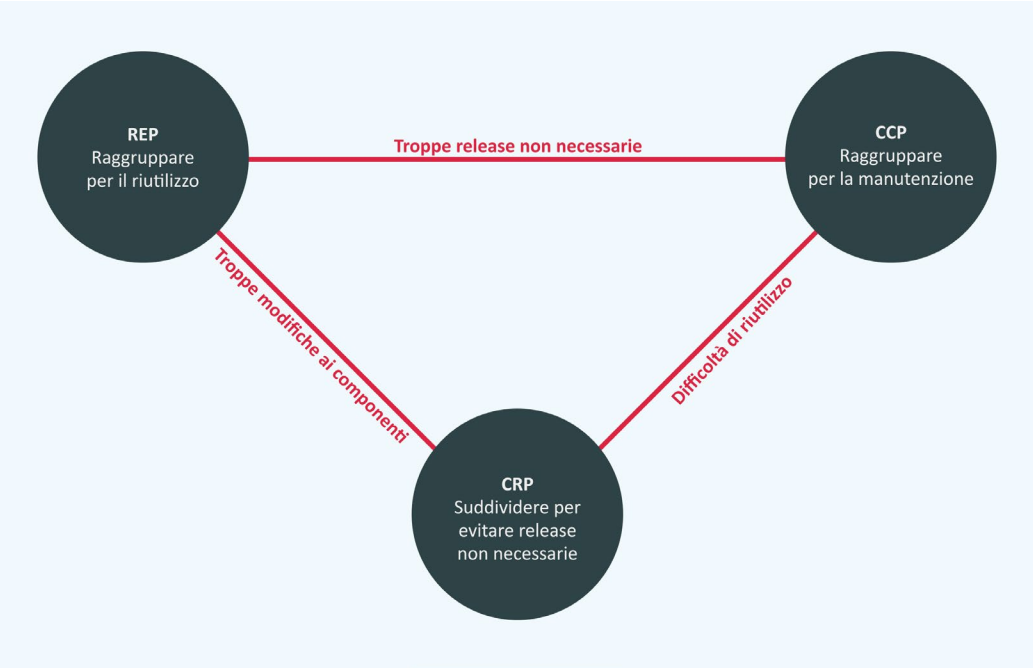
Un componente non dovrebbe avere più motivi per cambiare. Se le modifiche risultano isolate in un unico componente, allora solo lui dovrà essere ri-convalidato e ri-fornito, tutti gli altri che non dipendono da lui non ne hanno bisogno. Ciò fa sì che il carico di lavoro legato al rilascio, alla ri-convalida e al ri-deploy del software sia ridotto al minimo.

2.3.1.3. CRP – *Common Reuse Principle*

Non costringere gli utilizzatori di un componente a dipendere da cose delle quali non hanno bisogno.

Questo principio aiuta a decidere sia quali classi dovrebbero essere collocate in un componente, sia quali non mettere insieme. Le classi che dipendono maggiormente le une dalle altre dovrebbero trovarsi nello stesso componente.
Quando un determinato componente ne usa un altro, si crea una dipendenza tra i due. Anche se il componente *utilizzatore* adopera solo una classe del componente *usato*, la dipendenza tra i due componenti non è più debole.
Poichè esiste questo legame, ogni volta che qualsiasi classe del componente *usato* viene modificata, il componente *utilizzatore* dovrà probabilmente essere sottoposto a modifiche, compilato, convalidato e fornito, anche se la modifica non interessa quest'ultimo.
Di conseguenza, quando si dipende da un componente, bisogna assicurarsi di dipendere da tutto il componente e non solo da una classe. Per far ciò, bisogna che le classi appartenenti a un componente siano inseparabili, in modo che sia impossibile dipendere solo da un qualcosa e non dal resto.
Quindi, il principio *CRP* afferma che le classi che non sono strettamente legate tra loro non dovrebbero trovarsi nello stesso componente perché implicano una serie di altri problemi ai componenti utilizzatori.

In conclusione, i principi REP e CCP sono inclusivi perché entrambi tendono ad includere classi nel componente, mentre il principio CRP è esclusivo.
Per costruire un buon progetto è importante tenere a mente il diagramma di tensione per la coesione dei componenti.



2.13 - Diagramma di tensione relativo ai principi sulla coesione

Bisogna cercare di trovare una posizione ottimale in questo triangolo, la quale varierà inevitabilmente nel tempo. Nelle fasi preliminari i principi CCP e CRP sono più importanti rispetto al principio REP, perché prima ci si occupa dello sviluppo e successivamente del rilascio e riutilizzo. Per cui, inizialmente si tende a stare sul lato destro del triangolo sacrificando il riutilizzo e, successivamente, il progetto si sposterà verso sinistra poiché il progetto matura.

2.3.2 Accoppiamento dei componenti

I principi che si interessano dell'accoppiamento dei componenti sono tre e sono linee guida per trovare le giuste relazioni tra i diversi componenti.

2.3.2.1. ADP – *Acyclic Dependencies Principle*

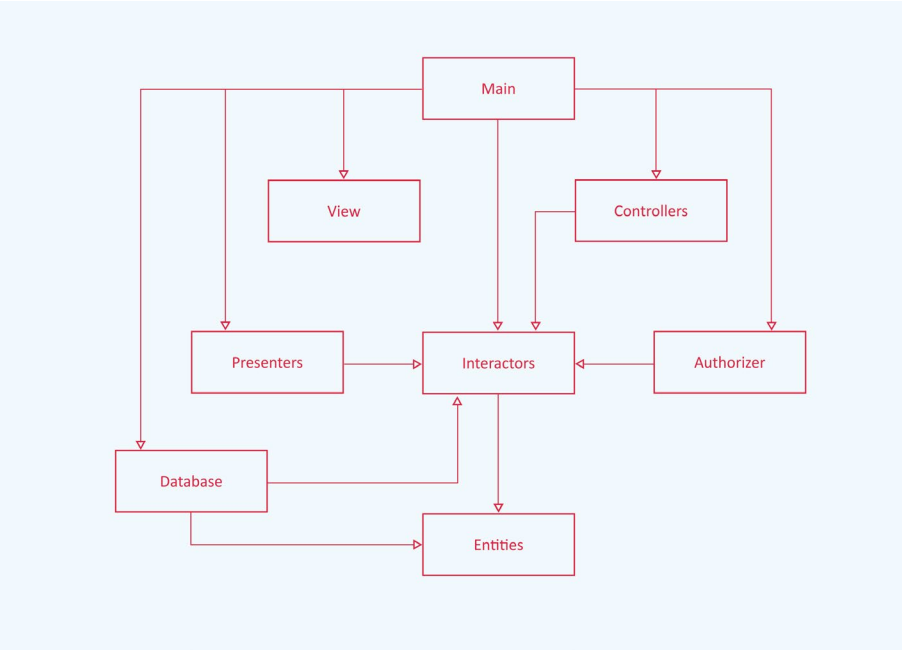
Non consentire l'insorgere di cicli nel grafico delle dipendenze fra i componenti.

Quando gli sviluppatori hanno un componente funzionante, lo rilasciano perché possa essere usato anche dagli altri sviluppatori. Man mano che si rendono disponibili nuove release di un componente, gli altri team possono decidere se adottare immediatamente la nuova release o meno.

Inoltre, l'integrazione si verifica a piccoli incrementi: non è necessario che gli sviluppatori debbano iunirsi e integrare il proprio lavoro in un unico momento.

Questo processo è semplice e razionale, ma, affinchè funzioni, occorre organizzare la struttura delle dipendenze dei componenti in modo che non vi siano cicli.

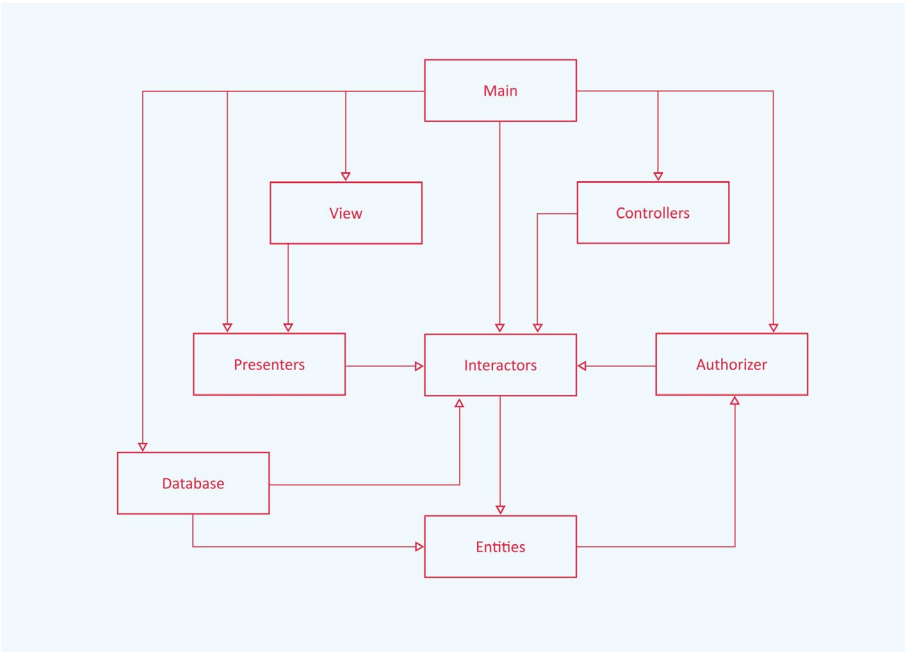
Si osservi la struttura rappresentata nella figura 2.14: indipendentemente dal componente da cui si parte è impossibile seguire le relazioni di dipendenza e ritornare a tale componente. Questa struttura non ha cicli.



2.14 - Schema di componenti senza cicli di dipendenze

Quando il team responsabile del componente *Presenter* genera una nuova release, non è difficile scoprire chi ne verrà influenzato, è sufficiente seguire a ritroso le frecce. Quindi, gli sviluppatori che stanno lavorando sui componenti *View* e *Main* dovranno decidere quando integreranno il proprio lavoro con la nuova release. Nessun altro componente sarà toccato.

Invece, nel caso in cui sia presente un ciclo di dipendenze (vedi figura 2.15 sotto), i problemi sono immediati.



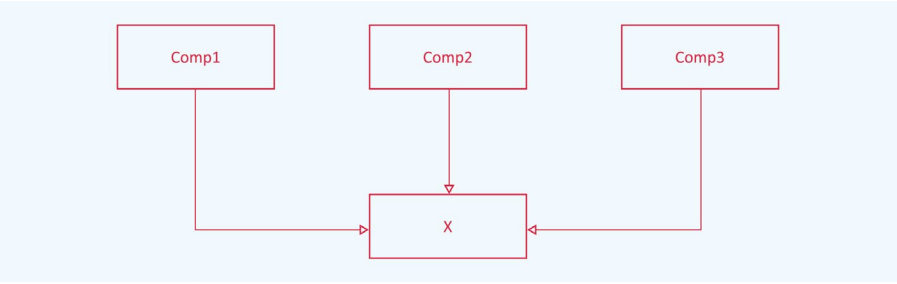
2.15 - Schema di componenti con cicli di dipendenze

Per esempio, i *developers* che operano su *Database* sanno che per produrre una release il loro componente deve essere compatibile con *Entities*, che a sua volta deve essere compatibile con *Authorizer*, ma quest'ultimo dipende da *Interactors*.
Questo rende la release molto più difficile da produrre. Inoltre, quando vi sono dei cicli nel grafico delle dipendenze, può essere molto difficile stabilire l'ordine in cui compilare i componenti. Probabilmente un ordine corretto non c'è.

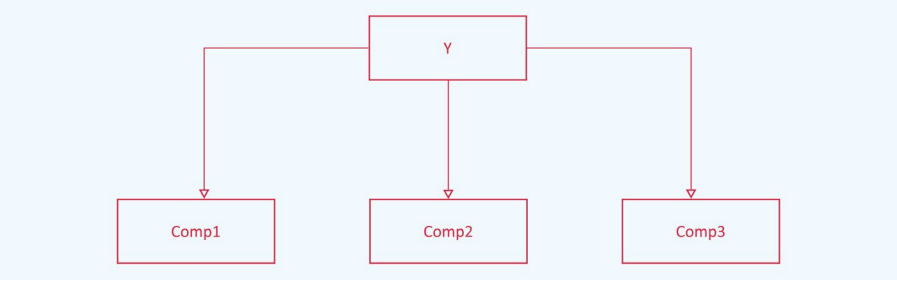
2.3.2.2. SDP – *Stable Dependencies Principle*

Puntate sulla stabilità.

La struttura di un'applicazione software non può essere completamente statica. Alcuni componenti della struttura saranno inevitabilmente volatili, cioè dovranno subire modifiche, mentre altri potranno rimanere statici.
Il termine stabilità è legato alla quantità di lavoro che deve essere eseguita per effettuare una modifica. Ciò può dipendere dalla complessità del modulo, dalle dimensioni, dalla chiarezza.
È importante che i componenti volatili non dipendano da componenti statici, altrimenti anche i volatili saranno difficili da modificare e la loro caratteristica perderebbe di significato.
Il principio SDP è una buona base per evitare questo genere di problema, assicura che i moduli che devono essere facili da modificare non dipendano da moduli che sono più difficili da modificare.
Un componente che ha molte dipendenze è molto statico, perché ci vuole molto lavoro per trasportare queste modifiche anche agli altri componenti che dipendono da esso. Al contrario, se un componente non ha nessuno che dipende da esso è volatile, si può quindi dire indipendente.



2.16 - X è un componente stabile



2.17 - Y è un componente molto instabile

Per misurare la stabilità di un componente si utilizzano le seguenti metriche che contano semplicemente il numero di ingresso e in uscita delle dipendenze in un componente:

- *Fan-in*: dipendenze in ingresso. Cioè il numero di classi esterne che dipendono dalle classi contenute nel componente in questione;
- *Fan-out*: dipendenze in uscita. Cioè il numero di classi interne al componente che dipendono da classi esterne;
- *I (instabilità)*: I = fan-out / (fan-in + fan-out). I è compreso nell'intervallo [0,1], dove I=0 indica un componente massimamente stabile, mentre I = 1 indica un componente massimamente instabile.

Il principio SDP afferma che la metrica I di un componente dovrebbe essere maggiore delle metriche dei componenti da cui esso dipende. Cioè, i componenti più instabili devono dipendere da componenti meno instabili.

2.3.2.3. SAP – *Stable Abstracions Principle*

Un componente dovrebbe essere tanto astratto quanto è stabile.

Come si è visto, quindi, il software si divide in componenti volatili e componenti stabili. La parte di software che non cambierà molto spesso sono le politiche di alto livello, per cui dovrebbero essere costituite da componenti massimamente stabili (I=0), mentre le decisioni operative di basso livello, che dipendono dalle politiche, devono essere rappresentate da componenti il meno instabili possibile (I=1). Tuttavia, se i dettagli di alto livello devono dipendere dalle politiche di alto livello, le quali sono stabili, allora anche il codice dei componenti instabili diverrà difficile da modificare. Questo legame può rendere l'architettura poco flessibile.

Per ovviare a questo problema bisogna, far affidamento sul principio SAP, il quale dice che è possibile e desiderabile creare classi che siano sufficientemente flessibili da poter essere estese senza richiedere modifiche: le classi astratte.

Pertanto, un componente per essere stabile deve essere anche astratto, ciò significa che deve contenere sia classi difficili da modificare sia classi astratte e interfacce, che permettono di rendere il codice flessibile e permettono al componente di essere esteso. D'altro canto, un componente instabile deve essere concreto, cioè avere classi con codice modificabile concretamente e con facilità.

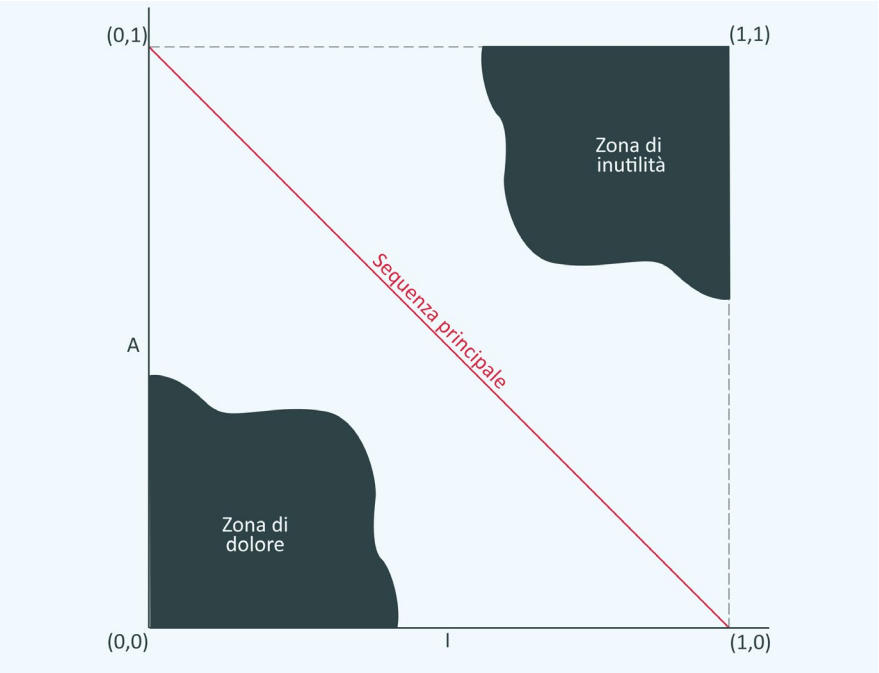
Dunque, le dipendenze sono rivolte nella direzione dell'astrazione.

Anche in questo caso esiste una metrica per misurare l'astrazione:

- *Nc*: numero di classi in un componente;
- *Na*: numero di classi astratte in un componente;
- *A (astrattezza)*: A = Na / Nc.

Il valore di A è sempre compreso in un intervallo [0,1], dove A=0 indica un componente senza nessuna classe astratta, mentre A=1 implica che il componente contiene solo classi astratte.

A questo punto si può definire la relazione tra stabilità (I) e astrattezza (A) tramite il grafico che rappresenta le zone di esclusione (figura 2.18).



2.18 - Le zone di esclusione

Si consideri un componente nei pressi di (1,1). Questo sarà massimamente astratto e massimamente instabile, ma è inutile poiché è progettato per avere delle classi che dipendono da esso, ma in pratica non ne ha. Tali componenti sono inutili, spesso sono una sorta di detriti con alcune classi non più utilizzate.
All'opposto, un componente nei dintorni di (0,0) è molto stabile e concreto, ciò significa che è difficile da modificare e non è astratto, quindi non può essere esteso. Ne risulta un componente molto rigido.

Di conseguenza, il compito dell'architetto software è quello di cercare di collocare i componenti sulla *sequenza principale* (termine assegnato da Uncle Bob), lontano dalle zone di dolore e di inutilità. La posizione più desiderabile è a uno dei due estremi della sequenza, dove sono perfettamente instabili e concreti oppure massimamente stabili ed astratti.

2.4 I Livelli della Clean Architecture

L'abilità dell'architetto software è quella di disporre i componenti in un grafo aciclico. I nodi del grafo rappresentano i componenti che contengono le politiche che si trovano allo stello livello. Gli archi sono le dipendenze tra tali componenti, connettono i componenti che si trovano a livelli differenti.
Una definizione di livello è "la distanza dagli input e dagli output". Per cui, più un componente è lontano dagli input e dagli output, maggiore è il suo livello. Mentre le politiche che gestiscono gli input e gli output appartengono ai livelli più bassi del sistema.
Ciò che segue descriverà nel dettaglio i livelli della Clean Architecture: Entità, Casi D'uso, Adattatori di Interfacciamento e Framework e Driver.

2.4.1 Entità

Un'entità è un oggetto che contiene una parte delle regole operative critiche di business. Le regole operative definiscono gli aspetti di business, descrivono le operazioni, le definizioni e i vincoli che si applicano a un'organizzazione. Esistono varie tipologie di regole operative, quelle critiche sono quelle che esisterebbero anche se non vi fosse un sistema automatico ad applicarle. Per esempio, il fatto che una banca addebiti l'N percento di interessi per un prestito è una regola operativa che permette alla banca di accumulare denaro. Non è rilevante se a calcolare gli interessi sia un computer o un impiegato.
Se non si sta scrivendo un'applicazione per un'azienda ma se si lavora ad una qualsiasi altra semplice applicazione, le entità devono comunque incapsulare le regole più generali e di alto livello. Per esempio, un'applicazione che mostra l'elenco delle serie tv in uscita su una specifica piattaforma nel giorno corrente, dovrà avere come entità un oggetto che rispecchia la serie tv. Quindi, un modello con il titolo, gli attori, la descrizione e così via.
Questa classe rappresenta l'operatività richiesta e viene separata da qualsiasi altro elemento presente nel sistema. Esiste questa regola e non deve essere toccata. Le entità sono del tutto ignare degli altri livelli, rappresentano un compito puro.

2.4.2 Casi d'uso

Non tutte le regole operative sono pure come le entità. Alcune regole operative sono vincolate dal sistema automatizzato in cui stanno operando e non verrebbero utilizzate in un ambiente naturale. Queste sono le cosiddette regole operative specifiche dell'applicazione e vengono descritte dai casi d'uso.
Un caso d'uso è una descrizione del modo in cui viene utilizzato un sistema automatizzato, contiene le regole che specificano come e quando verranno richiamate le regole operative critiche, contenute nelle entità. Sono i casi d'uso a controllare le relazioni fra le entità. Mentre le entità non sono a conoscenza dei casi d'uso che le controllano.
Riprendendo l'esempio precedente delle serie tv, mentre la serie tv è rappresentata

dall'entità, il caso d'uso di cui si ha bisogno è quello che descrive come ottenere l'elenco delle serie tv della giornata.
È importante sottolineare, però, che è irrilevante per i casi d'uso il modo in cui i dati entrano ed escono dal sistema. I dati arrivano nella forma desiderata e altri dati di output vengono prodotti. Il modo in cui, successivamente, questi dati verranno comunicati all'utente o a qualsiasi altro componente non deve interessare il caso d'uso.
"*Le regole operative sono il motivo per il quale esiste un sistema software, in quanto ne rappresentano le funzionalità centrali. […]*
Le regole operative dovrebbero rimanere pure, incontaminate, non toccate da dettagli come l'interfaccia utente o il database impiegati. Idealmente, il codice che rappresenta le regole operative dovrebbe essere il cuore del sistema, al quale si connettono tutti i dettagli minori" (Robert C. Martin, *Clean Architecture*).

2.4.3 Adattatori di Interfacciamento

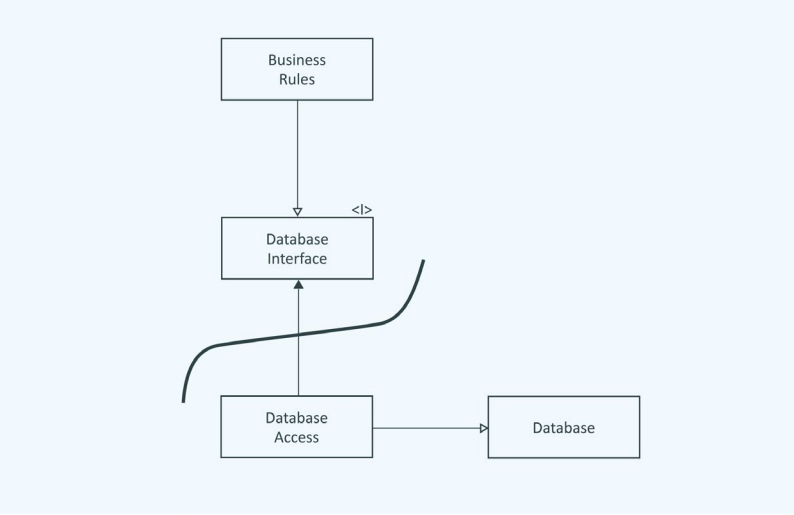
Gli *adapters* sono i traduttori tra il dominio e l'infrastruttura. Sono coloro che convertono i dati dal formato più appropriato per i casi d'uso e le entità, al formato più conveniente per un qualche attore esterno, come il database o il web.
Le viste e i controller apparterranno a questo livello, ma anche i gateway e i data mapper.
I gateway per il database sono interfacce che contengono i metodi relativi a ogni operazione di creazione, lettura, aggiornamento e cancellazione che l'applicazione può dover svolgere sul database. Per esempio, se l'applicazione deve conoscere l'elenco delle serie tv in uscita nel giorno corrente, allora l'interfaccia *SeriesGateway* avrà un metodo che accetta un argomento Data e restituisce un elenco di serie tv.
I data mapper, invece, sono oggetti che eseguono il trasferimento bidirezionale di dati tra un database e una sua rappresentazione dei dati nell'applicazione. In questo modo la rappresentazione in memoria e l'archivio dei dati rimangono indipendenti.

2.4.4 Framework e Driver

Questo layer è costituito da tutti i componenti di input e output: framework, database, UI, dispositivi, e così via. Generalmente non si deve scrivere molto codice in questo livello, tranne il codice di collegamento che comunica con il livello interno successivo.
Qui si trovano i componenti più volativi e poiché queste classi sono solite cambiare, questi componenti sono tenuti il più lontano possibile dalle politiche di alto livello.
In questo livello si trovano tutti i dettagli: il Web, il database, il framework.
La struttura dati che si sceglie per l'applicazione è significativa per l'architettura del sistema, ma il database non è il modello dei dati. Il database è un servizio che fornisce l'accesso ai dati, per cui, dal punto di vista dell'architettura, è irrilevante perché rappresenta un meccanismo, ovvero un dettaglio di basso livello.

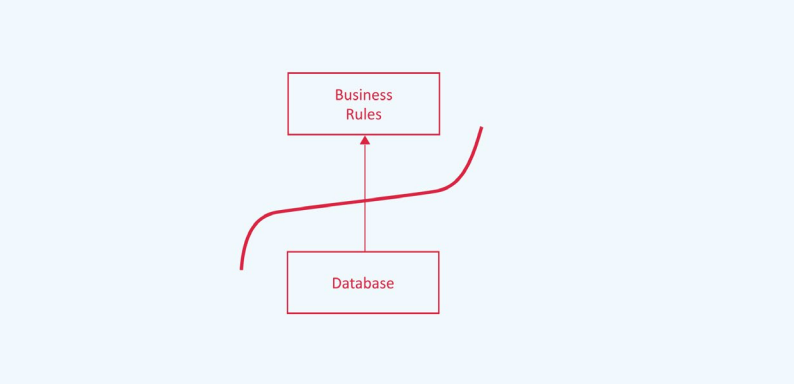
2.5 Attraversamento delle Delimitazioni

La regola principale della Clean Architecture è la *regola delle dipendenze*, la quale afferma che "*le dipendenze presenti nel codice sorgente possono puntare solo all'interno, verso le politiche di alto livello*" (Robert C. Martin, *Clean Architecture*). I livelli interni non devono conoscere nulla di quello che si trova al di sopra di essi, in particolare niente di quello che si trova nei livelli più esterni deve essere menzionato dai livelli più interni.
In questo modo ci si assicura che il framework, il database, la UI e tutto quello che riguarda la "tecnologia" scelta non influenzi la *business logic* dell'applicazione.
Come si viaggia, allora, tra i vari livelli?
In primo luogo, per obbedire alla regola delle dipendenze bisogna tracciare delle delimitazioni tra ciò che conta e ciò che non conta, cioè tra le regole operative e i plugin, cioè gli aspetti funzionali all'applicazione (DB, UI, framework, …). Nel momento in cui si sono separati questi aspetti in componenti differenti, tutto ciò che non è necessario conoscere si può mettere dietro a un'interfaccia.
Come si può vedere dalla figura 2.19, le logiche operative non hanno bisogno di conoscere la tipologia di database utilizzato, ma soltanto che esiste un insieme di funzioni che leggono e scrivono dati. Per cui, si utilizza un'interfaccia del database. Il *DatabaseAccess* implementa tale interfaccia e rigira l'operazione verso il database vero e proprio. In questo modo né le politiche di alto livello, né il database sanno nulla dell'esistenza dell'altro.



2.19 - La linea di delimitazione

La linea di delimitazione che divide le classi nei rispettivi componenti, va tracciata lungo la direzione di ereditarietà. Questo perché la classe *BusinessRules* che rispecchia le politiche di alto livello non può avere una dipendenza che va verso un componente di più basso livello. Perciò, *BusinessRules* e *DatabaseInteface* faranno parte del componente *Business Rules*, mentre *DatabaseAccess* e *Database* faranno parte del componente *Database*.
Il componente *Database* saprà dell'esistenza del componente *BusinessRules*, ma non viceversa.



2.20 - La linea di delimitazione

Avendo tracciato in questo modo la linea di delimitazione, ora le Business Rules non dipendono dal database, quindi possono usare qualsiasi tipologia. Se oggi si decidesse di usare un database SQL e domani, invece, si avesse bisogno di un database non relazionale, bisognerà intervenire solamente sul componente che riguarda il database, senza intaccare le logiche operative.

2.6 Conclusioni

Si sono tracciate le linee guida per progettare un sistema di alto livello in termini di qualità. Bisogna, però, tenere a mente che la Clean Architecture è solo uno dei possibili stili che si possono adottare per il proprio progetto, non l'unico ed imprescindibile.
Rimane, comunque, un'architettura complessa da applicare a livello pratico che richiede una vasta gamma di competenze e molta esperienza nel campo della programmazione.
Tuttavia, è bene cercare di tendere a questo stile architettonico seguendo i principi elencati precedentemente e avvicinarsi, così, pian piano a un progetto pulito che rispetti i requisiti di sviluppo, distribuzione e manutenzione.

Capitolo 3

Pattern Architetturali - MVC, MVP e MVVM

3.1 Stili Architetturali, Pattern o Design Pattern?

Fino ad adesso si è parlato di *Architettura del Software* e in particolare della Clean Architecture. Si è visto che quest'ultima è solo uno fra i tanti modi che un'azienda può adottare per sviluppare il proprio software. La Clean Architecture è uno **stile** architetturale, che indica per sommi capi come organizzare il codice. È il livello più alto di granularità e fornisce delle linee guida importanti per organizzare i componenti di alto e basso livello di un'applicazione e le relazioni che intercorrono fra essi.
Nelle scelte architetturali rientrano le decisioni sull'ambiente tecnico di sviluppo, le politiche, i frameworks e tutte le questioni viste in precedenza.

I **pattern** architetturali, invece, descrivono delle soluzioni specifiche per implementare un certo stile architettonico a livello di moduli. Ad esempio, "quali classi saranno effettivamente presenti e come interagiranno fra di loro al fine di implementare un sistema con un insieme specifico di livelli?"
I pattern architetturali di cui si parlerà in questo capitolo sono i seguenti:

- MVC (*Model-View-Controller*)
- MVP (*Model-View-Presenter*)
- MVVM (*Model-View-ViewModel*)

Il pattern *Model-View-ViewModel* è quello di rilevanza maggiore in relazione a questa tesi, poiché è il modello che si segue per sviluppare applicazioni basate sulla Clean Architecture. Il pattern MVVM è l'evoluzione dei due precedenti, per cui ognuno verrà trattato singolarmente per capire i cambiamenti e come i problemi che recavano le prime scelte siano stati risolti.

Per non creare dubbi si riporta anche la definizione di **Design Pattern**, un termine che spesso accompagna gli stili e ai pattern architetturali:

> *"I design pattern sono soluzioni tipiche per problemi comuni ricorrenti nel software design. Sono come progetti predefiniti che si possono personalizzare per risolvere un problema di progettazione ricorrente nel codice"*[1].

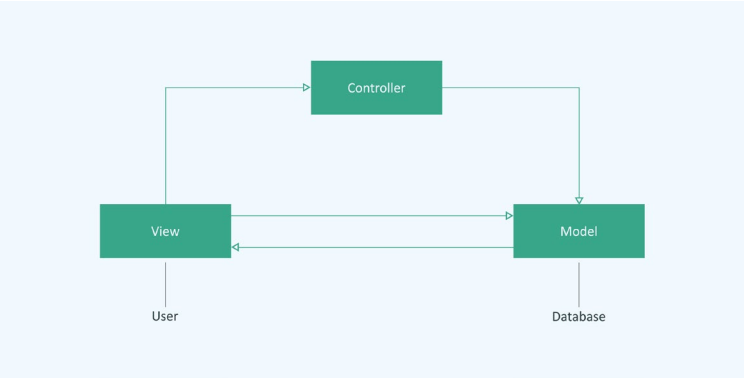
I design pattern influenzano una sezione del codice. Esempi di design pattern sono:

- Repository Pattern
- Facade Pattern
- Utilizzo di singleton

3.2 MVC

L'acronimo MVC intende Model-View-Controller, un pattern architetturale che ha come scopo quello di separare l'applicazione in tre componenti logici principali: *Model*, *View* e *Controller*. Ognuno di questi componenti è costruito per gestire un aspetto specifico dell'applicazione.
L'architettura MVC è stata introdotta per la prima volta da Trygve Reenskaug nel 1979. Tradizionalmente veniva usata per GUI desktop, ma attualmente il pattern MVC è diventato popolare sia per applicazioni desktop che per applicazioni mobile.

L'architettura MVC si può riassumere con il seguente schema:



3.1 - Pattern MVC

Model

Il componente *Model* gestisce i dati e la loro relativa logica, per questo è connesso con il database. Poiché il *Controller* non comunica mai direttamente con il database, è il *Model* che deve gestire le richieste del *Controller* e inoltrargli i dati desiderati.

1 https://refactoring.guru/design-patterns/what-is-pattern

View

Questo componente si occupa della rappresentazione dei dati all'utente. Le *View* vengono create dai dati raccolti dal componente *Model* e ricevuti dal *Controller*. Inoltre, monitorano i modelli per intercettare cambiamenti di stato ed effettuare l'eventuale aggiornamento. Per cui, *Model* e *View* interagiscono tra loro.

Controller

Il *Controller* è il componente che gestisce l'interazione dell'utente. Come descritto sopra, il *Controller* non deve preoccuparsi di gestire la logica dei dati, ma deve solamente annunciare al *Model* quali sono gli ordini in arrivo dall'utente. Dopo aver ricevuto i dati, li elabora e invia le informazioni alla *View*.

Vantaggi

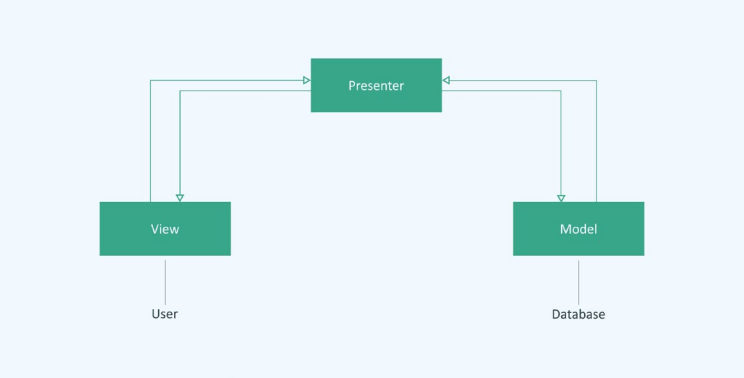
- Lo sviluppo dei vari componenti può essere eseguito parallelamente;
- Facile manutenzione del codice;
- I componenti possono essere distribuiti in modo indipendente.

Svantaggi

- La complessità è alta;
- Non adatto per piccole applicazioni;
- È richiesta la conoscenza di più tecnologie;
- Manutenzione di molti codici nel controller;
- Non c'è una completa separazione tra business logic e interfaccia.

3.3 MVP

Questo pattern architetturale è molto simile al Model-View-Controller. Anche qui, si separa l'applicazione in tre componenti principali: *Model*, *View* e *Presenter.*



3.2 - Pattern MVP

Model

Tale componente rappresenta le classi che descrivono i dati e la logica aziendale, quindi definisce le business rules che indicano come i dati possono essere modificati e manipolati.

View

View è il componente che interagisce direttamente con l'utente e non ha implementata nessuna logica. Trasmette solamente i comandi dell'utente al *Presenter*.

Presenter

È il componente di mezzo, agisce sia sul *Model* che sulla *View*. Riceve gli input degli utenti tramite la *View*, elabora i dati grazie all'aiuto del *Model* e ritrasmette i dati aggiornati alla *View*. È totalmente disaccopiato dalla *View* e comunica con questa tramite interfacce. Le classi presenti nella *View* implementano tale interfaccia e passano i dati nel modo desiderato da questa.

Vantaggi

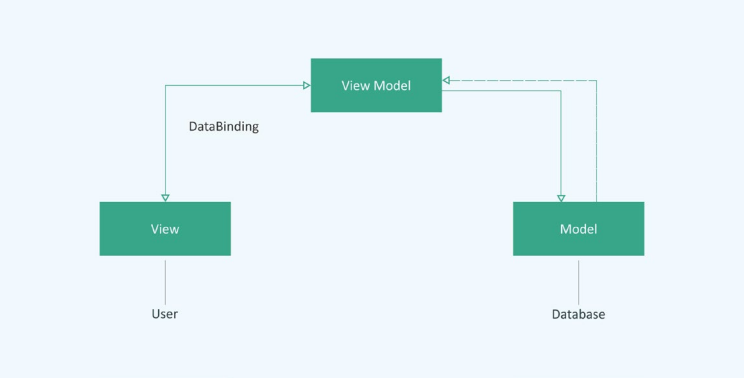
- La *View* e il *Model* sono completamente disaccoppiati;
- Test automatici del Presenter e del *Model* più semplici.

Svantaggi

- Il *Presenter* tende ad avere classi troppo grandi con troppe responsabilità;
- Complessità crescente;
- Non può venire utilizzato nel caso in cui l'interfaccia utente viene aggiornata senza l'interazione dell'utente.

3.4 MVVM

Il pattern Model-View-ViewModel è discendente del pattern MVP, ma in questo caso supporta il data binding bidirezionale tra la *View* e il *View Model*. Questo permette la propagazione automatica dei cambiamenti tra i due componenti.



3.3 - Pattern MVVM

Model

Rappresenta l'insieme di classi, strutture dati e algoritmi che contengono le politiche di alto livello.

View

È costituito dall'insieme di classi, strutture dati e algoritmi che interagiscono con il *View Model* per visualizzare i dati esposti e passano al *View Model* eventuali azioni da parte dell'utente che necessitano di elaborazione locale o remota.

View Model

Anche in questo caso è il componente intermediario tra il *Model* e la *View*. Tramite il *Model* ottiene i dati nativi e li trasforma in dati pronti per essere visualizzati e presentati all'utente.

Il meccanismo fondamentale contenuto nel *View Model* è il *data binding*: le modifiche ai dati apportate dall'utente attraverso la *View* vengono automaticamente riportate nel *View Vodel* e viceversa, le modifiche che avvengono ai dati del *View Model* vengono automaticamente rappresentate nella *View*.

Il *View Model* è completamente indipendente dalla *View*. Più viste possono avere dipendenze dallo stesso *View Model*, ma il *View Model* non dipende da nessuno.

Vantaggi

- Il codice è più facile da manutenere;
- Separazione della business logic dalla *View;*
- Lo sviluppo dei vari componenti può essere eseguito parallelamente;
- Unit testing più semplice;
- Aggiornamento automatico delle viste tramite pattern *Observable.*

Capitolo 4

Applicazione della Clean Architecture in Synesthesia

Dopo aver approfondito gli aspetti teorici, questo capitolo é dedicato alla declinazione operativa dei concetti discussi precedentemente.
L'applicazione della Clean Architecture, seguendo il pattern architetturale *MVVM,* verrà eseguita per lo sviluppo di un'applicazione mobile sul sistema operativo Android.

Si è scelto di lavorare in ambito mobile poiché, come è ben noto, negli ultimi anni la tecnologia mobile e gli smartphone hanno conosciuto una diffusione enorme. Esistono applicazioni per svolgere qualsiasi genere di compito, dalle svariate app per gestire i social network ad applicazioni utili ad indicare in tempo reale se si stanno lavando bene o meno i denti con il proprio nuovo spazzolino digitale, ad applicazioni ufficiali del Governo Italiano, quali l'app Immuni, per combattere la diffusione del COVID-19.

Si è preferito sviluppare in ambiente Android poiché rientra tra le competenze del corso di studi magistrale di Ingegneria del Cinema e dei Mezzi di Comunicazione e, poichè Android è una piattaforma per dispositivi mobili completa, aperta e gratuita.

Quello che segue è la spiegazione dell'architettura delle app Android sviluppate da Synesthesia, scritte in Kotlin e basate sulla Clean Architecture, seguendo il pattern *Model-View-ViewModel*.

4.1 Organizzazione dell'Architettura

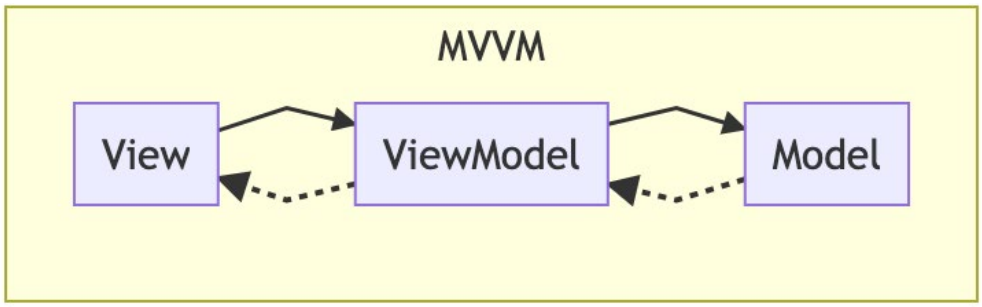
Come spiegato in precedenza, il pattern *MVVM* suddivide l'applicazione in tre macro-aree principali (*layer*) che sono:

Model
Insieme di classi, strutture dati e algoritmi che prelevano dati grezzi dall'esterno e li trasformano in oggetti nativi Kotlin.

ViewModel
Insieme di classi, strutture dati e algoritmi che interagiscono con il Model, ne ottengono dati nativi e li trasformano in dati pronti per essere visualizzati e presentati all'utente, eventualmente applicando la business logic imposta dal progetto.

View
Insieme di classi, strutture dati e algoritmi che interagiscono con il view model per visualizzare su schermo i dati esposti, senza necessità di modificarli e passano al view model eventuali azioni da parte dell'utente che necessitano di elaborazione locale o remota.

Il grafico di dipendenza dei vari layer è il seguente:



4.1 - Pattern MVVM

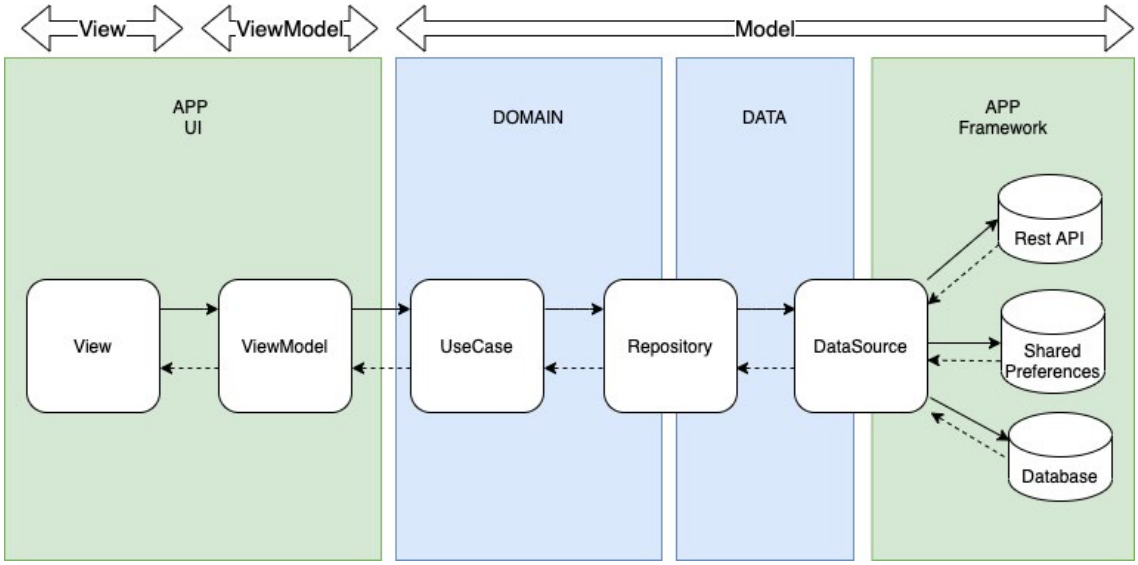
In particolare:

- le linee continue (da sinistra verso destra) indicano dipendenza diretta (o referenza diretta). Ad esempio, un oggetto appartenente al layer View ha accesso diretto (tramite variabile o metodo) a un oggetto appartenente al layer del view model;
- le linee tratteggiate (da destra verso sinistra) indicano l'osservazione di valori nel tempo. Ad esempio, il view model espone dei valori che vengono osservati dalla view senza che il view model ne sia a conoscenza ne abbia un riferimento diretto.

A livello di dipendenze, il pattern prevede che ogni singolo layer sia funzionante indipendentemente dalla presenza del layer alla sua sinistra. In teoria, il model può essere preso

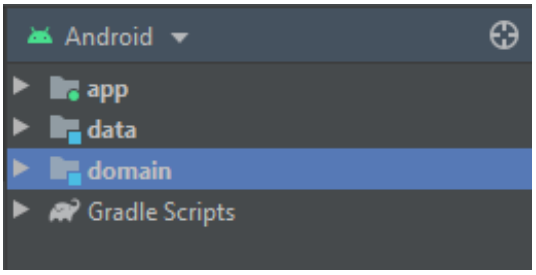
e spostato in un altro progetto e continuare a funzionare, senza view model e view del progetto precedente.

A livello di codice, i tre layer concettuali hanno una suddivisione un po' diversa. Utilizzando i concetti della Clean Architecture, e le caratteristiche della piattaforma Android, il progetto standard di Synesthesia è composto da tre moduli gradle (App, Data e Domain). App è un modulo di tipo `android-application`, mentre Data e Domain sono di tipo `Java-library`. Data e Domain non dipendono dal framework Android, contengono solo classi Kotlin e sono quindi completamente testabili con JUnit.
App, al contrario, dipende dal framework Android e si occupa quindi della presentazione e di recuperare i dati (da internet, dalla memoria del dispositivo, ecc).



4.2 - Architettura app Synesthesia

La distinzione tra i moduli che dipendono e quelli che, invece, non dipendono dal framework Android si può notare anche in Android Studio, il quale etichetta il package che rappresenta il modulo app con un cerchio verde, mentre gli altri due con un quadrato azzurro.



Domain

Contiene le Entity, gli oggetti che rispecchiano le logiche di business, gli UseCase, che rappresentano le logiche di business e le interfacce dei Repository, che contengono le dichiarazioni dei metodi che servono agli UseCase per recuperare i dati necessari.
Ogni UseCase svolge esclusivamente un compito soltanto, può utilizzare uno o più repository e altri UseCase.

Nel grafico della figura 4.2 a pag. 59, Entity e UseCase sono rappresentate dal quadrato UseCase.

Data

"Conosce" il layer Domain, ma non viceversa. Vede quindi le interfacce dichiarate in Domain e può implementarle.
Al suo interno ci sono le implementazioni dei repository dichiarati nel modulo Domain e le interfacce dei DataSource.
Un DataSource rappresenta una fonte di dati, ad esempio API remote o DB locali.
Un Repository può utilizzare uno o più DataSource.

App

Il modulo App conosce entrambi i moduli precedenti e contiene sia le implementazioni dei DataSource, sia tutta la parte di presentazione basata su pattern MVVM.
Il layer è diviso in due sezioni principali, framework e UI.

- Framework contiene l'implementazione dei DataSource. Le implementazioni dei DataSource sono nel modulo App perché è l'unico modulo che dipende dal framework Android e può quindi contenere implementazioni specifiche della piattaforma (es. database oppure chiamate di rete). In questo modulo sono presenti le response, che sono gli oggetti remoti che vengono ricevuti dalle API e i Mapper necessari a trasformarli nelle Entity. La ragione di questo sdoppiamento è la possibilità di essere indipendenti dalle strutture dati in arrivo dai backend. I datasource si occupano, tra le altre cose, di utilizzare i mapper per trasformare le response in Entity.
- UI: è composto da classi del framework android (Activity,Fragment,View), si occupa della UI vera e propria e dei viewmodel per collegare i dati alle view.
  I ViewModel utilizzati sono i ViewModel degli ArchitectureComponents di Google[1].
  Le activity o i fragment osservano cambiamenti di variabili osservabili all'interno dei viewmodel per aggiornare la view di conseguenza.
  Le variabili osservabili utilizzate all'interno dei viewmodel sono LiveData[2].
  Il vantaggio dell'utilizzo dei LiveData per questo compito è connesso al fatto che i LiveData sono strettamente legati al ciclo di vita delle activity/fragment, evitando di avere aggiornamenti su view non più visibili.
  Abbiamo introdotto il concetto di Resource, che è un wrapper di tre differenti LiveData, i quali forniscono aggiornamenti sugli stati di success, failure e loading.
  Osservando una singola resource all'interno di una activity o un fragment è possibile ricevere aggiornamenti per i tre diversi stati.

1 https://developer.android.com/topic/libraries/architecture/viewmodel
2 https://developer.android.com/topic/libraries/architecture/livedata

I viewmodel dichiarano le Resource che saranno visibili ad activity/fragment e utilizzano gli useCase per reperire i dati, validare gli input…
Tramite il metodo addDisposable collegano i risultati degli useCase alle Resource.

Si può notare dal grafico della figura 4.2 a pag. 59, che i componenti Repository e Data Source sono a metà tra un modulo e l'altro. Questo perché, come descritto nella spiegazione dei vari moduli, nel caso del Repository il modulo Domain contiene le interfacce dei Repository, mentre il modulo Data ne contiene l'implementazione. Lo stesso discorso vale per i Data Source: il modulo Data contiene le interfacce mentre il modulo framework le implementazioni.
Ciò mette in luce l'applicazione di uno dei principi cardine della Clean Architecture, il principio DIP (Dependency Inversion Principle):

"*moduli di alto livello non dovrebbero dipendere da moduli di basso livello. Entrambi dovrebbero dipendere dalle astrazioni. Le astrazioni non dovrebbero dipendere dai dettagli. I dettagli dovrebbero dipendere dalle astrazioni*".

Le astrazioni sono le interfacce, le quali rappresentano dei punti di contatto per permettere, per esempio, allo use case che si trova nel layer più interno di non sapere chi implementa il repository e quali siano le logiche di quel repository.
In questo modo si rispetta la regola della dipendenza, per cui i layer più interni non devono conoscere i layer più esterni, mentre il contrario sì.

4.2 Il Work Flow

Il flusso si può riassumere nel seguente modo: una schermata si appoggia a un view model che utilizza uno use case, il quale passa da un repository, che chiama un data source e quest'ultimo, infine, si affida alla chiamata vera e propria.

Il passaggio dalla view al view model è necessario per separare le proprietà dei dati della vista dalla logica.
Il framework Android gestisce il ciclo di vita delle view (Activity e Fragment), ciò significa che può crearle e distruggerle in risposta a determinate azioni dell'utente o ad eventi del dispositivo, senza che lo sviluppatore ne abbia controllo. Se un Activity contiene dei dati, nel momento in cui il sistema la distrugge o la ricrea, tutti quei dati archiviati verranno persi. Per questo motivo è opportuno utilizzare i view model. In queste classi si conservano i dati che si dovranno mostrare all'utente e si implementa la logica necessaria.
Il view model si serve degli use case, invece di cercare il metodo che gli serve direttamente nel repository, il quale magari può contenere decine di funzioni. Ogni use case implementa un solo caso d'uso, una sola funzione. Il view model importa i casi d'uso di cui necessita.

L'ultimo passaggio da chiarire è quello da Use Case a Repository a Data Source.
Il data source è una fonte di dati, ad esempio un server. Normalmente un'applicazione Synesthesia ha due data source: remoto (*API Rest*) e locale (*Shared Preferences*). In alcuni casi è necessario aggiungere un terzo data source per il database relazionale o non relazionale. Quindi, il data source suddivide i dati per tipologia di fonte. Il repository occorre per aggiungere un livello di astrazione e semplificare il codice.
Per esempio, un'applicazione di grandi dimensioni può avere un data source remoto che contiene cento metodi diversi perché ha cento API diverse da chiamare. Se uno use case necessitasse di anche solo uno dei metodi di quel data source, dovrebbe importarlo comunque tutto.
Con l'utilizzo dei repository si va a suddividere e raccogliere i metodi per concetti (utenti, film, serie tv, …) ed è poi lui che si appoggia ai data source di cui ha bisogno.
L'altro motivo, ovvio, di queste astrazioni è la separazione della business logic dal framework. Lo use case si aspetta un determinato risultato: per esempio, la lista delle serie tv preferite dell'utente. Per ottenerlo si affida a un repository ed è quest'ultimo che tramite un'astrazione si interfaccia con il data source. Se questo data source è uno Shared Preferences, perché siamo in Android, o un altro modulo simile che si trova su iOS, allo use case non interessa minimamente: i dati gli arriveranno comunque e saranno nel formato da lui richiesto.

I dati che attraversano i vari livelli, le delimitazioni di cui parla Robert C. Martin, come fanno a essere sempre nel formato corretto per quel determinato livello?
Con questa tipologia di architettura si creano sempre due diversi modelli:

- le Entity rappresentano la struttura dati che verrà utilizzata in tutta l'applicazione: i repository, gli use case, i view model e le view useranno questa tipologia di dato. Le entities sono immutabili, vengono modificate solamente se cambiano i requisiti o il comportamento dell'applicazione;
- le Response, invece, sono le strutture dati che arrivano direttamente dai servizi esterni. Sono dati "grezzi" il cui comportamento non è controllabile dall'applicazione, per questo motivo non sono utilizzati direttamente dal resto del app. Sono le risposte ottenute dai metodi dei data source.

Nel modulo Framework è presente una classe Mapper in grado di effettuare la conversione da Entity a Response e viceversa. Così facendo, nel momento in cui il server decide di cambiare un parametro di una Response, basta modificare solo il codice della classe che rappresenta l'oggetto remoto e la funzione che lo mappa nell'entità corrispondente. Dato che nel resto dell'applicazione si fa riferimento solo alla entity, non ci sarà bisogno di effettuare nessun cambiamento.

4.3 Dependency Injection

I diversi layer sono collegati tramite Dependency Injection, con l'utilizzo della libreria Dagger2[1]. Il package di Injection è all'interno del modulo app.
La Dependency Injection serve per far funzionare il livello di astrazione precedentemente discusso: classe interna che utilizza metodi di classe di livello più esterno tramite interfaccia. Quando una classe chiede un'interfaccia, le viene fornita la giusta implementazione di quell' interfaccia. Ma questo passaggio non viene eseguito in automatico e magicamente, c'è bisogno di qualcuno che fornisca l'implementazione di cui ha bisogno: questo è il compito della dependency injection.
Per esempio, uno use case si aspetta un repository ma non sa chi lo crea, come lo crea e cosa c'è dentro. Esso si aspetta solamente determinati requisiti ed implementa l'interfaccia di interesse e poi Dagger, che va a chiamare la corretta implementazione.

Il package Injection contiene diverse classi che rappresento dei DaggerModule, ognuno dei quali si occupa di fornire un determinato tipo di oggetti.
Un esempio di modulo Dagger è il seguente:

```
@Module
open class DataModule {

    @Provides
    @Singleton
     fun provideSampleRepository(r: EpisodeRepositoryImpl): Episode-
Repository = r

    @Provides
    @Singleton
      fun  provideRemoteDataSource(ds:  EpisodeRemoteDataSourceImpl):
EpisodeRemoteDataSource = ds

}
```

Esempio Dependency Injection

Dagger legge l'annotazione `@Module` in fase di compilazione per fare i suoi ragionamenti. `ProvideSampleRepository` significa che si sta provvedendo a fornire un'istanza di quel repository: quando lo use case chiede un `EpisodeRepository` verrà ritornato `r`, cioè l'implementazione `EpisodeRepositoryImpl`.

1 https://google.github.io/dagger/

L'istanza che viene fornita è *singleton*, cioè garantisce che soltanto un'unica istanza della classe stessa possa essere creata all'interno di un programma. Se nell'applicazione dieci classi diverse chiedono l'implementazione di `EpisodeRepository`, verrà fornita sempre la stessa istanza e non dieci istanze diverse di questa classe.
Con l'utilizzo di Dagger si riesce ad avere un unico punto centralizzato dove esplicitare tutte le injection necessarie.

4.4 RxJava e pattern Observable

Trasversalmente a tutti i layer, si utilizza *RxJava*[2] per rendere reattiva tutta la comunicazione tra i vari layer.
RxJava (*ReactiveX – Java*) è un framework di sviluppo di programmazione reattiva, ciò significa che i dati che viaggiano tra i vari layer sono trattati come un flusso di dati.
I vari componenti osservano questo flusso e reagiscono di conseguenza.
Per comprendere ciò che avviene, si utilizza la *Sample App* messa a disposizione da Synesthesia. Questa semplice applicazione ha lo scopo di mostrare l'elenco delle serie tv nel giorno corrente. Per farlo si appoggia a un API che restituisce i dati da internet.

In questa applicazione i dati si ottengono da qui:

```
interface TvMazeApi {

    // http://api.tvmaze.com/schedule?country=US&date=2019-09-12
    @GET("schedule")
    fun getSchedule(
            @Query("country") country: String,
            @Query("date") date: String
    ): Observable<List<EpisodeRemote>>

}
```

Interface API

È un'interfaccia di Retrofit[3], una delle librerie più usate per fare chiamate di rete su Android. Questa chiamata restituisce un *Observable*: uno stream di dati osservabile che chiunque può osservare.

2 https://github.com/ReactiveX/RxJava
3 https://square.github.io/retrofit/

A partire da qui, si ha un oggetto Observable che si porta dietro una lista di episodi.

Questo metodo viene chiamato dal `EpisodeRemoteDataSource`, ovvero l'implementazione del data source (la fonte di dati).

```
@Singleton
class EpisodeRemoteDataSourceImpl @Inject constructor(
        private val api: TvMazeApi,
        private val episodeMapper: EpisodeEntityMapper) : EpisodeRe-
moteDataSource {

    override fun getTodaySchedule(): Observable<List<EpisodeEntity>>
{
        return Observable
                .fromCallable {
                     SimpleDateFormat("yyyy-MM-dd", Locale.getDefau-
lt()).format(Date())
                }
                .concatMap { todayDate →
                    api.getSchedule("US", todayDate)
                            .map { episodeMapper.fromRemote(it) }
                }
    }
}
```

EpisodeRemoteDataSourceImpl

Il risultato della chiamata `api.getSchedule()` è a sua volta un Observable.
Perciò ogni metodo, a partire dal view model fino alla chiamata di rete, deve ritornare sempre un Observable, perché lo stream di dati non si può interrompere.

Chi richiede la lista di episodi è il `MainViewModel` nel metodo `loadData()`.
Il `MainViewModel` avrà la necessità di osservare questi dati e modificarli a suo piacimento (in questo caso verranno inseriti in una Recycler View). Per ascoltare questi dati si serve di un *Disposable*, un oggetto di RxJava.

```
fun loadData() {
        addEitherDisposable(refreshRelay
                .switchMap { forceRefresh →
                      getEpisodesUseCase.execute(GetEpisodesUseCase.
Params(forceRefresh))
                             .mapEither { episodes →
                                 this.episodes = episodes
                                 Model(episodes)
                             }
                },
                modelResource
        )
    }
```

MainViewModel

Un Disposable è l'insieme tra variabile osservabile e colui che la osserva (*Observable* e *Observer*).
Dire che un Observer sta osservando un Observable significa effettuare un'operazione di *Subscribe*: l'osservatore si sottoscrive alla variabile osservabile. Il risultato della Subscribe si chiama Disposable, ovvero un oggetto che può essere disposto, dismesso e quindi interrotto.

Ovviamente, il `MainViewModel` non comunica direttamente con il data source ma si serve di uno use case. In questo caso di `GetEpisodeUseCase`, che a sua volta restituisce un observable di una lista di `EpisodeEntity`.

```
open class GetEpisodesUseCase @Inject constructor(
        schedulerProvider: SchedulerProvider,
       private val episodeRepository: EpisodeRepository) : UseCase←
GetEpisodesUseCase.Params,   Either<List<EpisodeEntity>>>(scheduler-
Provider) {
    data class Params(val forceRefresh: Boolean)
   override fun buildObservable(params: Params) = episodeRepository.
getTodaySchedule(params.forceRefresh)
}
```

GetEpisodeUseCase

La lista arriva dal Repository. Come spiegato precedentemente, lo use case comunica so lamente con l'interfaccia e tramite Dagger viene fornita la giusta implementazione del repository.

```
interface EpisodeRepository {

   fun getTodaySchedule(forceRefresh: Boolean): Observable<Either<Li-
st<EpisodeEntity>>>

     fun getEpisode(forceRefresh: Boolean, episodeId: Int): Obser-
vable<Either<EpisodeEntity>>

}
```

EpisodeRepository Interface

Come si può notare, il ritorno del EpisodeRepository è sempre un Observable.

```
class EpisodeRepositoryImpl @Inject constructor(
       private val remoteDataSource: EpisodeRemoteDataSource) : Epi-
sodeRepository {

     override fun getTodaySchedule(forceRefresh: Boolean): Obser-
vable<Either<List<EpisodeEntity>>> {
          ...
        remoteDataSource.getTodaySchedule().subscribeEitherRelay(fe-
edRelay)
        }
        return ...
}

    override fun getEpisode(forceRefresh: Boolean, episodeId: Int):
Observable<Either<EpisodeEntity>> {
         ...
    }
}
```

EpisodeRepositoryImpl

Per ottenere i dati, il repository utilizza il data source necessario: EpisodeRemoteDataSource, il quale tramite il metodo getTodaySchedule effettua la chiamata vera e propria all'API. Anche in questo caso si ha l'interfaccia e l'implementazione:

```
interface EpisodeRemoteDataSource {

    fun getTodaySchedule(): Observable<List<EpisodeEntity>>

}
```

EpisodeRemoteDataSource Interface

```
@Singleton
class EpisodeRemoteDataSourceImpl @Inject constructor(
        private val api: TvMazeApi,
        private val episodeMapper: EpisodeEntityMapper) : EpisodeRe-
moteDataSource {

    override fun getTodaySchedule(): Observable<List<EpisodeEntity>>
{
        return Observable
                .fromCallable {
                     SimpleDateFormat("yyyy-MM-dd", Locale.getDefau-
lt()).format(Date())
                }
                .concatMap { todayDate →
                    api.getSchedule("US", todayDate)
                            .map { episodeMapper.fromRemote(it) }
                }
    }
}
```

EpisodeRepositoryImpl

Per concludere il flusso, l'API restituisce un Observable che è una lista di EpisodeRemote e non di EpisodeEntity. Come spiegato precedentemente, è il data source che si occupa di chiamare l'EpisodeEntityMapper per mappare la risposta nell'entità corretta che verrà utilizzata nel resto dell'applicazione.

```
class EpisodeEntityMapper @Inject constructor() : EntityMapper<Episo-
deRemote, EpisodeEntity>() {
override fun fromRemote(remote: EpisodeRemote): EpisodeEntity = wi-
th(remote) {
        return EpisodeEntity(
                airdate = airdate.orEmpty(),
                airstamp = airstamp.orEmpty(),
                airtime = airtime.orEmpty(),
                id = id ?: 0,
                image = ImageEntity( ... ),
                name = name.orEmpty(),
                number = number ?: 0,
                runtime = runtime ?: 0,
                season = season ?: 0,
                summary = summary.orEmpty(),
                url = url.orEmpty(),
                show = ShowEntity( ... ),
        )
    }
}
```

EpisodeEntityMapper

Capitolo 5

Conversione dell'Architettura della COWOApp

Per il corso di *Digital Interaction Design* di Ingegneria del Cinema e dei Mezzi di Comunicazione, è stato chiesto di sviluppare una potenziale applicazione mobile su Android con Kotlin destinata alla gestione di un generico spazio di Coworking.
Il capitolo corrente verterà sulla trasformazione dell'architettura di questa app, chiamata *COWOApp*, seguendo le linee guida delineate da Synesthesia per la progettazione di applicazioni Clean.

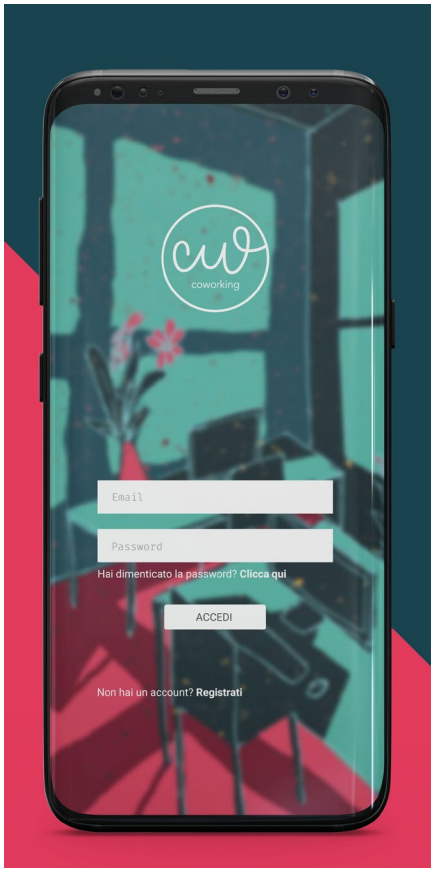
5.1 Funzionalità CowoApp

La COWOApp espone le seguenti funzionalità:

- logIn e Registrazione;
- bacheca con elenco di eventi e news dello spazio di coworking;
- elenco dei coworkers presenti;
- profilo modificabile con i propri dati;
- prenotazione di una postazione:
    - l'utente può scegliere fra tre diverse tipologie (*SMALL, MEDIUM* o *LARGE*);
    - l'utente può scegliere la durata dell'abbonamento (*GIORNALIERO, SETTIMANALE, MENSILE* o *ANNUALE*);
    - l'utente seleziona il periodo di inizio dell'abbonamento;
    - in base alle disponibilità, l'applicazione crea l'abbonamento richiesto.

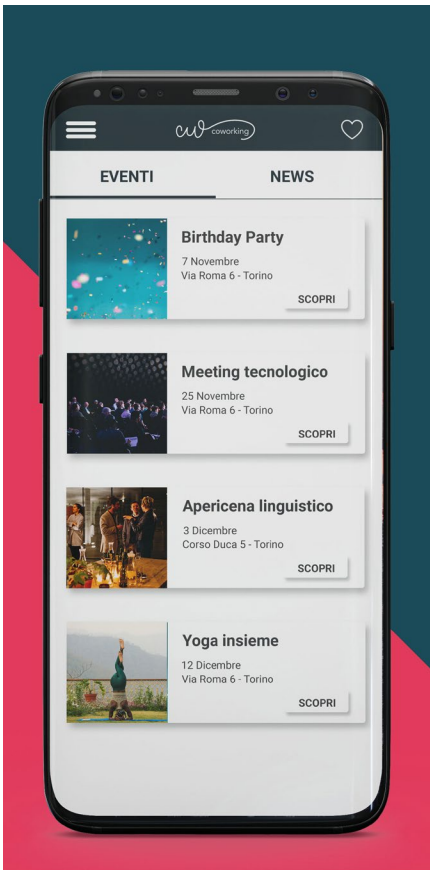
Inoltre, l'applicazione scrive e legge i dati da un database non relazionale: *MongoDB*[1].

1 https://github.com/mongodb/mongo

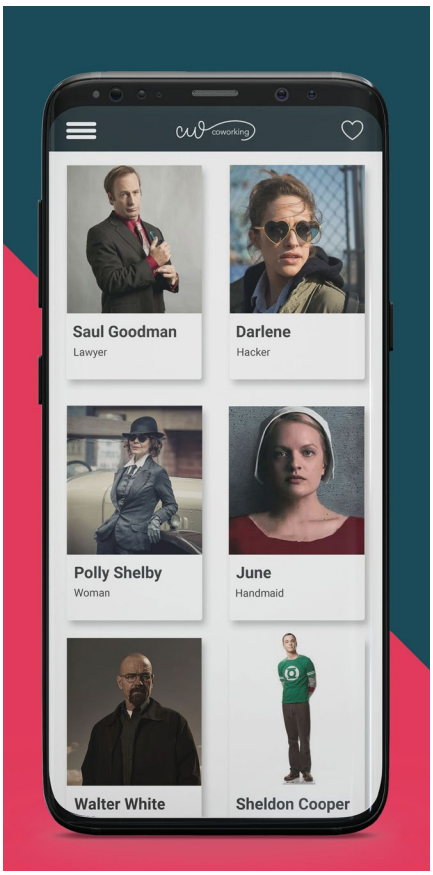


5.1 - LogIn

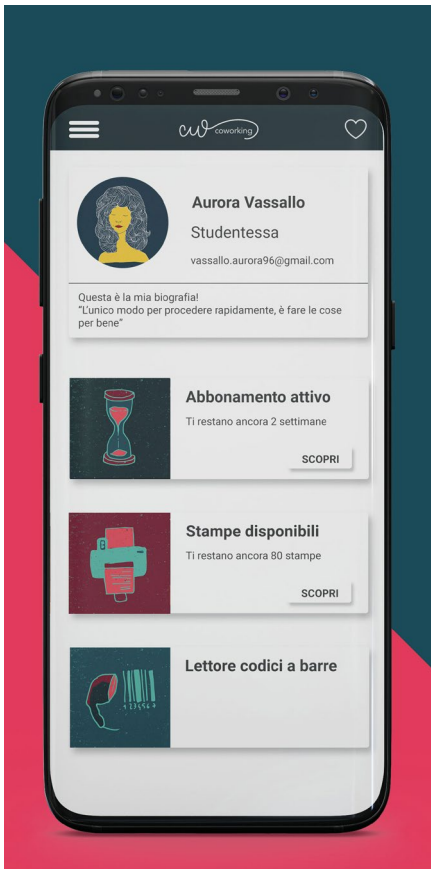
5.2 - Home



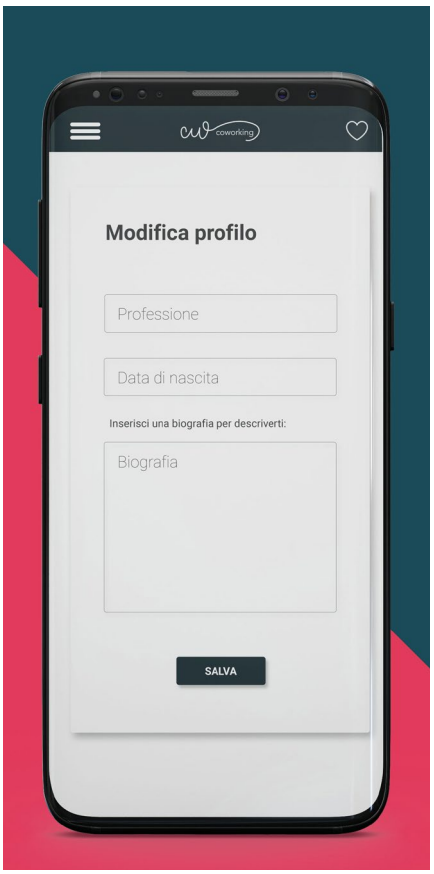
5.3 - Bacheca



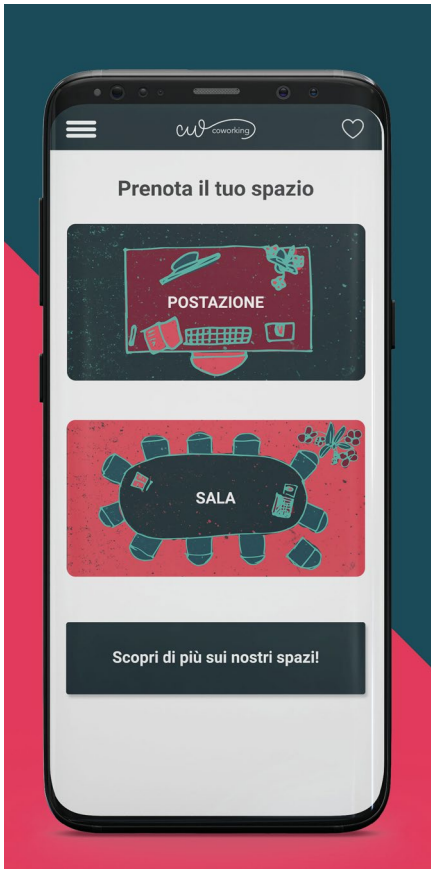
5.4 - Coworkers



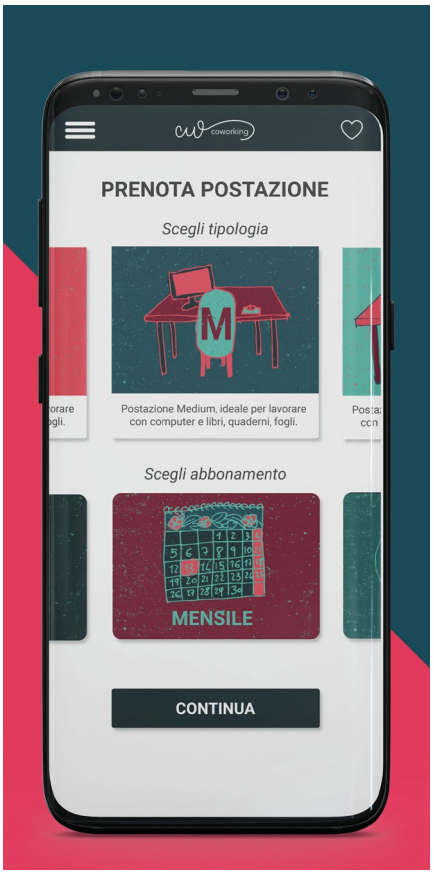
5.5 - Profilo



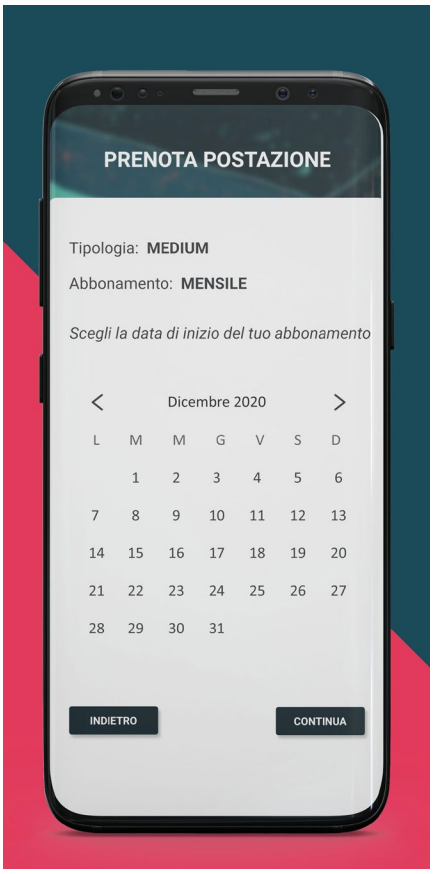
5.6 - Modifica Profilo



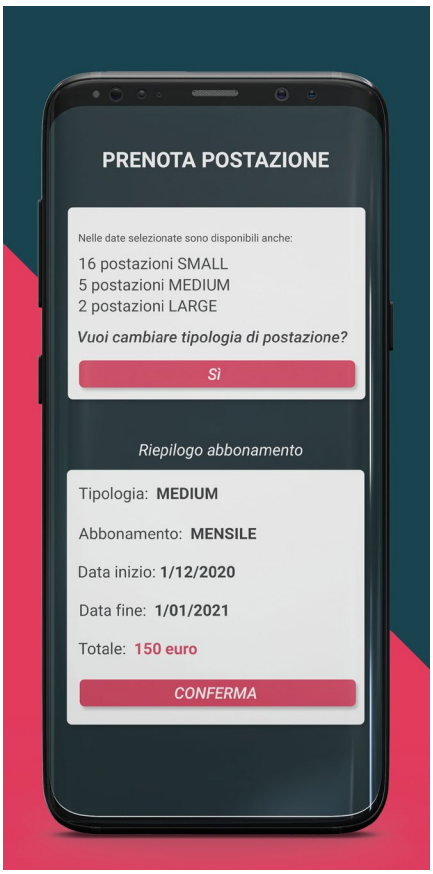
5.7 - Prenotazione Step1



5.8 - Prenotazione Step2



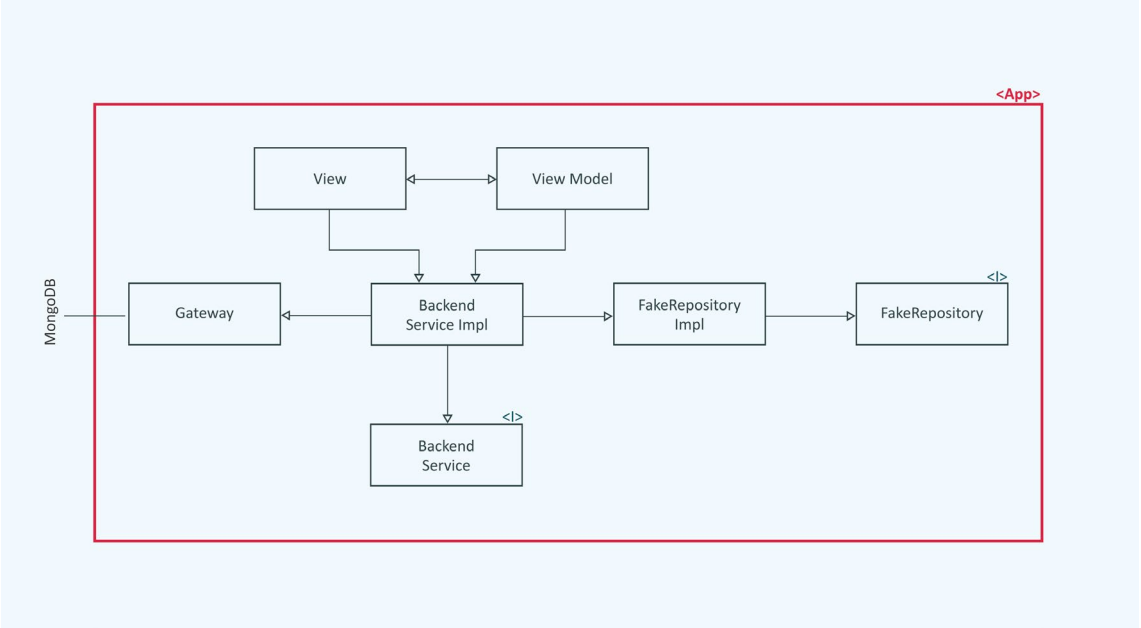
5.9 - Prenotazione Step3



5.10 - Prenotazione Step4

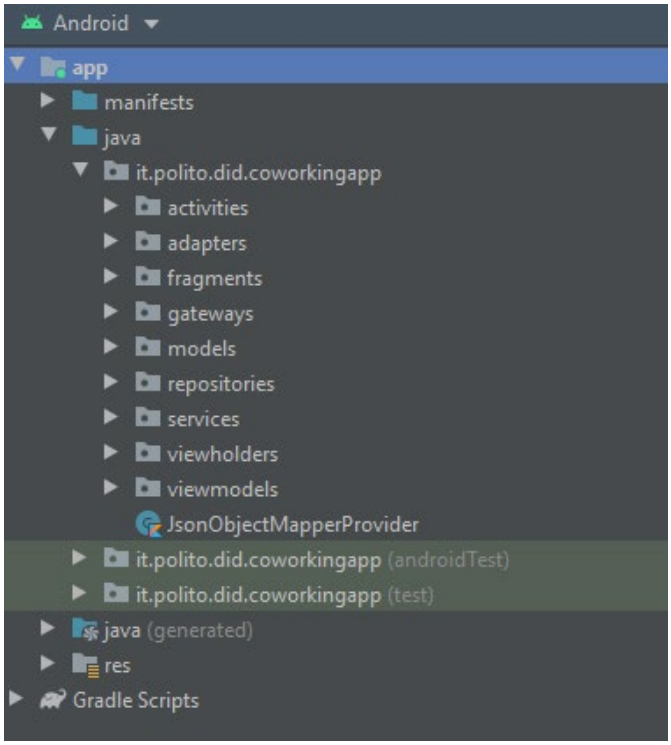
5.2 Architettura originale COWOApp

L'architettura di questa applicazione si può sintetizzare visivamente tramite il seguente schema:



5.11 - Architettura originale CowoApp

Quello che succede in Android Studio, invece, è visualizzabile nella foto che segue:



5.12 - Suddivisione delle classi nei package in Android Studio

Come si può osservare, tutto il codice è contenuto dentro un unico modulo: il modulo App. Questo, come evidenziato dal cerchio verde sul package, è un modulo completamente dipendente dal framework Android.

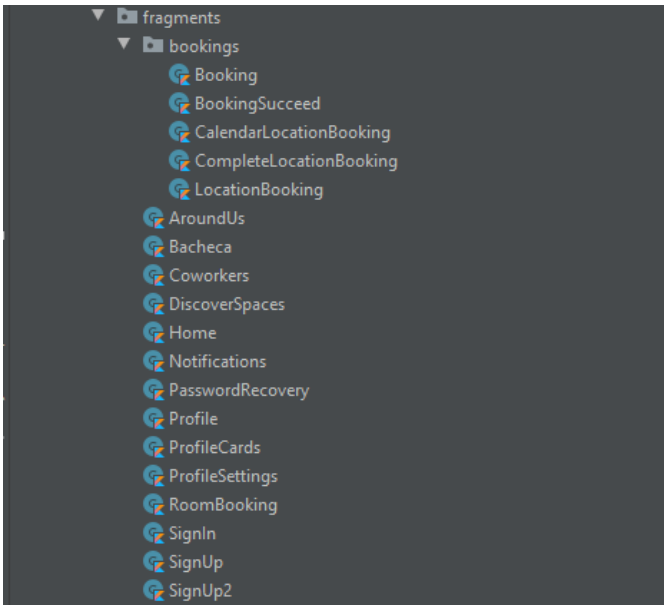
Activities, adapters, fragments e viewholders

I package activities, adapters e fragment contengono tutto ciò che riguarda le view. Nelle classi di questi package, in particolare nei Fragment, è contenuta la maggior parte della logica dell'applicazione.

Le activity sono tre: `BookingActivity`, `LogInActivity` e `MainActivity`. Queste classi si occupano di ospitare le varie schermate dei fragments e gestiscono la navigazione che l'utente può effettuare tramite menù.

Gli Adapter e i View Holder sono classi di Android che servono per gestire le Recycler View, ovvero le liste destinate a visualizzare collezioni di dati. In questo caso servono per gestire l'elenco delle news, degli eventi e dei coworkers.

I fragment rappresentano la totalità delle schermate che vengono mostrate all'utente.



5.13 - Package fragments

L'unico raggruppamento effettuato è "bookings". Questo package contiene i fragment che riguardano le view di prenotazione.

Tutti i fragment, esclusi `Bacheca` e `Coworkers`, contengono le regole di business e sono loro che chiamano direttamente il `BackendService`, cioè la classe che si interfaccia con il gateway.

Viewmodels

I view model sono tre: `BachecaViewModel`, `BookingViewModel` e `CoworkersViewModel`.

I view model della bacheca e dei coworkers contengono la lista degli eventi, delle news

e dei coworkers che ottengono chiamando il Service. Quindi in questo caso è proprio il viewmodel che si interfaccia con le classi che gestiscono i dati provenienti dall'esterno. Mentre il `BookingViewModel` è stato usato come "scatola" contenitiva dei dati che devono essere passati tra i vari fragment durante la prenotazione.
Questi dati, sia nel caso delle liste che nel caso della prenotazione, sono LiveData che vengono osservati dai rispettivi fragment.

Di seguito viene illustrato ciò che succede nelle due diverse situazioni:

- il view model che si occupa dei coworkers chiama direttamente il service, la view osserva la lista di LiveData;
- il view model contiene i dati della prenotazione, uno dei fragment (`CompleteLocationBooking`) recupera questi LiveData ed effettua la chiamata al service.

```
class CoworkersViewModel: ViewModel() {

      private val coworkersList: MutableLiveData<List<Coworker>> by
lazy {
         val liveData = MutableLiveData<List<Coworker>>()
         DefaultCowoBackendService.getCoworkers().let {
             liveData.value = it
         }
         return@lazy liveData
     }

     fun getCoworkersList() : LiveData<List<Coworker>> {
         return coworkersList
     }
}
```

CoworkersViewModel

```
class Coworkers : Fragment() {
     ...
     override fun onViewCreated(view: View, savedInstanceState: Bun-
dle?) {
         super.onViewCreated(view, savedInstanceState)
```

Coworkers Fragment

```
val model = ViewModelProviders.of(this).get(CoworkersViewMo-
del::class.java)
        model.getCoworkersList().observe(viewLifecycleOwner, Obser-
ver<List<Coworker>> { renderCoworkers(it) })

    }
}
```

Coworkers Fragment

```
class BookingViewModel : ViewModel() {
    val location = MutableLiveData<Location>()
    val subs = MutableLiveData<Subs>()
    val start = MutableLiveData<LocalDate>()
    val end = MutableLiveData<LocalDate>()
    val bookingsForDatesList = MutableLiveData<Map<Location, Int>>()

    fun setLocation(l: Location){
        location.value = l
    }

    fun setSubs(a: Subs){
        subs.value = a
    }

    fun setDate(s: LocalDate, e: LocalDate){
        start.value = s
        end.value = e
    }

    fun setBookingForDatesList(bookingsForDates: List<BookingsForDa-
tes>) {
      bookingsForDatesList.value = bookingsForDates.map { it.location
to it.available }.toMap()
    }
}
```

BookingViewModel

```
class CompleteLocationBooking : Fragment() {
    ...
    override fun onViewCreated(view: View, savedInstanceState: Bund-
le?) {
        super.onViewCreated(view, savedInstanceState)
           ...
    val viewModel = ViewModelProviders.of(activity!!).get(Booking-
ViewModel::class.java)
    viewModel.location.observe(viewLifecycleOwner,
            Observer<Location> { t →
                type.text = t.toString()
                locationInfo = t
            })
    ...
     confirmButton.setOnClickListener {
             ...
               val bookingInfo = BookingInfo(locationInfo, subsInfo,
parseDateStart, parseDateEnd)
            try{
                DefaultCowoBackendService.createBooking(bookingInfo)
               findNavController().navigate(R.id.action_completeLoca-
tionBooking_to_bookingSucceed)
            } catch(er: Exception){ println(er) }
        }
    }
```

CompleteLocationBooking Fragment

Le righe di codice scritte in ciascuna view sono molte e contengono tutta la logica dell'applicazione. Nei fragment e nei view model viene stabilito come, quando e perché una regola deve essere eseguita: verificare quali sono gli eventi, le news e i coworkers, effettuare l'accesso e la registrazione, ottenere il numero di prenotazioni per un periodo di tempo, creare una prenotazione.
Tutto ciò risulta confusionario, poiché chi si approccerà all'app per la prima volta non riuscirà a capire quali siano le funzionalità effettive di questa applicazione e come bisogna muoversi dal momento che una volta il service è chiamato direttamente dai fragment, mentre l'altra dai view model.

Services e Gateway

Il package services contiene l'interfaccia `CowoBackendService` e la sua implementazione, che è un *singleton*. Per cui, quando viene richiesto il service, in qualsiasi punto dell'applicazione, viene fornita sempre la stessa istanza.
I metodi all'interno del service rappresentano tutti i casi d'uso dell'applicazione, come è possibile osservare nel codice che segue:

```
interface CowoBackendService {
   fun getEvents(): List<Event>

   fun getNews(): List<News>

   fun getCoworkers(): List<Coworker>

   fun createUser(user: User): String

   fun signIn(user: UserSignIn): String

   fun editUSer(edits: UserEdits)

   fun getUserInfo(): User

   fun createBooking(bookingInfo: BookingInfo)

    fun getBookingForDates(bookingInfo: BookingInfo): List<Booking-
sForDates>

}
```

CowoBackendService

Questo service rappresenta la fonte di dati che si interfaccia con il gateway. Ottiene le risposte e le passa alle view o ai view model che gli hanno fatto la richiesta.
È da precisare che le varie schermate importano direttamente l'implementazione del service e non l'interfaccia. Quindi sono a conoscenza di tutto quello che avviene all'interno del service. L'interfaccia, di conseguenza, diventa inutile a fini pratici.
Il gateway, invece, è la classe che lascia l'applicazione e va a prendere i dati nel database.

Per questo progetto si è utilizzato il client *HTTP Jersey*[2] e la libreria *Jackson*[3], la quale permette di serializzare le risposte (oggetti Json) in oggetti Java.
Il codice sottostante mostra come viene configurato il client e come viene effettuata una chiamata.

```
private val client = JerseyClientBuilder.newBuilder()
        .register(JsonObjectMapperProvider::class.java)
        .register(JacksonFeature::class.java)
           .build()

    fun getEvents(): CompletableFuture<EventsResponse> {
        return CompletableFuture.supplyAsync {
            client.target(BASE_URL + EVENTS_PATH)
                .request()
                .get(EventsResponse::class.java)
        }
    }
```

Client Builder nella classe Gateway

Le risposte che si ottengono dal backend sono configurate come data class all'interno del gateway:

```
data class EventsResponse(val events: List<Event>)
data class NewsResponse(val news: List<News>)
data class CoworkersResponse(val coworkers: List<Coworker>)
data class SignUpResponse(val accessToken: String)
data class SignInResponse(val accessToken: String)
data class UserEditsResponse(val message: String)
data class UserInfoResponse(val user: User)
data class BookingsForDatesResponse(val result: List<BookingsForDa-
tes>)
```

Response ottenute dal backend

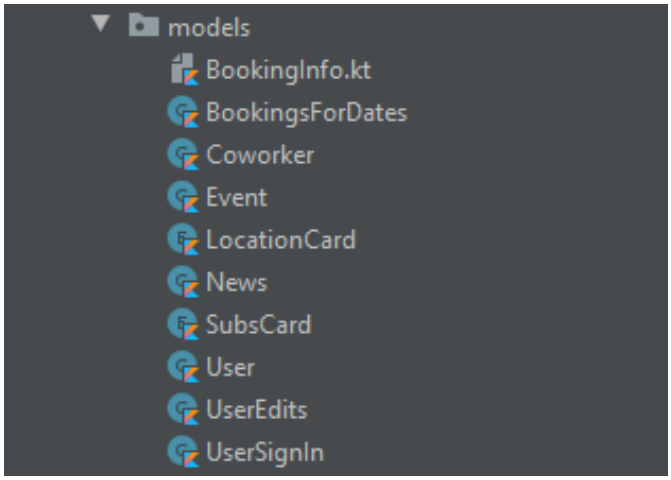
Queste risposte sono quelle che verranno utilizzate direttamente all'interno dell'applicazione.

2 https://eclipse-ee4j.github.io/jersey/
3 https://github.com/FasterXML/jackson

Per esempio, `EventsResponse` fornisce una lista di `Event` che sono un modello creato sulla base della risposta del backend. Proprio questo modello verrà utilizzato per creare la recycler view.

Models

Si tratta del package che contiene tutte le strutture dati che si trovano nell'applicazione. Oltre ad esserci alcune classi conformi alle risposte, costruite cioè sulla base degli oggetti che arrivano dal server, ce ne sono altre scritte apposta per effettuare una `POST` con il dato formattato secondo i requisiti del backend.



5.14 - classi appartenenti al package models

`BookingInfo`, `Coworker`, `Event`, `LocationCard`, `News`, `SubsCard` e `User` sono quelle che Robert C. Martin chiama "regole operative critiche", ma sono progettate sulla base delle response.

`BookingsForDates`, `UserEdits` e `UserSignIn` sono data class costruite per inviare la domanda al server nel formato che quest'ultimo desidera

Repositories

Questo package comprende l'interfaccia di un "fake" repository e la sua implementazione. È "fake" poiché viene utilizzato solamente per salvare le informazioni relative al token all'interno dell'applicazione.

Questo singleton viene chiamato dal service quando deve controllare se l'utente dispone dei permessi per effettuare una chiamata al backend.
Anche in questo caso, il service adopera direttamente l'implementazione del repository e non la sua interfaccia.

```
object FakeUserRepository : UserRepository {

    private var token: String? = null

    override fun setToken(token: String) {
        this.token = token
    }

    override fun getToken(): String? {
        return token
    }

}
```

FakeUserRepository

5.3 Conversione

L'applicazione, così com'è stata strutturata, funziona e risponde ai requisiti richiesti. Essendo sviluppata per un corso universitario, non c'è stata la necessità di organizzarla seguendo un'architettura particolare, ma l'esigenza è stata quella di farla funzionare e di imparare delle buone pratiche, quali l'utilizzo del pattern repository, dei singleton e delle interfacce. Nel momento in cui quest'app dovesse venire modificata da uno sviluppatore esterno, che non ha indicazioni riguardo la struttura, il work flow, i compiti di ciascuna classe, la questione può divenire complicata.
I problemi principali sono i seguenti:

- i fragment contengono tutta la logica dell'applicazione. In ognuna di queste view sono scritte moltissime righe di codice e diventa complesso trovare il metodo che esplicita il caso d'uso. In generale, è difficile capire qual è il compito di quel fragment;
- il flusso cambia a seconda della situazione. Nel caso della `Bacheca` e del `Coworkers`, sono i view model che inoltrano la chiamata al service, mentre nel resto dei fragment sono direttamente quest'ultimi a chiamarlo;
- i modelli che dovrebbero rappresentare le logiche operative critiche, sono molti di più di quelli che ci si aspetta. Per cui, classi come `UserEdits` e `UserSignIn` creano confusione.

I difetti elencati esistono a prescindere dal fatto che si stia seguendo un'architettura *clean* o meno.

Ora si procederà prendendo in considerazione tutti i requisiti per convertire l'architettura di questa app in una più facile da sviluppare, comprendere e manutenere.
La prima considerazione da fare è che tutte le classi sono contenute nell'unico componente `App`. In questa forma, l'app è interamente dipendente dal framework Android: dalle logiche di business al gateway.
In primis occorre slegarsi dal codice e dalle classi per andare a definire quali sono le regole di alto livello e quali sono, invece, i dettagli della COWOApp.

Le regole operative critiche, le Entity, sono:
- User;
- News;
- Event;
- Coworker;
- BookingInfo;
- Location (il tipo di postazione: small, medium o large);
- Subscribe (il tipo di abbbonamento: giornaliero, settimanale, mensile o annuale)

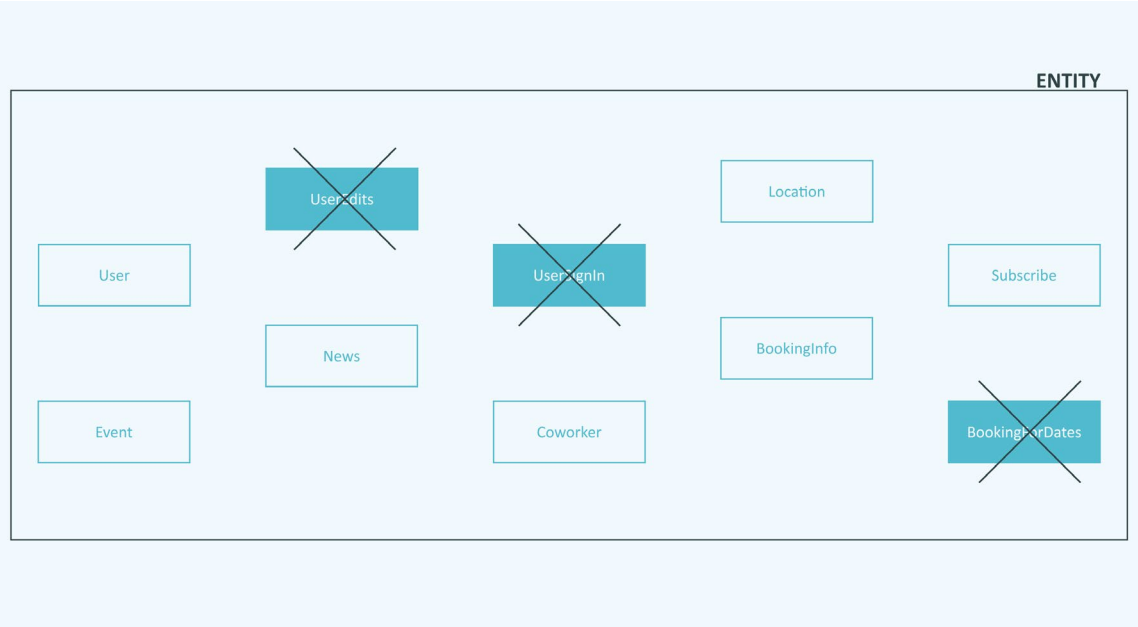
Questi sono gli attori che esistono a prescindere dell'esistenza dell'applicazione.

Le regole operative specifiche dell'applicazione, gli Use Case, sono:
- effetturare la registrazione all'app;
- effettuare il login;
- modificare i propri dati;
- ottenere una lista di eventi;
- ottenere una lista di news;
- ottenere una lista di coworkers;
- ottenere il numero di prenotazioni per uno specifico periodo di tempo;
- creare la prenotazione.

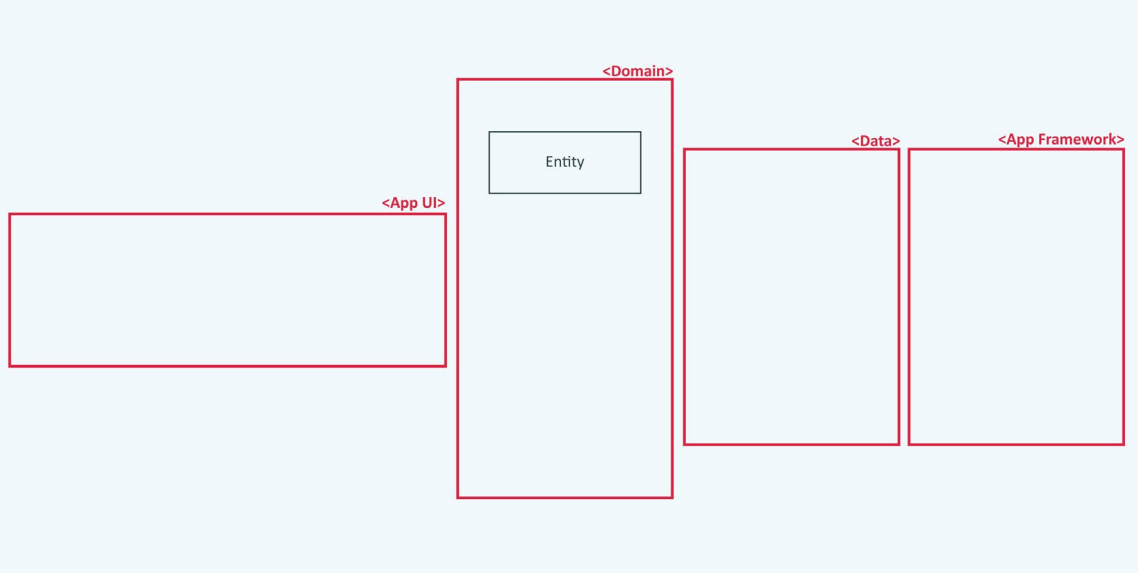
I dettagli di basso livello sono tutto ciò che riguarda la UI e il tipo di database utilizzato.

Ora bisognerà andare a capire quali classi già scritte si possono andare a riutilizzare, quali bisogna andare a creare, quali eliminare e, soprattutto, come inserirle nei rispettivi moduli: `Domain`, `Data` e `App` (Framework e UI).

I `models` dell'app originaria hanno la parvenza di Entity, perciò queste classi si possono spostare nel modulo `Domain`. Dall'elenco bisogna eliminare quelle non necessarie, mentre bisogna riformattare quelle rimaste inserendo gli attributi che rispecchiano le logiche aziendali (del coworking) e non le logiche del database.



5.15 - Entity da mantenere ed eliminare

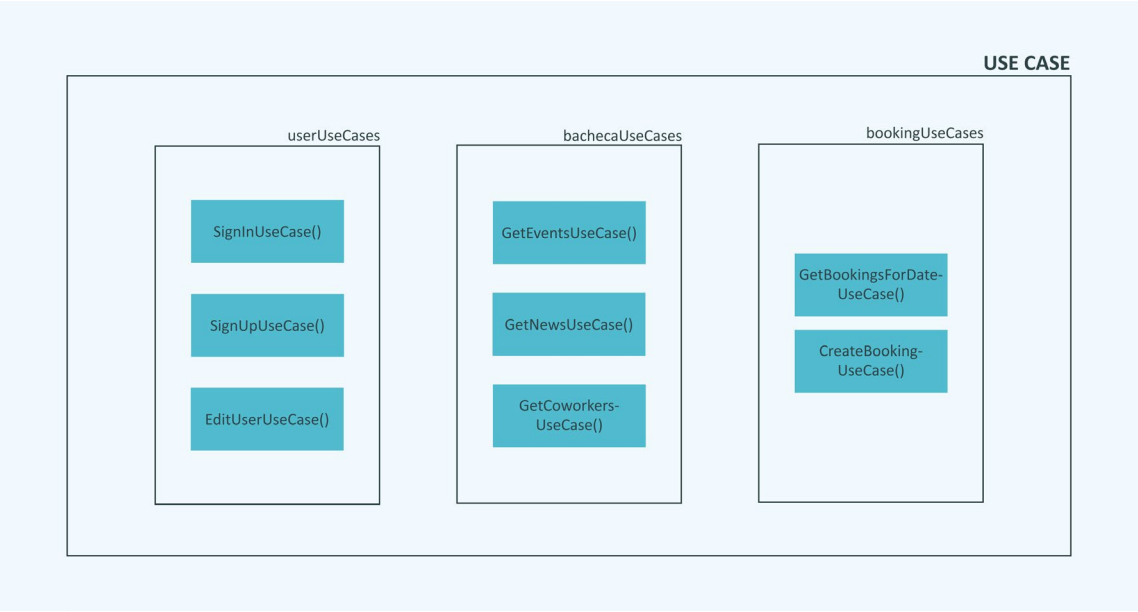


5.16 - Inserimento Entity nel modulo Domain

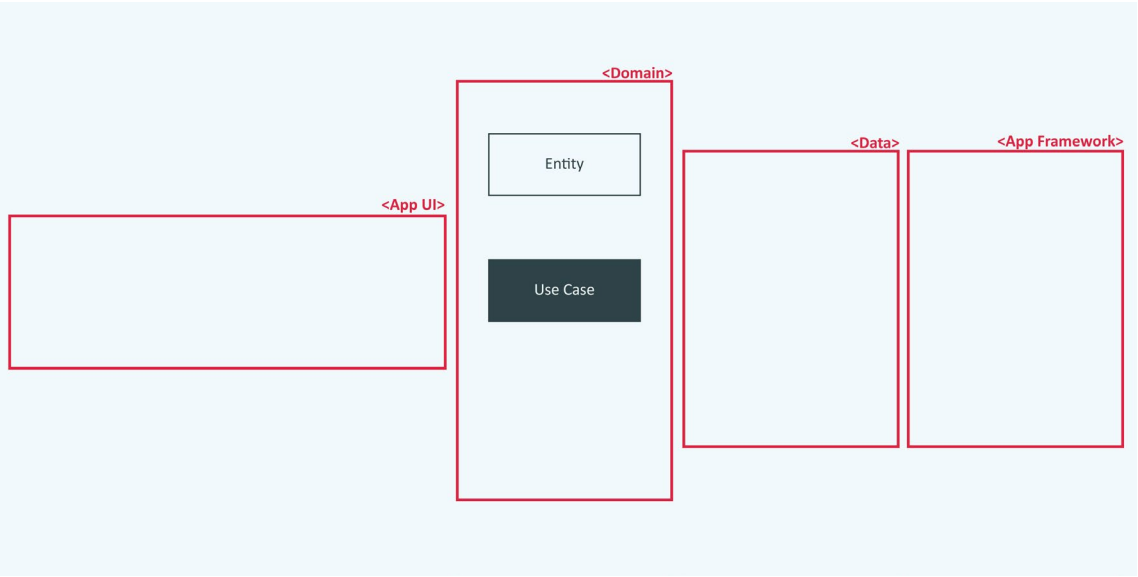
Gli use case vanno scritti da zero e si può pensare di raggrupparli in questo modo:

- `SignInUseCase()`, `SignUpUseCase()` e `editUserUseCase()` ⟶ `userUseCases`: sono le operazioni che riguardano l'utente;
- `GetEventsUseCase()`, `GetNewsUseCase()` e `GetCoworkersUseCase()` ⟶ `bachecaUseCases`: sono i metodi per ottenere liste che verranno presentate all'utente sotto forma di recyclerView. Tutti e tre gli use case funzionano nello stesso modo, ottengono solamente informazioni differenti;
- `GetBookingsForDateUseCase()`, `CreateBookingUseCase()` ⟶ `bookingUseCases`: sono le operazioni che riguardano la parte di prenotazione di una postazione.

Gli use case descritti vanno inseriti nel modulo `Domain`.



5.17 - Use Case da creare

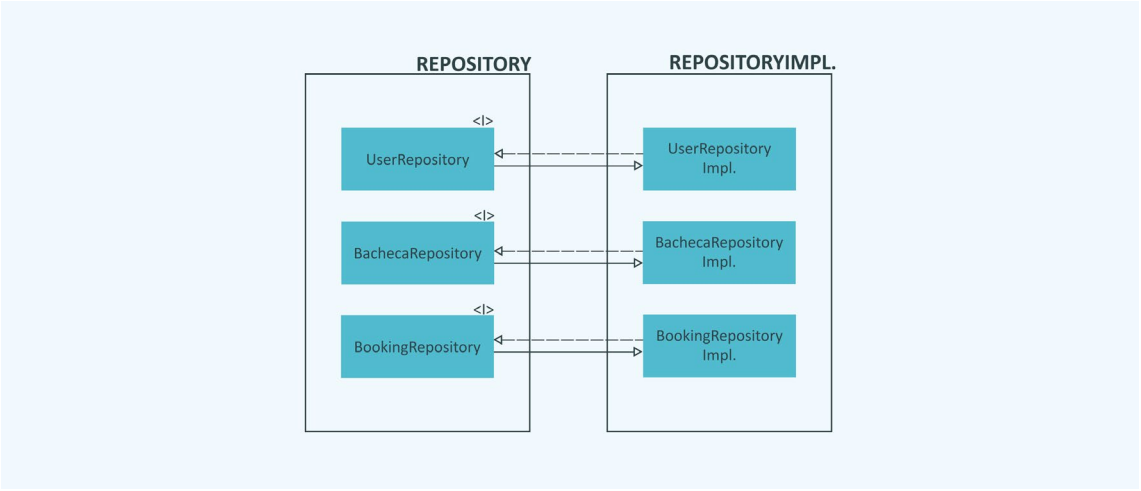


5.18 - Inserimento UseCase nel modulo Domain

Nel modulo `Data` devono essere presenti i repository. Come visto nei capitoli precedenti, i repository hanno il compito di aggiungere un livello di astrazione tra le regole di alto livello e ciò che riguarda i servizi esterni. Inoltre, i repository vanno suddivisi per concetto. Di conseguenza, si può pensare di creare tre diversi repository:

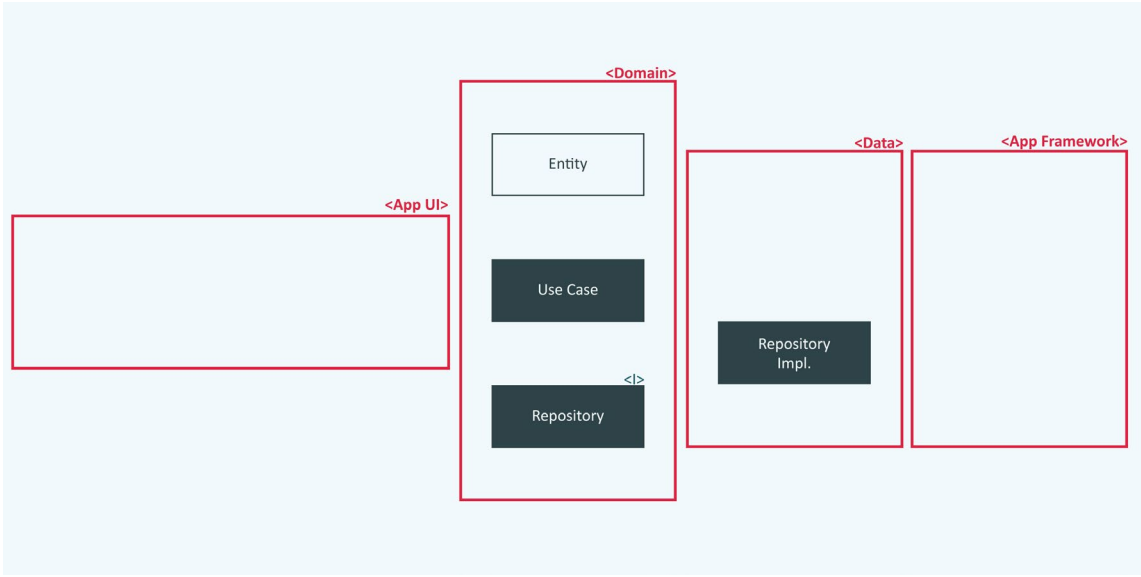
- UserRepository;
- BachecaRepository;
- BookingRepository.

Questi repository hanno bisogno delle loro interfacce, che saranno contenute nel modulo Domain. In questo modo, gli use case chiamano l'interfaccia del repository di cui hanno bisogno.



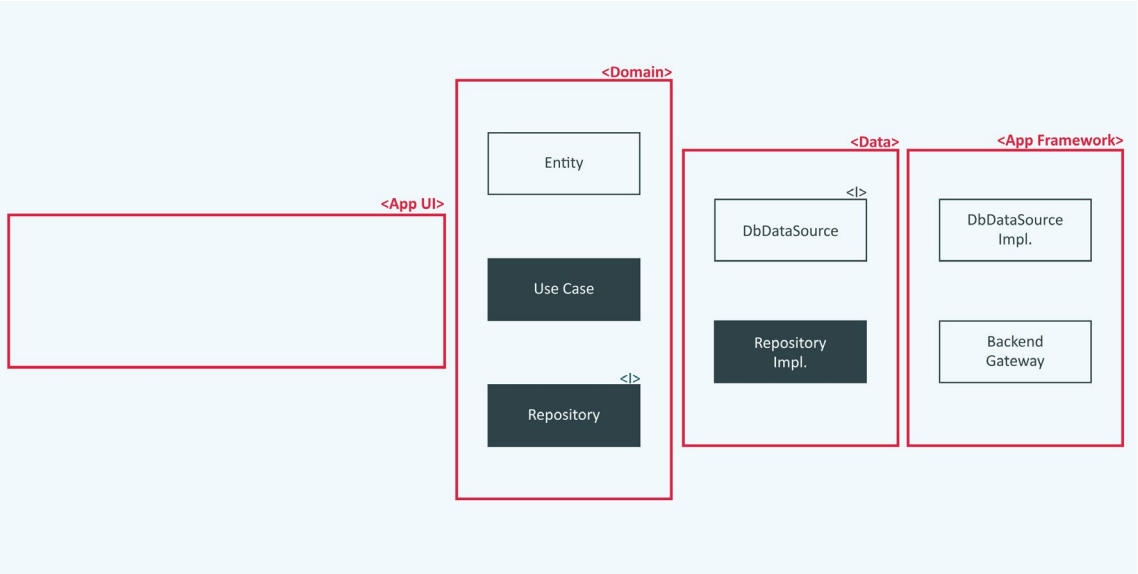
5.19 - Repository Interface e RepositoryImplementation da creare

Così come sono costruiti, cioè seguendo le regole di Synesthesia, questi repository e in particolare lo `UserRepository`, non hanno nulla a che fare con lo `UserRepository` già implementato nella COWOApp.
Quello era un "falso" repository e non chiamava nessun service. In questo caso, invece, i repository devono servirsi dell'interfaccia del service per inoltrare le richieste al database. Pertanto queste classi vanno costruite da zero.



5.20 - Inserimento interfaccia Repository nel modulo Domain e RepositoryImpl. nel modulo Data

Adesso bisognerà inserire le giuste classi nel modulo `App`.
Per quanto riguarda la sezione di Framework, occorre un solo data source poiché si impiega una sola fonte di dati (MongoDB).
È già presente sia la classe che svolge questa funzione, sia la sua interfaccia: `BackendService`. Dunque, l'implementazione verrà spostata nel modulo `App Framework`, mentre la sua interfaccia nel modulo `Data`. Per mantenere coerenza, i nomi di queste classi possono venire sostituiti con `DbDataSource` e `DbDataSourceImpl`.



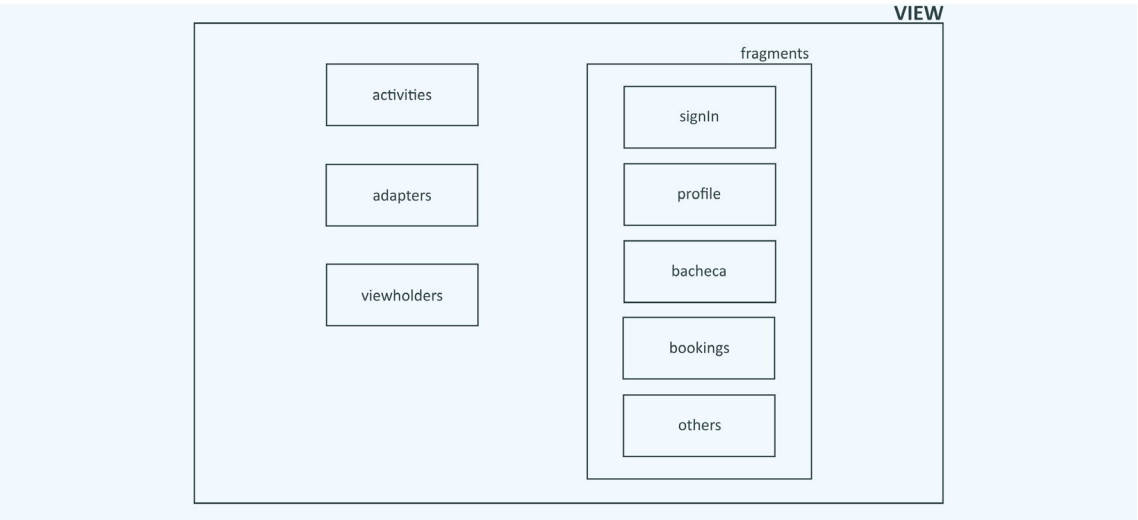
5.21 - Inserimento interfaccia DbDataSource nel modulo Data e DbDataSourceImpl. e Backend Gateway nel modulo App Framework

Il data source si servirà del gateway presente per andare a prendere i dati dal database. Perciò, la classe `BackendGateway` può essere lasciata così com'è e inserita nel modulo `App Framework`.

Nella parte di UI del modulo `App` vanno inseriti le view e i view model.
Nel package view vanno inseriti:

- activities;
- adapters;
- fragments;
- viewholders.

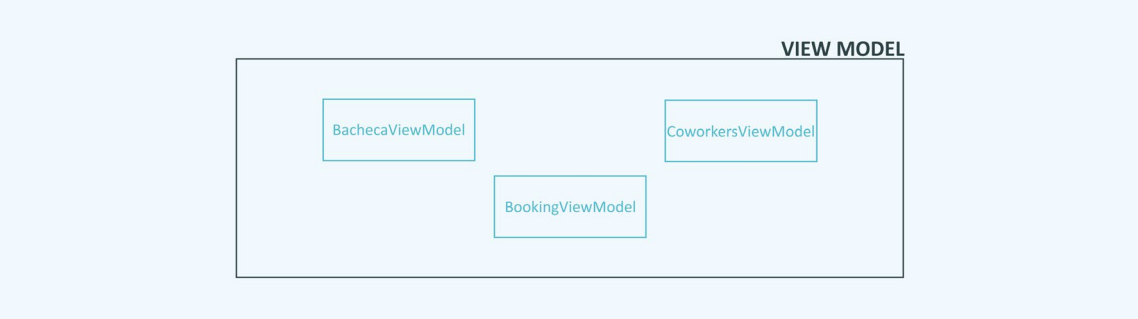
I fragments andrebbero a loro volta suddivisi in package più specifici, quali: `SignIn`, `Profile`, `Bacheca`, `Bookings` e `Others`.



5.22 - Suddivisione in package delle view

Nel package view model continuano a esistere i tre view model già costruiti:

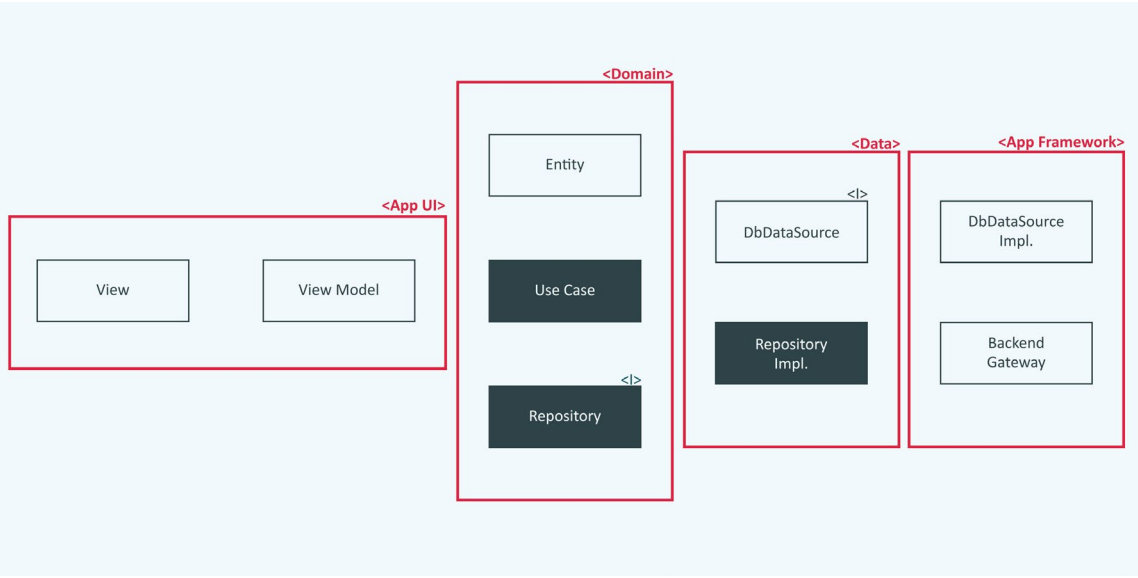
- BachecaViewModel;
- BookingViewModel;
- CoworkersViewModel.



5.23 - View Model da mantenere

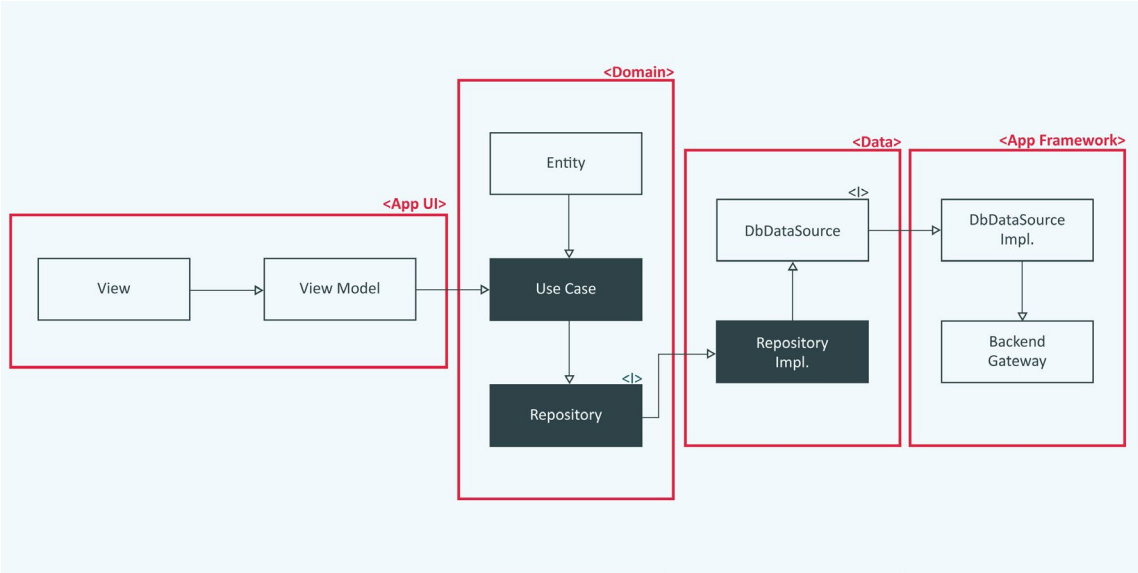
Non è necessario creare nuove classi o eliminarne di vecchie, tuttavia occorre modificare il codice all'interno di esse.
Le view non dovranno più implementare nessuna logica operativa. Il loro compito deve essere solamente quello di mostrare i dati all'utente e inoltrare le richieste al corrispondente view model. Saranno questi ultimi che, servendosi degli use case, otterranno i dati richiesti e li elaboreranno.

A questo punto le diverse tipologie di classi sono inserite nei rispettivi moduli:



5.24 - Inserimento View e View Model nel modulo App UI

Per completare lo schema grafico occorre inserire le frecce rappresentanti le relazioni di dipendenza.



5.25 - Architettura COWOApp convertita

Restano due annotazioni importanti da tenere in considerazione:

- è necessario aggiungere la sezione che riguarda il **mapping delle risposte**. Pertanto, bisogna aggiungere un package contenente le risposte ricevute dal backend e una classe che mappi queste risposte nell'Entity corretta.
  Così facendo, si evita di utilizzare strutture dati poco affidabili e che arrecano confusione nel resto dell'applicazione;
- ovviamente, per funzionare c'è bisogno della libreria Dagger2 per collegare i diversi layer tramite **Dependency Injection**. Quindi, nel componente App bisognerà inserire il modulo injection con le classi richieste.

Capitolo 6

Il Testing e la Clean Architecture

Il software testing è una pratica eseguita per fornire informazioni a tutte le parti interessate, clienti e produttori, sulla la qualità del prodotto rispetto ai requisiti e alle specifiche del sistema tramite un processo di verifica e convalida.
Lo scopo dei test è quello di dimostrare la presenza errori, cosicché gli sviluppatori possano correggerli e produrre un software idoneo all'uso. Gli errori che si vogliono individuare sono di tre categorie:

- *error*: quando è presente una differenza tra l'output desiderato e l'output ottenuto. Inoltre, sono considerati errori quelli di scrittura del codice;
- *fault*: anche conosciuto come bug, è il risultato di un errore che può causare il malfunzionamento del sistema;
- *failure*: l'incapacità del sistema di eseguire il task desiderato. Le failure si verificano quando esiste un fault nel sistema.

I test possono essere eseguiti a più livelli di complessità e le distinzioni tra le varie tipologie possono essere molte ed estremamente dettagliate. Tuttavia, si è soliti dividerli in tre categorie: *Unit Test*, *Integration Test* e *E2E Test*.
A seconda di come il testing viene effettuato, si parla di test automatici o di test manuali. I primi sono eseguiti con l'ausilio di tool specializzati, mentre nel secondo caso è una persona che svolge il test "*step by step*". Nel discorso che segue si terranno in considerazione solamente i test eseguiti automaticamente.

6.1 Unit Test

Questi test sono utilizzati per validare singole unità di software, ovvero il minimo componente dotato di un funzionamento autonomo, come una funzione o una classe.
Creare ed eseguire unit test permette di controllare il comportamento del codice in risposta a casi standard, limite e non corretti.

Un esempio di unit test può essere la verifica del corretto meccanismo di una funzione per validare la password inserita dall'utente. I requisiti voluti sono la presenza sia di lettere che di numeri e la lunghezza minima di otto caratteri. Per analizzare il funzionamento, il test deve fornire diversi input che rispecchiano i casi che si possono presentare: solo numeri, solo lettere, meno di otto caratteri, e così via. Per ogni input ci si aspetta un determinato output: password valida oppure password non valida. Se il test viene passato, la funzione opera nel modo corretto.

Uno unit testing accurato ha importanti vantaggi[1]:

- semplifica le modifiche: facilita la modifica del codice dell'unità in momenti successivi (*refactoring*) con la sicurezza che continuerà a funzionare correttamente;
- semplifica l'integrazione tra moduli diversi: limita i malfunzionamenti a problemi di interazione tra i moduli e non nei moduli stessi;
- supporta la documentazione: fornisce una documentazione "viva" del codice, perché è intrinsecamente un esempio di utilizzo dell'API del modulo.

6.2 Integration Test

Per verificare come le unità di codice (classi e funzioni) interagiscano tra loro è necessario eseguire gli integration test. Essenzialmente, si tratta di testare i singoli moduli come un gruppo al fine di valutare la conformità di un sistema e di esporre i difetti dei moduli al momento dell'interazione con altri moduli o con componenti esterne, come API e database.

Un esempio di integration test può essere la verifica del corretto funzionamento di una chiamata ad un servizio esterno in seguito ad una richiesta dell'utente effettuata cliccando su uno specifico bottone. Altri esempi sono la verifica del corretto processo di registrazione all'app oppure del login.

Gli integration test possono essere più o meno lunghi e complessi e, inoltre, gli approcci di sviluppo sono vari (Bing Bang, Bottom-up, Top-down, Sandwich). Le ragioni per cui vengono effettuati sono, però, altrettanto robuste[2] :

- ogni modulo può essere scritto da un singolo sviluppatore la cui logica di programmazione può essere differente da quella degli sviluppatori degli altri moduli. Questi test diventano essenziali per determinare l'esatta comunicazione tra i moduli;
- i requisiti possono cambiare durante lo sviluppo e non essere testati a livello di unità, per cui l'integration test diventa obbligatorio;
- l'incompatibilità tra i moduli software può creare errori;
- possono verificarsi incompatibilità tra hardware e software.

1 https://it.wikipedia.org/wiki/Unit_testing

2 https://www.javatpoint.com/integration-testing

6.3 End-to-end Test

Il testing E2E assicura che l'intera applicazione funzioni come richiesto dall'inizio alla fine, come suggerisce il nome. Quando si eseguono questi test è importante immaginare la prospettiva di un utente reale, cioè in che modo interagirebbe con l'app andando a testare tutti i flussi che potrebbero verificarsi.

In questa categoria rientrano gli **UI test** per testare l'interfaccia grafica, i quali automatizzano dei task ripetuti per assicurare che le interazioni della UI più critiche continuino a funzionare anche dopo aver aggiunto nuove funzionalità o dopo dei refactoring.

Esempi di test e2e possono essere: aggiungere al carrello un prodotto, eseguire una transizione completa su un sito web, flusso di navigazione.

A causa della complessità di ciò che si vuole testare, a volte non è possibile automatizzare questi test al 100% ma è necessario l'intervento di tester, figure professionali che svolgono i test manualmente.

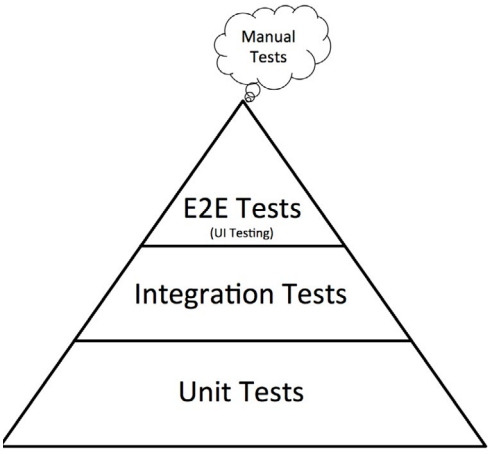
Come le altre tipologie, anche i test e2e producono dei benefici[3]:

- assicurano il corretto funzionamento dell'applicazione;
- espandono la copertura (*code coverage,* la percentuale di linee di codice del progetto che sono state eseguite dai test dopo un'esecuzione*)*;
- rilevano problemi associati ai sottosistemi;
- portano benefici sia agli sviluppatori, sia ai tester e sia ai manager.

6.4 Piramide di Testing e Clean Architecture

"The "Test Pyramid" is a metaphor that tells us to group software tests into buckets of different granularity. It also gives an idea of how many tests we should have in each of these groups.[4]"

La piramide dei test, teorizzata da Martin Flower, in sostanza delinea i tipi di test che dovrebbero essere inclusi in una suite di test automatizzati. Inoltre, descrive la sequenza e la frequenza con cui dovrebbero essere fatti questi test. Lo scopo è offrire un feedback immediato per assicurare che un refactoring al codice non interrompa la funzionalità esistenti.



6.1- Piramide dei test

3 https://www.katalon.com/resources-center/blog/end-to-end-e2e-testing/

4 Ham Vock*, The Pratical Test Pyramid,* martinFlower.com*,* 26 Febbraio 2018

Gli unit test sono alla base della piramide e rappresentano il sottoinsieme più grande, perciò dovrebbero essere scritti per primi. Bisogna tenere a mente, però, che ogni volta che viene aggiunto nuovo codice è necessario aggiungere nuovi unit test e runnarli. Così facendo, gli sviluppatori riusciranno ad avere feedback immediati.
Al secondo livello si trovano gli integration test, questo significa che rispetto agli unit test la quantità dovrà essere minore e non dovranno essere runnati così frequentemente, anche perché l'esecuzione sarà decisamente più lenta.
In cima sono collocati i test end-to-end, solitamente i più lunghi da eseguire. Possono anche essere considerati fragili, dal momento che devono testare una grande varietà di user scenarios.
Idealmente, dunque, bisognerebbe testare tutte le logiche tramite unit test, gli integration test aiutano a dare una buona copertura e, così, i test e2e possono limitarsi a verificare che l'applicazione funzioni correttamente senza andare a verificare passo dopo passo le varie logiche.

Nel mondo Android, un altro motivo per cui gli unit test vengono fatti in maggior numero è per il fatto che per essere eseguiti non hanno bisogno di alcun dispositivo mobile o emulatore, ma possono essere runnati direttamente dal computer tramite linea di comando e sono scritti in Java o Kotlin.
Mentre gli integration test e gli end-to-end test possono aver bisogno di elementi come activity, fragment e bottoni, quindi hanno bisogno di un dispositivo su cui essere eseguiti, perché questi componenti non esistono come concetti al di fuori di Android.
L'utilizzo della Clean Architecture favorisce notevolmente una corretta implementazione dei test: "*se l'architettura del vostro sistema è basata sui caso d'uso e se avete tenuto a debita distanza i framework dal nucleo dell'applicazione, allora dovreste essere in grado di eseguire uno unit-test in tutti questi casi d'uso senza l'assillo del framework. Non dovreste aver bisogno del server web per eseguire i vostri test. Non dovreste essere connessi al database per condurre i vostri test. I vostri oggetti entità dovrebbero essere comuni oggetti che non hanno dipendenze dal framework o dal database o da altri dettagli. Gli oggetti dei vostri casi d'uso dovrebbero coordinare i vostri oggetti entità. Infinte, tutti insieme dovrebbero essere testabili in situ, senza le complicazioni aggiunte dai framework.*[5]"
Siccome i moduli `data` e `domain` non sono dipendenti dal framework Android, è possibile testarli potenzialmente al 100% tramite unit test, mentre i test effettuati nel modulo `app` potrebbero aver bisogno di essere eseguiti su un dispositivo. Precisamente è possibile utilizzare ancora unit test per testare le implementazioni dei repository e i view model, mentre non è possibile per le view e i data source, questi ultimi poiché spesso utilizzano componenti offerti da Android (Room, Shared Preferences).

5 Robert C. Martin, *Clean Architecture,* Apogeo, 2018

È stato possibile sperimentare la scrittura di unit-test e UI test sulla Sample App messa a disposizione da Synesthesia. Di seguito vengono riportati due esempi.
Il primo test è uno unit-test, eseguiti con *Junit4*[6] e con il supporto della libreria *Mockito*[7], effettuato sulla seguente classe `GetEpisodeUseCase` del modulo domain:

```
open class GetEpisodeUseCase @Inject constructor(
        schedulerProvider: SchedulerProvider,
       private val episodeRepository: EpisodeRepository) : UseCase←
GetEpisodeUseCase.Params,  Either<EpisodeEntity>>(schedulerProvider)
{
    data class Params(val forceRefresh: Boolean, val episodeId: Int)

    override fun buildObservable(params: Params) =
           episodeRepository.getEpisode(params.forceRefresh, params.
episodeId)

}
```

GetEpisodeUseCase

Grazie alla semplicità consentita dalla Clean Architecture, è sufficiente testare che il metodo `buildObservable()` venga chiamato correttamente, così come riportato:

```
class GetEpisodeUseCaseTest {
    private var schedulerProvider = ImmediateSchedulerProvider()

    @Mock
    lateinit var episodeRepository : EpisodeRepository

    lateinit private var getEpisodeUseCase : GetEpisodeUseCase

    @Before
    fun setUp(){
       MockitoAnnotations.initMocks(this)
```

6 https://junit.org/junit4/
7 https://site.mockito.org/

```
            getEpisodeUseCase = GetEpisodeUseCase(schedulerProvider,
            episodeRepository)
    }

    @Test
    fun buildObservableTest(){
        val params = GetEpisodeUseCase.Params(true, 0)

        getEpisodeUseCase.buildObservable(params)

        verify(episodeRepository).getEpisode(eq(true), eq(0))
    }

}
```

GetEpisodeUseCaseTest

La classe ha delle dipendenze, `SchedulerProvider` ed `EpisodeRepository`, ma non è necessario creare delle vere istanze di queste classi, altrimenti bisognerebbe inserire un ciclo di dipendenze "infinito" e si perderebbe il senso del test. Per questo motivo vengono usati i *mock*, oggetti simulati che riproducono il comportamento degli oggetti reali in modo controllato.
Tutto ciò che viene inserito sotto l'annotation `@Before` viene eseguito sempre prima dei test, indicati, invece, con l'annotation `@Test`.
Quindi in questo caso, ogni qual volta si runna il test, viene creata una nuova istanza di `GetEpisodeUseCase` con gli attributi mockati. Nel test viene semplicemente chiamato il metodo `buildObservable()` e viene verificato che anche la chiamata al suo interno, effettuata sull'`episodeRepository`, venga eseguita correttamente.

Il secondo test riportato è un Android UI Test scritto con il supporto del tool *Espresso*[8] di Android, grazie al quale è possibile registrare i test usando l'applicazione.
In questo caso il test verifica che, dopo aver cliccato su uno specifico elemento della Recycler View, si apra un'altra View in cui esistono due specifiche Text View, con gli id title_dv e airdate_dv.

8 https://developer.android.com/training/testing/espresso

```
class DetailsTest {

    @Rule
    @JvmField
   var mActivityTestRule = ActivityTestRule(SplashActivity::class.java)

    @Test
    fun detailsTest() {
        Thread.sleep(700)

          val recyclerView = onView(allOf(withId(R.id.rv),
             childAtPosition(
             withClassName(`is`("android.widget.FrameLayout")), 2)))
          recyclerView.perform(actionOnItemAtPosition<ViewHolder>(0,
click()))

        Thread.sleep(700)

        val textView = onView(
    allOf(withId(R.id.title_dv), withText("Allianz Field: Level 4"),
                    withParent(allOf(withId(R.id.constraintLayout),
                        withParent(withId(android.R.id.content)))),
                    isDisplayed()))
        textView.check(matches(isDisplayed()))

        val textView2 = onView(
             allOf(withId(R.id.airdate_dv), withText("2019-11-14"),
                    withParent(allOf(withId(R.id.constraintLayout),
                        withParent(withId(android.R.id.content)))),
                    isDisplayed()))
        textView2.check(matches(withText("2019-11-14")))
    }

    private fun childAtPosition(
        parentMatcher: Matcher<View>, position: Int): Matcher<View> {
```

```
    return object : TypeSafeMatcher<View>() {
       override fun describeTo(description: Description) {
      description.appendText("Child at position $position in parent ")
                parentMatcher.describeTo(description)
            }

     public override fun matchesSafely(view: View): Boolean {
          val parent = view.parent
          return parent is ViewGroup && parentMatcher.matches(parent)
                        && view == parent.getChildAt(position)
            }
        }
    }
}
```

UI test

Test di questo genere servono per evitare errori di regressione, cioè aiutano ad accorgersi se le modifiche apportate in release successive hanno causato dei malfunzionamenti. Di conseguenza bisogna capire se aggiustare il codice o modificare il test, poiché, magari, quella funzionalità testata non è più necessaria.

In conclusione, il mondo dei testing è molto articolato e può essere più complesso di come descritto in questo capitolo, ma è fondamentale esserne a conoscenza per costruire un software robusto e affidabile.

Conclusioni

Inizialmente, l'adozione della Clean Architecture richiede un lavoro che può risultare oneroso in termini di tempo e abilità, tuttavia, come si è analizzato nel corso di questa tesi, dà i suoi frutti in tutto il ciclo di vita del software e aiuta notevolmente anche l'implementazione dei test, rendendo il software facilmente testabile.
È bene tenere a mente che la Clean Architecture è solo uno degli stili che si possono adottare per sviluppare un buon progetto software, non è l'unico ed imprescindibile. L'importante è essere consapevoli che prima di iniziare a scrivere codice, è necessario spendere del tempo nella progettazione dell'architettura basandosi su principi affidabili, come i principi SOLID e i principi di coesione e accoppiamento dei componenti.
Inoltre, come osservato nella conversione del caso studio, è essenziale saper distinguere le politiche di alto livello dai dettagli dell'applicazione e saper progettare le dipendenze in maniera efficiente.

"L'unico modo per procedere rapidamente, è fare le cose per bene."
Robert C. Martin

Appendice

I Flavor e le Build Variant

In questa appendice viene esposto un progetto in corso di Synesthesia, a cui è stato possibile partecipare per la realizzazione di alcuni piccoli task.
Per ragioni di copyright, il nome del cliente ed eventuali altre informazioni non verranno riportati.

Il cliente ha chiesto a Synesthesia di sviluppare quattro applicazioni mobile da distribuire a quattro tipologie di clienti/utenti differenti. Queste app sono, in concreto, la stessa applicazione ma ognuna ha delle proprie personalizzazioni. Le differenze riguardano elementi grafici come il cambio di colori e icone, l'aggiunta o la rimozione di alcune funzionalità, requisiti più sottili quali il baseurl del server da chiamare o l'api key di maps da fornire.
Per rispondere a questa richiesta, l'azienda ha scelto di utilizzare i *flavor*.

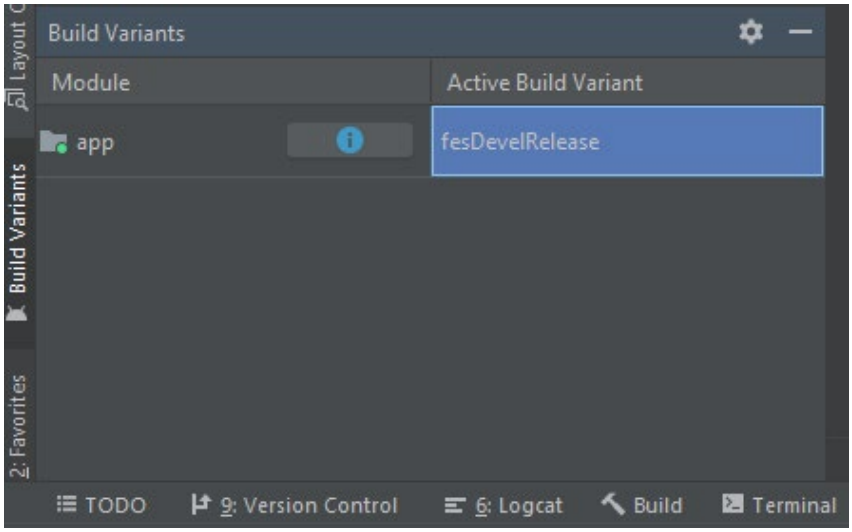
Flavor in italiano si traduce con sapore, gusto, aroma e in ambito di sviluppo mobile significa la stessa cosa. Non è altro che una variante, un tocco che si aggiunge al "piatto" per ottenere un gusto differente. Gli ingredienti base del piatto sono il main, il quale è accessibile da tutti i flavor e si possono creare directory diverse per ogni flavor dove aggiungere codice specifico per quella personalizzazione.

Esistono diverse situazioni in cui si ha bisogno di utilizzare i flavor; un caso è quello capitato a Synesthesia, un altro potrebbe essere la necessità di avere una versione gratuita e una versione a pagamento della stessa applicazione.

Per configurare i flavor si utilizza il plugin *Gradle* di Android, un toolkit di build avanzato per automatizzare e gestire il processo di build.
Nei file `build.gradle` vengono inserite le configurazioni del progetto. Esiste sempre un file in cui vengono definite tutte le dipendenze dalle diverse librerie a livello del progetto intero e altri file `build.gradle` che riguardano le configurazioni nei diversi moduli dell'applicazione.
Nel caso del progetto di Synesthesia, poiché è stato sviluppato seguendo le logiche della Clean Architecture, esistono tre file `build.gradle`: per il modulo app, per il modulo data e per il modulo domain.
È proprio nel file `build.gradle (Module: app)` che si va a inserire il codice per configurare i flavor e, di conseguenza, le *Build Variant*. I moduli data e domain non sono Android, ma Java, perciò non hanno il concetto di flavor.

Un flavor è caratterizzato da un tipo di dimension. Nel caso del progetto in questione, esistono due diversi tipi di dimension: app ed env.

- `app` è la singola applicazione chiamata con il suo nome. Dato che le applicazioni richieste sono quattro, sono presenti quattro flavor, che per ragioni di copyright si chiameranno: *purple, red, blue e yellow*.
- `env` è il tipo di ambiente, il buildType che applica diverse impostazioni di build e packaging. Per questo progetto sono presenti quattro buildType differenti: *devel (sviluppo), staging, preprod e prod (produzione)*.



A.1 - Pannello Build Variant in Android Studio

I flavor vengono combinati a matrice per costruire una build variant.
Secondo Google, le build variant sono il risultato dell'utilizzo di Gradle di un insieme specifico di regole per combinare impostazioni, codice e risorse configurate nei tipi di build e nelle versioni del prodotto.
Quindi, poiché ci sono due dimensions con quattro varianti ciascuna saranno disponibili sedici varianti diverse da runnare. Otto varianti per ogni applicazione: per l'applicazione Purple si avrà a disposizione la versione devel, staging, preprod e prod e così via per le altre.

	Purple	Red	Blue	Orange
devel	*purple.devel*	*red.devel*	*blue.devel*	*orange.devel*
staging	*purple.staging*	*red.staging*	*blue.staging*	*orange.staging*
preprod	*purple.preprod*	*red.preprod*	*blue.preprod*	*orange.preprod*
prod	*purple.prod*	*red.prod*	*blue.prod*	*orange.prod*

A.2 - Matrice di configurazione dei flavor

Di seguito viene riportato parte del codice presente nel file `flavors.gradle (Module:app)`, per comprendere meglio la creazione dei flavor:

```
android {

    flavorDimensions "app", "env"

    productFlavors {
        all {
            versionCode Integer.parseInt(project.VERSION_CODE)
            versionName project.VERSION_NAME
            manifestPlaceholders = [fabricApiKey: "..."]
        }

        purple {
            dimension "app"
```

Configurazione dei flavor nel file flavors.gradle

```
            applicationId "com.clientname.purple"
            buildConfigField "String", "CUSTOMER", '"Purple"'
            resValue "string", "application_name", "Purple"
        }
        devel {
            dimension "env"
            applicationIdSuffix ".devel"
            buildConfigField "boolean", "CRASHLYTICS_ENABLED", "true"
        }
    }
```

Configurazione dei flavors nel file flavors.gradle

Nel flavor bisogna definire la dimension a cui appartiene.
Nel caso di `app` bisogna configurare i parametri quali: `applicationId`, `buildConfigField` e `resValue`. Tra gli ambienti, invece, cambia il suffisso da aggiungere al package name.

In questo specifico progetto c'è un grado di complicazione in più, poiché ci sono determinate richieste che dipendono non solo dal tipo di dimension, ma dall'unione delle due dimensions.
Per cui ci possono essere delle differenze anche tra `purple.devel` e `purple.staging`.
Per esempio, sono stati forniti baseurl del server e api key di maps differenti per i diversi incroci. Per rispondere a questa esigenza, bisogna scrivere del codice di questo tipo:

```
applicationVariants.all { variant →
    if (variant.getName().contains("purpleDevel")) {
          variant.resValue "string", "application_name", "Purple Dev"
          buildConfigField "String", "endpoint", '"https:// ... /"'
          variant.resValue "string", "maps_api_key", " ... "
}
    else if (variant.getName().contains("purpleStaging")) {
       variant.resValue "string", "application_name", "Purple Staging"
        buildConfigField "String", "endpoint", '"https:// ... "'
        variant.resValue "string", "maps_api_key", " ... "
        }
}
```

Configurazione delle varianti nel file flavors.gradle

Tutto quello che è stato riportato fin qui serve per creare e modificare le configurazioni delle varianti. Ciò che non cambia, ancora, è la personalizzazione grafica. Per modificare i colori, le icone, i layout, si va ad operare nel modulo app.
Qui è presente la directory main in cui è presente tutto il codice necessario per costruire la base di tutte le quattro applicazioni. Dato che l'app è costruita seguendo le regole della Clean Architecture, in questa directory sono presenti tutte le logiche di basso livello viste in precedenza: view, datasource, package injection, e così via.
Nel package `res` di main sono descritti tutte le risorse grafiche (layout, values, animazioni, ecc.) su cui si basano tutti i flavor.
Nel momento in cui si crea la directory del nuovo flavor, per esempio Purple, si vanno a inserire nel package `res` di Purple le personalizzazioni che si vogliono avere.
La logica con cui Android legge nelle varie cartelle è la seguente:

`devel` ⟶ `purple` ⟶ `main`.

Cioè, legge nella cartella devel e controlla se è presente qualche modifica nelle resource, poi si sposta su purple ed effettua lo stesso controllo ed infine su main. Se ha trovato modifiche alle resource di purple, le sovrascrive a quelle di main e lo stesso per devel.
Per esempio nel file `colors.xml` di main esiste una variabile chiamata colorPrimary con il valore #00b1ae, mentre nel file analogo di Purple il valore è #b640a1 . Quindi l'azzurro sarà sovrascritto al viola.

Bibliografia e Sitografia

Alessio Fiore, *Quality of Service di un software*, italiancoders.it, 14 Febbraio 2018

Nikolay Ashanin, *The Path to Becoming a Software Architect*, medium.com, 1 Ottobre 2017

Peter Eeles, *What is a software architecture?*, ibm.com, 15 Febbraio 2006

Software Architecture & Software Security Design, synospsys.com

Suragch, *Clean Architecture for the rest of us,* pusher.com, 4 Gennaio 2019

The Clean Architecture, The Clean Code Blog by Robert C. Martin, 13 Agosto 2012

Giuseppe Capodieci, *SOLID Design Principles*, losviluppatore.it

Ugonna Thelma, *The S.O.L.I.D. Principles in Pictures*, medium.com, Maggio 2018

Hgraca, *Architectural Styles vs. Architectural Patterns vs. Design Patterns*, herbertograca.com, 28 luglio 2017

Zanfina Svirca, *Everithing you need to know about MVC architecture*, towardsdatascience.com, 29 Maggio 2020

Florina Muntenescu, *Android Architecture Patterns Part 1: Model-View-Controller*, 1 Novembre 2016

Florina Muntenescu, *Android Architecture Patterns Part 2: Model-View-Presenter*, 2 Novembre 2016

Florina Muntenescu, *Android Architecture Patterns Part 3: Model-View-ViewModel*, 4 Novembre 2016

Eran Kinsbruner, *Manual Testing vs. Automated Tesing, perfecto.io*, 13 Agosto 2019

Shreya Bose, *Testing Pyramid : How to jumpstart Test Automation*, browserstack.com, 21 Gennaio 2020

https://refactoring.guru/design-patterns/what-is-pattern

https://www.tutorialspoint.com/software_engineering/software_testing_overview.html

https://it.wikipedia.org/wiki/Unit_testing

https://www.javatpoint.com/integration-testing

https://www.katalon.com/resources-center/blog/end-to-end-e2e-testing/