ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

APPLICAZIONI PER ANDROID WEAR: PROCESSO DI SVILUPPO, ARCHITETTURA E REALIZZAZIONE DI UN CASO DI STUDIO

Relazione finale in
PROGRAMMAZIONE AD OGGETTI

Relatore
Prof. MIRKO VIROLI

Correlatore
CLAUDIO BUDA

Presentata da
MARCO BALDASSARRI

Terza Sessione di Laurea
Anno Accademico 2015 – 2016

PAROLE CHIAVE

Wearable Computing

Android Wear

Clean Architecture

VIPER

Scrum

Alla mia famiglia,
che con affetto e vicinanza mi ha sempre sostenuto.
Ai miei amici,
che ogni giorno mi fanno vedere la vita in risoluzione 8K.

Indice

Abstract

In un'epoca in cui la società è costantemente connessa alla Rete, è presto nata l'esigenza di accedere all'informazione in una maniera più immediata ed intuitiva. Questo ha portato alla nascita e allo sviluppo dell'Internet of Things e degli Embedded Systems. A giorno d'oggi in particolare, sia nel contesto italiano che internazionale, è in grande crescita il numero di imprese del mondo ICT che investono risorse su una branca dell'IoT, che è quella del Wearable Computing, ovvero delle nuove tecnologie indossabili.

Questa tesi si pone essenzialmente l'obiettivo di indagare quali siano i nuovi processi e le architetture che coinvolgono lo sviluppo dei sistemi Software moderni e orientati al contesto Wearable e Mobile, facendo un confronto con le esigenze del passato. A tale scopo si apporteranno vari esempi di esperienza diretta nell'ambito di un tirocinio aziendale. Si andranno poi a definire dettagliatamente le fasi di Analisi, Design ed Implementazione che hanno portato alla costruzione di una Wearable Application usando le tecnologie fornite da Google per lo sviluppo su Android Wear.

Introduzione

Nell'ambito dell'Internet of Things, sono molteplici e in via di sviluppo gli ambiti applicativi che destano interesse per la ricerca scientifica e aziende del settore. Un ambito in continua evoluzione e che negli ultimi anni ha portato ad importanti cambiamenti ed innovazione è quello del Wearable Computing. Le aziende informatiche stanno investendo molto su questo nuovo campo, ingenti risorse economiche ed umane sono attualmente impiegate per lo sviluppo di soluzioni Software e Hardware a supporto dei wearable devices, sia per specifici fini industriali che per il mercato consumer. Per fare un esempio, le più grandi compagnie IT come Google e Microsoft hanno recentemente prodotto occhiali intelligenti che permettono all'utilizzatore di catapultarsi in una realtà aumentata che viene sovrapposta a quella vissuta dall'indossatore. Un altro device di rilevanza e il cui utilizzo è attualmente in grande crescita è senza dubbio lo Smartwatch, sul quale Google e Apple stanno sviluppando nuovi Sistemi Operativi ottimizzati e aperti agli sviluppatori. Il colosso di Mountain View in particolare, ha rilasciato negli ultimi anni un Sistema chiamato Android Wear ed entro il 2017 rilascerà la versione 2.0, ricca di nuove accattivanti feature grafiche ed operative. Gli orologi intelligenti sono stati visti fin da subito come un estensione degli Smartphone, consentono quindi una user experience migliorata grazie al fatto che l'utente può accedere alle informazioni di cui ha bisogno in maniera più immediata, senza dover compiere il gesto di tirare fuori dalla tasca il proprio telefono. In questo senso l'azienda Mango Mobile Solution (d'ora in poi, per semplicità: Mango) [1], una Startup che opera nell'ambito dello sviluppo di soluzioni Mobile e IoT su scala nazionale, ha voluto propormi un'esperienza di tirocinio per Tesi direzionata allo studio delle nuove tecnologie legate ad Android Wear e all'implementazione di una Wearable App che verrà utilizzata dal cliente finale.

Il presente lavoro di Tesi consiste quindi in prima analisi nella descrizione dell'intero processo aziendale sul quale si appoggia Mango e delle Architetture innovative e robuste utilizzate per lo sviluppo del Software. In secondo luogo verranno spiegate dettagliatamente le fasi che hanno portato all'implementa-

[1] http://mangomobi.com/

zione del progetto finale. I capitoli della tesi sono quindi organizzati come segue: la prima parte è dedicata ad un background più discorsivo riguardante la storia e i casi d'uso più comuni del Wearable Computing [8], con particolare riferimento al mondo degli Smartwatch Android Wear, a come vengono utilizzati dagli utenti nei vari contesti, quali sono i prodotti Hardware disponibili nel mercato e quali sensori integrano i Watch più venduti.

Il secondo capitolo verte su un background con un taglio più metodologico e tecnologico orientato all'esperienza di tirocinio in azienda: vengono descritti in dettaglio i metodi Agili, i linguaggi di modellazione, i tool di supporto allo sviluppo, alla progettazione e al deployment utilizzati in Mango. Si parlerà quindi largamente della nuova Clean Architecture [21], di tutti i componenti di cui è costituita e di come, grazie al Single Responsability Principle su cui si basa, possa avere un impatto estremamente positivo sull'ingegnerizzazione dei sistemi nei contesti moderni.

Per portare un esempio di implementazione pratica della Clean Architecture, gran parte del secondo capitolo è dedicata all'Architettura VIPER [22], con particolare riferimento all'uso di VIPER su Android. Basandosi sistematicamente sui principi di buona progettazione, tale estensione viene attivamente utilizzata da Mango per lo sviluppo, a prescindere dalle piattaforme e dal contesto nel quale si va ad operare. Verranno spiegate le motivazioni che hanno portato l'azienda a fare questa scelta e il pattern VIPER verrà brevemente confrontato con i pattern più classici, come Model-View-Controller.

A supporto della modellazione e della progettazione di sistemi con Architettura VIPER vengono in aiuto gli schemi C4 [19], considerati un sostituto ad alto livello del classico linguaggio UML, ormai scarsamente utilizzato dalle aziende soprattutto perché non adatto a contesti Agili. Tra i tool verrà citato Bitbucket [2] per il versioning del codice, Trello [3] per la gestione Agile dei vari task da parte dei membri del Team e Fabric [4], utilizzato per un reporting efficace dei bug una volta che il software viene messo in produzione.

Sempre nel secondo capitolo, si pone poi l'attenzione sul framework Android Wear [5] rilasciato da Google per gli sviluppatori, fornendo una panoramica dei principali componenti utilizzati per l'interazione con l'utente, per la ricezione di notifiche e per lo scambio dei dati tra Smartwatch e Handheld device. Si discuterà delle differenze tra la prima e la seconda versione in arrivo e in che direzione Google vuole portare il Wearable Computing applicato ai Watch.

Nell'ultimo capitolo si parla in dettaglio della fase di Analisi, di Design e di Implementazione che hanno portato allo sviluppo della Wearable App. Si

[2] https://bitbucket.org/
[3] https://trello.com/
[4] https://get.fabric.io/
[5] https://developer.android.com/wear/

tratta di un'estensione per indossabile di un'App già esistente su piattaforme Android e iOS per fare promozione, marketing e comunicazione nell'ambito teatrale.

Tra i risultati del lavoro svolto durante l'attività di tirocinio vi è la creazione di un applicativo in grado di rendere fruibile all'utente gli eventi in programma del teatro a cui è iscritto, leggere direttamente dall'orologio informazioni dettagliate (come ora di inizio, descrizione e costi del biglietto) sugli spettacoli in programma. L'utilizzatore potrà inoltre interagire aprendo uno spettacolo sul telefono direttamente dall'orologio, o marcare quello spettacolo come preferito. Le fasi di design e implementazione avranno da un lato un taglio fortemente Agile, dove si fa leva sul flusso delle varie Sprint pianificate durante i quattro mesi di tirocinio; dall'altro seguirà più in dettaglio la divisione del codice in moduli VIPER.

Complessivamente, mi ritengo molto soddisfatto sia dell'attività svolta in azienda sia della stesura di questa tesi. Ho avuto l'opportunità di mettere in pratica i concetti appresi durante il corso di studi, implementando un progetto funzionante e totalmente spendibile da Mango. Ho avuto occasione inoltre di entrare a contatto con nuove tecnologie ed approcci innovativi alla progettazione e alla programmazione del Software, nonché di lavorare in Team sfruttando un'ottica basata sui metodi Agili, oggi molto richiesti dal mercato del lavoro.

Capitolo 1

Wearable Computing e IoT

1.1 Internet of Things

"We turn on the lights in our house from a desk in an office miles away. Our refrigerator alerts us to buy milk on the way home. A package of cookies on the supermarket shelf suggests that we buy it, based on past purchases. The cookies themselves are on the shelf because of a "smart" supply chain. When we get home, the thermostat has already adjusted the temperature so that it's toasty or bracing, whichever we prefer. This is the Internet of Things" [1]

A partire dalla nascita del progetto ARPANET nel 1969, l'avvento di Internet ha rivoluzionato l'intera società e il nostro modo di comunicare [2]. Una rete di router, server e macchine hanno reso l'intero mondo sempre più interconnesso grazie al passaggio di informazioni da una parte all'altra della Terra in frazioni di secondo.
Fino al 1999 si è sempre parlato di comunicazione machine-to-machine, ma proprio in quell'anno Kevin Ashton, ricercatore del MIT, nel discutere di progetti relativi alle tecnologie RFID, ha coniato il termine Internet of Things. Sebbene il concetto non sia stato nominato prima del 1999, era comunque già presente da decenni: il primo esempio fu quello di un distributore di Coca Cola all'Università di Carnegie Melon nei primi anni '80. I programmatori potevano connettersi alla macchina attraverso Internet e controllarne lo stato per verificare se ci fossero ancora bibite fresche disponibili [3].

Per Internet of Things (Internet delle Cose), si intende un sistema di dispositivi intelligenti interconnessi che possono scambiarsi informazioni utili per un determinato fine. Viene a cadere la vecchia percezione che l'uomo ha di Internet, crolla la barriera tra mondo fisico e digitale, si passa dall'idea di una comunicazione *machine-to-machine* ad una comunicazione più ad ampio

raggio; l'uomo stesso è preso in considerazione ed interagisce con altri Smart Objects presenti nell'ambiente circostante [4].

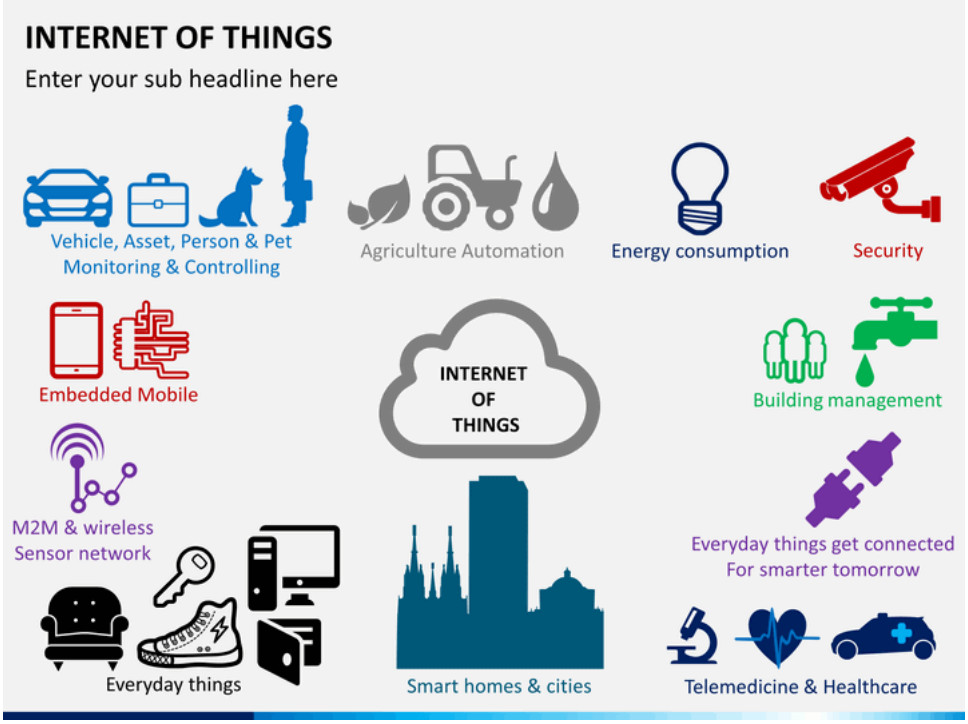


Figura 1.1: Immagine esplicativa di alcuni scenari applicativi di IoT

Si parla di smart things perché agli oggetti viene aggiunta una qualche forma di intelligenza: grazie all'utilizzo di un'appropriata infrastruttura di rete, di un hardware fatto di piccoli sensori e attuatori e di interfacce software dedicate alla comunicazione efficace tra dispositivi embedded, gli oggetti vengono vengono connessi e viene assegnato loro un ruolo attivo e ben definito all'interno del sistema. Ogni oggetto diventa quindi un generatore di dati e al contempo un esecutore di azioni più o meno fisiche a fronte di opportuni eventi e trigger da parte di altri device. I principali ambiti applicativi dell'IoT sono la sanità, i trasporti e la logistica, gli edifici intelligenti (musei, cinema, teatri, uffici, case), l'agricoltura, e in un'ottica più orientata al futuro, si può parlare anche di Smart City e robotica.

Le cose quindi, nell'ambito IoT, possono essere oggetti di qualsiasi genere e variano a seconda del contesto in cui ci troviamo: da un sensore di temperatura regolabile a distanza fino ad automobili che si comportano diversamente a seconda delle condizioni fisiche del conducente. Anche gli Smartphone stessi possono essere considerati parte dell'ecosistema degli smart devices, ma spesso vengono utilizzati come centro di controllo accessibile all'utente. Negli anni la

rete Internet è stata raggiunta anche nelle zone del mondo più remote e nei paesi più sottosviluppati. Alcune stime recenti riportano che in media ogni nucleo familiare possiede 7 device connessi in Rete, che siano Smartphones, tablet, console o PC. [6] Con l'avvento dell'IoT questo numero è aumentato ulteriormente. Uno studio recente afferma infatti che il numero di devices connessi aumenterà esponenzialmente fino ad arrivare a 50 miliardi entro il 2020, a fronte dei 10 miliardi del 2015 [5]. E' sopraggiunto quindi un problema legato all'architettura stessa del protocollo IPv4: il numero sempre crescente di devices sta causando una riduzione degli indirizzi IP a disposizione. Per far fronte a questo problema, nel 2011 è stato progettato e reso pubblico il nuovo protocollo IPV6, al quale molti ISP hanno già da tempo aderito. Il protocollo riesce a sostenere 2^{128} indirizzi a fronte dei 2^{32} di IPv4; è ritenuto quindi un fattore fondamentale nello sviluppo dell'Internet of Things. Steven Leibson afferma che "Potremmo assegnare un indirizzo IPV6 a ciascun atomo sulla faccia della Terra, ed avere abbastanza indirizzi rimasti per coprire più di altre 100 Terre".

Ovviamente non a tutti i sensori deve necessariamente essere assegnato un indirizzo IP pubblico, ma possono comunuque comunicare tra di loro creando una piccola rete locale la quale comunica con altre reti locali e così via. In questo senso l'IoT è visto come una Network of Networks. Le automobili moderne, ad esempio, hanno al loro interno reti multipli di sensori per controllare il funzionamento del motore e per gestire la sicurezza del conducente, così come molti edifici residenziali e commerciali hanno piccole reti separate per il controllo dell'aria condizionata, delle comunicazioni telefoniche ecc.

1.2 Wearable Computing

Un altro esempio molto significativo di Rete di Rete in ambito IoT è sicuramente quello del Wearable Computing. Il termine si riferisce a quella branca dell'Internet of Thing che si occupa di device di dimensioni molto ridotte e indossabili dall'essere umano; dispositivi che messi insieme possono creare una vera e propria *personal wireless local-area-network*, dove il fine ultimo è quello di raccogliere dati e renderli accessibili. Grazie al Wearable Computing è molto facilitata l'interazione uomo-macchina in quanto consente all'utente di avere un accesso immediato all'informazione che cerca proprio perché indossa il device. L'utilizzatore può interagire con tali smart objects, i quali si comporteranno diversamente a seconda del contesto e della situazione, fornendo così un'esperienza utente molto vicina a quella di un assistente personale intelligente. La Wearable Technology è nata quindi con lo scopo di aiutare le attività quotidiane dell'essere umano nella vita di ogni giorno, lasciando più

libertà nell'uso di mani e occhi, cosa sulla quale lo smartphone viene meno. Per questo motivo la maggior parte dei device indossabili sono concentrati sull'area delle mani e del polso (come nel caso di braccialetti, anelli, e Smartwatch) o della testa (smart glasses, orecchini, collagne, auricolari), ma la tecnologia lascia libero spazio all'immaginazione, si sta quindi iniziando a parlare anche di *Smart Clothing* come scarpe, magliette, jeans e cuffie o simili [7].

1.2.1 Applicazioni pratiche

A seconda del contesto e del device al quale ci si riferisce, negli anni sono emersi svariati rami al quale fa riferimento il Wearable Computing, branca dell'Informatica nata anche grazie alla sempre più consistente diminuzione nelle dimensioni dell'hardware presente in commercio. Tali tecnologie hanno rivoluzionato positivamente molti contesti applicativi fornendo un vero e proprio supporto all'essere umano. Alcuni degli esempi più rilevanti possono essere:

- Fitness e Wellness: le funzione tipiche di questo campo sono il monitoraggio dell'attività fisica, la cattura attraverso sensori dei dati fisiologici e la gestione degli obiettivi giornalieri dell'atleta, ma anche la condivisione della propria posizione, la tracciabilità del percorso effettuato durante una corsa o una camminata. Lo Smartwatch e le *smart training shoes* ne sono due esempi tipici. Nella maggior parte dei casi, entrambi integrano sensori di movimento, GPS e accelerometro.

- Medicina e Sanità: il contesto sanitario è stato uno dei primi successi del Wearable Computing ed è stato fonte di ispirazione di tutti gli altri campi. L'uso tipico è quello del monitoraggio a distanza dello stato del paziente, al quale sono apposti sensori connessi in Rete. Gli esempi più comuni sono l' ECG (elettrocardiogramma) o sensori che rilevano la saturazione dell'ossigeno e la pressione arteriosa.

- Servizi Militari: i dispositivi installati sulle uniformi di ogni soldato permettono un accesso facilitato a informazioni tattiche, di geolocalizzazione e di controllo. Essi inoltre riescono a comunicare tra di loro, creando una rete che risulta utile per avere un più chiaro riconoscimento di chi sia il nemico nelle vicinanze.

- Industria: a seconda di quale sia il tipo di azienda, sono molti i contesti nei quali un impiegato si può trovare a beneficiare dell'uso di un dispositivo indossabile, i quali, oltre che aiutare nello svolgimento dei vari task, garantiscono una maggiore sicurezza sul lavoro, comfort e maggiore disponibilità all'interazione sociale con gli altri lavoratori. Un esempio

pratico può essere quello della Realtà Aumentata, ambito di ricerca molto attivo negli ultimi anni: un impiegato, nell'eseguire il proprio lavoro, potrebbe fare uso di alcune "istruzioni" visuali proiettate dai propri occhiali smart. Un altro esempio è quello di Mercedes-Benz, azienda produttrice di automobili di qualità, la quale ha usato lo Smartwatch per visualizzare informazioni utili al conducente sullo stato dell'automobile sulla posizione dove è stata parcheggiata [8].

1.2.2 Breve storia del Wearable Computing

Siamo portati a pensare che date le piccole dimensioni dell'hardware indossabile a cui siamo abituati oggigiorno, la tecnologia che vi si cela dietro sia frutto di recenti scoperte in ambito di ricerca, ma questo non è il caso degli Smartwatch, smart glasses e di tutti gli altri dispositivi presenti sul mercato, i quali prototipi iniziali risalgono ai primi anni '80. In questo decennio anche la cinematografia ha avuto la sua parte, producendo film aventi come tema ricorrente il futuro e la fantascienza, da cui molti scenziati hanno tratto inspirazione negli anni successivi.

Storicamente i *Wearable Devices* venivano utilizzati per scopi militari, medici e di ricerca scientifica. Uno dei precursori che ha contribuito alla loro diffusione è senza dubbio Steve Mann, ricercatore statunitense nato nel 1962 e studente del Massachusetts Institute of Technology. La passione per Informatica e fotografia ha portato lo studioso ad essere riconosciuto come il padre della tecnologia Wearable e della realtà aumentata.

Figura 1.2: Immagine di Steve Mann durante i suoi esperimenti sul Wearable Computing

Molti fatti sono accaduti a partire da quegli anni, verranno tuttavia riportati i punti più salienti:

- 1979: l'azienda leader nel campo dell'elettronica Sony introduce il primo Walkman col fine di poter riprodurre musica attraverso audiocassette in mobilità [9].

- 1981: Steve Mann, nel portare a termine una ricerca sulla *computational photography*, utilizza un vecchio *computer-6502* per costruire una macchina fotografica montabile sulla testa. Si tratta della prima scoperta interessante riguardante la tecnologia indossabile e sarà un punto di ispirazione per tutte le scoperte future.

- 1994: con l'ausilio di un casco lo stesso Mann monta una webcam collegata ad un router portatile e riesce nella trasmissione delle immagini in diretta direttamente sul Web. Questo evento è un chiaro segno di apparizione dell'antenato dei Google Glass. Nello stesso anno viene alla luce il primo prototipo di *wrist computer* e presentato durante la CHI-94 conference di Boston. I progressi nella wearable technology hanno portato al lancio della prima conferenza sul wearable, tenuta a Seattle nel 1996 [10].

- 2000: l'avvento del nuovo millennio ha portato alla nascita di grandi innovazioni in ambito wearable. Tra queste spicca il *Tinmith wearable computer*, un sistema creato da Bruce H. Thomas e Wayne Piekarski che fungeva come supporto di ricerca per la realtà aumentata.

- 2002: in questo anno viene introdotto il *Poma Wearable PC* della Xybernaut, il quale però non ebbe molto successo e non venne quindi commercializzato, ma fu di ispirazione per molti progetti wearable attivati di recente.

- 2009: introdotto il *W200 Wearable Computer*, un mini PC indossabile come bracciale con display touch a risoluzione 320x240, sistema operativo Windows CE intercambiabile con Linux. Non mancava anche un modulo GPS e uno per la connessione wired e wireless sia Wi-Fi che Bluetooth. E' stato progettato per essere adoperato nei servizi di emergenza, sicurezza e logistica. Assistiamo quindi ad un primo esemplare di device realmente utilizzato, anche se non ancora in ambito consumer [11].

- 2006: Apple e Nike realizzano i primo activity traker della storia. In quegli anni viene fondata l'azienda FitBit da James Park e Erik Friedman, oggi società leader nel settore del fitness.

- 2010: inizia un decennio molto importante e pieno di innovazione. Si abbassano i prezzi e aumenta l'efficienza delle batterie, memorie, processori, quindi l'avanzamento della wearable technology diviene più semplice. Nei primi mesi del 2010 vengono presentati iPod Nano di Apple, forniti con una fascia da braccio per poterli indossare durante un allenamento fisico, e iniziano a vedersi i primi prototipi di Google Glass, sicuramente molto più interessanti dal punto di vista tecnologico e attraenti per il pubblico rispetto ad altri occhiali smart già esistenti. Inoltre, in questo anno assistiamo soprattutto ad un primo popolamento degli Smartwatch, presto inseriti nel mercato consumer.

- 2012: gli Smartwatch e i braccialetti prendono piede sul mercato, montano al loro interno dei sensori integrati per monitorare le attività corporee dell'indossatore; sono tutti device mirati ad aumentare l'esperienza utente durante l'allenamento fisico. Nei primi mesi dell'anno Nike lancia il Nike+ FuelBand il quale mostra delle informazioni testuali su un piccolo schermo. Successivamente viene presentato primo prototipo *Pebble* grazie ad una raccolta fondi di 100 mila dollari usando un sito di crowdfunding che renderà l'azienda molto apprezzata. Oltre ai sensori per l'activity tracking negli anni successivi verrà aggiunta la comunicazione con gli smartphone Android e iPhone, per visualizzare informazioni utili all'utente direttamente sul polso.

- 2013: Sergey Brin presenta ufficialmente i nuovi Google Glass; il pubblico si mostra molto più interessato rispetto ad altri occhiali intelligenti già presenti, grazie soprattutto alla tecnologia innovativa e al design poco invasivo per il viso. Tuttavia non verranno mai commercializzati per problemi di surriscaldamenti nella parte della batteria e per motivi di privacy. Nel mese di settembre stesso anno Samsung lancia Samsung Galaxy Gear, che insieme ad altri modelli simili permette di ricevere messaggi e leggere le email direttamente sul dispositivo.

- 2014: Google annuncia Android Wear con molte nuove feature e grafica user friendly, la diffusione è immediata grazie anche alla partnership con i big dell'elettronica come Motorola, Samsung, LG, HTC e ASUS.

- 2015: a Settembre di questo anno Apple svela il nuovo Apple Watch al WWDC. Insieme ad Android Wear questi ultimi due anni sono stati molto importanti per la diffusione degli Smartwatch e in generale per le nuove tecnologie in ambito Wearable Computing [12].

1.3 Android Smartwatch

1.3.1 Introduzione

Uno dei device più utilizzati nell'ambito del Wearable Computing è lo Smartwatch, la quale diffusione aumenta ogni anno, grazie agli investimenti di molte aziende leader del settore. Apple con *iWatch* e Google con *Android Wear* sono i due produttori principali di software e hardware legati al mondo degli Smartwatch. In questa sede si vuole fornire una breve panoramica sugli Smartwatch acquisiti da Google.

1.3.2 Sensori

Ogni marca di Smartwatch con Sistema Operativo Android Wear integra un diverso set di sensori, è quindi difficile definirne una lista esaustiva. Di seguito vengono elencati i principali:

- Accelerometro: misura l'accelerazione associata al movimento
- Giroscopio: rileva i movimenti in tre direzioni. Insieme all'Accelerometro è utile per capire a quali movimenti è sottoposto il device
- Sensore Barometrico: misura le variazioni di pressione atmosferica, utile per una più accurata geolocalizzazione
- GPS: determina la posizione del dispositivo sulla Terra, generalmente usato per la geolocalizzazione
- Bussola: misura il campo magnetico, insieme agli altri sopra elencati è un sensore di supporto al posizionamento
- Sensore di luce ambientale: aumenta/diminuisce la luminosità in base alla quantità di luce esterna
- Wi-Fi: per la connessione alla rete locale senza fili
- NFC: Near Field Communication, protocollo per stabilire una comunicazione a distanza molto ravvicinata con un altro device. Spesso usato per consentire operazioni di accettazione come pagamenti da carta di credito senza l'utilizzo della carta magnetica.
- Pedometro: ha la funzione di contare i passi effettuati dall'utente che lo indossa

- Bluetooth: indispensabile per la comunicazione con lo smartphone. I produttori fanno spesso utilizzo della versione LE (Low Energy) per consentire un risparmio della batteria.

- Sensore di Temperatura Ambientale: misura la temperatura e l'umidità esterne.

- Cardiofrequenzimetro: riesce a rilevare il battito cardiaco dell'indossatore

1.3.3 Principali utilizzi

Grazie alla presenza dei sensori, si allarga il ventaglio dei contesti nei quali può essere presente uno Smartwatch. I casi d'uso sopra elencati ne sono ottimi esempi. Ciò nondimeno, uno degli utilizzi maggiori, che ha portato alla sua diffusione, è quello di fungere da "assistente personale" o da oggetto di svago. L'utente può, anche tramite dei comandi vocali, ricevere ed effettuare chiamate, leggere email e messaggi, visionare le proprie Note o Appuntamenti nel Calendario, leggere News in tempo reale o usare le più comuni App di *Instant Messaging*. Su Smartwatch sono anche molto utilizzate le App relative alla Musica. Esempi che non si può fare a meno di menzionare sono MusicMatch, per vedere i testi della canzone che si sta ascoltando, Shazam, dove il focus è la velocità di utilizzo, per non perdere il nome della canzone che sta passando alla radio, e Spotify, con la possibilità di effettuare le classiche operazioni di play, pausa, next, prev, scelta della playlist, ricerca e cambio canzone. Come già accennato in precedenza, un altro caso d'uso molto presente è quello del fitness. Grazie ai sensori integrati è possibile tracciare la propria posizione e ricevere altri dati importanti durante l'attività fisica senza aver bisogno di interagire con un telefono. In qualunque contesto applicativo si tende a demandare al device la maggior parte del lavoro, quindi anche se lo Smartwatch ha a disposizione un GPS per tracciare la posizione, è preferibile usare quello del telefono se la connessione tra i due è attiva. In presenza di un allenamento fisico, per ragioni di comodità capita spesso che i due device siano scollegati; i risultati dopo l'allenamento sono direttamente visualizzabili dall'orologio ma si possono facilmente sincronizzare con lo smartphone una volta ricollegato tramite Bluetooth, in modo da poter visionare le proprie statistiche tramite App su smartphone, favorendo anche una GUI più comoda su cui interagire coi propri dati. Tra le App più comuni e utilizzate troviamo Google Fit, Fitbit, Runtastic.

1.3.4 Bluetooth

Il protocollo di comunicazione comunemente utilizzato dagli Smartwatch è il Bluetooth, considerato uno degli standard globali nell'ambito della comunicazione wireless su cui si appoggia anche l'Internet of Things e il Wearable Computing. L'etimologia del termine e l'indistinguibile simbolo hanno origini storiche. La compagnia svedese Ericson nel 1994 decise di dare il nome di un antico re vichingo a questa nuova tecnologia. Si tratta di Harald Bluetooth, vissuto tra il 940 ed il 981 D.C. e riconosciuto per il suo successo nell'impresa di unificare sotto un unico impero due paesi all'epoca molto diversi tra loro come la Danimarca e la Norvegia. Così come vennero uniti due mondi differenti, Ericson riuscì nell'impresa di unire senza fili due device differenti, creado una comunicazione ad onde radio a basso raggio (massimo 10 metri) ma ad ottime prestazioni in termini di consumo energetico dei due dispositivi. Il basso consumo è uno dei motivi principali per cui spesso si sceglie questo standard in qualsiasi contesto riguardante l'IoT. Non poteva che aderirvi anche lo Smartwatch, che generalmente ha una durata della batteria molto limitata per via delle piccole dimensioni. Negli anni anche il protocollo stesso è stato migliorato dando alla luce *Bluetoothh Low Energy* (anche chiamato *Bluetooth Smart* o *BLE*). La nuova versione richiede un utilizzo energetico ancora più ridotto, garantendo a embedded devices e quindi anche agli Smartwatch una vita più lunga.

Capitolo 2

Background Tecnologico e Metodologico

In questo viene fatta un'analisi delle principali metodologie utilizzate per lo sviluppo di applicativi Software in ambito aziendale, con alcuni richiami all'esperienza di tirocinio avvenuta in Mango Mobile Solutions.

2.1 Android Wear

2.1.1 Introduzione

In questo paragrafo verranno elencate alcune delle principali caratteristiche del Sistema Operativo Android Wear con particolare riferimento al framework di supporto sviluppato da Google e rilasciato nell'apposita sezione del sito *Android Developers*. Verranno anche discussi i metodi di comunicazione con lo Smartphone e gli scenari futuri nei quali l'azienda si sta direzionando.

Android Wear è una versione del Sistema Operativo Android sviluppata interamente col fine di essere eseguito sugli Smartwatch. Una prima preview per gli sviluppatori è stata introdotta il 18 Marzo 2014, pochi mesi più tardi Google annuncia la versione ufficiale durante l'evento *Google I/O* [13].

Molte aziende del settore tra le quali Samsung, LG, Asus, HTC e Motorola si sono presentate come partner per la diffusione dei nuovi Smartwatch. Accoppiando tramite Bluetooth uno Smartwatch Android Wear con uno Smartphone di versione 4.3 o successiva si ha accesso a molte funzionalità che aumentano l'esperienza utente, tra cui l'utilizzo (anche tramite comando vocale Google Now) di applicazioni scaricabili da Google Play Store direttamente su *watch*. E' supportato anche l'accoppiamento ad una versione iOS 8.2 o successiva ma con funzionalità molto limitate [14].

Figura 2.1: Immagine esplicativa raffigurante il funzionamento del processo di installazione

Anche se molti device hanno integrato il modulo Wi-Fi, nella maggior parte dei casi Android Wear non è connesso ad internet direttamente ma fa riferimento allo Smartphone, il quale si preoccupa di compiere operazioni di rete come recuperare i dati da un Web Service. Come si può evincere dall'immagine sopra riportata, in Android Wear 1.x l'utente scarica ed installa sul proprio device l'App dal Google Play Store. Il Sistema sarà poi in grado di riconoscere l' *Embedded Wearable APK* e quindi copiarlo ed installarlo in Android Wear. Nel caso in cui un'App riceva un aggiornamento il procedimento si ripete. Allo stato attuale l'App su Wearable non esiste senza un apk di supporto installato sul telefono. Di seguito vengono elencate le principali features che hanno caratterizzato la prima versione del framework per lo sviluppo su Android Wear.

2.1.2 Notification

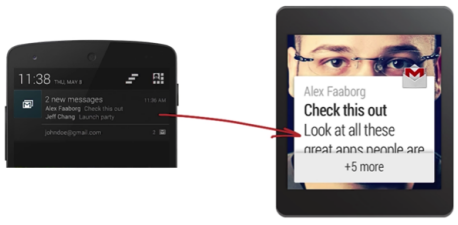


Figura 2.2: Immagine di una tipica notifica in arrivo su sistema Android Wear

Se l'App sullo Smartphone implementa le Notifiche, esse saranno automaticamente spedite tramite bluetooth al Wearable e mostrate tramite *Card* (uno dei componenti grafici principali di Android Wear), non trascurando eventuali azioni associate alla notifica stessa per regalare la migliore esperienza utente possibile.

Più in generale possiamo affermare che la maggior parte delle notifiche visualizzate sul telefono viene mostrata anche sull'orologio. Sono incluse quelle

relative a chiamate senza risposta, messaggi di testo, promemoria di eventi e altro ancora. Quando viene ricevuta una notifica l'orologio emette soltanto una vibrazione. Con poche linee di codice è possibile anche aggiungere eventuali altre azioni visualizzabili solo sullo Smartwatch, come ad esempio l'utilizzo del comando vocale per la risposta alle notifiche. Per questo scopo si fa riferimento alla classe `WearableExtender`.

Android Wear 1.x permette anche la personalizzazione grafica delle Notifiche tramite codice. Una delle caratteristiche è infatti la possibilità di sfruttare lo spazio nella parte più alta dello schermo per espandere una Card e visualizzare tutto il testo. Questo styling è gestito dalla classe `BigTextStyle`. Lo sviluppatore può anche desiderare che vengano presentate delle informazioni aggiuntive quando viene effettuato lo swipe verso sinistra della notifica, tra cui anche la possibilità di inserire un input vocale che verrà rispedito allo Smartphone. Il caso d'uso tipico della risposta vocale ad una notifica è quello della chat: arriva un messaggio, tramite la prima view può essere visualizzato, facendo swipe verso una seconda view è possibile avviare *Google Now* o presentare dei messaggi testuali di risposta veloce definiti dallo sviluppatore. Per aiutare nella creazione di view multiple viene introdotto il concetto di *Pages*, implementabile tramite alcuni metodi della classe `NotificationCompat.Builder`. Ogni extra Page aggiunta non verrà mai visualizzata sullo Smartphone. Se arrivano più notifiche dalla stessa applicazione è possibile raggrupparle tutte per risparmiare spazio e visualizzarle soltanto nel momento in cui viene fatto tap sul gruppo. Questa feature è chiamata *Stack* ed è facilmente implementabile facendo uso di alcuni metodi della classe NotificationCompat.Builder. Se richiesto, la notifica può essere visualizzata soltanto su Smartwatch o soltanto su Smartphone. Questo è possibile tramite il metodo `setLocalOnly` di NotificationCompat.Builder.

```
// Create a WearableExtender to add functionality for wearables
NotificationCompat.WearableExtender wearableExtender =
        new NotificationCompat.WearableExtender()
        .setHintHideIcon(true)
        .setBackground(mBitmap);

// Create a NotificationCompat.Builder to build a standard
    notification
// then extend it with the WearableExtender
Notification notif = new NotificationCompat.Builder(mContext)
        .setContentTitle("New mail from " + sender)
        .setContentText(subject)
        .setSmallIcon(R.drawable.new_mail)
        .extend(wearableExtender)
```

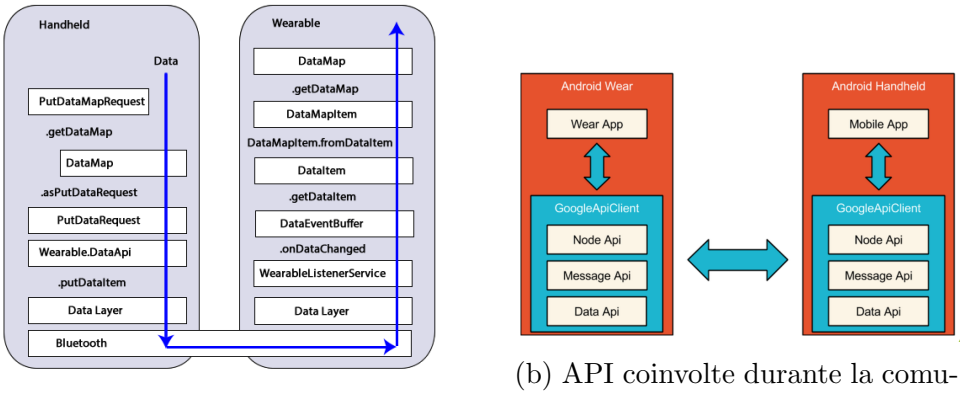
```
        .build();

[...]
NotificationCompat.WearableExtender wearableExtender =
        new NotificationCompat.WearableExtender(notif);
boolean hintHideIcon = wearableExtender.getHintHideIcon();
```

Listato 2.1: Esempio di creazione e personalizzazione di una notifica per Wear

2.1.3 Watch Faces

Una Watch Face è ciò che l'utente visualizza quando nessun'altra App è attiva. E' possibile creare delle Watch Face personalizzate, che possono presentarsi con un design colorato, sfondi dinamici, animazioni e integrazione dei dati. Esempi tipici di dati presentati sono la temperatura esterna, i passi o i Km effettuati nell'arco della giornata, le calorie bruciate e così via. I due concetti chiave da tenere in considerazione nelle Watch Faces sono *Interaction* e *Ambient Mode*. Quest'ultima modalità viene attivata quando l'orologio non è in uno stato di movimento o quando l'utente non vi interagisce. Si vuole quindi massimizzare il risparmio energetico rimuovendo eventuali animazioni, scegliendo un set di colori scuri e riducendo i la frequenza di sincronizzazione dei dati e di aggiornamento della View mostrante l'ora. Interaction mode viene invece attivata quando l'utente presta attenzione all'orologio o quando riceve una notifica, o a fronte di un movimento dello stesso. In questo caso vengono renderizzati colori e animazioni, i dati aggiornati e viene aumentata la frequenza di aggiornamento. Essendo sempre attive, le Watch Faces sono implementate come *Android Services*; il Service va quindi dichiarato nel Manifest prima di poterne estendere le funzionalità. Per implementare una Watch Face occorre estendere le classi `CanvasWatchFaceService` e `CanvasWatchFaceService.Engine` , incluse nel `Wearable Support Library` (libreria principale di Android Wear). Una volta estese le classi sopra citate va effettuato l'*Override* di vari metodi; i più importanti sono `onCreate` e `onDraw`. I nomi sono già di per sé esplicativi: onCreate serve per istanziare i vari oggetti che andranno a costituire la Watch Face, come ad esempio un `Bitmap` per lo sfondo o le animazioni. Successivamente il Sistema invoca onDraw, nel quale avviene il rendering grafico degli oggetti instanziati in precedenza. E' in questo metodo che vanno opportunamente gestiti i comportamenti delle due modalità, al fine di consentire un buon risparmio energetico del device.

2.1.4 Scambio Dati e Sincronizzazione



(a) Scambio DataItem

(b) API coinvolte durante la comunicazione

Figura 2.3: Figure rappresentanti lo scambio di dati tramite Data Layer API

La libreria *Wearable Data Layer API*, tramite un meccanismo ad eventi, fornisce un canale di comunicazione tra telefono ed indossabile.

Il framework mette a disposizione dello sviluppatore due principali meccanismi per spedire i dati dall'App al Watch e viceversa.

- `DataItem` (DataAPI): meccanismo automatico di sincronizzazione dei dati. Anche se il device è disconnesso, il framework garantisce che i dati siano correttamente inviati una volta avvenuto un nuovo accoppiamento tra i due dispositivi. Questo livello di affidabilità è completamente gestito da Data Layer API. Tramite Bluetooth viene spedita una parte di *Payload* e uno specifico *Path* rappresentante la destinazione. Il Payload è costituito da un array di byte nei quali si può settare qualunque tipo di dato, ma ha una limitazione di 100KB. Il Path è una stringa univoca utilizzata sia per identificare il device destinatario sia per riconoscere il dato spedito nel caso si vogliano spedire dati di natura diversa. Un esempio di URI può essere "`/path/node-id/data`". Per evitare di dover lavorare con dati grezzi come array di byte, viene a supporto un altro oggetto chiamato `DataMap`, il quale espone il Data Item tramite un semplice set di coppie chiave-valore.

```
 // Sender
  private void increaseCounter() {
      PutDataMapRequest putDataMapReq =
          PutDataMapRequest.create("/count");
      putDataMapReq.getDataMap().putInt(COUNT_KEY, count++);
      PutDataRequest putDataReq =
          putDataMapReq.asPutDataRequest();
      PendingResult<DataApi.DataItemResult> pendingResult =
              Wearable.DataApi.putDataItem(mGoogleApiClient,
                  putDataReq);
  }

[...]

  //Receiver
  @Override
    public void onDataChanged(DataEventBuffer dataEvents) {
        for (DataEvent event : dataEvents) {
            if (event.getType() == DataEvent.TYPE_CHANGED) {
                // DataItem changed
                DataItem item = event.getDataItem();
                if (item.getUri().getPath().compareTo("/count")
                    == 0) {
                    DataMap dataMap =
                        DataMapItem.fromDataItem(item)
                                                .getDataMap();
                    updateCount(dataMap.getInt(COUNT_KEY));
                }
            } else if (event.getType() ==
                DataEvent.TYPE_DELETED) {
                // DataItem deleted
            }
        }
    }
```

Listato 2.2: Esempio di utilizzo dell'oggetto DataItem

Come si può evincere dal codice riportato sopra, tramite alcune classi di supporto fornite dall'API è possibile specificare il Path con il quale identificare il dato da spedire e quindi creare un *wrapper* dell'oggetto DataMap. A questo punto il DataItem è pronto per essere spedito.

Il ricevente estrae il DataMap dal DataItem ricevuto e tramite l'opportuna chiave (che deve essere la stessa usata dal mittente) estrae il dato

ricevuto e lo usa per i propri scopi.

La particolarità di questo meccanismo risiede nel fatto che il Sistema riesce a riconoscere lo stato dei dati presenti in uno specifico Path. Nel caso in cui non sia stata apportata alcuna modifica rispetto alla spedizione precedente, i dati non vengono inviati al destinatario. Google ha deciso di utilizzare questo approccio per consentire di limitare il più possibile le operazioni di Rete al fine di favorire una maggiore durata della batteria di entrambi i dispositivi.

Per inviare dati binari come immagini attraverso Bluetooth, al Data Item può essere agganciato un oggetto chiamato `Asset`. Gli Asset gestiscono automaticamente il *caching* dei dati per prevenire la ritrasmissione e conservare la banda Bluetooth. Come già accennato un DataItem è limitato a 100KB, tuttavia gli Asset possono essere grandi a piacere in quanto i byte vengono automaticamente suddivisi e spediti man mano. E' senza dubbio sconsigliato inviare immagini di grandi dimensioni, potrebbero avere un impatto negativo sulla *user experience* e sulla memoria dell'indossabile.

- `Messages` (MessagesAPI): sono semplici Array di Byte e non essendoci un meccanismo di sync, in questo caso non vi è garanzia alcuna che i dati arrivino al destinatario: se è disconnesso o distante, il dato andrà semplicemente perso. I Messages sono pensati per una comunicazione *one-way*, utile in scenari di RCP (Remote Procedure Calls), dove si vuole chiedere all'interlocutore remoto di compiere determinate azioni. Anche i Messages sono costituiti da Payload e Path; in questo caso però il Payload è arbitrario ma va gestito manualmente dallo sviluppatore. Sebbene non vi sia una regola ben stabilita, solitamente i DataItem vengono utilizzati per spedire grandi quantità di dati in maniera affidabile, mentre i Messages vengono usati per notificare, lanciare Activity o chiedere all'altro device di performare determinati task (es. richiesta di dati). Di seguito vengono riportati alcuni scenari di casi d'uso tipici.

	Data Items	Messages
Notifica su wearable quando un ristorante è nelle vicinanze		X
Configurazione del colore di background di una WatchFace da telefono	X	
Il Wearable registra i dati GPS e li trasmette al telefono	X	
Il telefono vuole mostrare informazioni sulla strada in cui si trova direttamente su wearable		X
Il telefono vuole spedire una notifica al wearable riguardo un importante task che deve effettuare l'utente	X	

In Android Wear, multipli Smartwatch possono essere connessi al telefono dell'utente. Ogni device connesso nella Rete è considerato un "Nodo". Dati i multipli dispositivi connessi occorre identificare a quale spedire i messaggi. Ad esempio, una Wearable App che registra la voce dovrà inviare il dato al device con più potenza di calcolo e capacità a livello di batteria (lo smartphone), dovrà quindi riconoscerlo all'interno della Rete. Per questo motivo viene in aiuto la libreria *NodeAPI*, anch'essa facente parte del Data Layer API. Tramite poche semplici righe di codice è possibile ricevere una lista di tutti i device connessi, quindi scegliere il Nodo più appropriato.

```
NodeApi.GetConnectedNodesResult result =
    Wearable.NodeApi.getConnectedNodes(mGoogleApiClient).await();
List<Node> nodes = result.getNodes();
for (Node node : nodes) {
    if (node.isNearby()) {
        nodeId = node.getId();
        //send Message using MessageApi
        Wearable.MessageApi.sendMessage(mGoogleApiClient, nodeId,
            ITEM_PATH, itemMenuId.toString().getBytes()).await();
        if (!result.getStatus().isSuccess()) {
            Log.e(TAG, "Error sending request message");
        } else {
            Log.i(TAG, "Success! Message sent to: " +
                node.getDisplayName());
        }
    }
}
```

Listato 2.3: Esempio di richiesta di una lista di Nodes presenti nella rete

Per notificare una parte o l'altra della ricezione di un Message o di un DataItem il framework fa uso di un'architettura ad eventi: l'orologio (come anche il telefono) riceve i dati tramite un listener che cambia di stato non appena gli viene notificata la presenza di dati. In entrambi le parti occorre estendere un Android Service chiamato `WearableListenerService` ed eseguire l'Override di due metodi in particolare: `onDataChanged()`, che viene chiamato alla ricezione di un nuovo DataItem, e `onMessageReceived()`, che viene chiamato alla ricezione di un Messagge.

2.1.5 Best Practice

Di seguito vengono riportati dei casi tipici di corretta ingegnerizzazione del Software in ambito Android Wear, vale a dire tutti quegli scenari in cui occorre fare attenzione per evitare problemi di performance o consumi anomali della batteria di entrambi i device.

- Invio dei dati: poniamo l'esempio in cui stiamo implementando una Weather App dove vengono mostrate le temperature. Dovranno essere spedire allo Smartwatch più temperature e quindi si farà riferimento a più DataItem separatamente. Ma ogni DataItem ha un suo header aggiunto dalla Rete e dalla piattaforma stessa; è quindi importante salvare tutti i valori all'interno di un singolo messaggio (che avrà un solo header) per spedire meno dati. Nel caso di invio tramite DataItem, grazie al framework verrà garantito un *acknowledgement* della ricezione su device. Nel caso di Messages, occorre controllare manualmente, soprattutto perché in questo caso la trasmissione non è mai garantita.

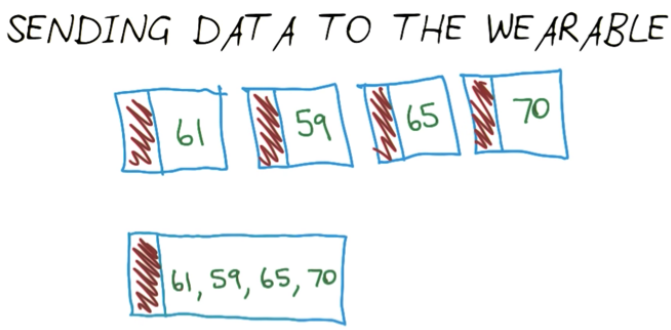


Figura 2.4: Immagine raffigurante il raggruppamento dell'header in un unico messaggio

- CPU Balancy: Avendo il telefono un processore molto più potente e una durata della batteria maggiore, è importante lasciare il Wearable il più possibile in uno stato idle (quindi anche entrare in Ambient Mode) e implementare facendo sempre attenzione a dove vengono distribuite le operazioni di calcolo che richiedono un ingente consumo della CPU.

- Networking: Android Wear non permette l'accesso diretto ad Internet, quindi è il telefono ad occuparsi di fare chiamate ad un eventuale Web Service. Successivamente è buona pratica che il telefono si preoccupi di non spedire al device l'intero *JSON* ricevuto ma soltanto le informazioni veramente rilevanti. Le ragioni principali di questa scelta architetturale sono le seguenti:

1. Durata Batteria: il risparmio energetico è importante. Uno Smartwatch di fascia media che non utilizza sensori se non il modulo Bluetooth per connettersi al telefono ha una durata media che varia dalle 24 alle 36 ore.
2. Affidabilità: Il Wearable potrebbe non essere sempre nei pressi del telefono quindi non può essere garantita una connessione ad Internet costante.
3. Connectivity: il device potrebbe connettersi al telefono in maniere differenti, Android Wear SDK viene in aiuto per astrarre questo concetto e creare un layer aggiuntivo che permetta una comunicazione efficace.

2.1.6 Uno sguardo al futuro

Una prova che Google sta puntando molto sul Wearable Computing è l'introduzione della nuova versione del Sistema Operativo Android Wear 2.0, la quale data di rilascio è prevista entro la fine del 2017. Sarà il primo grande aggiornamento a partire dal lancio del sistema nel 2014. Attualmente è presente la versione Preview disponibile agli sviluppatori. La novità principale che verrà introdotta in Android Wear 2.0 risiede nell'autonomia degli Smartwatch rispetto agli Smartphone. Nasce il concetto di Standalone Apps anche su Wear: non vi sarà più bisogno di uno Smartphone di supporto, si avrà a disposizione un *Wear Play Store* dedicato dal quale sarà possibile scaricare ed installare le Wearable App. Allo stato attuale l'utente è costretto ad utilizzare spazio inutile sul proprio telefono se vuole avere installata un'App sul proprio orologio; in futuro questo limite sarà superato. L'accesso alla Rete da parte di Android Wear 2.0 non sarà più demandato al Wearable Data Layer API, ma ogni Android Wear App potrà gestire le proprie richieste di Rete autonomamente. Un esempio è quello di *Google Cloud Messaging*, usato per l'invio delle notifiche push: grazie alla separazione dei due Sistemi, sarà possibile scegliere su quale device inviare le notifiche. Lo sviluppatore potrà quindi anche pubblicare una *Watch-only App* sullo Store dedicato, il quale sarà integrato direttamente nel device. Questa esigenza è nata dalla community di sviluppatori al fine di dare agli utenti un modo più semplice e immediato di scoprire le nuove App offerte dal mercato. Il tentativo di separare i due mondi è una chiara e graduale intenzione da parte dei produttori di ridurre al minimo l'uso dello Smartphone a fronte di un indossabile di più immediata consultazione. Un altro dei motivi che ha portato a spostare sullo Smartwatch la possibilità di effettuare richieste di Rete è la maggiore compatibilità con i telefoni con Sistema Operativo iOS: lo sviluppatore di sistemi Apple non deve preoccuparsi

di studiare un framework di appoggio da integrare nella propria iPhone App; sarà il Web Server stesso a gestire la comunicazione e a spedire i dati necessari.

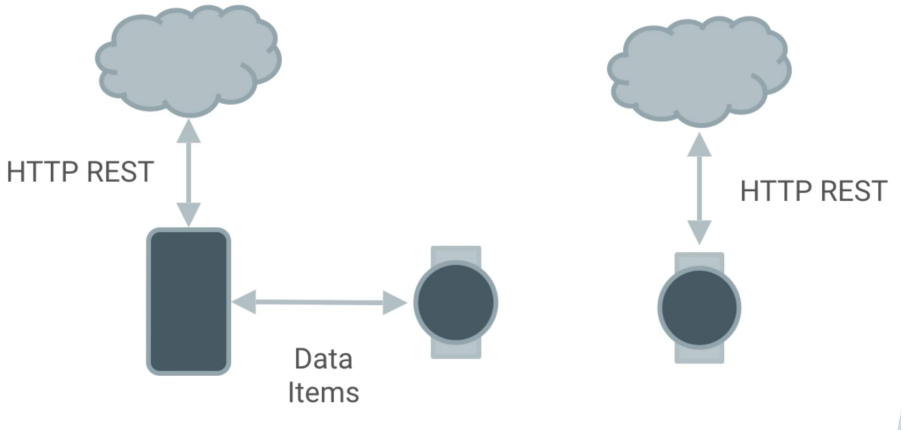


Figura 2.5: Immagine dell' architettura di Rete in Android Wear 2.0

Tra le novità spicca anche una GUI migliorata e una più facile gestione soprattutto per quanto concerne gli Smartwatch di forma tonda. In particolare verrà introdotta una piccola tastiera virtuale utilizzabile direttamente dallo schermo del Watch oltre alla scrittura a mano libera e ai messaggi preimpostati. Le icone delle notifiche saranno più piccole rispetto alla versione precedente, lasciando lo spazio per una Progress Bar nella parte inferiore dello schermo che mostra quante altre Card sono presenti nello stack. Le notifiche sono state ridisegnate per favorire una maggiore durata della batteria, fornendo un testo chiaro su uno schermo scuro. Rivisitati anche i Navigation Drawer e gli Action Drawer, due componenti grafici destinati a semplificare la navigazione tra le viste [15].

2.2 Metodi Agili in azienda e Tool di supporto

2.2.1 Introduzione

A giorno d'oggi la maggior parte delle aziende nel settore IT fa uso di metodi Agili per fornire il prodotto al cliente. L'essenza stessa di questo nuovo approccio porta a volersi focalizzare sulla distribuzione del proprio prodotto al cliente nei tempi stabiliti. E' inevitabile che per tentare di rispettare i brevi tempi di consegna si tende a dare molto più spazio alla fase di implementazione: pochissimo tempo è dedicato alla fase di progettazione e di testing. Non saranno in questa sede elencate le buone caratteristiche di un'architettura, si

vuole altresì dare un'impronta più orientata ai problemi e le sfide quotidiane ai quali gli ingegneri incorrono durante lo sviluppo dei sistemi Software in ambito aziendale e ai quali io stesso mi sono imbattuto durante la mia esperienza di tirocinio in azienda.

L'esperienza di 4 mesi che ho avuto in Mango mi ha portato a capire che spendere anche solo qualche ora sulla stesura dell'architettura del sistema che si intende sviluppare risulta essere utile in futuro. Risulta utile anche in contesti Agili, in quanto garantisce un risparmio non indifferente di tempo in fase di rilascio di nuove feature. E' stato notato che usare un approccio ben definito porta ad un codice più pulito, leggibile e comprensibile (si evitano i cosiddetti God Objects, Classi con migliaia di righe di codice), una maggiore flessibilità nel caso si voglia cambiare qualche componente o libreria. Va citata anche la necessità di modellare avendo uno sguardo sempre rivolto ai cambiamenti futuri, cioè su come quella determinata scelta architetturale potrebbe impattare sulle decisioni che si prenderanno in base al cambiamento delle tecnologie. Pertanto si cerca sempre di dividere a seconda dei compiti che ogni componente deve svolgere, di generalizzare e di scegliere dei nomi appropriati per le interfacce (spesso generici, non relativi alla tecnologia usata), in maniera da lasciare la possibilità in futuro di rimodellare soltanto ciò che le nuove tecnologie richiedono che sia cambiato, ma mantenendo comunque intatta la struttura, lo scheletro del sistema. Il tentativo costante è quello di aderire ad un'architettura che renda il codice facilmente riusabile e testabile, che eviti duplicazioni di parti, che sia flessibile e facilmente comprensibile dai nuovi addetti ai lavori, quindi adatto al contesto aziendale moderno.

2.2.2 Sviluppare Software di qualità con Android

Le metodologie di sviluppo, i pattern e le architetture utilizzate in Mango sono risultate molto utili per facilitare lo sviluppo sulla piattaforma Android e quindi anche Android Wear. Ogni Android Engineer è consapevole del fatto che realizzare un progetto può risultare complesso, specie se di grande portata. La complessità deriva soprattutto dalla richiesta da parte del cliente, che spesso chiede di avere un sistema sempre più articolato e pieno di feature. Di seguito sono riportate alcune caratteristiche comuni che rendono una generica applicazione Android non facilmente gestibile e anzi, a volte molto complessa:

- Retro-compatibilità: permettere che sia installabile ed utilizzabile dalla maggioranza dei device presenti in commercio. Questo è già di per sé una sfida in quanto ogni telefono ha moduli hardware differenti (ad esempio, un produttore potrebbe decidere di non utilizzare NFC), inoltre ogni versione di Android ha requisiti e permessi diversi che spesso devono essere controllati a runtime per prevenire malfunzionamenti.

- Gestione interfaccia utente e Multithreading: come in qualsiasi altro contesto applicativo moderno, anche Android richiede che l'esperienza utente sia ben gestita attraverso l'uso di Thread o Task separati.

- Uso di risorse: aspetto molto importante in ambito Mobile e quindi anche Wear. Anche se il mercato sta offrendo device con potenza computazionale sempre più vicina a quella di una macchina desktop, quando si scrive codice occorre comunque tenere in considerazione le performance, l'utilizzo della memoria, della CPU e della banda a disposizione. Una gestione non attenta di queste può portare ad un drastico utilizzo della batteria e dei dati internet a disposizione, limitando così la soddisfazione dell'utente.

- Salvataggio dati in locale, sincronizzazione dei dati con il server remoto: ad oggi quasi tutte le applicazioni hanno bisogno di comunicare con un Web Service, quindi devono includere nella propria logica applicativa una sezione completamente dedicata allo scambio di dati in formato JSON o XML tramite protocollo HTTPRest. Inoltre spesso è necessario anche implementare un meccanismo di sincronizzazione dei dati presenti nel Server remoto con un database locale, utile per prevenire che vi siano troppe richieste di rete, che possono causare un drenaggio della batteria ma anche perché capita che la Rete non sia disponibile: se non ci fossero i dati salvati in un local db, in caso di connettività limitata o assente l'App diverrebbe inutilizzabile.

- UI accattivante: oltre che tenere d'occhio tutto il resto, anche la grafica e l'esperienza utente deve essere uno dei punti di forza. Ogni applicazione dovrebbe attenersi alle regole grafiche imposte dal Material Design di Google, fare attenzione alle animazioni, transazioni, colori, dimensioni ecc. Il tutto deve essere fatto non trascurando le performance del dispositivo.

- Utilizzo dell'hardware: alcune App si trovano a dover gestire nel migliore dei modi i propri componenti hardware (fotocamera, oscilloscopio, accelerometro, ecc) ma anche interagire con sensori posti nell'ambiente circostante, come ad esempio gli iBeacon.

Inoltre, un qualsiasi progetto su piattaforma Android (che sia Mobile, Wear, TV, Car o Thing) fa uso dell'SDK fornito da Google, il quale ha incorporati alcuni componenti che sono fondamentali alla costruzione e al funzionamento del sistema. Alcuni esempi di questi sono:

- Activities, Services, Broadcast Receivers, Content Providers, Bundles, Intents

- Fragments, Views, Notifications, Resources
- Databases, IPC, Threads, Storage
- Librerie di supporto di Google o di terze parti
- APIs esterne per fornire un supporto allo sviluppo dell'interfaccia grafica, dei sensori hardware, delle comunicazioni di Rete, ecc [16].

Come in qualunque altro Sistema o ambiente, la complessità nello sviluppo aumenta se si decide di iniziare a sviluppare con un approccio senza una regola ben definita, e quindi senza seguire un pattern architetturale di riferimento.

E' indubbio che l'utilizzo dei componenti sopra elencati è molto forte all'interno di un'Android App; spesso lo sviluppatore si trova quindi ad inserire la propria logica applicativa all'interno dei componenti stessi, generando il più delle volte confusione. Date le circostanze quindi, se ogni parte fosse ben distinta, l'intero sistema avrebbe un impatto positivo. Per limitare la complessità, la domanda da porsi in ambito mobile (data l'esistenza di vari componenti) è come separare le parti, ovvero definire dove inserire la business logic dell'applicazione e dove apporre tutto il resto. Nel mio caso mi sono trovato a dover comprendere l'intero funzionamento del progetto Android lato handheld, il quale, non a caso, ingloba tutti i componenti e le feature sopra elencate. La difficoltà di apprendimento è stata molto bassa: alcuni schemi C4 di documentazione mi sono stati forniti come ausilio; inoltre, grazie alla Viper Architecture, ogni componente era ben suddiviso e quindi di facile comprensione. In questa sezione ho dovuto aggiungere un nuovo package, le quali classi si occupano di preparare in maniera opportuna i dati da inviare allo Smartwatch: questa operazione è stata molto veloce: grazie ad una buona progettazione non è stato necessario rifattorizzare altre classi esistenti, ma solo aggiungerne di nuove al bisogno. Di conseguenza, anche lo sviluppo su Android Wear è risultato più semplice.

2.2.3 Scrum

A partire dai primi anni '90, per gestire complessi processi di sviluppo Software molte aziende del settore fanno uso di Scrum, definito dai fondatori Ken Schwaber e Jeff Sutherland come un framework basato sui pilastri del manifesto Agile che ha l'intento di garantire, attraverso un approccio basato sull'esperienza, la massima produttività e creatività nel rilascio dei prodotti. Ogni attività del processo Scrum ruota attorno al concetto di Sprint, un arco di tempo che va da 1 a 4 settimane nel quale ci si focalizza sul rilascio di una story (una singola funzionalità) del prodotto da poter presentare al cliente

per promuovere la collaborazione [17]. Uno Scrum Team è un team multifunzionale che non ha bisogno di aiuti esterni per raggiungere un determinato obiettivo. I ruoli all'interno di Scrum sono suddivisi in Product Owner, Team di sviluppo e Scrum Master. Il product Owner rappresenta gli interessi del cliente e si occupa di definire e gestire le singole feature del prodotto da sviluppare nonché di massimizzarne il valore rendendo più efficiente possibile il lavoro del team di Sviluppo. Conosce quindi tutti i dettagli delle funzionalità richieste dal cliente, le gestisce e fa del suo meglio per far sì che il resto del Team raggiunga l'obiettivo di svilupparle entro i tempi stabiliti. Spesso nelle piccole realtà aziendali il Product Owner si occupa anche di svolgere la funzione di Scrum Master, definito come una sorta di "guardiano" di tutti i processi Scrum. Questa figura gestisce i meeting con gli altri collaboratori e tutte le varie fasi legate al framework. Il Development Team si occupa di portare a termine (marcare come "Done") i vari task di ogni Sprint e quindi realizzare il prodotto finito. Per garantire efficienza e diminuire problemi di organizzazione non si dovrebbe andare sotto un minimo di 3 e un massimo di 9 persone coinvolte [18]. Il framework mette a disposizione vari momenti di incontro del Team. Tali eventi costituiscono una parte fondamentale dell'intero processo poiché portano ad un netto risparmio di tempo e aiutano a prevenire eventuali situazioni di disagio e di allarme. Anche se si è portati a pensare che il dialogo, soprattutto in presenza di scadenze imminenti, porti via tempo prezioso allo sviluppo, in realtà aumenta di molto la produttività e le scelte future, indispensabili per il corretto andamento del prodotto. Viene riportata in seguito una descrizione cronologica di ogni evento Scrum:

- Sprint Planning: In questa fase viene definito il lavoro che deve essere fatto dal Team di sviluppo durante lo Sprint. Viene preso in considerazione il Backlog e i suoi punti focali e le feature divise in vari task, ai quali vengono assegnate delle ore di lavoro. Il Backlog è una lista dinamica di requisiti, funzioni e miglioramenti concordata tra Product Owner e cliente. Viene fatto anche un design del sistema, necessario a convertire gli elementi del Backlog in parti di prodotto funzionanti.

- Sviluppo: Parte dedicata allo sviluppo delle funzionalità concordate nella fase iniziale dello Sprint a cui vengono aggiunte ore di "demo", review del codice, QA (quality assurance), correzione di bug e testing.

- QA: è una delle parti dello sviluppo ma data la sua importanza, merita una descrizione. Un membro del team è addetto a leggere la descrizione delle funzionalità e verificare sui device che siano state effettivamente implementate. Solitamente vengono scelti dei giorni prestabiliti per accertare la qualità del prodotto. Nei grandi contesti aziendali viene assunto

un addetto il quale unico compito è fare QA; nelle PMI come Mango invece, solitamente è il Product Owner che fa le veci del "QAyer".

- Sprint Review: evento di breve durata tenuto a cavallo tra la fine di una Sprint e l'inizio dello Sprint Planning. In questa fase si descrive cosa è stato fatto (gli elementi in "Done") durante la Sprint, cosa è andato o non è andato a buon fine. Lo Scrum Master si preoccupa di mantenere un aperto dialogo su come è possibile migliorarsi in ogni aspetto, al fine di non ripetere gli stessi errori nella Sprint successiva. Per far ciò ad ogni membro del Team viene chiesto di stilare una lista di "Pro", di "Contro" e di "Action" (azioni da compiere per migliorare i contro) nonché di rilasciare un voto che solitamente va da 1 a 5. Viene poi appuntata una media per eccesso dei voti di ognuno.

- Sprint Retrospective: oltre che assegnare dei voti sul lavoro svolto e parlare dei possibili miglioramenti, prima di iniziare con lo Sprint Planning, il Team è chiamato a discutere apertamente di eventuali dubbi e pensieri sul livello relazionale e di processo. E' una parte più psicologica rispetto alle precedenti, dedicata all'individuo e finalizzata al raggiungimento di un'esperienza di lavoro più efficace e costruttiva. Nel caso del tirocinio al quale ho preso parte, i voti andavano da 1 a 5 e insieme a molte altre informazioni vengono scritti all'interno di una scheda di Trello.

2.2.4 Trello

In questa sezione verrà descritto il software *Trello* con particolare riferimento all'esperienza diretta di tirocinio. Trello è uno dei software di supporto al metodo Scrum esistenti in commercio. E' molto utile per avere una visione chiara dello stato di avanzamento di ogni task assegnato per il conseguimento di una Sprint. Durante la fase di Sprint Planning ogni componente del Team di Sviluppo fa una stima delle ore di cui ha bisogno (tra quelle a disposizione) per lo sviluppo di una sottoparte della Story, in Trello vengono appuntate le ore totali e le ore effettive di lavoro. Nel caso di Mango, una volta effettuata la pianificazione delle ore, ogni task è spostato nella colonna "TODO"; quando un membro del Team inizia lo sviluppo di un task a lui assegnato, lo sposta in "IN PROGRESS"; una volta terminato si sposta in "QA". Se viene terminato in un tempo maggiore o inferiore rispetto a quello inizialmente concordato, si sposta comunque in QA lasciando invariata la pianificazione delle ore e quelle effettivamente sfruttate. E' importante che rimanga tutto nello stato precedente poiché potrà essere discusso e migliorato nella successiva fase di Sprint Review. Una volta che è stato effettuato il *code review* e la QA su quella determinata feature, se ci sono errori da correggere vengono allocate delle ore per la

correzione dei *Bugs*, altrimenti il task viene spostato in "DONE", si effettua un *merge* nel master principale del software *DVCS* utilizzato per la gestione dei repository, e si passa allo sviluppo della feature successiva. Nel caso di Mango ad ogni feature da sviluppare (quindi ad ogni task di Trello) viene associata la creazione di un nuovo *Branch* sulla piattaforma Bitbucket. Il branch avrà lo stesso nome del task di Trello per una migliore identificazione.

Durante il Planning viene anche creata una scheda in Trello chiamata "Agile Activities Sprint", dove vengono definite le ore totali assegnate ad ognuno (l'iniziale di ogni collaboratore viene apposta tra parentesi tonde) e poi divise per ogni progetto. In questa scheda vengono appuntate anche le ore dedicate al Planning stesso nonché quelle per il Code Review e Backlog. Come si può evincere dalla figura, 34 ore totali sono state assegnate allo sviluppatore MB per il progetto in questione. Di queste, 30 sono le ore operative poiché 2 sono state allocate per altre attività legate a Scrum e 2 per il review del codice. Delle 30 ore operative, 4 saranno assegnate per il task "CardList Module", 8 per il "CardDetail Module" e così via. Una volta terminate queste ore di Code Review e Planning, ognuno aggiorna la scheda nella parte destra del titolo. Tale operazione che viene ripetuta per ogni scheda individuale visualizzata nelle varie colonne di Trello.

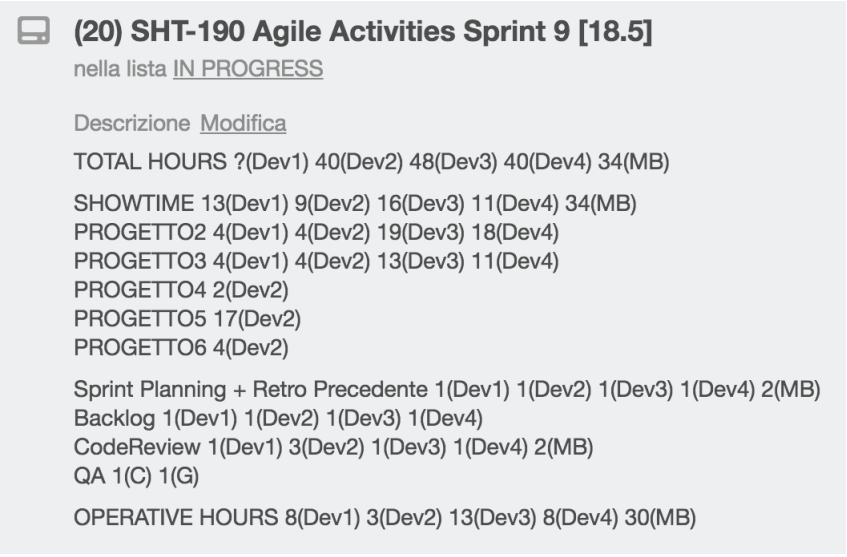


Figura 2.6: Immagine di una Scheda di Trello

2.2.5 Draw.io

Un altro esempio di strumento che ho avuto l'opportunità di sperimentare è *Draw.io*, una Web App molto utile nei contesti Agili che permette di disegnare

schemi e diagrammi di qualsiasi tipo e dove è forte la collaborazione e la condivisione in tempo reale anche a distanza. E' stato principalmente utilizzato nella fase di design di ogni Sprint: in questo modo tutti i membri del Team anche non fisicamente presenti in ufficio possono rimanere al corrente su quali pattern si decide di utilizzare e su come è stata pensata l'architettura delle interfacce all'interno del proprio progetto. Le ore dedicate alla progettazione si sono rivelate un momento di fondamentale importanza e questo strumento, misto ad un linguaggio comune dettato dal modello C4, ne hanno facilitato molto la comprensione e quindi la successiva implementazione.

2.2.6 Fabric.io

Nella fase di collaudo del sistema installato il cliente ha la possibilità di testare ed imparare ad usare il nuovo prodotto e quindi tutte le feature implementate fino a quel momento. Questa fase, che è a cavallo tra lo sviluppo e la messa in esercizio, viene anche detta fase di *beta-testing* ed è molto importante poiché consente ai diretti interessati di riscontrare errori bloccanti o altri malfunzionamenti, evidenziare problemi di operatività o relativi a funzionalità non implementate correttamente. A supporto di questa parte del processo di rilascio del software viene in aiuto Fabric, un tool recentemente acquisito da Google. Fabric offre un meccanismo per automatizzare la distribuzione in beta delle varie App nonché un *crash reporting* dettagliato che consente all'azienda di scovare bug e risolverli in maniera veloce ed efficiente. Il motto di Fabric è "*Understand how your app is doing. In real-time.*" Dallo sviluppo al rilascio, Fabric può portare dei benefici all'azienda che lavora in contesti Agili, aiutandola ad avere una visione chiara di come l'App risponde in termini di performance e di utilizzo da parte di un numero elevato di utenti. In Mango ho potuto notare che l'utilizzo di questo strumento aumenta la facilità di comunicazione e di collaborazione con il cliente, che rimane più soddisfatto proprio perché più coinvolto nel processo di produzione del proprio prodotto.

2.3 C4

In una gestione dei progetti tradizionale il gruppo di lavoro che si trova ad affrontare la fase di progettazione del Sistema attribuisce molto valore alla documentazione e ai diagrammi formali, come ad esempio UML (Unified Modeling Language).

La varietà di skills e background differenti dai quali proviene ogni professionista del Team, nonché l'avvento dei nuovi approcci Agili, hanno portato ad una netta diminuzione dell'uso della documentazione e di un approccio standard alla progettazione a favore di una maggiore libertà espressiva, che abbia

come focus il problema stesso, piuttosto che il modo con cui viene presentato. Le aziende moderne vedono UML come uno strumento potente ma al contempo troppo dettagliato, lungo da creare (i tempi sono importanti in Scrum) e soprattutto instabile: manutenere un livello così dettagliato di documentazione è divenuto pressoché inutile. Quando il Team arriva nella fase di Backlog e deve lavorare sul design di una Story per la nuova Sprint, si trova a dover discutere dei vari problemi legati al Sistema e ad escogitare una soluzione efficace su come risolverli. Ogni membro avrà quindi bisogno di trovare una modalità veloce ed efficace di comunicare agli altri le proprie idee. Da questo contesto nasce la necessità di creare un linguaggio di immediata comprensione, che sia leggero, semplice ma al contempo pragmatico. Per questo motivo nasce C4, un modello avente una descrizione a diagrammi della struttura del Software più ad alto livello. I diagrammi sono essenzialmente costituiti da rettangoli e frecce. Proprio perché il focus non è sulla notazione, entrambi gli elementi possono essere rappresentati a proprio piacimento: per fare un paragone con UML, non vi è alcuna differenza semantica nel disegnare una freccia tratteggiata piuttosto che una freccia continua. Viene introdotto il concetto di astrazione a più livelli, molto intuitivo per ogni essere umano anche non esperto del settore. Si parte da un alto livello di descrizione, che dice che cos'è il Sistema e come esso possa fornire un valore al business. Si va poi sempre più nel dettaglio, "esplodendo" una o più parti del diagramma precedente. Essendo più astratta, la nuova struttura rende la progettazione più stabile nel tempo e permette al lettore di capire in maniera immediata quali siano le parti che compongono il Sistema e come interagiscono. Come riporta il manifesto ufficiale: *"A software system is made up of one or more Containers, each of which contains one or more Components, which in turn are implemented by one or more Classes"* [19]. Da questo prende spunto il nome *C4*, proprio perchè essenzialmente formato da quattro C, ognuna con un significato ben definito sempre più dettagliato:

- Context: come accennato sopra, in questa parte principale viene introdotta l'entità del Sistema Software da progettare, chi sono gli attori coinvolti e chi i costruttori e qual'è l'ambiente circostante in cui esso si vuole posizionare. E' solitamente costituito da un riquadro centrale rappresentante il Sistema e da frecce che si direzionano verso le varie entità in gioco, che possono essere altri sistemi software di appoggio o persone [20].

- Container: mostra una panoramica generale delle scelte tecnologiche e della divisione delle responsabilità. Il sistema è decomposto in vari container e l'enfasi è su come questi container comunicano tra di loro. Ad esempio si può mostrare l'immagine stereotipata di un utente che interagisce sia con "iOS App" che con "Android App"; entrambi i container

interagiscono a loro volta sia con i container “HTTP Server” che “Facebook SDK”. Già da qui si può stilare una divisione dei compiti di ogni sviluppatore, in quanto è già possibile indicare esattamente dove andare a scrivere codice per implementare una determinata feature.

- Components: si sceglie un container e lo si espande, mostrandone la struttura interna. Ci si chiede ad esempio di quali servizi e componenti è costituito quel container. Sopra ogni riquadro generalmente vengono annotati i linguaggi utilizzati e una breve descrizione.

- Class: in alcuni casi, per descrivere la struttura interna di un componente viene in aiuto il tradizionale diagramma delle Classi di UML. Solitamente è necessario arrivare a questa profondità descrittiva quando il componente è particolarmente complesso da comprendere. Vengono disegnate le Interfacce associate al componente e si discute su quale sia la soluzione architetturale (pattern) più elegante da utilizzare. Lo sguardo è sempre rivolto al futuro, si cerca quindi di creare Classi il più possibile indipendenti dalla tecnologia usata, che nel tempo tende a variare.

Per quanto concerne la documentazione i diagrammi vengono disegnati su una lavagna e al termine della progettazione è buona norma scattare una foto e apporla come allegato alla Scheda di Trello dedicata alla fase di Backlog.

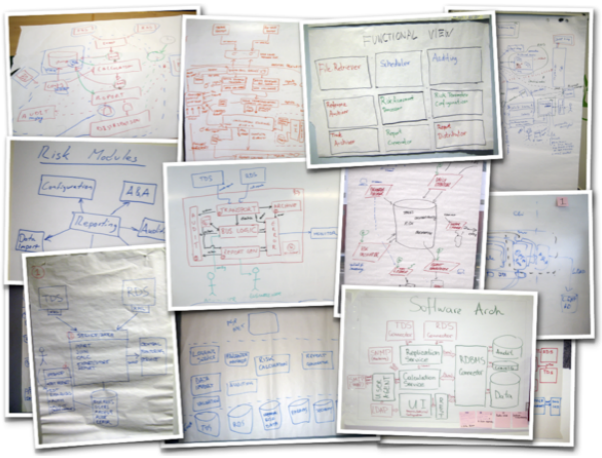


Figura 2.7: Foto di alcuni diagrammi C4

2.4 Clean Architecture

2.4.1 Introduzione

Durante il processo di sviluppo di un Sistema Software più o meno complesso, investire tempo e risorse per definirne l'architettura è senza dubbio un'ottima scelta, anche se si decide di lavorare in contesti Agili. Nel mondo dell'Informatica le tecnologie tendono a subire un cambiamento spesso veloce e radicale. In contemporanea alle nuove scoperte tecnologiche sia nell'ambito dell'Hardware che del Software, inevitabilmente anche le architetture e i pattern utilizzati vengono modificati o a volte totalmente riprogettati per conformarsi ai nuovi standard. Se l'architettura è in linea con i pattern utilizzati in una determinata tecnologia, cresce la garanzia che tutti i principi di buona progettazione vengano rispettati. Un esempio concreto è quello della Clean Architecture, una nuova architettura, comparsa per la prima volta nel 2012 in un sito dedicato, nata in seguito all'avvento del Web, dei dispositivi mobili e dell'Internet of Things, ma che può essere estesa a qualunque contesto anche più classico come una Desktop Application. I professionisti del settore si trovano agevolati nello sviluppare Sistemi anche di grandi dimensioni.

2.4.2 Descrizione

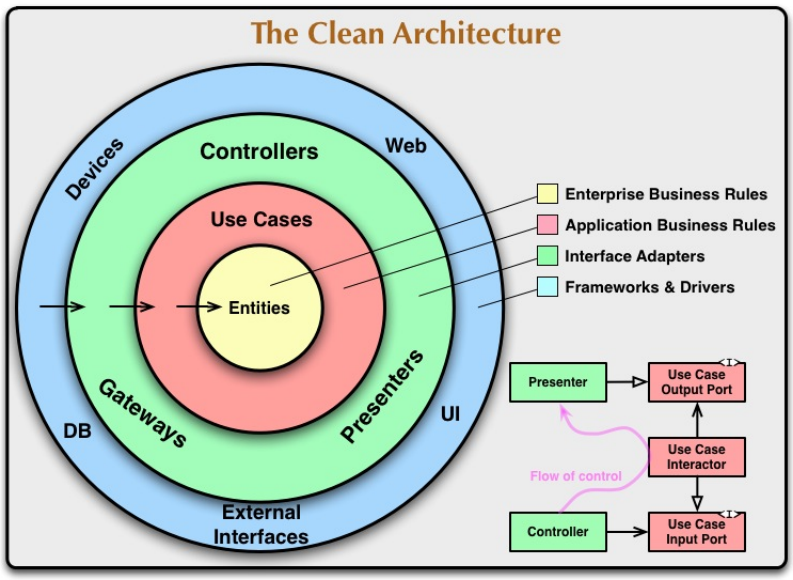


Figura 2.8: Immagine della struttura Clean Architecture

La Clean Architecture ha una struttura *a cipolla*: come si può evincere dalla figura, i cerchi concentrici rappresentano le differenti aree di competenza e più ci si allontana dal centro, più aumenta l'astrazione del software. L'intera struttura della Clean Architecture gira intorno ad un'unica regola, chiamata *The Dependency Rule*, la quale afferma che le dipendenze vanno solo verso l'interno, ovvero le unità più interne non hanno alcuna consapevolezza dell'esistenza di quelle più esterne. In altre parole, tutto ciò che è dichiarato esternamente non può essere menzionato più internamente, proprio perché non ne può essere a conoscenza [21]. Essendo ogni componente Software indipendente dagli altri, il codice risulta molto più riusabile, estremamente testabile, flessibile e di facile gestione, cioè non particolarmente sensibile ai cambiamenti.

Un esempio di componente è quello dei Framework: avere un Sofware indipendente dalle librerie usate costituisce un vantaggio enorme, soprattutto per quanto concerne le scelte future ed i cambiamenti da apportare per qualsiasi motivo (esempio:"questa libreria non è più mantenuta dalla Comunità, occorre sostituirla"). L'indipendenza dal Database è anch'essa fondamentale: molte Software house stanno migrando verso soluzioni *NoSql* come MongoDB; il passaggio è sicuramente meno tedioso se il resto del proprio sistema è indipendente dalla base dati utilizzata. Anche una UI può essere cambiata facilmente, si può passare da una Desktop based UI ad una Web UI senza dover cambiare le *business rules*.

Andando ad esplorare più in dettaglio gli aspetti salienti della Clean Architecture, ciò che troviamo nel nucleo centrale sono le *Entities*, considerate come *business objects* o anche definite *business model* dell'applicazione. Le Entities incapsulano tutte le regole di business, sono rappresentate da oggetti con dei metodi definiti al loro interno o possono essere un set di strutture dati e funzioni, come ad esempio una classe Studente, Impiegato, o Stipendio. Essendo parte del nucleo, il loro stato rimane sempre lo stesso, sono gli oggetti meno soggetti ad un cambiamento: se nel sistema si cambia la veste grafica o si implementa un nuovo algoritmo di Sicurezza, questo non andrà a modificare il layer delle Entities; la classe Studente rimarrà la stessa. Al contrario, se viene cambiato un metodo definito in un Entity, tutto il contesto esterno ne risente.

Salendo di livello, troviamo gli *Use Cases*, nei quali vengono definite le *Business rules* dell'applicazione, ovvero le azioni che l'applicativo è destinato a compiere, i comportamenti che deve avere: la lista di Clienti deve essere ordinata per data di inserimento? cosa succede se un pagamento viene avviato? Come si comporta l'oggetto Docente se Studente non passa l'esame per tre volte consecutivamente? ecc. Questo layer ha un'interazione forte con le Entities, dialoga tramite le cosiddette *Boudaries*, delle interfacce che permettono la comunicazione tra i livelli e che in questo caso possono consentire agli Use Cases di recuperare i dati necessari forniti dagli Entities e gestirli. Robert Ce-

cil Martin, comunemente conosciuto con il nome di "Uncle Bob", durante una conferenza NDC tenuta a Londra nel 2013, soprannomina questi Use Cases come *Interactors*, definendoli come "*the choreographers of the Entities*". Questo nome verrà utilizzato nelle implementazioni future che si basano sulla Clean Architecture, come nel caso di Viper. Un cambiamento in questo layer non stimolerà alcun cambiamento su quello più interno, tuttavia possiamo aspettarci ad esempio che venga cambiato lo stato dell'interfaccia utente.

A fare da tramite tra la business logic ed il layer più esterno, costituito da entità come un Database, un Web Service o una GUI, vengono in soccorso gli *Interface Adapters*. Sono delle interfacce di supporto che si occupano di convertire il dato dal formato più conveniente per l'Interactor (che generalmente corrisponde all'oggetto dell'Entity) in un formato più adatto ad esempio ad una visualizzazione su schermo. Se ad esempio si fa uso di Hibernate per comunicare con il proprio Database, questo è il posto giusto dove gestirne le classi. Nessuna parte del codice di questo layer è a conoscenza di come sia fatta la parte grafica o la struttura di un Database. Non fa altro che preoccuparsi di manipolare i dati per non demandare il lavoro ad altri componenti. Nei casi applicativi in cui il destinatario a cui demandare gli oggetti "trasformati" sia un'interfaccia grafica visualizzabile all'utente, possiamo trovare in questo layer molte similitudini con il Pattern MVP (Model-View-Presenter). Per fare un paragone, il Model in questo caso sarà l'Interactor, la business logic dell'applicazione, mentre il Presenter è l'insieme di classi che si occupa di rendere più fruibili alla View i dati provenienti dall'Interactor. Il livello più esterno, al quale si è voluto dare il nome *Framework & Drivers*, è generalmente considerato come un dettaglio, in quanto non è visto come la parte del software più onerosa in termini di scrittura del codice. Come già accennato, questo livello può essere costituito da un Web Framework, o da una GUI, una base di dati, un collegamento con un device fisico (come ad esempio un iBeacon) e così via.

La Clean Architecture non non ha un pattern rigido, nessuna regola vieta di aggiungere dei cerchi, l'importante è applicare sempre la regola della Dipendenza. Come si è potuto notare, grazie a questa regola ogni parte svolge un compito ben specifico e definito. Questo porta ai vantaggi già citati e favorisce lo sviluppo in ambienti molto dinamici e in continua evoluzione come quello del Mobile.

2.4.3 VIPER

La Clean Architecture risulta uno strumento di qualità, ma rimane comunque un'astrazione. Una delle sue applicazioni pratiche nell'ambito Mobile è Viper, utilizzato soprattutto in progetti di grandi dimensioni. Inizialmente è stato pensato per gestire i progetti in ambito iOS, quindi per lo sviluppo sui

device di casa Apple, ma è stato presto esteso anche allo sviluppo sulle altre piattaforme.

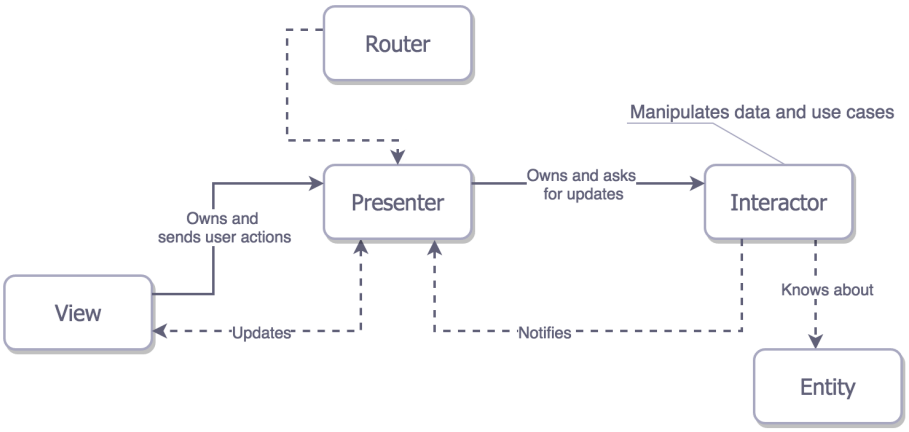


Figura 2.9: Immagine rappresentante la connessione tra i componenti VIPER

VIPER è l'acronimo di *View, Interactor, Presenter, Entity, Router*. Di seguito verrà fornita una descrizione di ogni componente con particolare riferimento a contesti reali nell'ambito dello sviluppo Android.

- Interactor: ereditando dalla Clean Architecture, anche in Viper esiste il concetto di Use Cases, che sono responsabili di gestire la Business logic incapsulando comportamenti piccoli ma ben definiti. Generalmente la feature di ogni Sprint va sviluppata creando un nuovo package all'interno dell'IDE contenente la struttura delle classi in stile VIPER, quindi l'Interactor di ogni modulo rappresenta un singolo Use Case all'interno dell'App. Contiene la Business logic legata alle Entities o ad operazioni di Rete o più in generale a meccanismi per prendere i dati da una certa fonte. In Android l'Interactor dialogherà con delle dipendenze esterne a Viper e proprie del Framework di Google, come ad esempio un Service o dei Manager utili al *fetching* dei dati. Il lavoro svolto dall'Interactor dovrebbe rimanere totalmente indipendente dalla View.

- Entities: i cosiddetti *POJO* (Plain Old Java Objects), da non confondere con il *data access layer*, del quale è invece responsabile l'Interactor. Sono oggetti Java utilizzati per rappresentare la realtà del sistema. Un oggetto dell'Entity non è mai manipolato dal Presenter o da altri componenti di Viper: solo l'Interactor ha la responsabilità di gestirli.

- Presenter: è considerato il motore della UI, invoca i metodi dell'Interfaccia dell'Interactor e una volta che lo stesso gli restituisce i dati, contiene

la logica necessaria per renderli fruibili alla View. Come si può evincere dall'immagine sopra riportata, oltre che mandare richieste all'Interactor, può quindi aggiornare la View a fronte di alcuni eventi (spesso scatenati dall'utente). Nei progetti reali i campi di una classe dell'Entity sono solitamente di numero superiore rispetto a quelli realmente necessari alla View, inoltre sono spesso in un formato non adatto. Per fare un esempio concreto, la classe Persona (Entity) potrebbe contenere un campo `long` rappresentante la data di nascita in formato *timestamp*: il Presenter genererà un oggetto (in Mango viene chiamato viewModel) il quale oltre che contenere solo i campi necessari per la View del modulo attivo in quel momento, avrà un campo `birthdate` in formato `String` e ben formattato, pronto per essere dato in pasto alla GUI.

- View: la View è passiva, non fa altro che aspettare che il Presenter gli dia i dati per poterli mostrare sul display. Il Presenter non conosce gli elementi propri della View, che in Android possono essere un'Activity, un Fragment, un Adapter, un Button e così via. Il Presenter conosce soltanto il contenuto dei dati che la View va a mostrare; sarà la View a preoccuparsi di come questi dati devono essere visualizzati.

- Routing: è il responsabile delle connessioni tra i moduli Viper, si preoccupa dell'andamento del flusso di lavoro, ovvero di come passare da un modulo all'altro (o se vogliamo, in termini Agili, da una feature all'altra). Il Router è gestito unicamente dalla View e dal Presenter. Esempio classico è quello di una app rappresentante una lista di elementi: nel momento in cui l'utente clicca su un determinato elemento della lista, la View, tramite un *listener* associato alla pressione dell'elemento, notificherà il Presenter, il quale chiamerà un metodo dell'Interfaccia del Router (che molto probabilmente avrà una signature molto simile a `onItemClicked(int index)`; la classe che implementerà l'interfaccia del Router si preoccuperà, tramite un Android Intent, di aprire una nuova Activity, scatenando l'introduzione di un nuovo modulo Viper. Tale modulo avrà il compito di ripetere l'intero processo di Viper (quindi anche coinvolgendo Interactor ed Entities) per andare a fare il *fetching* di tutti i dettagli associati all'elemento cliccato dall'utente, che verranno poi mostrati a video una volta terminata la richiesta (che sia una richiesta di rete o una query su un Database locale). In Android, la classe che implementa l'Interfaccia del Router è solitamente l'Activity stessa, la quale si preoccuperà di lanciare altre Activity. Nei progetti dove è coinvolto Viper, l'Activity è considerata il cuore, un ingranaggio di fondamentale importanza che permette di gestire il flusso stesso di tutto il sistema che si va a sviluppare.

Come si può notare, anche in Viper vi è una netta divisione dei compiti. Il motivo principale per cui si è deciso di utilizzare Viper in Mango è proprio il "Single Responsibility Principle", molto simile alla Dependency rule della Clean Architecture: ogni partizione di Viper ha la sua responsabilità, il codice di ogni parte si può sostituire senza difficoltà e senza generare l'instabilità dell'intero Sistema. Viper risulta quindi uno strumento utile per gestire al meglio ai cambiamenti futuri. Un esempio calzante è quello di Android Wear, che come già accennato, subirà nelle versioni successive un cambiamento nel metodo di comunicazione con la Rete: la versione 2.0 di Android Wear porterà ad un'indipendenza tra Smartwatch e Smartphone. Gli sviluppatori dovranno usare altri metodi per il recupero dei dati, potrebbe cadere il concetto di DataItems e Messages ed essere sostituito con altri meccanismi. Se si sviluppa con Viper, questo non costituisce alcun problema: le signature delle Interfacce dell'Interactor, che si occupa di prendere i dati, rimangono le stesse; va sostituita solo la loro implementazione. Inoltre, cambiare il modo in cui avviene la connessione alla Rete non porterà ad un cambiamento della struttura delle Entities, né si dovranno apportare variazioni al Presenter e alla veste grafica.

2.4.4 Confronto Viper con MVx

L'esperienza diretta degli ingegneri del Software con i nuovi contesti come il Web e il mondo Mobile, ha portato ad una decisione di abbandonare i pattern architetturali classici per dare spazio ai nuovi, come nel caso della Clean Architecture e VIPER. Il motivo principale di questa scelta è il Single Responsibility Principle, totalmente assente nei pattern del passato. Si vuole dare tuttavia uno sguardo più approfondito ai problemi legati ai pattern tradizionali che hanno portato a questa scelta.

A prescindere dal contesto, molte delle applicazioni erano progettate adottando l'architettura MVC (Model-View-Controller). L'architettura MVC impone che tutte le parti siano strettamente correlate tra di loro; anche il Model può conoscere lo stato della View, e questo costituisce uno svantaggio non indifferente nello sviluppo dei sistemi software, riducendo testabilità e riusabilità.

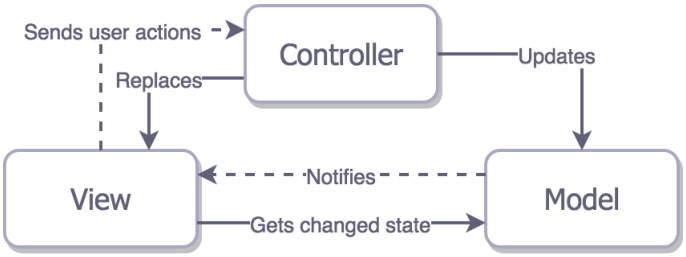


Figura 2.10: Immagine del tradizionale pattern MVC

In MVC, il Model rappresenta il dominio, ovvero l'accesso ai dati, la View rappresenta il layout da presentare sullo schermo, mentre il Controller, oltre che gestire il caricamento stesso dell'applicazione, si occupa di determinare quale View dovrà essere mostrata in risposta ad una determinata azione, come la pressione di un tasto. La View rimane *stateless*, è semplicemente ricreata dal Controller una volta che avviene un cambiamento nel Model. Viene preso come esempio una Web app, ove ad ogni pressione di un link è associato un URL adeguatamente gestito dal Server, nel quale risiederà un Controller che risponde a quel *path* con una determinata azione da compiere; una volta che ha completato le sue operazioni, il Controller risponde con la View corretta. La non separazione delle parti in MVC comporta che maggior parte della logica dell'applicazione non appartenga al Model o alla View, bensì al Controller. Si finisce inevitabilmente per fare un uso spropositato dello stesso, causando un problema che in gergo viene chiamato *Massive-View-Controller*: a volte più di mille righe di codice non sono abbastanza per soddisfare la complessità di un Controller ricco di feature, e provare a snellirlo diventa un compito arduo per lo sviluppatore.

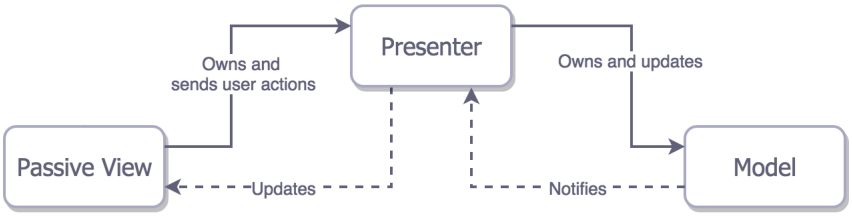


Figura 2.11: Immagine del pattern MVP

Un primo accenno di soluzione ai problemi sopra citati si ha con MVP (Model-View-Presenter). In questo pattern la logica della View, che in MVC è presente nel Controller insieme ad altro codice, viene separata in un'altra entità

chiamata *Presenter* o anche *ViewController*. Inoltre il Model non "dialoga" con la View e la stessa View è disaccoppiata dal Presenter; i cicli di vita delle entità sono separati e possono dialogare soltanto attraverso un'Interfaccia di *callback*. Come già accennato il Presenter, a differenza del Controller di MVC, contiene solo la logica collegata alla View, si occupa quindi di retituire ciò che servirà alla View per visualizzare l'informazione necessaria; sarà la View a preoccuparsi di mostrare il dato. Nasce quindi una sorta di "controller" separato e completamente dedicato alla View.

Prendendo spunto da questi pattern, il concetto di separazione si è esteso ancora, fino ad arrivare alla Clean Architecture e quindi a VIPER.

Capitolo 3

Progetto

3.1 Analisi

In questa sezione si descrivono i requisiti che l'applicazione deve rispettare e le funzionalità che deve implementare.

Anche nel mondo dell'intrattenimento la tecnologia è sempre più presente e ha lo scopo primario di raggiungere il pubblico in maniera più diretta e di dare un supporto efficace agli addetti ai lavori. Lo Smartphone in particolare è un media più interattivo e vicino all'utente rispetto ai classici Social Network e siti Web. Si è potuto notare come una mobile App sia una delle fonti più rapide per accedere ad un determinato servizio, lo strumento più utile e diretto per fare promozione e comunicazione. Per questo motivo è stato creato *Showtime*, sistema sviluppato su più piattaforme e interamente dedicato al mondo del teatro e dello spettacolo. Grazie all'utilizzo di questo servizio, un teatro di qualsiasi nomina e dimensione ha sia la possibilità di puntare sul Marketing, quindi di attrarre clienti più facilmente e di far parlare di sé, sia l'opportunità di avere una gestione più corretta ed ordinata di tutto ciò che concerne l'ambiente teatrale, come ad esempio l'acquisto di un biglietto o la consultazione degli spettacoli in programmazione. Showtime mette a disposizione delle funzionalità di base che poi vengono ampliate a seconda delle varie richieste dei teatri che vi aderiscono. Si tratta di un sistema multi-piattaforma che permette l'utilizzo della mobile App da parte degli utenti e al contempo fornisce una console Web per i gestori dei vari teatri. Il desiderio di voler dare all'utente un accesso ancora più immediato ai propri servizi ha spinto Mango a voler direzionarsi sulla creazione di una Wearable App di supporto, che è stato l'oggetto principale del progetto che sono andato a sviluppare.

3.1.1 Analisi dei Requisiti

In questa sezione si vogliono delineare le funzionalità della Mobile App che i proprietari dei teatri hanno richiesto all'azienda, e quindi la descrizione dei requisiti che la Smartwatch App dovrà soddisfare.

L'utente utilizzatore dell'App Showtime deve innanzitutto avere a portata di mano le informazioni aggiornate riguardanti il teatro di suo interesse e ricevere quindi le giuste notifiche a seconda del luogo in cui si trova. La Mobile App, già esistente e funzionante, ha quindi le seguenti feature:

- Consultazione lista spettacoli ed eventi: visualizzazione di una lista divisa per categoria di tutti gli spettacoli ed eventi presenti o futuri organizzati dal teatro. La categoria è implementata diversamente a seconda delle richieste del cliente, può essere divisa in maniera temporale, cioè per stagione teatrale, o per genere dello spettacolo, o per sale fisiche presenti all'interno della struttura.

- Consultazione dettagli spettacoli ed eventi: per ogni spettacolo sono visualizzabili i relativi dettagli dello stesso, tra cui giorni e fasce orarie, titolo, sottotitolo e descrizione dell'evento, serie di foto e immagini rappresentative.

- Notifiche push: ricezione di notifiche push su Smartphone riguardanti spettacoli di interesse.

- Wish-List: salvare una lista personale di spettacoli preferiti apponendo un *flag* su un determinato evento; quindi possibilità di consultazione della propria Wish-list.

- Accesso utente: possibilità da parte dell'utente di effettuare il login all'interno del sistema Showtime facendo uso dei più comuni social network o di classico *username* e *password*. Il login comporta l'ampliamento dell'utilizzo dell'App. Solo se l'utente è correttamente autenticato ha accesso alle feature riportate di seguito.

- Acquisti In-App: l'utente può acquistare uno spettacolo potendo scegliere settore e poltrona.

- QR-Code: generazione di un QR-Code a seguito dell'acquisto da poter mostrare all'addetto prima dell'ingresso in sala.

- iBeacon: una volta che il device dell'utente è fisicamente all'interno della sala, se agganciato ad un iBeacon apposto in posizioni strategiche, può ricevere notifiche sullo spettacolo in corso, promozioni e news riguardanti la stagione teatrale.

- Notifiche specifiche: informare l'utente dell'imminente inizio di uno o più spettacoli acquistati. Spedire notifiche quando l'utente si avvicina geograficamente al luogo dove verrà svolto uno o più spettacoli salvati in Wish-list.

La velocità con cui oggi si accede alle informazioni è divenuta la chiave per il successo di qualsiasi contesto applicativo. Il cliente, cosciente di questa nuova realtà, ha quindi richiesto per i suoi utenti una maniera ancora più veloce per fruire delle informazioni legate agli spettacoli e in futuro anche di poter accedere in sala senza dover tirare fuori lo Smartphone e cercare il QR-Code dello spettacolo acquistato. Per soddisfare questa richiesta si è pensato all'ausilio dello Smartwatch. Il progetto prevede l'implementazione di una Android Wear App per la consultazione degli spettacoli e del proprio abbonamento. Si vuole dare all'utente non solo la possibilità di ricevere notifiche inerenti all'avvicinarsi di uno spettacolo acquistato, ma anche di consultare tutti gli spettacoli in abbonamento attraverso un semplice tocco sul quadrante, andando quindi ad aumentare le informazioni consultabili già oggi attraverso App Smartphone. In particolare, l'applicazione dovrà soddisfare i seguenti requisiti:

- Menu Principale: mostrare all'utente un menu contenente l'accesso agli spettacoli in programma, agli eventi e ai biglietti acquistati. Anche se possono apparire diversi, vengono distinti i due termini: per "Spettacoli" si intende tutto quell'insieme di performance classiche, organizzate da compagnie teatrali che si esibiscono in una o più serate. Per "Eventi" invece (anche chiamati "Casa dello Spettacolo" secondo volontà del cliente), si vogliono intendere quelle performance tenute in un'unica serata e da un artista singolo. Un Evento può essere di carattere musicale, speech, dibattito, incontro o aperitivo.

- Cartellone stagione: di ogni spettacolo dovranno essere visualizzati immagine di locandina, titolo, sottotitolo, data e ora dell'evento e una breve descrizione. Il comportamento dovrà essere simile a quello dell'App per telefono, si dovrà quindi disporre di un'apposita sezione dedicata ai dettagli di ogni spettacolo.

- Funzioni da Watch: l'utente deve poter aprire un determinato spettacolo sul telefono a partire da un *tap* sullo Smartwatch o aggiungere l'evento alla propria Wish-List.

- Notifiche: ricevere notifiche personalizzate direttamente su Smartwatch riguardanti lo spettacolo.

- Tickets: l'utente loggato può visualizzare la lista dei propri biglietti acquistati direttamente sull'orologio.

- Check-in: l'utente loggato può reperire dalla sezione tickets il QR-Code da mostrare prima dell'inizio di uno spettacolo.

Nel sistema da sviluppare inoltre, sono stati rilevati anche i seguenti requisiti non-funzionali:

- Grafica: mantenere il più possibile lo stesso stile grafico utilizzato nella phone App, tra cui colori, dimensioni dei caratteri, immagini e divisioni logiche delle varie sezioni.

- Tempi di caricamento: per aumentare la User Experience, i tempi di caricamento degli spettacoli da caricare e visualizzare su schermo non devono superare i 10 secondi.

- QR-Code: la visualizzazione del QR-Code deve essere a tutto schermo e si dovrà chiedere all'utente di settare la luminosità del device al 100% per una migliore visualizzazione.

- Framework: il framework da utilizzare per lo scambio dei dati e grafica dovrà essere conforme alle specifiche e ai pattern Android Wear versione 1.x con eventuali test sulla versione 2.0 Developer Preview.

- IDE: l'ambiente di sviluppo del sistema applicativo deve essere Android Studio, nella sua versione più recente.

- Android Wear 1.x: svolgendosi il tirocinio in un periodo a cavallo tra la versione 1.x e 2.x di Android Wear, si deve cercare il più possibile di utilizzare i componenti della versione più recente progressivamente rilasciati nelle varie release Preview per gli sviluppatori.

Di seguito viene riportato un diagramma dei casi d'uso per descrivere le azioni che l'utente è abilitato a compiere.

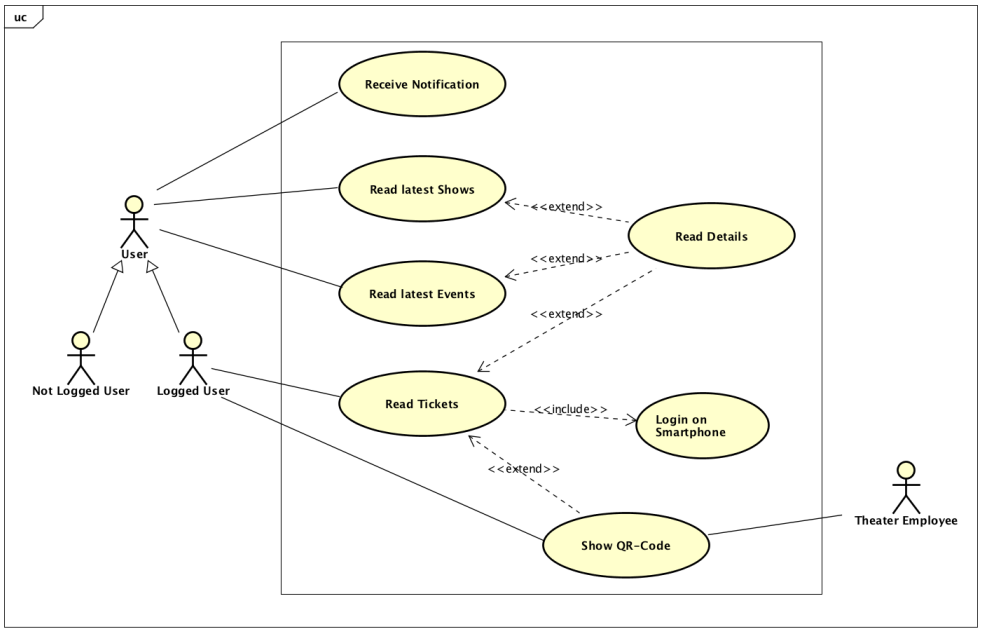


Figura 3.1: Schema dei Casi d'Uso

3.2 Design

In questo capitolo verranno discusse le scelte progettuali e di design, che hanno portato all'implementazione della Wearable App. Verrà rispettato un approccio *top-down*, si partirà da un aspetto più complessivo, riguardante le architetture e le tecnologie usate per soddisfare i requisiti per poi scendere nel dettaglio di modellazione delle varie parti dell'applicativo sviluppato.

3.2.1 Panoramica del sistema

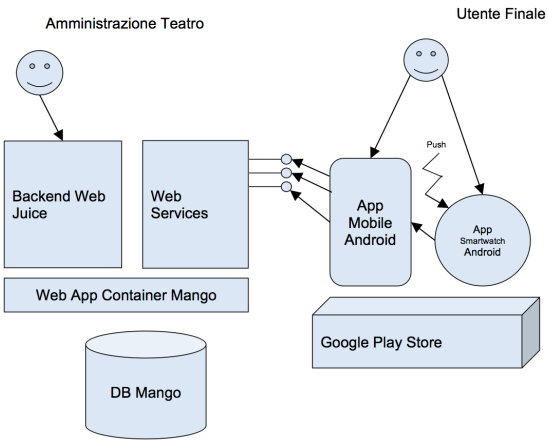


Figura 3.2: Immagini dei principali componenti coinvolti

Le tecnologie utilizzate nelle parti illustrate del sistema sono tutte *Open Source*, e comprendono a lato Server un Database relazionale (DBMS), utilizzato come punto nevralgico per il salvataggio di tutti i dati anagrafici degli utenti autenticati nonché delle informazioni concernenti i vari teatri. A gestire la base dati vi è un Web App Container utilizzato come Application Server su cui sono installate le Applicazioni Web. Tale Backend, a cui è stato dato il nome simbolico di *Juice*, è stato sviluppato con l'ausilio di un Web Framework a supporto delle richieste HTTP. Come è possibile notare nella figura riportata sopra, per quanto riguarda il lato client, è pervasivo l'utilizzo di Google Android. Lo Smartphone si occupa di effettuare le richieste al Web Service Juice e di ricevere le notifiche push generate con l'ausilio di *Google Cloud Messaging*, quindi di recuperare i dati necessari all'uso di tutte le feature sopra elencate. Lo Smartwatch invece non comunicherà direttamente con Juice ma bensì con il telefono tramite connessione Bluetooth LE. Il motivo principale di questa scelta è legato uno dei requisiti non funzionali indicati nella fase di Analisi: come si è potuto intuire dai capitoli precedenti, soltanto dalla versione 2.0 sarà

possibile effettuare operazioni di Rete direttamente dal Watch. Nella versione 1 l'orologio si dovrà appoggiare allo Smartphone per spedire richieste e ricevere i dati necessari. Come nella maggioranza dei device presenti in commercio, non tutti gli Smartwatch risulteranno compatibili con il nuovo *update* della piattaforma: il modello di orologio scelto per lo sviluppo è un *Sony Smartwatch 3* nel quale è già presente un modulo Wi-Fi ma si prevede che non sarà possibile effettuare l'upgrade del Sistema ad Android Wear 2.0. La versione 1.5 attualmente installata tuttavia risulterà un buon compromesso in quanto pur avendo una differente modalità di comunicazione, molti componenti della versione 2 legati soprattutto all'interfaccia grafica saranno retro-compatibili, permettendone quindi il corretto uso anche sul dispositivo citato. Per lo sviluppo della parte relativa al telefono è stato utilizzato un OnePlus modello 3T, nel quale è installato Android Nougat 7.0, corrispondente alla versione 24 di *Android SDK Platform-Tools*, ovvero l'insieme di tutti quegli strumenti forniti da Google per lo sviluppo in ambito Android. Per poter creare applicazioni in ambito Wear e quindi utilizzare tutte le classi e le interfacce fornite, la versione minima dell'SDK deve essere la 20. Anche in questo caso si pone un limite all'utilizzo dell'hardware, i telefoni Android più datati non possono né associarsi né comunicare con uno Smartwatch. L'intero sistema software è costituito, e sarà ampliato, da codice sorgente object-oriented in linguaggio Java.

3.2.2 Componenti Android

Tra i componenti dell'SDK Android utilizzati per lo sviluppo della Wearable App e della parte phone, quelli che verranno descritti in seguito hanno una rilevanza maggiore rispetto agli altri in quanto hanno contribuito al corretto funzionamento dello scambio dei dati e della visualizzazione su schermo. Il più importante è senza dubbio il `WearableListenerService`, già accennato nel Capitolo 2. Il compito fondamentale dell'oggetto WearableListenerService è quello di consentire lo scambio di dati tra i due device. In Android i Service sono quei componenti che possono portare a termine operazioni in background anche di lunga durata e non hanno alcun bisogno di una interfaccia grafica. Sono molto utilizzati in generale per avviare il download di file dalla Rete o per consentire la riproduzione musicale e lasciare all'utente la possibilità di interagire con il telefono. Sono totalmente indipendenti dall'applicazione che li lancia, possono sopravvivere anche se l'App viene chiusa o se non è in *foreground* (in esecuzione ma non in cima allo stack delle applicazioni aperte, quindi non visibile graficamente). Il WearableListenerService, come si può intuire dal nome, è un Service completamente gestito dal framework che rimanere in ascolto sul cambio di stato che avviene all'arrivo degli già citati `Message` o `DataItem`. Non essendo un normale Service ma una sua estensione,

il programmatore non si deve preoccupare di crearne una propria "versione" e lanciarlo in un determinato punto del workflow dell'applicazione, ma è sufficiente dichiararlo nel Manifest, creare una classe che estenda tale servizio ed effettuare l'Override di `onMessageReceived()` e `onDataChanged()`, lasciando così che sia Android stesso ad occuparsi del suo ciclo di vita.

In entrambi i device si è scelto inoltre di implementare un BroadcastReceiver, componente risultato utile nel tentativo di rispettare l'architettura Viper e la comunicazione tra ogni layer. Il Sistema Android e le App installate possono ricevere e spedire messaggi di broadcast e tramite un pattern publish-subscribe altre App possono rimanere in ascolto dei messaggi spediti. Per fare un esempio, Android manda messaggi di broadcast quando viene avviato, quando la batteria è scarica, o quando viene attivata la modalità aereo, e così via. Dall'altra parte un'App potrebbe rimanere in ascolto di alcuni di questi messaggi e reagire di conseguenza. E' anche possibile crearsi dei messaggi di Broadcast personalizzati nella propria App, spedirli in un punto del workflow e in un altro registrarsi e rimanere in ascolto. L'invio e la ricezione di tali messaggi avviene tramite un oggetto chiamato `BroadcastReceiver`. Nella fase di implementazione verrà spiegato l'utilizzo che se ne è fatto e in che modo è collegato con Viper. Un altro elemento molto usato per la visualizzazione degli spettacoli è la `WearableListView`, una versione della ListView ottimizzata per i piccoli schermi dei wearable. Mostra verticalmente una lista di elementi e ingrandisce l'elemento corrente quando l'utente si ferma nello scorrere in basso o in alto. Nella versione successiva di Android Wear, Google dichiarerà *deprecated* questa implementazione per Watch di ListView, dando spazio al nuovo `RecyclerView`, il quale svolge la stessa funzione di WearableListView ma con un consumo della memoria più efficiente, rendendo facilmente fruibili anche liste di grandi dimensioni. Il meccanismo per migliorare le performance si basa sul riuso della View durante lo scorrimento: la View che prima mostrava un certo set di dati viene salvata in memoria tramite un meccanismo di caching e riutilizzata successivamente per mostrare i dati aggiornati. Un Adapter si occupa di mantenere il set di elementi della lista aggiornato e di restituire la stessa View utilizzata in precedenza. RecyclerView è un chiaro esempio retrocompatibilità: sarà attivo a partire dalla versione 2.0 ma sarà utilizzabile anche da applicazioni che girano su versioni del Sistema precedenti e quindi su orologi più datati.

Un altro elemento graficamente interessante è il `GridViewPager`, costituito da una matrice di *Card* navigabili dall'utente in qualsiasi direzione. Solitamente è utilizzato per mostrare i dettagli di un certo *item* di una lista e dividerli in diverse sezioni, consentendo un'esperienza utente migliore. Infine, per tentare di sfruttare il più possibile i nuovi componenti grafici in arrivo si è deciso di utilizzare il `WearableActionDrawer`, rilasciato dalla Preview di Android Wear

2.0 durante l'ultima settimana di tirocinio, anch'esso retrocompatibile e creato allo scopo di facilitare all'utente l'esecuzione di una o più azioni utili grazie alla comparsa di un menù contestuale tramite uno *swipe* verso l'alto.

3.2.3 Pattern Utilizzati

Oltre al Single Responsibility Principle, che ha coinvolto tutte le scelte architetturali del sistema, sono essenzialmente due i pattern utilizzati e riconosciuti dal Team:

- Pattern Factory: essenzialmente utilizzato nelle varie parti del sistema per la costruzione degli oggetti ViewModel, considerati "versioni" più adatte alla View dei dati presi dalla business logic. Il pattern Factory è tra i più utilizzati nell'ambito dello sviluppo in Java ma anche in altri linguaggi comuni, ed è essenzialmente sfruttato per la creazione di Java Objects. La particolarità risiede nel fatto che il client che lo invoca non è al corrente della logica nascosta dietro la creazione dell'oggetto. In questo progetto una sola tipologia di oggetto verrà istanziata, ma in generale la flessibilità del Pattern Factory consente, grazie all'uso di una Interface o di una Abstract Class, di istanziare diverse implementazioni dello stesso. Citando la *Gang Of Four*: *"Define an interface for creating an object, but let subclasses decide which class to instantiate. The Factory method lets a class defer instantiation it uses to subclasses"*.

- Singleton: è considerato un pattern creazionale, viene utilizzato per garantire che venga creata una sola istanza di una certa classe e che questa istanza sia accessibile da più parti dell'applicazione, quindi che sia pubblica. E' stato implementato un Singleton nella classe `ItemsStore` che detiene la lista di tutti gli Item recuperati dal telefono.

- Event-driven Programming: il paradigma ad eventi consente di eseguire un determinato set di istruzioni a seguito di un input da parte dell'utente o di un altro componente del software. Nel progetto, questo approccio verrà largamente utilizzato per una gestione non bloccante della GUI, ma anche per consentire una corretta comunicazione tra i vari componenti dell'architettura VIPER. Ne fa uso anche framework per lo scambio dei dati tra i due device, che viene "svegliato" solo all'invio o alla ricezione di un dato in una delle due parti.

3.2.4 Architettura in Dettaglio

In questa sezione si farà uso di diagrammi C4 per descrivere nel dettaglio l'architettura del sistema. Si partirà da una visione più generica e di contesto

fino ad arrivare al diagramma delle classi in UML, come suggerito dalla filosofia C4.

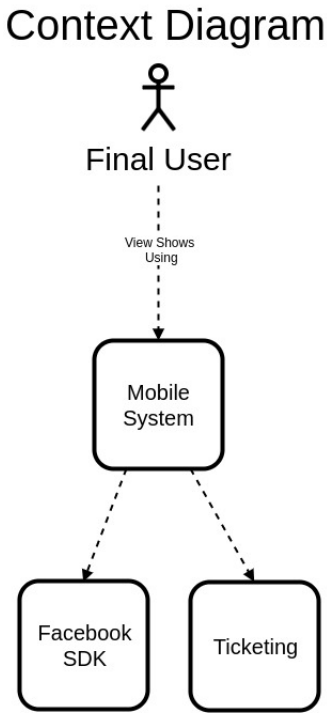


Figura 3.3: Context Diagram

Il primo schema è il Context Diagram, che come di consueto mette al centro dell'attenzione l'intero sistema software su cui si andrà a lavorare. Si è voluto lasciare esterni due dei più importanti servizi di terzi utilizzati.

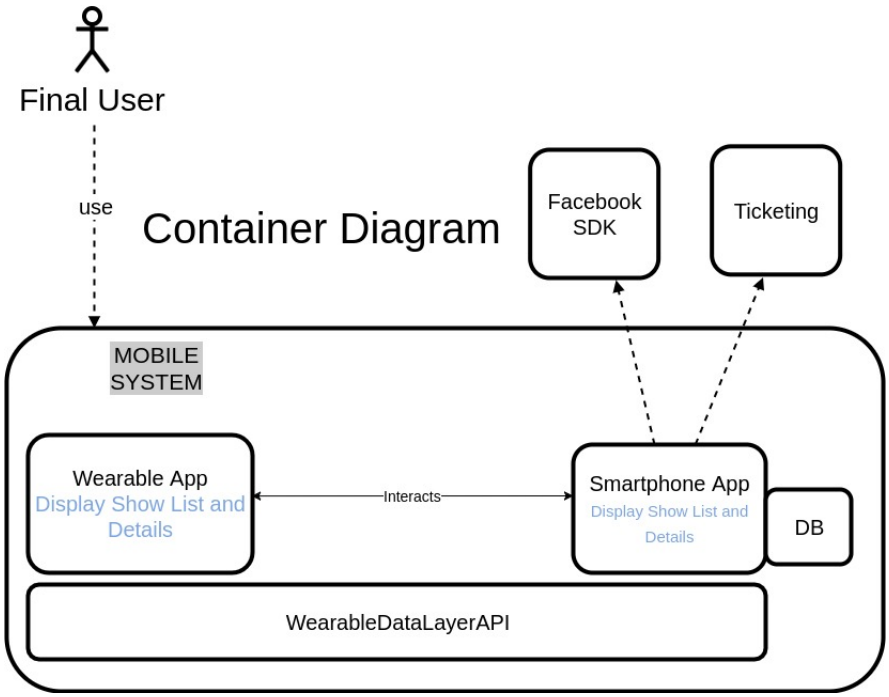


Figura 3.4: Container Diagram

Nel Container Diagram viene mostrata una divisione delle responsabilità e le principali scelte tecnologiche. Espandendo il box centrale del diagramma precedente infatti si possono notare le due entità principali, la Wearable App e la Smartwatch App, entrambi in comunicazione grazie al Wearable Data Layer API, di cui si è già discusso largamente nel capitolo 2. Quest'ultimo riquadro è posto alla base delle due App in gioco poiché utilizzato da entrambi per la comunicazione. Si è scelto di non espandere il riquadro DB perché non centrale per i fini di design, ma riguarda tutta la parte di WebService Juice e un db locale all'interno dell'App. Così come i servizi esterni di ticketing e login/sharing tramite Facebook, anche la parte DB relativa ai dati è stata posta vicino all'App Smartphone perché si vuole sottolineare che tutte le comunicazioni con l'esterno effettuate per il fetching dei dati non sono demandate al watch.

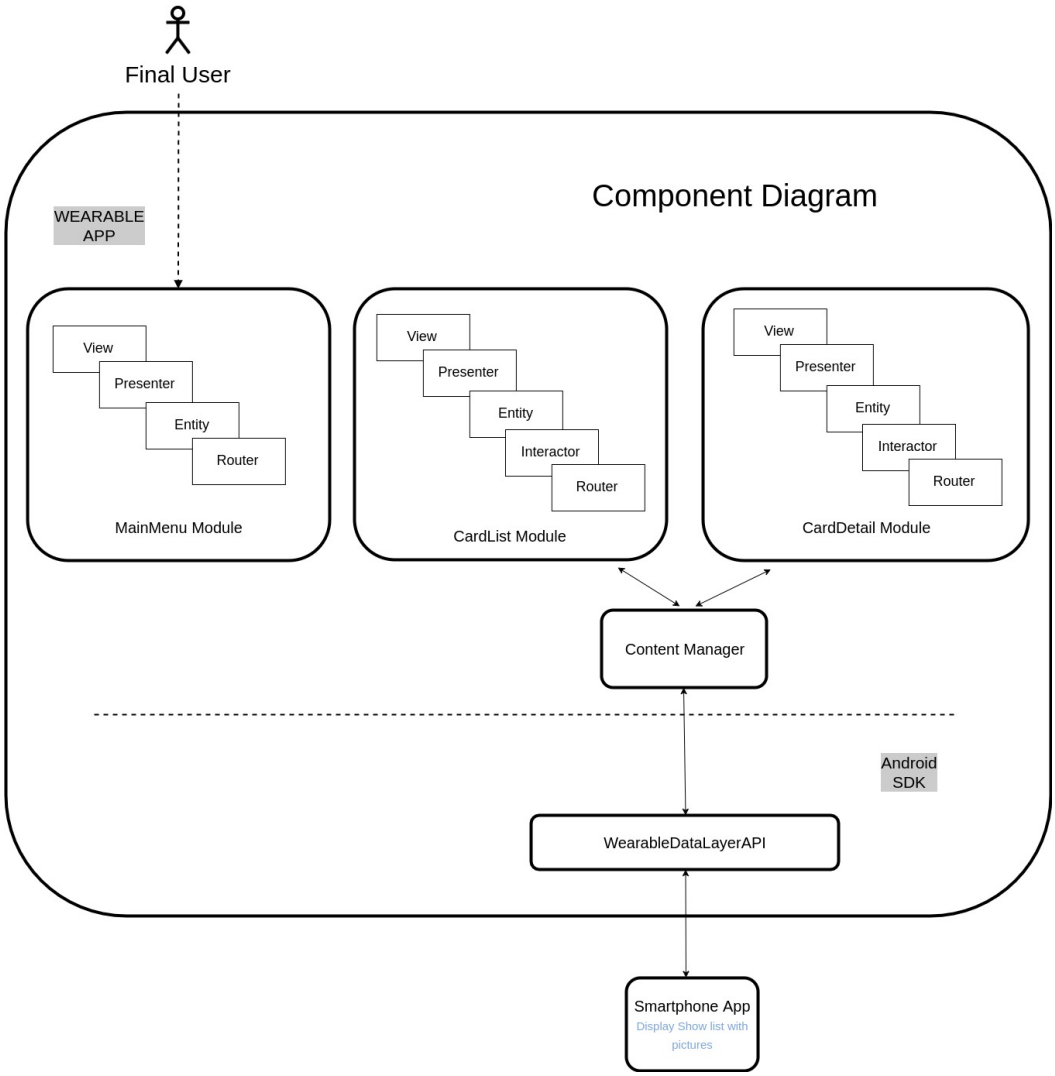


Figura 3.5: Component Diagram

Scendendo ancora più in profondità si vuole mostrare la struttura interna della Wearable App. Già a partire da questo punto si inizia ad intravedere l'efficacia di Scrum e di Viper. Si è deciso di dividere le varie parti da sviluppare creando il concetto di modulo Viper. Ogni modulo Viper è quindi una feature del software finale assegnata durante la fase di Planning e al quale è stato allocato un tempo di una o due Agile Sprint. Come si può evincere dalla figura, in ogni modulo viene replicata l'architettura Viper, tranne nel primo, in cui si è deciso di omettere la parte di Interactor perché non necessaria. In particolare il "MainMenu Module" si occupa di mostrare all'utente una semplice lista statica di 3 elementi generati a runtime all'apertura della Watch App. Alla pressione di una di queste voci, l'utente ha accesso ad un'altra View, anch'essa

mostrante una lista contenente però tutti gli spettacoli o eventi o tickets presi direttamente dal telefono. Si tratta del modulo "CardList", che a differenza del precedente possiede un Interactor il quale si occuperà del fetching dei dati da telefono e li salverà in memoria. Nell'accedere alla sezione dei dettagli di un determinato elemento della lista, si scatena un nuovo modulo Viper al quale è stato dato il nome di "CardDetail", che si occuperà della visualizzazione dei dettagli su schermo e di far interagire l'utente con il telefono, sempre a partire dall'Interactor. Un componente essenziale, creato appositamente per la comunicazione e al quale è stata dedicata un'intera Sprint, è il Content Manager. Sia CardList che da CardDetail fanno uso di questo componente, che in stretto collegamento con le classi del Data Layer API, gestisce la comunicazione sia per i dati in uscita che in entrata della Wearable App verso l'Handheld. Si passerà ora alla sezione *Class* di C4, dove viene descritta in dettaglio l'architettura interna di ogni modulo Viper citato nel diagramma dei Componenti. Per una rappresentazione più leggibile e per motivi di spazio, verranno omessi gli argomenti dei metodi scrivendo soltanto il tipo del valore di ritorno. Per lo stesso motivo le classi verranno inserite solo se ritenute importanti alla comprensione. Nella maggior parte dei casi quindi verranno messe in evidenza soltanto le Interfacce tenendo conto che ogni classe che si sottoscrive al contratto di una determinata Inteface avrà il suo stesso nome con apposto alla fine il suffisso "Impl". Rispettando la regola fondamentale della Clean Architecture, Viper si serve di interfacce di callback per passare i dati dai sottomoduli più interni verso quelli esterni (ad esempio dall'Interactor al Presenter), quindi l'utilizzo di tali listener all'interno del progetto è presto divenuto pervasivo. Pertanto si è scelto di mostrare negli schemi seguenti solo i più importanti, aventi tutti il suffisso "Callback" alla fine del nome dell'Interfaccia.

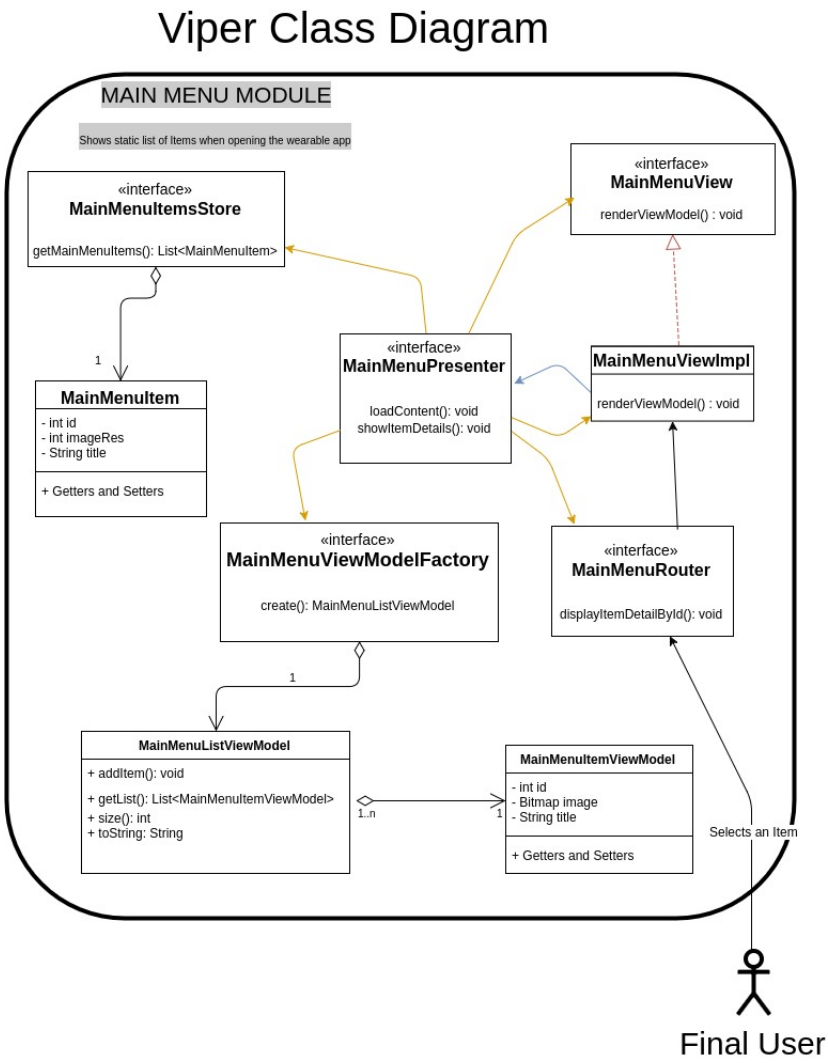


Figura 3.6: Class Diagram del modulo MainMenu

Come si può notare dalla figura, MainMenuRouter corrisponde alla parte Router di VIPER; tale interfaccia sarà implementata dall'Activity principale che sarà il punto di partenza del workflow dell'applicazione nonché il punto di passaggio verso il modulo VIPER successivo. il Router quindi, all'apertura dell'App, si dovrà occupare di creare una View atta a visualizzare una lista di elementi statici a video. L'interfaccia di tale View verrà chiamata MainMenuView e la sua implementazione dovrà richiedere i dati al Presenter chiamando il metodo `loadContent()`, il quale si preoccupa di scomodare la parte di Entity, costituita dall'interfaccia `MainMenuItemsStore`. L'implementazione di questa Interfaccia consente di restituire una lista creata a runtime di tre oggetti di tipo `MainMenuItem`. Si noti che in tutto il progetto per generalizzare il con-

cetto di elemento in una lista di elementi, viene attribuito il nome di Item. La scelta di questo nome generico è scaturita da una chiara richiesta da parte degli altri membri del Team di Sviluppo di tentare di replicare sulla parte Wear ciò che era stato già implementato sulla parte Phone, attribuendo anche gli stessi nomi ai vari concetti ed Interfacce. Ad un Item, proprio perché generico, gli si può attribuire una diversa semantica: può essere uno spettacolo se posto nella lista degli spettacoli, un evento se presente nella lista degli eventi, un ticket, e così via. MainMenuItem contiene soltanto tre parametri: un id per identificare l'Item, un titolo e un'immagine associata all'Item. Al parametro `imageRes` è stato attribuito il tipo di semplice `int` poiché si è previsto che nella fase di implementazione l'immagine sarebbe stata direttamente presa dalle risorse del progetto e non caricate da altre fonti esterne (Android associa un numero identificativo di tipo `int` alle risorse grafiche staticamente salvate nel progetto di Android Studio).

Una volta che l'Entity restituisce al Presenter la lista di Item, delega alla classe `MainMenuViewModelFactory` il compito di creare una versione più adatta alla visualizzazione su schermo degli Item ricevuti. Come si può intuire dal nome, è stato utilizzato il pattern Factory che prende in ingresso un MainMenuItem e restituisce un oggetto `MainMenuItemViewModel` (si è progettato di usare il termine *viewModel* per identificare una versione più "stile UI" del dato che si ha a disposizione). In questo contesto si passerà al Factory una List di MainMenuItem e verrà restituito un oggetto di tipo `MainMenuListViewModel`, contenente una lista di MainMenuItemViewModel. Sì è deciso di creare un oggetto che rappresenti una lista di viewModel piuttosto che utilizzare una *Java Collection* (come ad esempio `List<MainMenuItemViewModel>`) perché si vuole passare un unico oggetto alla View che rappresenti la lista. Uno degli svantaggi (se non l'unico) di Viper è che a livello di implementazione tende a generare una quantità consistente di classi. E' stata quindi l'esperienza su Viper degli altri membri del Team a suggerire questa scelta, con lo scopo finale di avere un codice più pulito e comprensibile.

A questo punto il Presenter ha in mano la giusta lista di elementi ben formattati da restituire alla View; chiama quindi il metodo `renderViewModel()` di MainMenuView, che si occuperà di aggiornare i dati della lista creata all'apertura dell'App. Durante la fase di design del modulo MainMenu si è discusso di cosa dovesse essere costituita la lista di elementi da visualizzare all'utente. E' stato deciso di mostrare tre elementi principali il quale titolo dovrà essere: "Spettacoli", "Casa del Teatro" e "Tickets". Il secondo può risultare ambiguo ma già accennato sopra è semplicemente il nome che il cliente ha chiesto di utilizzare per identificare gli Eventi in programma.

Si prevede che l'utente faccia tap su uno di questi tre Item. A seconda dell'elemento cliccato si avrà accesso ad informazioni diverse ma generate dal-

lo stesso modulo Viper, chiamato *CardList Module*. La View conosce quale elemento è stato cliccato, ne passerà quindi l'id al Presenter, il quale avrà un riferimento al Router. Il Presenter a questo punto invocherà il metodo `displayItemDetailById()` passando come argomento l'id dell'elemento selezionato. L'implementazione di questo metodo dovrà consentire l'apertura di una nuova Activity che implementerà l'interfaccia `CardListRouter`, dando inizio ad un nuovo ciclo Viper.

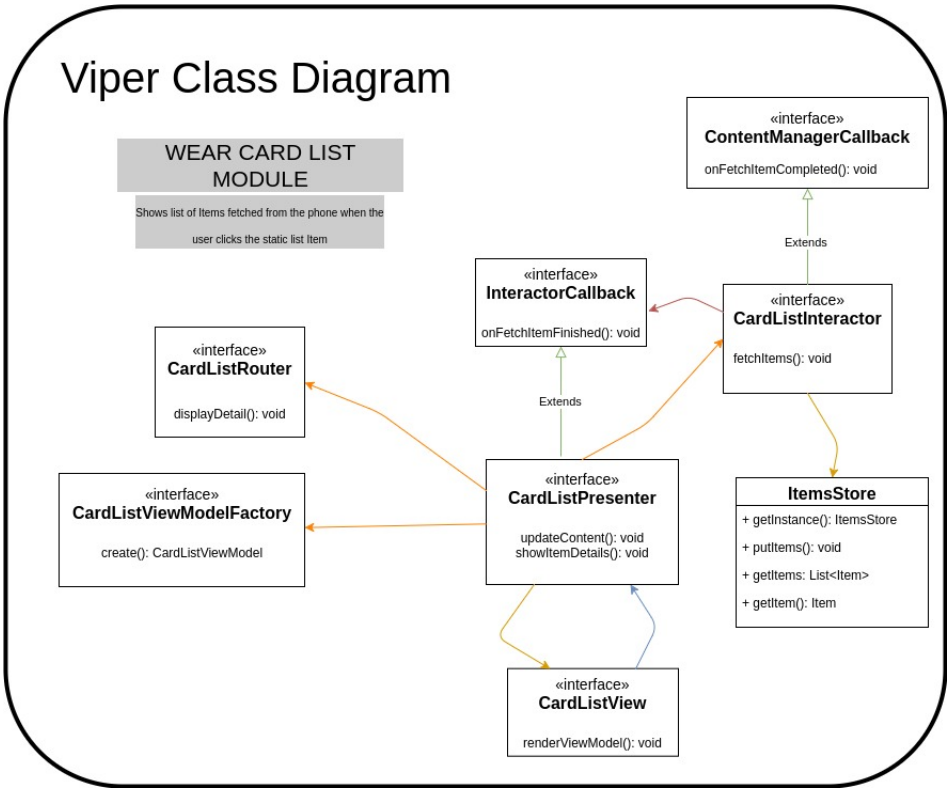


Figura 3.7: Class Diagram del modulo CardList

Nel modulo CardList l'architettura è essenzialmente ripetuta, tranne che sull'Interactor. Nel modulo precedente si è deciso di omettere l'Interactor perché considerato semplice: in quel caso i dati da prendere erano pochi e venivano generati direttamente sullo Smartwatch a runtime. Ben diverso è il caso di questo importante modulo, che si dovrà occupare di eseguire il *fetching* dei dati dal telefono utilizzando Data Layer API. L'Interactor effettua questa operazione tramite il metodo `fetchItems()`, il quale astrae il concetto di richiesta di ottenere dati dal telefono. Come si può notare il valore di ritorno è `void`. Avendo studiato il funzionamento del framework di Android Wear prima della fase di progettazione, si è presto intuito che la richiesta di

dati non poteva essere un'operazione bloccante; vige anzi un'architettura ad eventi: tale Interactor estende infatti l'Interfaccia `ContentManagerCallback`. Il metodo `onFetchItemCompleted()` dovrà in qualche modo essere chiamato dalla classe che farà uso del Data Layer API una volta che i dati sono arrivati senza errori sul dispositivo indossabile. Tramite un altro Callback chiamato `InteractorCallback`, l'Interactor passerà gli Item ricevuti al Presenter, che si occuperà di creare un viewModel da restituire alla View, la quale visualizzerà gli Item ricevuti in una lista.

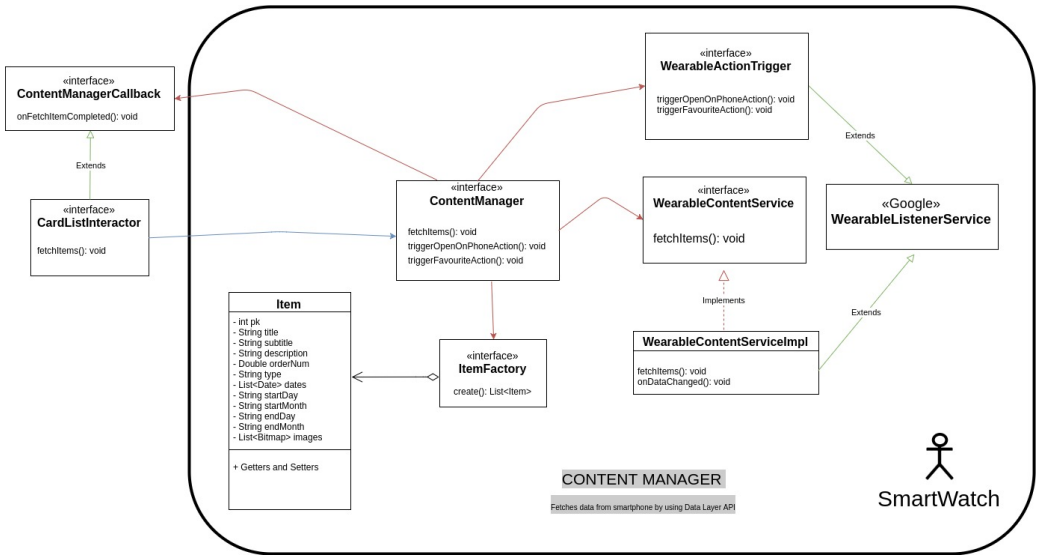


Figura 3.8: Class Diagram del ContentManager lato Smartwatch

Più di una Sprint è stata dedicata alla progettazione e all'implementazione del modulo ContentManager, che si distacca dall'architettura Viper ma funge come pool di classi di supporto alla comunicazione con il telefono.

Con un approccio speculare, anche nel telefono verrà progettato lo stesso ContentManager che però si occuperà di prendere gli Item dal Web Service Juice, trasformarli in una versione ottimizzata per il wearable ed inviarli. Con un'ottica orientata al futuro, è stato deciso di privare completamente CardListInteractor dal compito di prendere i dati, operazione che invece è stata demandata al ContentManager. In questo caso CardListInteractor si occupa solo di chiederli; non gli è dato sapere in che modo poi questi dati verranno scaricati. Già dalla versione 2.0 si potrebbe stravolgere il ContentManager, passare dal fetching tramite Bluetooth a chiamate HTTP tramite Wi-Fi. Un domani il ContentManager lato wearable potrebbe voler comunicare con un iBeacon, o interagire tramite altri meccanismi di comunicazione che saranno sviluppati in futuro, e così via. Molto peso è stato dato quindi al concetto

di astrazione dei vari componenti nonché di divisione dei compiti in piccole sottoparti, per consentire una resistenza maggiore del sistema ai cambiamenti futuri. Il ContentManager quindi si dovrà occupare di avere un'istanza della classe `WearableContentService`, identificato come l'oggetto che dovrà attivamente gestire la comunicazione tra i due device. Per farlo dovrà infatti estendere il famoso `WearableListenerService` offerto dal Wearable Data Layer API, che metterà a disposizione i metodi necessari alla ricezione del dato una volta richiesto tramite `fetchItems()`. Una volta che il WearableContentService riceve i dati, in qualche modo li dovrà restituire al ContentManager. Si è previsto che saranno oggetti di tipo Item rappresentanti gli Spettacoli ma completamente confezionati all'interno di oggetti forniti dal framework di Google, come i già citati DataItem, DataMap, Asset ecc. L'interfaccia ItemFactory è stata pensata con lo scopo trasformare questi oggetti del framework in oggetti *plain Java*. Tramite il metodo *create()*, tale Factory eseguirà quindi una vera e propria "traduzione", restituendo una List di Item, più facilmente gestibili dall'Interactor e quindi successivamente da tutte le altre parti di VIPER.

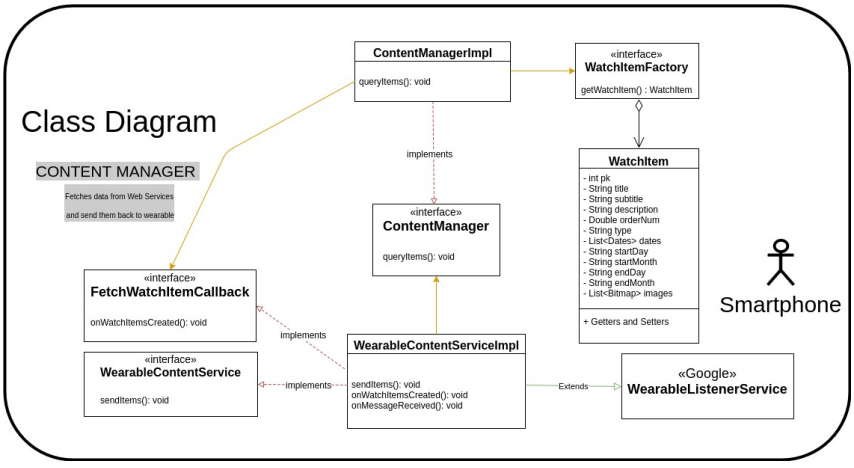


Figura 3.9: Class Diagram del ContentManager lato Smartphone

Per completezza si vuole riportare anche lo schema riguardante la modellazione lato Smartphone. Il Service WearableContentServiceImpl è sempre attivo, rimarrà quindi in ascolto di richieste da parte del wearable anche ad applicazione chiusa. Questa caratteristica di essere sempre in uno stato di running soddisfa un caso d'uso molto comune nell'utilizzo dello Smartwatch, ovvero avere accesso ad informazioni senza che l'App nel telefono sia effettivamente aperta. Nel caso riceva una richiesta di dati da parte dello Smartwatch, anche qui si è voluto delegare al ContentManager il compito di eseguire il fetching degli Item richiesti. Nel seguire le linee guida di altre parti della phone App già implementate in passato, si è previsto che questa operazione venga

effettuata da thread separati, i quali tramite un meccanismo ad eventi risveglieranno il WearableContentService ad operazione conclusa. Il ContentManager in questo caso si occupa quindi sia di effettuare la query al Web Service che di trasformare il dato (grazie al componente `WatchItemFactory`) in una versione più adatta allo Smartwatch. Quando il ContentManager avrà a disposizione la lista di WatchItem li passerà al WearableContentService tramite il metodo `onWatchItemsCreated()` dell'Interfaccia `FetchWatchItemCallback`. La logica per l'impacchettamento dei dati nel formato DataItem dovrà essere gestita direttamente dal WearableContentService, il quale si occuperà anche di inviarli. Una volta che l'Interactor di CardList avrà ricevuto i dati, oltre notificare il Presenter tramite l'InteractorCallback, li dovrà salvare in memoria perché siano utilizzati dal modulo CardDetail (rappresentato nella figura sottostante) e presumibilmente da altri moduli in futuro. Per il salvataggio dei dati farà uso dell'oggetto `ItemsStore`, che sarà poi utilizzato da CardDetail per mostrare i dettagli relativi ad un determinato Item.

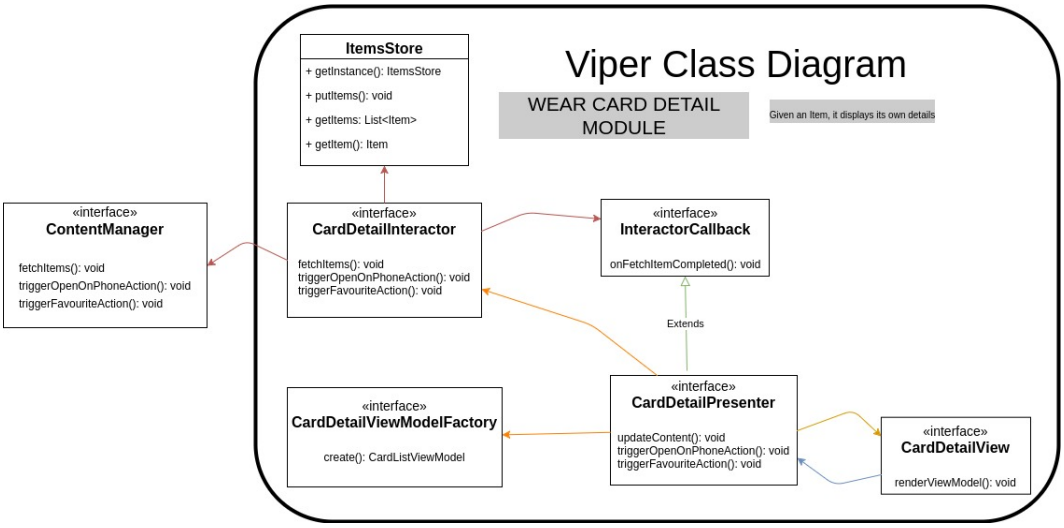


Figura 3.10: Class Diagram del Card Detail Module

Alla pressione di uno degli Item mostrati da CardList, il processo precedente si verrà a ripetere: la CardListView chiamerà quindi il CardListPresenter, il quale tramite l'oggetto `CardListRouter` farà in modo di aprire il nuovo modulo CardDetail, passando tramite un id il riferimento all'Item. L'Interactor di quest'ultimo modulo andrà di conseguenza a cercare nell'ItemsStore l'elemento avente l'id selezionato. Il *mock-up* prevede di dare la possibilità all'utente di aprire sul telefono o di marcare come preferito l'Item che si sta visionando. A fronte di uno dei due eventi quindi, il CardDetailView dovrà chiedere al Presenter di invocare i rispettivi metodi nell'Interactor, e quindi nel Con-

tentManager, il quale tramite la classe `WearableActionTrigger` si occuperà di spedire al telefono una delle due richieste.

3.2.5 Design Scambio Dati

Oltre che progettare classi, pattern e interfacce, durante la fase di design si è voluto discutere anche delle best practice e dei metodi migliori per massimizzare il risparmio energetico e l'uso della memoria RAM del device. Come punto essenziale si è subito individuata la questione dello scambio dei messaggi tra i due device. In questo contesto le strade da percorrere erano essenzialmente due:

- Fetching parziale: una volta entrato nel modulo CardList, l'Interactor chiede al telefono soltanto i parametri di ogni Item essenziali per la visualizzazione su schermo. Ci si sarebbe ridotti ad avere un semplice Item costituito di una immagine, un id e un titolo. Soltanto una volta entrati in CardDetail, un'altra chiamata al telefono consente di scaricare i parametri rimanenti di ogni Item, da poter poi visualizzare su schermo (sottotitolo, date evento, descrizione, altre immagini, ecc).

- Fetching totale: una volta entrato nel modulo CardList, l'Interactor chiede al telefono l'intero Item, contenente tutti i parametri, salvare questi dati in memoria e visualizzarne soltanto i parametri essenziali. Una volta entrati in CardDetail, l'Interactor deve soltanto recuperare da un Java Object i restanti parametri di ogni Item, evitando così di dover performare un'ulteriore operazione di Rete. Questo avrebbe limitato l'utilizzo del Bluetooth, risparmiando batteria.

Come si è potuto scaturire dalla descrizione dei vari moduli, in fase di planning si è optato per la seconda opzione: il CardList è l'unico avente il compito di recuperare gli spettacoli, gli eventi o i ticket dal telefono col fine di mostrarli all'utente. Anche le informazioni visualizzate nella sezione dei dettagli sono direttamente prese da una lista di Item già salvati nella memoria dell'indossabile (ItemsStore). Nessuna ulteriore operazione di richiesta di dati viene effettuata nei confronti del telefono. Le due Action accennate poc'anzi sono state considerate due feature meno pesanti in termini di quantità di dati da scambiati tra i due device. Di seguito viene riportato un diagramma di sequenza che descrive in maniera schematica l'interazione tra i due dispositivi.

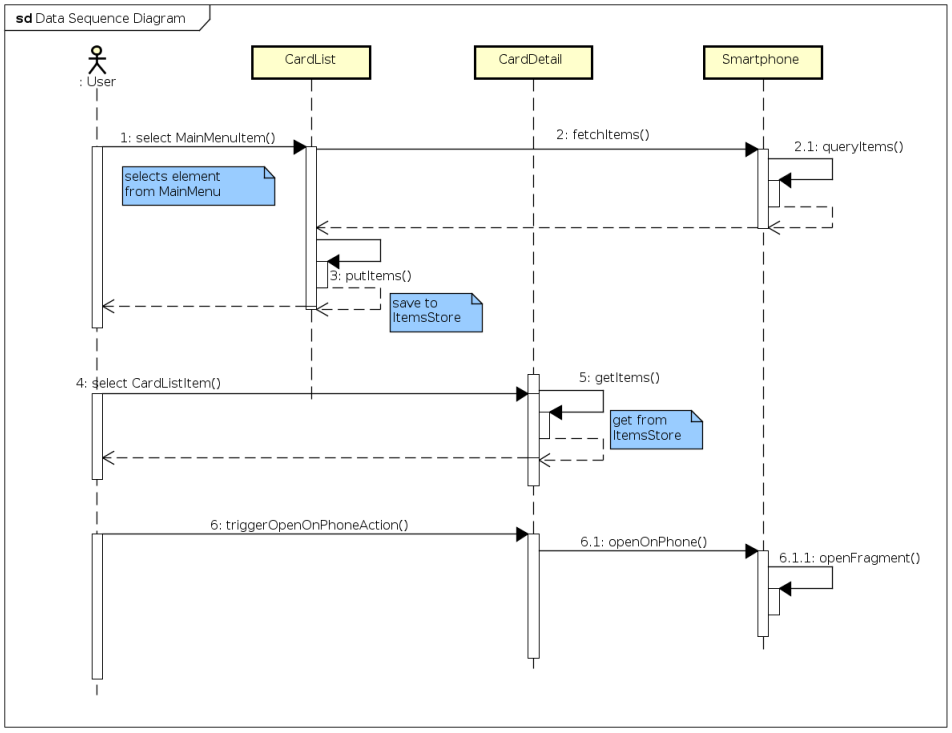


Figura 3.11: Data Sequence Diagram

3.3 Implementazione

Verrà fornita una descrizione dell'implementazione del progetto Showtime Android Wear seguendo l'andamento delle varie Sprint pianificate durante il tirocinio.

3.3.1 Demo App

Le prime quattro Sprint sono state dedicate sia allo studio del framework Android Wear con successiva presentazione agli altri membri del Team, sia all'implementazione di una Wear App avente lo scopo di testare le nuove funzionalità grafiche e di comunicazione offerte dal framework. Veniva chiesto di visualizzare sul telefono una lista di elementi e mostrarne i dettagli in una nuova Activity. Gli stessi dati dovevano essere visualizzati su watch all'apertura della Wearable App tramite una WearableListView e quindi aprire una nuova Activity alla pressione di un elemento e mostrarne i dettagli per mezzo di un GridViewPager. Molta attenzione è stata posta nella parte relativa alle operazioni di Rete. Una delle richieste del Project Manager era quella di tro-

vare, grazie anche all'aiuto di alcuni esempi Open Source presenti in portali specializzati come *GitHub*, il metodo migliore per effettuare operazioni di rete, quindi chiedere e ricevere dati in maniera efficiente. Alla presentazione della Demo App è seguito il design dell'architettura del progetto e lo studio delle Architetture Software utilizzate da Mango.

3.3.2 Ambiente di sviluppo

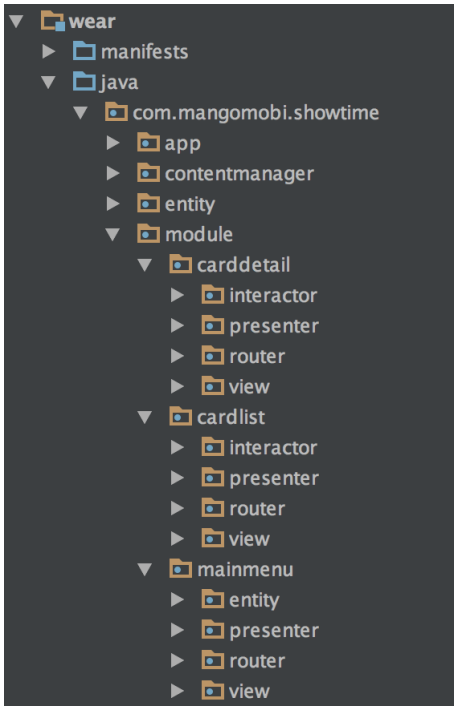


Figura 3.12: Immagine della struttura Viper su Android Studio

In Android Studio si è tentato di rispecchiare il più possibile la suddivisione in task creati in fase di Planning e salvati in Trello. Dando uno sguardo al Component Diagram, risulta naturale ed immediato tradurre la fase di Design in quella di Implementazione: ad ogni componente è stato associato un *Java package* contenente la struttura VIPER ove prevista. Dalla figura si può notare la struttura ad albero: nella root sono stati posti i package generici riguardanti le parti in comune a tutti i moduli, mentre nel package "module" sono stati collocati i moduli VIPER. Il package "App" contiene tutte le Activity, considerate le basi su cui si regge il resto del sistema. Implementando le Interfacce dei Router, le Activity fungono anche da coordinatori del flusso dell'applicazione. Il package `contentanager` non è stato implementato

tenendo conto dell'architettura VIPER; è stato visto bensì come un'estensione dell' Interactor di uno o più moduli, è stato posto quindi nella root allo stesso livello delle Activity. Stesso discorso vale per le Entity: si è subito intuito che ogni modulo VIPER possiede la stessa parte di Entity, costituita da una semplice classe Item rappresentante l'elemento da mostrare e la classe ItemsStore, che impacchetta una lista di Item. Come già largamente discusso nella fase di Design, dalle varie Sprint sono sorte quattro parti principali da sviluppare: MainMemu, CardList, CardDetail e ContentManager. Verranno presentati in seguito degli spezzoni di codice dei moduli sviluppati presentandone le parti più significative.

3.3.3 MainMenu

Figura 3.13: Screenshot del modulo MainMenu

Per il MainMenu si è scelto di mostrare la classe MainMenuViewImpl per dare un riscontro pratico al funzionamento di VIPER. La View è la prima parte di VIPER creata dalla MainMenuActivity, una volta pronta richiede i dati al Presenter e registra gli ascoltatori degli eventi scatenati dall'utente.

```
public class MainMenuViewImpl extends Fragment
                                        implements MainMenuView {

    private MainMenuListViewAdapter.OnItemClickListener mListener;
    private MainMenuListViewAdapter mAdapter;
    private WearableListView mWearableListView;
```

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    mListener = new MainMenuListViewAdapter.OnItemClickListener()
        {
        @Override
        public void onItemClicked(MainMenuItemViewModel
            mainMenuItemViewModel) {
            MainMenuPresenter menuPresenter =
            (MainMenuPresenterImpl)
                ((MainMenuActivity)getActivit())
            .getPresenter(MainMenuPresenter.PRESENTER_TAG);
            menuPresenter.showItemDetails(mainMenuItemViewModel);
        }
    };
}


@Override
public View onCreateView(LayoutInflater inflater, ViewGroup
    container, Bundle savedInstanceState) {
    View view = inflater.inflate(R.layout.fragment_main_menu,
        container, false);
    mWearableListView = (WearableListView)
        view.findViewById(R.id.wearable_list_view);
    return view;
}

@Override
public void onViewCreated(View view, Bundle savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);
    MainMenuPresenterImpl menuPresenter = (MainMenuPresenterImpl)
             ((MainMenuActivity) getActivity())
    .getPresenter(MainMenuPresenter.PRESENTER_TAG);
    if (menuPresenter != null) {
        menuPresenter.loadContent();
    }
}

@Override
public void onDetach() {
    super.onDetach();
    mListener = null;
}
```

```
    @Override
    public void renderViewModel(MainMenuListViewModel
        mainMenuListViewModel) {
        mAdapter = new MainMenuListViewAdapter(getActivity(),
            mainMenuListViewModel, mListener);
        mWearableListView.setAdapter(mAdapter);
    }
}
```

Listato 3.1: Codice della classe MainMenuViewImpl

Come si può intuire dai nomi dei metodi il Presenter viene creato soltanto a seguito dell'avvenuta creazione dell'interfaccia grafica; nel metodo `onViewCreated()` quindi viene creata un'istanza di MainMenuPresenter e invocato il metodo `loadContent()`.

```
public class MainMenuPresenterImpl extends Fragment implements
    MainMenuPresenter {

    private MainMenuItemsStore mMainMenuItemsStore =
        MainMenuItemsStore.getInstance();
    private MainMenuListViewModel mMainMenuListViewModel;
    private MainMenuViewModelFactory mMainMenuViewModelFactory;
    private MainMenuView mMainMenuView;
    private MainMenuRouter mRouter;


    @Override
    public void onAttach(Context context) {
        super.onAttach(context);
        mRouter = (MainMenuRouter) context;
    }

    @Override
    public void loadContent() {
        mMainMenuViewModelFactory = new
            MainMenuViewModelFactoryImpl();
        mMainMenuView = (MainMenuView) ((MainMenuActivity)
            getActivity()).getView(MainMenuView.MAIN_MENU_VIEW_TAG);
        mMainMenuListViewModel =
            mMainMenuViewModelFactory.create(getActivity(),
            mMainMenuItemsStore.getMainMenuItems());
        mMainMenuView.renderViewModel(mMainMenuListViewModel);
    }

    @Override
    public void showItemDetails(MainMenuItemViewModel
        mainMenuItemViewModel) {
        mRouter.displayCardListById(mainMenuItemViewModel.getId());
    }
}
```

Listato 3.2: Codice della classe MainMenuPresenterImpl

Una volta che il Presenter ha creato un viewModel adatto per il MainMenu, lo passerà alla View chiamando il metodo `renderViewModel()`. Soltanto in risposta a questo metodo la View setterà l'Adapter della WearableListView. In Android, l'unico scopo dell'Adapter è quello di fare da *DataSource*, cioè di popolare e mantenere i dati che gli vengono passati i quali verranno poi visualiz-

zati nella ListView. Come riportato dalla documentazione ufficiale di Android Developers: "*the Adapter provides access to the data items. The Adapter is also responsible for making a View for each item in the data set.*" Quando l'utente premerà su uno degli Item, verrà invocato il metodo `onItemClicked()` dell'Interfaccia `OnItemClickListener` costruita direttamente all'interno dell'Adapter. L'implementazione del metodo `showItemDetails` si occuperà di delegare al Router l'andamento del flusso dell'applicazione, quindi di aprire la nuova CardDetailActivity tramite il MainMenuRouter (implementato da MainMenuActivity).

L'Adapter, estendendo la classe WearableListView.Adapter, ha accesso a vari metodi per la gestione dei dati da mostrare nella lista, tra cui `OnBindViewHolder()`, che prende i parametri del viewModel passatogli dal Presenter e li abbina ai componenti grafici presenti in una Row della lista, attraverso la classe `MainMenuRowView`.

```
public class MainMenuListViewAdapter
                            extends WearableListView.Adapter {

    private Context mContext;
    private MainMenuListViewModel mMainMenuListViewModel;
    private OnItemClickListener mListener;

    public MainMenuListViewAdapter(Context context,
        MainMenuListViewModel mainMenuListViewModel,
        OnItemClickListener listener) {
        mContext = context;
        mMainMenuListViewModel = mainMenuListViewModel;
        mListener = listener;
    }

    @Override
    public WearableListView.ViewHolder onCreateViewHolder(ViewGroup
        viewGroup, int i) {
        return new WearableListView.ViewHolder(new
                                    MainMenuRowView(mContext));
    }

    @Override
    public void onBindViewHolder(WearableListView.ViewHolder
        viewHolder, int i) {
        MainMenuRowView mainMenuRowView =
                    (MainMenuRowView) viewHolder.itemView;
        final MainMenuItemViewModel item =
```

```
            mMainMenuListViewModel.get(i);

        mainMenuRowView.getImage().setImageBitmap(item.getImage());
        mainMenuRowView.getText().setText(item.getTitle());
        viewHolder.itemView.setOnClickListener(new
            View.OnClickListener() {
            @Override
            public void onClick(View v) {
                mListener.onItemClicked(item);
            }
        });
    }

    @Override
    public int getItemCount() {
        return mMainMenuListViewModel.size();
    }

    public interface OnItemClickListener {
        void onItemClicked(MainMenuItemViewModel
            mainMenuItemViewModel);
    }
}
```

Listato 3.3: Codice della classe MainMenuListViewAdapter

3.3.4 CardList

Figura 3.14: Screenshot del modulo CardList

Il Modulo CardList adotta esattamente la stessa logica per quanto concerne la parte di interfaccia grafica, e quindi si presenterà con un codice simile. Anche in questa parte si vuole esaltare l'efficacia e la flessibilità di Viper, ma focalizzandosi sulla creazione del ViewModel. Come spiegato nella fase di Progettazione, il modulo CardList si occupa di chiedere i dati al telefono e salvarli in memoria nel watch. La modalità con cui verranno presi i dati verrà mostrata in seguito, si vogliono mostrare ora i parametri della classe Item e la classe ItemsStore che ne detiene una lista.

```
public class Item {
    private Integer pk;
    private Integer type;
    private Double orderNum;
    private String title;
    private String subtitle;
    private String description;
    private List<Date> dates;
    private String startDay;
    private String startMonth;
    private String endDay;
    private String endMonth;
    private List<Bitmap> images;
    /* Getters and Setters*/
```

Listato 3.4: Codice della classe Item

```
public class ItemsStore {
    private static final ItemsStore itemsStore = new ItemsStore();
    private List<Item> listItems = new ArrayList<>();

    private ItemsStore() {
    }

    public static ItemsStore getInstance() {
        return itemsStore;
    }

    public List<Item> getItems() {
        return this.listItems;
    }

    public void putItems(List<Item> items) {
        listItems.addAll(items);
    }

    public void putItem(Item item) {
        listItems.add(item);
    }

    public Item getItem(int pk) {
        for (Item item : listItems) {
            if (item.getPk().equals(pk)) {
                return item;
            }
        }
        return null;
    }
}
```

Listato 3.5: Codice della classe ItemsStore

Una volta che questi oggetti vengono salvati nell'ItemsStore, il CardListPresenter dovrà creare un CardListViewModel da restituire alla View. Lo stesso lavoro verrà eseguito in seguito dal modulo CardDetail, creando un suo CardDetailViewModel, che avrà ovviamente un'implementazione diversa dal precedente, proprio perché adattato per un contesto grafico differente. Si sono voluti mostrare i parametri dell'oggetto Item per mostrare che alcuni rimangono invariati, mentre altri, come ad esempio la lista di date da mostrare a video, verranno completamente rimodellati. Di seguito quindi vengono forniti

i due Factory per la creazione di entrambi i ViewModel per dare al lettore una maggiore consapevolezza delle differenze e più in generale di quanto sia modificabile un oggetto proveniente dalla business logic.

```
public class CardListViewModelFactoryImpl
                        implements CardListViewModelFactory {

    private CardListViewModel mCardListViewModel =
                                    new CardListViewModel();

    @Override
    public CardListViewModel create(
                        Context context, List<Item> items) {

        for (Item item : items) {
            int pk = item.getPk();
            Bitmap image = item.getImages().get(0);
            String title = item.getTitle();
            String startDay = item.getStartDay();
            String startMonth = item.getStartMonth();
            String endDay = item.getEndDay();
            String endMonth = item.getEndMonth();
            CardListItemViewModel cardListItemViewModel =
                    new CardListItemViewModel(pk, title, startDay,
                        startMonth, endDay, endMonth, image);
            mCardListViewModel.addItem(cardListItemViewModel);
        }
        return mCardListViewModel;
    }
}
```

Listato 3.6: Codice della classe CardListViewModelFactoryImpl

```
public class CardDetailViewModelFactoryImpl
                        implements CardDetailViewModelFactory {

    private Context mContext;
    private List<Drawable> bitmapList;
    private List<Bitmap> mImages;
    private CardDetailViewModel mViewModel;

    public CardDetailViewModelFactoryImpl() {
        mViewModel = new CardDetailViewModel();
        bitmapList = new ArrayList<>();
```

```
    }

    @Override
    public CardDetailViewModel create(Context context, Item item) {
        mContext = context;
        mImages = item.getImages();
        mViewModel.setTitle(item.getTitle());
        mViewModel.setSubtitle(item.getSubtitle());
        mViewModel.setDescription(item.getDescription());
        setBitmapList();
        setDateList(item.getDates());
        return mViewModel;
    }

    private void setBitmapList() {
        Resources res = mContext.getResources();
        for (Bitmap bitmap : mImages) {
            bitmapList.add(new BitmapDrawable(res, bitmap));
        }
        mViewModel.setImages(bitmapList);
    }

    public void setDateList(List<Date> dateList) {
        List<DateViewModel> dateViewModelList = new ArrayList<>();
        Collections.sort(dateList);
        for (int i = 0; i < dateList.size(); i++) {
            DateViewModel dateViewModel = new DateViewModel();
            Date date1 = dateList.get(i);
            Date date2 = null;
            if (i < dateList.size() - 1) {
                date2 = dateList.get(i + 1);
            }
            dateViewModel.setDate(
                    WearableUtils.getDateToString(date1));
            dateViewModel.setStartHour(
                    WearableUtils.getHourToString(date1));
            if (isSameDay(date1, date2)) {
                dateViewModel.setEndHour(
                        WearableUtils.getHourToString(date2));
            } else {
                dateViewModel.setEndHour("");
            }
            dateViewModelList.add(dateViewModel);
        }
```

```
        mViewModel.setDateViewModelList(dateViewModelList);
    }

    private boolean isSameDay(Date date1, Date date2) {
        DateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");
        if (date2 != null) {
            return dateFormat.format(date1)
                    .equals(dateFormat.format(date2));
        }
        return false;
    }
}
```

Listato 3.7: Codice della classe CardDetailViewModelFactoryImpl

3.3.5 CardDetail

(a) Screenshot CardDetail 1

(b) Screenshot CardDetail 2

L'implementazione del pattern Factory appena presentata è la parte sicuramente più interessante di questo modulo. La business logic di CardDetail è infatti quasi totalmente inesistente in quanto si limita a prendere i gli Item necessari dalla classe ItemsStore. Per completezza si vuole porre l'attenzione sui componenti forniti dal framework per gestire la parte grafica, e quindi mostrare la classe `CardDetailViewImpl`, la quale fa uso sia dell'ormai datato `GridViewPager`, sia di nuovi elementi presentati nella versione 2.x di Android Wear, come ad esempio il `WearableDrawerLayout` e il `WearableActionDrawer`, oggetti che verranno utilizzati per mostrare un me-

nu a comparsa ove è possibile far interagire l'utente tramite opzioni di scelta multipla.

```
public class CardDetailViewImpl extends Fragment
                      implements CardDetailView,
                      WearableActionDrawe.OnMenuItemClickListener {

    private WearableDrawerLayout mWearableDrawerLayout;
    private WearableActionDrawer mWearableActionDrawer;
    private int mItemPk;
    private WearableActivity mContext;
    private GridViewPager mPager;
    private List<NavigationMenu> mNavigationMenu;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mItemPk = getArguments().getInt(Constants.ITEM_PK);
        mNavigationMenu = new ArrayList<>();
        mNavigationMenu.add(new NavigationMenu(Constants.PHONE_PATH,
                R.drawable.ic_smartphone, getResources()
                    .getString(R.string.common_open_on_phone)));
        mNavigationMenu.add(new
            NavigationMenu(Constants.FAVOURITE_PATH,
            R.drawable.favourite,
                getResources().getString(R.string.make_favourite)));
    }

    @Override
    public void onAttach(Context context) {
        super.onAttach(context);
        if (context instanceof Activity) {
            mContext = (WearableActivity) context;
        }
    }

    @Nullable
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup
        container, Bundle savedInstanceState) {
        super.onCreateView(inflater, container, savedInstanceState);
        View view = inflater.inflate(R.layout.fragment_card_detail,
            container, false);
        final Resources res = getResources();
```

```
        mWearableDrawerLayout = (WearableDrawerLayout)
            view.findViewById(R.id.drawer_layout);
        mWearableActionDrawer = (WearableActionDrawer)
            view.findViewById(R.id.bottom_action_drawer);
        mWearableActionDrawer.setOnMenuItemClickListener(this);
        ViewTreeObserver observer =
            mWearableDrawerLayout.getViewTreeObserver();
        observer.addOnGlobalLayoutListener(new
            ViewTreeObserver.OnGlobalLayoutListener() {
            @Override
            public void onGlobalLayout() {
                mWearableDrawerLayout.getViewTreeObserver().
                removeOnGlobalLayoutListener(this);
                mWearableDrawerLayout.peekDrawer(Gravity.BOTTOM);
                mWearableDrawerLayout.closeDrawer(Gravity.BOTTOM);
            }
        });
        mPager = (GridViewPager) view.findViewById(R.id.pager);
        mPager.setOnApplyWindowInsetsListener(new
            View.OnApplyWindowInsetsListener() {
            @Override
            public WindowInsets onApplyWindowInsets(View v,
                WindowInsets insets) {
                final boolean round = insets.isRound();
                int rowMargin = res.getDimensionPixelOffset(
                                        R.dime.page_row_margin);
                int colMargin = res.getDimensionPixelOffset(
                       round ? R.dimen.page_column_margin_round
                              : R.dimen.page_column_margin);
                mPager.setPageMargins(rowMargin, colMargin);
                mPager.onApplyWindowInsets(insets);
                return insets;
            }
        });
        DotsPageIndicator dotsPageIndicator = (DotsPageIndicator)
            view.findViewById(R.id.page_indicator);
        dotsPageIndicator.setPager(mPager);
        return view;
    }

    @Override
    public void onViewCreated(View view, Bundle savedInstanceState) {
        super.onViewCreated(view, savedInstanceState);
        CardDetailPresenterImpl cardListPresenter =
```

```
            (CardDetailPresenterImpl) ((CardDetailActivity)
            getActivity())
                .getView(CardDetailPresenter.CARD_DETAIL_PRESENTER_TAG);
        if (cardListPresenter != null) {
            cardListPresenter.updateContent(mItemPk, getActivity());
        }
    }

    @Override
    public void renderViewModel(CardDetailViewModel
        cardDetailViewModel) {
        mPager.setAdapter(new
            CardDetailGridPagerAdapter(getActivity(),
            mContext.getFragmentManager(),
                           cardDetailViewModel,mWearableDrawerLayout));
    }

    @Override
    public boolean onMenuItemClick(MenuItem menuItem) {
        CardDetailPresenterImpl cardDetailPresenter =
            (CardDetailPresenterImpl) ((CardDetailActivity)
            getActivity()).getPresenter(
                    CardDetailPresenterImpl.CARD_DETAIL_PRESENTER_TAG);
        final int itemId = menuItem.getItemId();
        Intent intent = new Intent(getActivity(),
            ConfirmationActivity.class);
        switch (itemId) {
            case R.id.menu_open_on_phone:
                if (cardDetailPresenter != null) {
                    cardDetailPresenter.triggerOpenOnPhoneAction(
                        Constants.PHONE_PATH, mItemPk, getActivity());
                }
                intent.putExtra(
                    ConfirmationActivity.EXTRA_ANIMATION_TYPE,
                        ConfirmationActivity.OPEN_ON_PHONE_ANIMATION);
                intent.putExtra(
                    ConfirmationActivity.EXTRA_MESSAGE, "");
                startActivity(intent);
                break;
            case R.id.menu_favourite:
                if (cardDetailPresenter != null) {
                    String itemTitle = ItemsStore.
                            getInstance().getItem(mItemPk).getTitle();
                    cardDetailPresenter.triggerFavouriteAction(
```

```
                    Constants.FAVOURITE_PATH, mItemPk, itemTitle,
                        getActivity());
                }
                intent.putExtra(
                    ConfirmationActivity.EXTRA_ANIMATION_TYPE,
                        ConfirmationActivity.SUCCESS_ANIMATION);
                intent.putExtra(ConfirmationActivity.EXTRA_MESSAGE,"");
                startActivity(intent);
                break;
        }
        mWearableDrawerLayout.closeDrawer(mWearableActionDrawer);
        return true;
    }
}
```

Listato 3.8: Codice della classe CardDetailViewImpl

Come si può intuire dal metodo `onMenuItemClick()`, a seconda di quale opzione sarà scelta dall'utente, viene chiamato un metodo diverso del Presenter, il quale chiamerà metodi diversi dell'Interactor. CardDetailInteractor passerà quindi al ContentManager stringhe diverse a seconda dell'opzione cliccata, e quindi finirà per inviare un Message al telefono contenente giusto input, che sarà correttamente riconosciuto dall altro device. Anche il DataGridView ha un suo Adapter, che avrà il compito di far visualizzare nella struttura matriciale le giuste informazioni in ogni cella.

3.3.6 ContentManager Wear

L'intero package ContentManager è dedicato allo scambio di dati tra Smartphone e Smartwatch. Come già detto in precedenza sarà la business logic di CardList a chiedere i dati al ContentManager da mostrare nella WearableListView. CardListInteractor quindi invocherà il metodo `fetchItems()` riportato nello spezzone di codice sottostante. Se invece ci troviamo nel modulo CardDetail verrà invocato uno degli altri due metodi sottostanti.

```
public class ContentManagerImpl implements ContentManager {

    private WearableContentService mWearableContentService;
    private WearableActionTrigger mWearableActionTrigger;
    private ContentManagerCallback mCallback;
    private BroadcastReceiver broadcastReceiver = new
        BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
```

```
            List<Item> items = (List<Item>)
                intent.getSerializableExtra
            (Constants.BROADCAST_DATA_MAP_ARRAY);
            mCallback.onFetchItemCompleted(items);
        }
    };
    public ContentManagerImpl() {
        this.mWearableContentService = new
            WearableContentServiceImpl();
    }

    @Override
    public void fetchItems(int itemMenuId, Context context, final
        ContentManagerCallback callback) {
        mCallback = callback;
        LocalBroadcastManager.getInstance(context)
        .registerReceiver(broadcastReceiver, new
            IntentFilter(Constants.ACTION_ITEM_CHANGED));
        mWearableContentService.fetchItems(itemMenuId, context);
    }

    @Override
    public void triggerOpenOnPhoneAction(String action, int itemPk,
        Context context) {
        mWearableActionTrigger = new WearableActionTriggerImpl();
        mWearableActionTrigger.triggerOpenOnPhoneAction(action,
            itemPk, context);
    }

    @Override
    public void triggerFavouriteAction(String action, int itemPk,
        String itemTitle, Context context) {
        mWearableActionTrigger = new WearableActionTriggerImpl();
        mWearableActionTrigger.triggerFavouriteAction(action,
            itemTitle, context);
    }
}
```

Listato 3.9: Codice della classe ContentManagerImpl

A seconda del metodo chiamato, il ContentManager creerà istanze delle classi che si occupano della comunicazione. Verrà descritta in dettaglio soltanto la classe `WearableContentServiceImpl` e tralasciata la classe WearableActionTriggerImpl, in quanto entrambi usano gli stessi metodi per le operazioni

di rete. Quest'ultima inoltre, ha una logica molto più semplice: non deve fare altro che inviare un Message al telefono chiedendo di eseguire determinate operazioni (come l'apertura di una schermata o una query al database per impostare un Item come preferito). Il WearableContentService invece contiene una logica più complessa e articolata: oltre che chiedere i dati al telefono, dovrà anche gestirne la ricezione e trasformare gli oggetti ricevuti in oggetti facilmente gestibili dal CardListInteractor. Di seguito viene quindi riportato il codice WearableContentServiceImpl, considerata la classe più importante dell'intero sistema.

```
public class WearableContentServiceImpl extends
    WearableListenerService
        implements WearableContentService,
                   GoogleApiClient.ConnectionCallbacks,
                   GoogleApiClient.OnConnectionFailedListener {

    private static final String TAG =
        WearableContentService.class.getSimpleName();
    private GoogleApiClient mGoogleApiClient;
    private String mNodeId;
    private Context mContext;
    private boolean mResolvingError = false;
    private ArrayList<Item> mItems;
    private ItemFactory mItemFactory = new ItemFactoryImpl();

    @Override
    public void fetchItems(int itemMenuId, Context context) {
        mContext = context;
        mGoogleApiClient = new GoogleApiClient.Builder(context)
                .addApi(Wearable.API)
                .addConnectionCallbacks(this)
                .addOnConnectionFailedListener(this)
                .build();
        mGoogleApiClient.connect();
        requestItems(itemMenuId);
    }

    @Override
    public void onCreate() {
        super.onCreate();
        mContext = this;
        mGoogleApiClient = new GoogleApiClient.Builder(this)
                .addApi(Wearable.API)
```

```
            .addConnectionCallbacks(this)
            .addOnConnectionFailedListener(this)
            .build();
    mGoogleApiClient.connect();
}

@Override
public void onConnected(Bundle connectionHint) {
    mResolvingError = false;
}

public void requestItems(final Integer itemMenuId) {
    new Thread(new Runnable() {
        @Override
        public void run() {
            if (mGoogleApiClient != null &&
                !(mGoogleApiClient.isConnected() ||
                mGoogleApiClient.isConnecting())) {
                mGoogleApiClient
                .blockingConnect(
                    Constants.GOOGLE_CLIENT_CONN_TIMEOUT,
                TimeUnit.MILLISECONDS);
            }
            NodeApi.GetConnectedNodesResult result =
            Wearable.NodeApi
            .getConnectedNodes(mGoogleApiClien).await();
            List<Node> nodes = result.getNodes();
            for (Node node : nodes) {
                if (node.isNearby()) {
                    Log.i(TAG, node.getDisplayName() + " " +
                        node.getId());
                    mNodeId = node.getId();
                    Wearable.MessageApi
                    .sendMessage(mGoogleApiClient, mNodeId,
                    Constants.ITEM_PATH,
                        itemMenuId.toString().getBytes()).await();
                    if (!result.getStatus().isSuccess()) {
                        Log.e(TAG, "Error sending request message");
                    } else {
                        Log.i(TAG, "Success! Message sent to: " +
                            node.getDisplayName());
                    }
                }
            }
```

```
            mGoogleApiClient.disconnect();
        }
    }).start();
}

@Override
public void onConnectionSuspended(int cause) {
    Log.i(TAG, "Connection to Google API client was suspended");
}

@Override
public void onConnectionFailed(ConnectionResult result) {
    if (!mResolvingError) {
        if (result.hasResolution()) {
            try {
                mResolvingError = true;
                Log.e(TAG, GOOGLE_API_CLIENT_ERROR_MSG);
                result.startResolutionForResult((CardListActivity)
                    mContext, REQUEST_RESOLVE_ERROR);
            } catch (IntentSender.SendIntentException e) {
                mGoogleApiClient.connect();
            }
        } else {
            Log.e(TAG, "Connection to Google API client has
                failed");
            mResolvingError = false;
        }
    }
}

@Override
public void onDataChanged(DataEventBuffer dataEventBuffer) {
    Log.d(TAG, "onDataChanged: " + dataEventBuffer);
    for (DataEvent event : dataEventBuffer) {
        if (event.getType() == DataEvent.TYPE_CHANGED &&
            event.getDataItem() != null &&
        Constants.WATCH_ITEM_PATH
            .equals(event.getDataItem().getUri().getPath())) {
            DataMapItem dataMapItem =
                DataMapItem.fromDataItem(event.getDataItem());
            ArrayList<DataMap> watchItemsDataMap =
            dataMapItem.getDataMap()
                .getDataMapArrayList(Constants.EXTRA_ITEMS);
            new CreateItemListAsyncTask()
```

```
                .execute(watchItemsDataMap);
            }
        }
    }

    private class CreateItemListAsyncTask extends
        AsyncTask<ArrayList<DataMap>, Void, ArrayList<Item>> {

        @Override
        protected ArrayList<Item>
            doInBackground(ArrayList<DataMap>... watchItemsDataMap) {
            return (ArrayList) mItemFactory.create(mGoogleApiClient,
                watchItemsDataMap[0]);
        }

        @Override
        protected void onPostExecute(ArrayList<Item> items) {
            super.onPostExecute(items);
            mItems = items;
            Intent intent = new Intent();
            intent.setAction(Constants.ACTION_ITEM_CHANGED);
            intent.putExtra(Constants.BROADCAST_DATA_MAP_ARRAY,
                items);
            LocalBroadcastManager
                .getInstance(mContext).sendBroadcast(intent);
        }
    }
}
```

Listato 3.10: Codice della classe WearableContentServiceImpl

Una delle prime cose che saltano all'occhio è il fatto che questa classe estende il più volte citato `WearableListenerService`, il quale fornisce i metodi di callback `onMessageReceived()` e `onDataChanged()`, che rispettivamente gestiscono la ricezione di oggetti di tipo Message (libreria MessageAPI) e di tipo DataItem (libreria DataItemAPI). Questa classe inoltre implementa le interfacce fornite dall'oggetto `GoogleApiClient` per gestire la connessione al framework e per avviare la comunicazione via Bluetooth tra i due device.

A connettersi è proprio il metodo `fetchItems()`, chiamato dall'Interactor di CardList. Oltre che connettersi, sfrutta la libreria NodeAPI per trovare il device più vicino ed inviare la richiesta. Dovendo inviare un semplice input al telefono, non dovranno essere spedite grandi quantità di dati; basterà una semplice stringa identificativa a cui lo Smartphone attribuirà il giusto signifi-

cato. Il componente più adeguato per questo scopo è il MessageAPI. l'oggetto Message, come già descritto in precedenza, possiede un payload molto leggero ed è costituito da un array di byte che viene inviato ai Nodi specificati in maniera totalmente inaffidabile: si tratta di spedire una richiesta, se tale richiesta non va a buon fine l'utente può essere notificato e ripetere l'operazione in un secondo momento, quando la connessione con l'altro Nodo sarà migliore. Diverso è il caso in cui vanno inviati grandi quantità di dati, dove è richiesta una certa affidabilità di consegna. Il ContentManager dello Smartphone infatti invierà gli Item contenenti gli spettacoli sotto forma di DataItem, nel quale sono impacchettati anche gli Asset, oggetti contenenti le immagini associate ad ogni spettacolo. L'operazione di ricerca del Nodo e di spedizione avviene in un *Thread* separato per consentire alla GUI di non rimanere bloccata. Tuttavia, come è si può notare, sia al metodo per la ricerca del Nodo che a quello per l'invio del messaggio viene apposto un `.await()` in coda. Il metodo await fa parte della classe PendingResult, oggetto che viene restituito all'invocazione del metodo `Wearable.MessageApi.sendMessage()`, e consente al thread di bloccarsi finchè l'operazione non viene conclusa. Si tratta di una *Synchronous call* molto importante per gestire il risultato dell'operazione, che come si può intuire, è utile per implementare diversi comportamenti a seconda di come si è conclusa l'operazione. Il framework mette anche a disposizione una chiamata asincrona, che può essere implementata anche nello stesso thread della GUI. Sono due diversi modi per implementare la stessa operazione: ciò che conta è ottenere un oggetto che rappresenti il risultato per poter agire di conseguenza. Per meglio afferrare il concetto, di seguito viene riportato un esempio di chiamata asincrona usata per gestire il risultato di un'operazione di invio di un DataItem, che evita l'utilizzo del metodo await.

```
pendingResult.setResultCallback(new ResultCallback<DataItemResult>()
    {
    @Override
    public void onResult(final DataItemResult result) {
        if(result.getStatus().isSuccess()) {
            Log.d(TAG, "Data item set: " +
                result.getDataItem().getUri());
        }
    }
});
```

Listato 3.11: Asynchronous call per ottenere un PendingResult

Tornando al Listato 3.10, gli Item vengono inviati tramite DataItem, quindi il metodo di callback lato wear sarà `onDataChanged()`. Il codice scritto in questo metodo consente di far partire un nuovo Thread (in Android la classe

AsyncTask permette una migliore gestione del Multithreading) nel quale un altro Factory method (questa volta chiamato ItemFactory) verrà in ausilio per la trasformazione da pool di oggetti riconosciuti dal DataLayerAPI ad insieme di oggetti squisitamente Java, come un `ArrayList<Item>`.

Nell'AsyncTask, il metodo di callback `onPostExecute()` viene invocato quando il task è concluso, passando come parametro la lista degli Item creati da ItemFactory. A questo punto WearableContentService ha ricevuto i dati che gli servivano e li detiene anche trasformati nella forma più corretta. Dovrà trovare un modo per spedirli al ContentManager che ha invocato `fetchItems()`, il quale chiamerà l'Interactor, che a ritroso risalirà tutti i layer di Viper fino ad arrivare alla View, la quale finalmente mostrerà i dati. Non è stato possibile utilizzare una semplice interfaccia di Callback come per gli altri layer, questo per una ragione legata al WearableListenerService. In Android, ogni componente ha un suo ciclo di vita autonomamente gestito, il framework Android Wear in questo caso crea una nuova istanza di tale Service ogni volta che viene invocato uno dei due metodi in ascolto per la ricezione di dati (onMessageReceived e onDataChanged). A causa di questo comportamento, qualsiasi argomento si passi al metodo fetchItems, sarà andato perduto una volta che del Service viene creata una nuova istanza. Risulta quindi impossibile implementare un Callback. In una situazione normale, ContentManagerImpl poteva eseguire l'implement di una ipotetica interfaccia "ServiceCallback", e passare quindi `this` come argomento al metodo `fetchItems()`.Una volta ricevuti i dati, sarebbe stato sufficiente usare tale callback invocando il suo metodo di risposta, ipoteticamente chiamato "`onDataReceived()`". Essendo questa soluzione impossibile da codificare, il metodo `onPostExecute()` si è dovuto quindi attrezzare diversamente: ha usato un BroadcastReceiver. Leggendo il codice ContentManagerImpl infatti, si nota subito che prima di chiamare fetchItems un broadcastReceiver viene registrato, rimarrà quindi in ascolto finchè non verrà effettivamente ricevuto il dato. Tramite il metodo `onReceive()` della classe `BroadcastReceiver` si può quindi gestire correttamente la lista di Items ricevuta dal WearableContentService. Questa voltà sarà possibile usare un Callback (`ContentManagerCallback`). CardListInteractor implementerà tale interfaccia e passerà i dati ricevuti al Presenter, e così via fino a risalire alla View.

```
public class CardListInteractorImpl implements CardListInteractor {

    @Override
    public void fetchItems(int itemMenuId, Context context, final
        InteractorCallback callback) {
        ContentManager contentManager = new ContentManagerImpl();
        contentManager.fetchItems(itemMenuId, context, new
```

```
            ContentManagerCallback() {
            @Override
            public void onFetchItemCompleted(List<Item> items) {
                ItemsStore.getInstance().putItems(items);
                callback.onFetchItemFinished(items);
            }
        });

    }
}
```

Listato 3.12: Codice della classe CardListInteractor

3.3.7 ContentManager Phone

Nella parte Phone, il processo è molto simile, i nomi dei metodi e delle classi e la loro semantica sono gli stessi. Cambia essenzialmente la fonte da dove vengono presi i dati e la costruzione del giusto oggetto da inviare al Wearable. Il metodo `onMessageReceived()` entrerà in funzione all'invio della richiesta di inviare gli Item. La classe `WearableContentServiceImpl` posta nella parte Phone del progetto di Android Studio, si attrezza quindi per chiedere al suo ContentManager (da non confondere con il ContentManager lato watch) di recuperare gli Items costituiti da testo e immagini facendo una query al Web Service Juice e tramite un Factory costruire un oggetto WatchItem, una versione di Item più adatta al contesto Wearable. Ad operazione conclusa gli Item, che siano spettacoli, o gli eventi, o i ticket vengono impacchettati (questa volta in DataItem, o più precisamente in DataMap) e spediti tramite il metodo `Wearable.DataApi.putDataItem()`. Anche in questo caso si fa uso del metodo await e di un PendingResult per gestire il risultato dell'operazione. Per motivi legati alla quantità ingente di righe di codice, viene riportato soltanto il metodo che a partire da una lista di WatchItem prepara un oggetto DataItem e ne effettua l'invio all'indossabile.

```
    private void sendDataToWearable(List<WatchItem> watchItemList) {
        ConnectionResult connectionResult =
        mGoogleApiClient.blockingConnect(
            Constants.Wearable.GOOGLE_API_CLIENT_TIMEOUT_S,
                                    TimeUnit.SECONDS);
        int count = watchItemList.size();
        ArrayList<DataMap> dataMapList = new ArrayList<>(count);
```

```
for (WatchItem watchItem : watchItemList) {
    DataMap rootItemDataMap = new DataMap();
    DataMap itemFieldsDataMap = new DataMap();
    DataMap itemImagesDataMap = new DataMap();
    itemFieldsDataMap.putInt(Constants.Wearable.EXTRA_PK,
        watchItem.getPk());
    itemFieldsDataMap.putInt(Constants.Wearable.EXTRA_TYPE,
        watchItem.getType());
    itemFieldsDataMap.putDouble(
        Constants.Wearable.EXTRA_ORDER_NUM,
                    watchItem.getOrderNum());
    itemFieldsDataMap.putString(
        Constants.Wearable.EXTRA_TITLE,
                        watchItem.getTitle());
    itemFieldsDataMap.putString(
        Constants.Wearable.EXTRA_SUBTITLE,
                        watchItem.getSubtitle());
    itemFieldsDataMap.putString(
        Constants.Wearable.EXTRA_DESCRIPTION,
                    watchItem.getDescription());
    itemFieldsDataMap.putStringArrayList(
        Constants.Wearable.EXTRA_DATES,
                        watchItem.getDates());
    itemFieldsDataMap.putString(
    Constants.Wearable.EXTRA_START_DAY, watchItem
                                .getStartDay());
    itemFieldsDataMap.putString(
        Constants.Wearable.EXTRA_END_DAY,
                        watchItem.getEndDay());
    itemFieldsDataMap.putString(
        Constants.Wearable.EXTRA_START_MONTH,
                    watchItem.getStartMonth());
    itemFieldsDataMap.putString(
        Constants.Wearable.EXTRA_END_MONTH,
                    watchItem.getEndMonth());
    for (int j = 0; j < watchItem.getImages().size(); j++) {
        itemImagesDataMap.putAsset(
            Constants.Wearable.EXTRA_IMAGE + j,
        WearableUtils.createAssetFromBitmap(
                    watchItem.getImages().get(j)));
    }
    rootItemDataMap.putDataMap(
        Constants.Wearable.ROOT_ITEM_FIELDS,
                            itemFieldsDataMap);
```

```
            rootItemDataMap.putDataMap(
                Constants.Wearable.ROOT_ITEM_IMAGES,
                                itemImagesDataMap);
            dataMapList.add(rootItemDataMap);
        }

        if (connectionResult.isSuccess() &&
            mGoogleApiClient.isConnected() && !dataMapList.isEmpty())
            {
            PutDataMapRequest dataMapRequest =
            PutDataMapRequest.create(
                       Constants.Wearable.WATCH_ITEM_PATH);
            dataMapRequest.getDataMap().putDataMapArrayList(
                Constants.Wearable.EXTRA_ITEMS, dataMapList);
            dataMapRequest.getDataMap().putLong(
                Constants.Wearable.EXTRA_TIMESTAMP,
                                new Date().getTime());
            PutDataRequest request = dataMapRequest.
                                    asPutDataRequest();
            request.setUrgent();
            DataApi.DataItemResult result =
                Wearable.DataApi.putDataItem(mGoogleApiClient,
                request).await();
            if (!result.getStatus().isSuccess()) {
                Log.e(TAG, "Error sending data using DataApi (error
                    code = %d)" + result.getStatus().getStatusCode());
            } else {
                Log.e(TAG, "sending data using DataApi OK");
            }
        } else {
            Log.e(TAG, String.format(
                Constants.Wearable.GOOGLE_API_CLIENT_ERROR_MSG,
                    connectionResult.getErrorCod()));
        }
        mGoogleApiClient.disconnect();
    }
```

Listato 3.13: Codice di uno dei metodi della classe WearableContentServiceImpl dello Smartphone

Per eseguire il fetching delle immagini da Rete è stata utilizzata una libreria esterna di supporto chiamata *Glide*. Questo efficiente *image loader* permette in poche righe di codice di scaricare l'immagine e manipolarla secondo le esigenze dello sviluppatore. Nel caso della Wearable App, per tentare di non

incorrere in un'eccezione `OutOfMemory` per l'utilizzo eccessivo della memoria, occorre limitare di parecchio la risoluzione dell'immagine scaricata e settare delle dimensioni più adatte allo schermo quadrato o rotondo dell'indossabile. Per come è stato progettato Juice, prima viene eseguito il fetching degli Item, ogni oggetto Item avrà al suo interno un campo contenente l'URL dove è stata salvata l'immagine in formato binario. Una seconda operazione si dovrà quindi prendere la briga di scaricare queste immagini. Viene riportato il codice che è stato utilizzato dal ContentManager del telefono per ottenere una lista di Bitmap (immagini) a partire da un Item.

```
private List<Bitmap> getImages(Item item) {
    List<Bitmap> bitmapList = new ArrayList<>();
    List<Image> imageList = item.getImages();
    if (imageList != null) {
        try {
            for (int i = 0; i < imageList.size(); i++) {
                if (i != 1 && i != 2) {
                    Bitmap itemImage = Glide.with(mContext)
                            .load(imageList.get(i).getRealImageUrl())
                            .asBitmap()
                            .override(OVVERRIDE_SIZE , OVVERRIDE_SIZE)
                            .diskCacheStrategy(DiskCacheStrategy.SOURCE)
                            .into(WEAR_IMAGE_WIDTH,
                                    Constants.Wearable.
                                    WEAR_IMAGE_HEIGHT)
                            .get();
                    bitmapList.add(itemImage);
                }
            }
        } catch (InterruptedException e) {
            Log.e(TAG, "Error during bitmap download", e);
        } catch (ExecutionException e) {
            Log.e(TAG, "Error during bitmap download", e);
        }
    }
    if(bitmapList.isEmpty()){
          bitmapList.add(BitmapFactory
          .decodeResource(mContext.getResources(),
              R.drawable.bellini_big_placeholder));
    }
    return bitmapList;
}
```

Listato 3.14: Codice di uno dei metodi della classe ContentManagerImpl

Tali immagini saranno poi impacchettate insieme agli altri campi all'interno dell'oggetto WatchItem e dati in pasto al WearableContentService, che avrà il compito di inviarli.

3.3.8 Testing

Per mancanza di tempo, non sono stati effettuati test utilizzando tool specifici per mettere sotto stress l'applicativo. Tuttavia la fase di QA delle ultime Sprint è stata necessaria e indispensabile per garantirne il corretto funzionamento. La Wearable App è stata sottoposta a review da parte del Product Owner che ha testato la qualità del prodotto seguendo i vari task posti su Trello indicanti le varie feature da sviluppare. Per ogni feature è stata stilata una lista di *bug* da correggere. Quelli di maggiore rilevanza sono stati bug grafici e relativi alla gestione della memoria. Vengono riportati di seguito alcuni esempi di bug individuati durante il testing:

- Image Crop: le immagini sono della stessa forma (rettangolare) di quelle utilizzate su Smartphone, occorre trovare un modo per ridimensionarle a runtime una volta arrivate su Smartwatch creandone una versione di forma quadrata che impedisca al titolo dei vari Item nella CardList di anteporsi all'immagine.

- OutOfMemory: la grande quantità di immagini da scaricare causa un `OutOfMemoryError` durante la trasformazione da Asset a Bitmap una volta che le immagini sono arrivate su Smartwatch. Occorre ridurre la qualità delle immagini prima di inviarle o testare altre soluzioni architetturali.

- Lingua: cambiare lingua delle informazioni da visualizzare a seconda della lingua settata dal sistema Android Wear

- Date: rifattorizzare CardDetailViewModel in modo da evitare di creare Card orizzontali diverse per ogni orario di inizio o fine spettacolo se la data dello spettacolo è la stessa. Nella visualizzazione della lista di date di un evento, associare quindi una data per ogni Card visualizzandone anche orari di inizio e fine.

Conclusioni

Sviluppi Futuri

Grazie a ciò che è stato fin ora sviluppato, l'azienda è entrata a pieno contatto con tutte le tematiche e le questioni relative ad Android Wear. Si è aperto quindi un nuovo mondo per Mango, che può progredire e valutare di entrare nel mercato del Wearable Computing e dello sviluppo su device indossabili, creando soluzioni specifiche col fine di aumentare la soddisfazione del cliente.

Si prevede in futuro di sviluppare le feature designate nella fase di Analisi che non si ha avuto il tempo di implementare. Prima tra tutti è l'introduzione della parte relativa alla lettura dei ticket degli spettacoli acquistati direttamente da Smartwatch con anche la possibilità di generare un QR-Code da mostrare in biglietteria. Un altro sviluppo prioritario riguarda la volontà di conformarsi ai nuovi standard grafici e tecnologici della nuova versione della piattaforma Android Wear 2.0, ampliando quindi la connettività non solo al collegamento Bluetooth con il telefono ma anche a chiamate dirette al Web Service tramite Wi-Fi e dando all'intera applicazione una nuova veste grafica grazie all'utilizzo dei nuovi elementi grafici messi a disposizione.

Valutazioni finali

Si è portati a pensare che un Developer già abile nello sviluppo su piattaforma Android si trovi agevolato nello sviluppo su Android Wear. Imparare ad usare il framework di Android Wear non è tuttavia scontato anche per i più esperti.

Si è trattato di un progetto complesso da realizzare, soprattutto per mancanza di documentazione e di esperienza pregressa da parte del Team di sviluppo. Le maggiori difficoltà sono state riscontrate nella comprensione del funzionamento della comunicazione e della gestione della memoria dell'indossabile. Questi e altri problemi minori hanno rallentato il processo di sviluppo, toglien-

do del tempo per l'implementazione di tutte le feature descritte nell'analisi dei requisiti.

Al di là di queste difficoltà, il risultato è stato ottimo. E' stato creato un sistema funzionante, già pronto per essere rilasciato al cliente ed in grado di rendere fruibili all'utente finale importanti informazioni relative agli spettacoli e agli eventi in programma. L'impatto sul mondo aziendale mi ha dato la possibilità di mettere in pratica tutti i concetti acquisiti durante il corso di studi. Ho avuto inoltre l'opportunità di entrare a contatto con nuove tecnologie ed approcci innovativi alla progettazione e alla programmazione del Software, come C4, VIPER e la Clean Architecture. Non è mancato un rafforzamento delle capacità di lavoro di gruppo; questo grazie alla forte collaborazione del Team e alla gestione del processo di sviluppo con un'ottica orientata a Scrum e ai metodi Agili. Sono quindi molto soddisfatto dell'attività svolta in azienda e della stesura di questa tesi.

Ringraziamenti

Ringrazio Claudio Buda per la professionalità e l'umanità con cui mi ha accompagnato nel percorso di tirocinio. Era mio scopo portare a termine una tesi che mi lasciasse un'esperienza tangibile e da poter riutilizzare nei contesti lavorativi futuri. Grazie a lui l'obbiettivo è stato raggiunto con successo.
Ringrazio il prof. Mirko Viroli per avermi trasmesso in passato un grande interesse per la Programmazione ad Oggetti e ora per la collaborazione e per tutti i consigli che mi sono stati dati nella stesura della Tesi.
Ringrazio gli ottimi amici incontrati durante questo percorso universitario, i quali hanno sempre contribuito a tenere viva la mia sconfinata passione per l'Informatica. Infine ringrazio tutte le persone che mi vogliono bene e che mi sono state vicine in questi anni senza le quali non avrei mai potuto affrontare con lo spirito giusto una facoltà di questa portata.

Bibliografia

[1] *The Internet of Things*
https://mitpress.mit.edu/books/internet-things

[2] *A Brief History of the Internet of Things*
http://www.dataversity.net/brief-history-internet-things/

[3] Internet of Things (IoT)
http://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT

[4] *The Internet of Things - How the Next Evolution of the Internet Is Changing Everything*
http://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf

[5] *That 'Internet of Things' Thing*
http://www.rfidjournal.com/articles/view?4986

[6] *Online all the time – Average British household owns 7.4 internet devices*
https://www.theguardian.com/technology/2015/apr/09/online-all-the-time-average-british-household-owns-74-internet-devices

[7] *Wearable Computing*
http://www.webopedia.com/TERM/W/wearable_computing.html

[8] *Wearable Computing and its Application*
https://pdfs.semanticscholar.org/0199/ceeb9d7bd66815538a1b62deffd1da55bc86.pdf

[9] *A brief history of wearable computing*
https://www.media.mit.edu/wearables/lizzy/timeline.html

[10] *La storia dei dispositivi indossabili*
http://www.fastweb.it/smartphone-e-gadget/la-storia-dei-dispositivi-indossabili/

[11] *Wearable tech: A brief history and a look into the future*
http://www.androidauthority.com/wearable-computing-history-238324/

[12] *Storia degli Smartwatch*
http://www.portalesmartwatch.it/storia-degli-smartwatch/

[13] *Motorola, LG announce upcoming Android Wear smartwatches*
http://www.theverge.com/2014/3/18/5522340/motorola-lg-announce-upcoming-android-wear-smartwatche

[14] *Android Wear for iOS gives iPhone owners more smartwatch options*
http://www.cio.com/article/2978194/smartwatches/android-wear-for-ios-gives-iphone-owners-more-smartwatch-options.html

[15] *Verso l'autonomia degli smartwatch Android*
http://mangomobi.com/verso-lautonomia-degli-smartwatch-android/

[16] *Android Clean Architecture*
http://luboganev.github.io/blog/clean-architecture-pt1/

[17] *Essere Agile in un'azienda MonoTeam*
http://mangomobi.com/essere-agile-in-unazienda-monoteam/

[18] The Definitive Guide to Scrum: The Rules of the Game

[19] *Getting Started with C4*
http://codermike.com/starting-c4

[20] Simon Brown - The Art of Visualizing Software Architecture

[21] *The Clean Architecture*
https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html

[22] *Architecting iOS Apps with VIPER*
https://www.objc.io/issues/13-architecture/viper/