UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Elaborato d'esame in Software Architecture Design

CovidSafe

Anno Accademico 21/22

Studenti

Luca Torre
Luigi Garofalo
Niccolò Salvatore
Nunzio Francesco Riccetti

Indice

Introduzione

Lo scopo di questo documento è di raccogliere, analizzare e definire le esigenze ad alto livello di astrazione degli utenti e le features del sistema.
Alla luce dello stato pandemico generato dalla diffusione del virus SARS-CoV-2 è emersa la necessità di impiegare ogni mezzo possibile, incluso quello digitale, per contrastare la propagazione incontrollata dell'agente patogeno. Tale virus presenta una carica virale molto alta e modalità di contagio particolarmente efficaci (si diffonde per via aerea), le quali rendono urgenti misure di contenimento e monitoraggio della popolazione da esso affetta o minacciata, al fine di ridurre la trasmissione dell'agente patogeno.
Date queste premesse, si è deciso di realizzare un'applicazione per dispositivi mobili che offra dei servizi volti al monitoraggio ed alla segnalazione di casi Covid-19 alle autorità competenti ed alle persone che hanno intrattenuto contatti con individui contagiati, nonché al monitoraggio di luoghi chiusi al fine di rilevarne il numero di individui presenti in tempo reale e funzioni a fini commerciali (informazioni generiche, possibilità di prenotazione, etc.).

1. Processo di Sviluppo

Per la realizzazione del progetto si è proceduto, quando possibile, a predisporre giornate lavorative in presenza, sostituite all'occorrenza da riunioni telematiche tramite la piattaforma Teams.
Il processo di sviluppo adottato può essere generalmente considerato UP (Unified Process), ovvero un processo di sviluppo **iterativo** ed **evolutivo**, sebbene siano stati incorporati molti cerimoniali del framework per lo sviluppo agile **SCRUM** soprattutto per aumentare il grado di controllo sull'intero processo software.
Generalmente, un processo iterativo a consegna incrementale tende, a far aumentare l'entropia del progetto rendendolo pressocché invisibile, ovvero diventa molto difficile misurare lo stato di avanzamento dello stesso.

Per quanto riguarda, invece, il processo SCRUM, si è deciso di decidere i ruoli all'interno del team:

- Scrum Master: Luigi - Colui che decide cosa deve essere fatto, quali funzionalità deve avere un prodotto e quando rilasciarle sul mercato.
- Back-End Developers: Nunzio e Niccolò - Coloro che fanno in modo che il software sia robusto, funzionale, affidabile e veloce sulla base di un codice ben costruito. Lavorano principalmente su lato server.
- Front-End Developer: Luca - Colui che fa in modo che l'esperienza utente sia svolta senza intoppi con una navigazione fluida e veloce. Conosce i fondamenti dello User Experience Design. Lavora principalmente su lato client.

Il primo passo è stato quello di definire due giorni di workshop dei requisiti.

- Il primo giorno è stato dedicato ad attività di **brain storming** per comprendere al meglio e definire quali fossero tutti i possibili requisiti funzionali e non funzionali. L'attività di confronto ha portato alla stesura di un elenco dei casi d'uso, dei principali requisiti non funzionali e vincoli di progetto, al fine di imporre un certo grado di ordine e schematicità rispetto alla più caotica attività di brainstorming. Tali requisiti sono stati ordinati per

rischio e **priorità** in maniera tale che fosse fin da subito chiaro su quale sottoinsieme di requisiti lavorare inizialmente.
- Il secondo giorno, quindi, sono stati specificati i requisiti definiti nella giornata precedente in forma testuale e grafica (diagramma dei casi d'uso, documenti di specifica dei requisiti).

Successivamente, poiché dall'analisi dei requisiti è emerso che il principale fattore di rischio fosse collegato al tracciamento in real time dei dispositivi vicini all'utente (con vincoli di precisione della misura molto stringenti) si è voluto procedere, al fine di valutare la fattibilità del progetto, ad uno studio approfondito delle tecnologie disponibili sul mercato (riportata di seguito nel paragrafo "panoramica delle tecnologie").

Stabilita quale fosse la tecnologia più adatta alle nostre esigenze (Bluetooth), si è proceduto ad una fase di **prototipazione rapida** del tipo "usa e getta" per acquisire confidenza con le potenzialità ed i limiti degli strumenti del framework scelto (AltBeacon), oltre che ovviamente per valutare la fattibilità stessa del progetto.
Questa fase è durata, tra implementazione e studio teorico, circa 3 giorni.
Si è successivamente selezionato il caso d'uso che attraversasse trasversalmente l'intera architettura del sistema, ovvero la dichiarazione della propria positività al virus.
La scelta del caso d'uso è stata effettuata valutando una serie di fattori, ovvero che:

1. Il principale scopo dell'applicazione è quello di rilevare ed identificare i dispositivi che si trovano a distanza minore di due metri e di consentire all'utente di poter dichiarare la propria positività, mentre le altre funzionalità sono secondarie rispetto all'obiettivo sopra descritto. Dunque, questo caso d'uso costituisce il nucleo fondante dell'intero progetto.
2. Poiché il caso d'uso scelto implica lo sviluppo del sottosistema di rilevamento dei contatti, di persistenza degli identificativi e di invio delle notifiche di avvenuta positività, esso attraversa trasversalmente l'intera architettura software.
3. L'inesperienza con la tecnologia Bluetooth BLE ci ha portato a considerare più rischioso questo caso d'uso, non potendoci affidare al nostro bagaglio conoscitivo.
4. I vincoli più stringenti si applicano in questo caso d'uso (privacy, tempo di persistenza dei dati di 14 giorni).
5. Era noto il fatto che le attività in background su dispositivi Android fossero più o meno limitate a seconda del modello di smartphone e della versione del sistema operativo.

È stato redatto inoltre un **product backlog** che raggruppasse in maniera sintetica tutti i requisiti. In seguito, sono stati stabiliti quattro sprint ciascuno di durata di due settimane al fine di soddisfare tutti i requisiti di interesse e per implementare il caso d'uso scelto.
Ad ogni sprint è stato associato uno **Sprint Backlog** (tramite Trello) costituito da un sottoinsieme (aggiornato dinamicamente) dei requisiti del Product Backlog, usato come linea guida nell'implementazione ed aggiornato opportunamente a seconda delle esigenze (cosa che accadeva giornalmente).

- Il primo sprint è stato dedicato alla realizzazione del modulo di trasmissione e della ricezione del segnale Bluetooth attraverso la libreria AltBeacon.
- Nel secondo sprint è stato costruito il sistema di persistenza dei dati di interesse su dispositivo.
- Il terzo sprint è stato dedicato alla realizzazione del modulo di comunicazione con il lato server del sistema software.
- Il quarto sprint è stato dedicato alla costruzione della componente server del sistema software.

Ciascuna iterazione è stata organizzata con la seguente struttura:

- All'inizio della prima settimana di ciascuna iterazione, ci si è concentrati sulla progettazione del sottosistema da sviluppare, delineandone il design e l'architettura. Questa fase durava generalmente due giorni, oltre i quali si procedeva alla fase di implementazione.
- I 3 giorni centrali della settimana erano dedicati pienamente all'implementazione. Ove possibile, sono stati condotti dei test sulle singole classi.
- Il sabato mattina veniva effettuata una riunione riepilogativa ed organizzativa in cui veniva valutato il lavoro svolto durante la settimana ed i risultati raggiunti. Se alcuni task non fossero stati completati sarebbero stati spostati alla settimana successiva.
- Alla fine dell'ultima settimana dello Sprint veniva effettuata una **Sprint Review** per valutare il lavoro svolto durante lo Sprint e spostare eventualmente un subset dei requisiti ad un'iterazione aggiuntiva.

Infine, ogni mattina si è tenuto giornalmente un meeting della durata di circa 15 minuti al fine di riepilogare ed organizzare il lavoro da portare a termine in giornata, prendendo come riferimento lo Sprint Backlog.
Il testing è stato effettuato a vari gradi di granularità. Generalmente ogni modulo è stato opportunamente testato partendo da diverse condizioni iniziali al fine di valutarne la correttezza e la robustezza. Con l'avanzare del progetto, la fase di testing si è estesa poiché ovviamente è aumentata la correlazione tra i moduli.
Infine, l'intero sistema è stato testato alla ricerca di eventuali bug o problemi che potevano essere sfuggiti nella fase di implementazione.

1.1 Panoramica delle tecnologie

L'applicazione necessita di un meccanismo di rilevamento della posizione relativa dei dispositivi che la circondano. Per tal motivo sono state analizzate varie tecnologie ed è stata successivamente scelta quella che meglio si adattasse alle esigenze richieste presentando il miglior rapporto vantaggi/svantaggi in termini di performance, fruibilità, costi, etc.

Si definisce quindi:

- Localizzazione assoluta: la posizione calcolata rispetto ad un sistema di riferimento univoco e fisso, ad esempio attraverso latitudine e longitudine.
- Localizzazione relativa: la posizione è calcolata in termini di distanza rispetto ad un altro dispositivo.

Inoltre, viene fatta distinzione tra sistemi:

- Indoor: per localizzare oggetti o persone all'interno di un edificio.
- Outdoor: per localizzare oggetti o persone all'esterno.

Poiché bisognava ottenere un'applicazione basata su di un sistema di tracciamento relativo con buone prestazioni sia in ambito indoor che outdoor, sono state analizzate di seguito le varie tecnologie disponibili sul mercato:

1.1.1 Global Positioning System (GPS)

La **geo localizzazione** è il processo tramite il quale è possibile mettere in relazione una certa informazione con un punto specifico della superficie terrestre, individuato tramite metodi di localizzazione (quindi il reperimento delle **coordinate latitudinali e longitudinali**) di un apparato elettronico, sfruttandone le caratteristiche fisiche o ricevendo informazioni dettagliate dallo strumento stesso.
Il **GPS** è ad oggi il sistema più preciso, Il suo **principio di funzionamento** si basa su un metodo di posizionamento sferico, che parte dalla misura del tempo impiegato da un segnale radio a percorrere la **distanza satellite-ricevitore**. In particolare, ogni satellite trasmette in continuazione due informazioni chiave: la propria **posizione** e **l'orario esatto** di trasmissione del messaggio. Queste informazioni servono al ricevitore per riuscire a calcolare la propria posizione triangolandola rispetto a quella inviata dai tre satelliti. E quindi necessario l'intervento di un quarto satellite che serve al ricevitore per sincronizzare l'orologio interno, in modo che i calcoli effettuati dal ricevitore siano corretti.
Per questo motivo il quarto trasmettitore invia periodicamente l'orario corretto, e permette quindi di calcolare la distanza degli altri satelliti in base allo scarto di tempo necessario alla trasmissione del messaggio.
Vantaggi:

1. **grande precisione all'aperto**: grazie all'elevato numero di satelliti che ruotano attorno alla terra, l'accuratezza del sistema GPS è calcolata intorno a 1-5 metri, ma dipende dalla posizione.

Svantaggi:

1. **scarsa precisione in ambito civile**: il sistema offre maggiore precisione per gli ambiti militari rispetto a quelli civili. La precisione in ambito civile, infatti, varia **tra i 10 ed i 100 metri.**

2. **Inefficacia in luoghi chiusi** (contesto indoor): ciò avviene a causa della **mancanza di segnale tra i satelliti ed il dispositivo** di ricezione ed alle gravi interferenze di rimbalzo che si hanno tra gli edifici.
3. **Alto consumo di batteria:** ciò avviene a causa dell'**impiego massiccio di risorse hardware** e di connessioni ai satelliti. Inoltre, la relativa lentezza con la quale il dispositivo riceve la posizione iniziale rende questa tecnologia inutilizzabile per applicazioni che richiedono prolungata durata ed un discreto grado di reattività.

Dal momento che tale tecnologia si basa inoltre su quello che è stato definito posizionamento assoluto, sarebbe stato necessario elaborare un algoritmo in grado di ricavare le posizioni relative tra i dispositivi interagenti mediante l'applicazione, il che avrebbe aumentato i tempi di latenza.
Per tali ragioni, unitamente al fatto che i requisiti imponessero un tracciamento indoor ed outdoor e presupponessero una durata della batteria ragionevolmente lunga, **questa soluzione è stata scartata.**

1.1.2 Localizzazione tramite triangolazione da rete cellulare

La localizzazione cellulare è meno precisa di quella GPS, ma è praticabile anche in quei dispositivi mobili che non sono dotati di ricevitore GPS.
Questo metodo di calcolo della posizione sfrutta le **antenne radio** a cui ogni cellulare è connesso per la **ricezione GSM (Global System for Mobile Communications)** o **UMTS (Universal Mobile Telecommunication System)** per i servizi di telefonia mobile.
Le antenne sono posizionate sulle cosiddette celle, le quali hanno un raggio di trasmissione che dipende da molti fattori, i più importanti dei quali sono la **densità di terminali connessi** contemporaneamente e la **potenza trasmissiva**.
Il fattore di densità delle celle è molto importante per determinare la posizione di un ricevitore in maniera precisa: tutti gli algoritmi sono infatti vincolati dal numero di celle che possono rilevare e dalla potenza di segnale ricevuto.

Vantaggi:

1. maggiori possibilità di diffusione, anche su dispositivi datati.

Svantaggi:

1. **Precisione troppo variabile**: questo sistema risulta altamente inaffidabile, specie in contesti in cui la precisione è un fattore fondamentale ai fini del corretto funzionamento dell'applicazione.
2. **Inefficacia in ambienti chiusi**: ancora una volta, questo sistema risulta del tutto impraticabile in contesti che contemplano il posizionamento di tipo indoor.

Gli svantaggi appena elencati rendono questa metodologia **inutilizzabile per gli scopi descritti.**

1.1.3 Bluetooth

Il **Bluetooth**, studiato anche nella sua implementazione **Low Energy (BLE)** e nell'ultima versione disponibile (**5.1**), offre una pico-rete di 100 metri e deve la sua diffusione al grande sviluppo che ha reso possibile la connettività a bassi costi tra più dispositivi provvisti di tale tecnologia.
Esso può essere utilizzato idealmente sia per **soluzioni di tracciamento assoluto** sia per **soluzioni di tracciamento relativo** (attraverso l'invio di messaggi in broadcast, dal punto di vista del trasmettitore, e di ricezione opportunamente filtrata, dal punto di vista del ricevitore).
Tale **tecnologia è applicabile sia in ambienti chiusi che aperti**.

Vantaggi:

1. **Setup e configurazione veloce**: la sempre maggiore attenzione dedicata a questa tecnologia negli anni ha fatto sì che prosperassero un buon numero di **framework** attraverso cui configurare a livello software i chipset.
2. **Basso consumo di batteria**: i dispositivi Low Energy (chiamati **Beacons**) non sono altro che **chipset Bluetooth progettati** appositamente **per ridurre il consumo della batteria**.
3. **Possibilità di determinare la distanza relativa tra dispositivi interagenti**: tale distanza può essere facilmente stimata misurando la potenza del segnale ricevuto ed effettuando una serie di elaborazioni numeriche sulla stessa.
4. **Costi**: i beacons costituiscono una tecnologia di prossimità il cui costo di produzione risulta essere relativamente basso.

Svantaggi:

1. **Performance**: **l'accuratezza** del sistema dipende dal range di azione del segnale e dalla sua opportuna configurazione. Generalmente, infatti, maggiore è l'area coperta in fase di scansione, minore è l'accuratezza della misura.

Dopo un'attenta analisi è emerso che:

1. Il raggio di azione utile agli scopi della nostra applicazione è considerevolmente ridotto (circa 2 metri), per cui è possibile **limitare la degradazione delle performance** fino ad un grado accettabile.
2. La necessità di progettare un sistema di tracciamento relativo ci ha concesso di **ignorare i costi relativi alla costruzione di un'infrastruttura dedicata** di sensori Bluetooth (costi presenti nel caso in cui fosse stato necessaria invece un sistema di tracciamento assoluto).
3. L'**alta diffusione** di questi componenti hardware all'interno della maggioranza degli smartphones ed il basso impatto energetico consentono una copertura pressoché totale dell'utenza target senza particolari restrizioni di utilizzo dello smartphone (in termini energetici).

4. L'**ampio numero di framework e l'esaustiva documentazione** di molti di essi consente un **rapido apprendimento**, abbattendo tempi di sviluppo e di implementazione.

Per tali ragioni quindi si è scelto di utilizzare il **Bluetooth (nella sua variante Low Energy)** come tecnologia di tracciamento.

1.2 Standard Bluetooth e Bluetooth Low Energy

Bluetooth è uno **standard tecnico-industriale di trasmissione dati per reti personali** senza fili (WPAN: Wireless Personal Area Network). Fornisce un metodo standard, economico e sicuro per scambiare informazioni tra dispositivi diversi attraverso una frequenza radio sicura, a corto raggio, in grado di ricercare i dispositivi coperti dal segnale radio entro un raggio di qualche decina di metri, mettendoli in comunicazione tra loro.
Con la pubblicazione dello standard **Bluetooth 5.1 è diventato possibile** individuare la direzione di un dispositivo con la precisione del centimetro, grazie a innovativi chipset Bluetooth, colloquialmente definiti **Beacon**. I **servizi di localizzazione Bluetooth** sono di due tipi: **soluzioni di prossimità** e **sistemi di posizionamento.**

I **sistemi di posizionamento** si suddividono a loro volta in**:**

- **real-time locating systems** (**RTLS**).
- **indoor positioning systems** (**IPS**), cioè sistemi di posizionamento negli spazi chiusi.

Le **soluzioni di prossimità** consentono di determinare quando due dispositivi sono vicini ed approssimativamente a quale distanza.

Questa è la soluzione adottata ai fini dello sviluppo del progetto in questione.

1.2.1 Principio di Funzionamento

Attraverso questa versione dello standard è possibile misurare, in orizzontale, l'**angolo di arrivo** (**AoA**) e l'**angolo di partenza** (**AoD**) di un dispositivo in movimento, mentre la misurazione in verticale è relativa all'altitudine (**elevation**). La variazione sulla misura è di al massimo 5°. Le misure dell'angolo di arrivo e dell'angolo di partenza si basano su due tecniche diverse ed entrambe richiedono che uno dei due dispositivi disponga di antenne a schiera (array). L'array deve essere presente nel dispositivo ricevente quando va misurato l'angolo di arrivo (AoA), mentre deve essere presente nel dispositivo trasmittente quando va misurato l'angolo di partenza (AoD).

1.2.2 Caratteristiche Bluetooth Low Energy (BLE)

Il BLE utilizza la **banda 2.4 GHz**, ma resta **costantemente in modalità riposo** fino a quando non viene avviata una connessione. I **tempi di connessione** sono molto **ridotti** rispetto al

normale Bluetooth. Tutti questi elementi rendono il BLE efficace per la maggior parte degli utilizzi negli attuali smartphone, dagli smartwatch all'interazione con oggetti dell'Internet of Things (IoT), ma anche e soprattutto per **applicazioni di tracciamento.**

1.2.3 Protocol Data Unit BLE

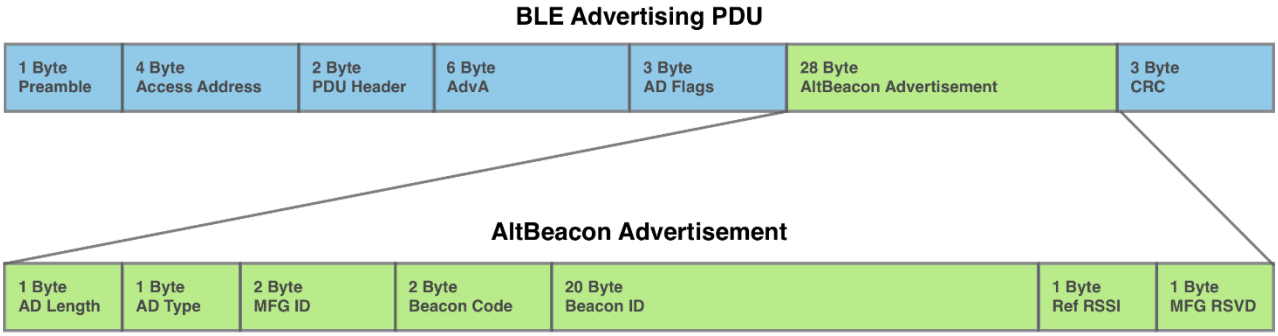
L'informazione standard dei beacon consiste in un **UUID** (**U**niversally **U**nique **ID**entifier) ed in due interi chiamati **minor** e **major** che sono valori assegnati per identificare i dispositivi con una accuratezza maggiore rispetto all'utilizzo del solo UUID.
Spieghiamo il loro utilizzo con un semplice esempio:
se un'azienda decide di installare dei dispositivi beacon all'interno dei propri negozi e crea un'applicazione che può comunicare con l'utente una volta che questi sia entrato in un negozio specifico, essa definirebbe:

- un UUID che è unico per la loro applicazione e per i beacon dentro i negozi (l'UUID rappresenta l'azienda).
- All'interno delle strutture, ciascuna contrassegnata da un valore unico di major, verrebbe configurato ogni dispositivo a usare un differente valore minor.

Con le informazioni trasmesse da ciascun beacon un'applicazione, dopo averle lette, può dire quanto vicino (o lontano) siano rispetto al telefono ed effettuare delle azioni, come inviare avvisi, offrire sconti, accendere o spengere le luci, aprire la porta, e così via.
Un beacon è quindi un dispositivo BLE che trasmette piccole quantità di dati ad intervalli di tempo regolari.

- A **livello hardware**, i beacon sono dispositivi BLE-compatible che trasmettono dati con il protocollo Bluetooth 5.1.
- A **livello software**, i beacon sono messaggi inviati da questi dispositivi trasmittenti, che sono poi rilevati e processati da un dispositivo ricevente, come un'applicazione mobile.

Nel dettaglio il pacchetto inviato è costituito nel seguente modo (sono state prese in considerazione le specifiche della libreria utilizzata nel progetto, ma le variazioni rispetto alle altre famiglie software sono minime):



Il pacchetto di advertisement è incapsulato in un **PDU** (**P**rotocol **D**ata **U**nit), ovvero l'unità d'informazione scambiata tra due entità pari in un protocollo di comunicazione di un'architettura di rete a strati.
Il **pacchetto di advertisement** (specifico per ogni azienda manifatturiera), strutturato secondo il protocollo AltBeacon, presenta i seguenti campi:
AD Length: lunghezza del pacchetto di Advertisement,1 byte.
AD Type: specifica il tipo di pacchetto di advertisement, 1 byte.
MFG ID: rappresentazione little endian dell'identificativo della compagnia manifatturiera che ha prodotto il beacon (Google, Apple, Eddystone, etc.), così come registrato presso il Bluetooth SIG Database, 2 byte.
Beacon Code: rappresentazione big endian del valore 0xBEAC, nel caso di AltBeacon.
Reference RSSI: rappresentazione dell'intensità media del segnale ad un metro dal trasmettitore, 1 byte.
MFG RSVD: bit riservati, interpretabili diversamente a seconda dell'azienda manifatturiera che ha prodotto il beacon.

1.2.4 Impatto sulla batteria

Bluetooth Low Energy (si ricorda essere una tecnologia e non uno standard) è progettato per abilitare dispositivi con **basso consumo energetico**. Studi certificati hanno dimostrato che periferiche che utilizzano Beacon di prossimità funzionano per 1-2 anni con una batteria a moneta da 1.000 mAh. Ciò è possibile grazie all'**efficienza energetica** del protocollo Bluetooth Low Energy che trasmette solo piccoli pacchetti rispetto al classico Bluetooth, il che lo rende adatto anche per l'audio e per dati su banda larga.
Per contro, una scansione continua per gli stessi Beacon in un ruolo centrale può consumare 1.000 mAh in poche ore. I dispositivi Android e iOS hanno anche un **impatto sulla batteria molto diverso a seconda del tipo di scansioni** e del numero di dispositivi Bluetooth Low Energy nelle vicinanze. Con i più recenti chipset e con l'avanzamento del software, il **consumo di energia è diventato trascurabile** negli scenari quotidiani di utilizzo.
Per tale motivo si è scelto, nel corso dello sviluppo dell'applicazione, di perseguire un **compromesso tra efficienza energetica ed efficienza funzionale**, prediligendo un tempo di scansione relativamente ridotto (grazie **all'opportuno settaggio della Potenza del Segnale**, ricavato da studi euristici su scala nazionale) **ed intervalli regolari di scanning**, in maniera tale da non drenare precocemente la batteria del dispositivo.

1.2.5 Stima sulla distanza

Diversi modelli di dispositivi Android ricevono il segnale in maniera differente ed ogni modello potrebbe avere un differente chipset Bluetooth ed una differente antenna, per cui ciascun modello potrebbe ricevere un segnale più o meno forte rispetto ad un altro modello nella medesima posizione relativa alla sorgente. Per far fronte a questo problema, la libreria **AltBeacon** (approfondita al paragrafo successivo) utilizza **differenti formule** per calcolare la distanza, a seconda del modello Android di riferimento. Ciò nonostante, poiché non tutti i

modelli sono coperti dalla libreria, nel caso in cui un modello sia assente, la libreria utilizzerà la formula standard.
Tale formula si basa su **RSSI**, che è stato dimostrato empiricamente essere il miglior predittore di distanza e si basa sulla potenza del segnale.

La distanza, infine, viene calcolata secondo la formula:

$$d = A\left(\frac{r}{t}\right)^{B} + C$$

ove:
d = **distanza** in metri.
r = **RSSI** misurato dal dispositivo.
t = **RSSI di riferimento**, rispetto ad un metro.
A, **B**, **C** = **costanti**.

1.2.6 Variazione della misura

Ogni framework fornisce stime della distanza in metri. I valori sono da considerarsi stime basate sulla potenza del segnale Bluetooth e bisogna aspettarsi un certo grado di incertezza sulla base di:

- **mobilità** del dispositivo.
- **eventuali oggetti che si frappongono** tra sorgente e ricevitore.
- **potenza** del segnale stesso.

A circa un **metro di distanza**, è lecito aspettarsi una stima con variazioni comprese tra i 50 centimetri ed i 2 metri.
Nell'applicazione è stato aggirato il problema di una stima fallace avvantaggiandosi degli studi portati a termine dal team di ricerca e sviluppo dell'applicazione ***Immuni***, utilizzando un valore di potenza (**73 dBm**) tale da garantire una precisione sufficiente ed un certo grado di stabilità in un intorno dei **due metri**.

1.2.7 AltBeacon Library

L'**AltBeacon Library** è la principale **libreria Open Source** per il rilevamento dei beacons in Android. Un'applicazione può infatti richiedere di ricevere notifiche quando uno o più beacons vengono rilevati o scompaiono dal raggio di azione del dispositivo; inoltre, essa può richiedere di ricevere aggiornamenti relativi al rilevamento di uno o più beacons fino ad approssimativamente alla frequenza di 1 Hz. In tal modo **è possibile trasmettere beacons anche in background da dispositivi Android.**
Di default, **la libreria rileva i beacon che rispettano lo standard AltBeacon**, ma può essere opportunamente configurata per rilevare altri formati.

Dalla versione 2.16 in poi sono ampiamente supportati tutti i dispositivi Android 4.3-10.x. Android 10 invece ha un nuovo modello di permessi relativi alla localizzazione, rendendo l'utilizzo del framework leggermente meno immediato.
Ciascun dispositivo Android di versione 4.3+ che monta un chipset Bluetooth Low Energy può rilevare beacons con questa libreria.
Le applicazioni esistenti che utilizzano la versione 2.12 o versioni precedenti della libreria devono aggiornarla al fine di poter ricercare beacon in background su devices che hanno come versione di sistema operativo Android 8+.
Per trasmettere come beacon, **è necessario Android 5+ e un firmware che supporti la Bluetooth Low Energy Peripheral Mode**.
La libreria supporta sia la rilevazione che la notifica dello standard beacon Exposure Notification Service annunciato da Apple e Google in tempi recenti per far fronte alle necessità sorte con la nascita del tracciamento digitale in ottica anti-Covid.
La libreria è inoltre in grado di notificare all'applicazione quando rileva beacons in background compatibili con il formato Eddystone e fornire regolari aggiornamenti relativi al loro rilevamento quando questi sono nelle vicinanze. Di fatto la libreria è in grado di decifrare:

- **Eddystone-UID (identifier frame),**
- **Eddystone-TLM (telemetry frame)**
- **Eddystone-URL (URL frame).**

1.2.8 Meccanismo di identificazione del sistema software

Nel caso dell'applicazione oggetto dell'elaborato, si è scelto di utilizzare il framework messo a disposizione da **AltBeacon**. Le ragioni di tale scelta sono radicate nella:

- **Compatibilità del framework** con i principali sistemi operativi per telefonia mobile (IOS, Android) rispetto ai framework proprietari.
- Relativa **semplicità di utilizzo**.
- **Documentazione** del framework **estensiva e completa** che ci ha consentito un rapido apprendimento dei principi base e delle funzionalità principali.

Si è provveduto a generare un **UUID customizzato** al fine di identificare la suddetta applicazione ed il servizio da essa offerto.
A ciascun utente, durante la fase preventiva di registrazione, viene associata una coppia di interi univoca (valori minor, major) al fine di identificare ciascun utente che usufruisce dell'applicazione. Ciascun valore può essere codificato con **due byte**.

1.3 Prototipazione usa e getta

Una volta stabilita la tecnologia che meglio rispondesse alle esigenze di progetto ed il framework più adatto allo sviluppo della componente core del sistema software (sistema di rilevazione dei contatti in real time), è stata avviata una fase di **prototipazione rapida usa e**

getta volta a valutare la **fattibilità** del progetto ed i maggiori rischi ad esso legati, oltre che per **familiarizzare con gli strumenti** principali forniti dal framework e prevenire l'eventuale cambiamento, per quanto possibile.
In questa fase si è proceduto ad **implementare i soli requisiti funzionali** collegati al modulo di **tracking** (che era quello che risultava essere meno chiaro e più complesso) **ignorando i requisiti non funzionali e** anche la **gestione degli errori**, focalizzando l'attenzione sulle possibili strategie di sviluppo del modulo, individuandone la migliore e cercando, per quanto possibile, di padroneggiare gli strumenti messi a disposizione dal framework.

```
50      public void onCreate() {
51          super.onCreate();
52
53          //TRASMISSION
54          BeaconManager beaconManager = org.altbeacon.beacon.BeaconManager.getInstanceForApplication(this);
55          final Beacon beacon = new Beacon.Builder()
56                  .setId1("aaaaaaaa-bbbb-bbbb-aaaa-123456789aaa")
57                  .setId2("1")//UTILIZZIAMO MINOR E MAJOR IN COPPIA PER CREARE UN IDENTIFICATIVO UNIVOCO PER OGNI UTENTE
58                  .setId3("2")
59                  .setManufacturer(0x0118)
60                  .setTxPower(-59)
61                  .setDataFields(Arrays.asList(new Long[] {0l}))
62                  .build();
63          BeaconParser beaconParser = new BeaconParser()
64                  .setBeaconLayout("m:2-3=beac,i:4-19,i:20-21,i:22-23,p:24-24,d:25-25");
65          BeaconTransmitter beaconTransmitter = new BeaconTransmitter(getApplicationContext(), beaconParser);
66          beaconTransmitter.startAdvertising(beacon, new AdvertiseCallback() {
67              @Override
68              public void onStartSuccess(AdvertiseSettings settingsInEffect) {
69                  super.onStartSuccess(settingsInEffect);
70                  logToDisplay("I'm trasmitting beacon "+ beacon.getManufacturer());
71              }
72
73              @Override
74              public void onStartFailure(int errorCode) {
75                  super.onStartFailure(errorCode);
76              }
77          });
```

In prima battuta si è proceduto a generare un oggetto di tipo Beacon, la cui istanza rappresenta il chipset hardware Bluetooth ed incapsula informazioni manipolabili dal programmatore quali:

- **Potenza del segnale**
- **Beacon ID** (rappresentato da **ID1, ID2, ID3** che identificano rispettivamente l'UUID del servizio, il minor ed il major)
- **Codice dell'azienda manifatturiera** (attraverso il metodo **setManufacturer**)
- **BeaconLayout** (specifica **la divisione in byte della PDU**, che è unica per ogni azienda manifatturiera).

In questo modo, con l'ausilio della documentazione ufficiale, è stato possibile capire meglio come manipolare le informazioni ed i gradi di libertà garantiti dal framework nonché il meccanismo di trasmissione del segnale.

```
@Override
public void onBeaconServiceConnect() {

    final ArrayList prova = new ArrayList<Integer>();

    RangeNotifier rangeNotifier = new RangeNotifier() {
        @Override
        public void didRangeBeaconsInRegion(Collection<Beacon> beacons, Region region) {
            if (beacons.size() < 2) {
                Log.d(TAG, "didRangeBeaconsInRegion called with beacon count:  "+beacons.size());
                Beacon firstBeacon = beacons.iterator().next();


                if(firstBeacon.getDistance()<2){

                    prova.add(firstBeacon.getId2());
                    prova.add((firstBeacon.getId3().toInt()));
                    logToDisplay(prova.get(0).toString());
                    logToDisplay(prova.get(1).toString());
                }
                else{
                    logToDisplay("fuori range");
                }


                System.out.println("test dell'ID DEI BEACON" + firstBeacon.getId1());
                System.out.println("dati ottenibili : " +
                        "ID " + firstBeacon.getId1() +
                        "MAJOR " + firstBeacon.getId2() +
                        "MINOR " + firstBeacon.getId3() +
                        "BLUETOOTHADDRESS" + firstBeacon.getBluetoothAddress() +
                        "TYPECODE " + firstBeacon.getBeaconTypeCode() +
                        "MANUFACTURER "+ firstBeacon.getManufacturer() +
                        "MEASURAMENTCOUNT "+ firstBeacon.getMeasurementCount() +
```

In seconda battuta è stata implementato il modulo di ricezione che ci ha consentito, ancora una volta, di comprendere nel dettaglio il suo meccanismo e le peculiarità dei metodi utilizzati.
In particolare, è stato esplorato il processo di rilevazione della distanza testandone i limiti, reattività e la precisione.

```
beaconManager.setEnableScheduledScanJobs(true);
beaconManager.setBackgroundBetweenScanPeriod(0);
beaconManager.setBackgroundScanPeriod(1100);
Log.d(TAG, "setting up background monitoring for beacons and power saving");
Region region = new Region("backgroundRegion",null, null, null);
regionBootstrap = new RegionBootstrap(this, region);
backgroundPowerSaver = new BackgroundPowerSaver(this);
```

2. Strumenti utilizzati

In questo capitolo verranno presentati i principali strumenti adoperati al fine della realizzazione dell'intero progetto.

2.1 Spring

Spring è il più popolare framework per lo sviluppo di applicazioni Java Enterprise ed è un framework leggero: grazie alla sua struttura estremamente modulare è possibile utilizzarlo nella sua interezza o solo in parte, senza stravolgere l'architettura del progetto.
Fornisce, inoltre, una serie completa di strumenti per gestire la complessità dello sviluppo software, fornendo un approccio semplificato sia ai più comuni problemi di sviluppo (accesso ai database, gestione delle dipendenze, etc.) che di testing.
Il framework Spring ha una struttura modulare:

- Il modulo core che fornisce le funzionalità fondamentali del framework e si basa su Dependency Injection, Programming, la gestione delle transazioni, la struttura di applicazioni Web, l'accesso ai dati, la messaggistica, i test e altro.
- Altri moduli facoltativi: dalla configurazione alla sicurezza, dalle web app ai big data.

Spring permette la progettazione e sviluppo di applicazioni web grazie al suo modulo Spring MVC il quale mette a disposizione le funzionalità core di Spring su un pattern architetturale MVC.
Segue una spiegazione dei principi basilari su cui si basa Spring.

2.1.1 Inversion of Control (IoC)

L'Inversion of Control è un principio architetturale basato sul concetto di invertire il controllo del flusso di sistema rispetto alla programmazione tradizionale.
Nella Programmazione tradizionale lo sviluppatore definisce la logica del flusso di controllo, specificando le operazioni di creazione, inizializzazione degli oggetti ed invocazione dei metodi.
Nell' Inversion of Control si inverte il control flow, facendo in modo che non sia più lo sviluppatore a doversi preoccupare di questi aspetti, ma il framework, che reagendo a qualche "stimolo" se ne occuperà per suo conto.

2.1.2 Dependency injection (DI)

Spring implementa la IoC tramite Dependency Injection che è una specifica implementazione del principio dell'inversion of control ed è rivolta ad invertire il processo di risoluzione delle dipendenze, facendo in modo che queste vengano iniettate dall'esterno.
La DI prevede che tutti gli oggetti all'interno dell'applicazione accettino le dipendenze, ovvero gli (altri) oggetti di cui hanno bisogno, tramite costruttore o metodi setter. Non sono quindi gli stessi oggetti a creare le proprie dipendenze, ma esse vengono iniettate dall'esterno.

2.1.3 Spring IoC Container

Spring inietta le dipendenze necessarie alle classi della nostra applicazione. Questo avviene sia per i componenti del framework, sia per gli oggetti da noi definiti. L'ecosistema all'interno del quale le applicazioni Spring vivono viene definito IoC container. Lo IoC container si occupa di istanziare gli oggetti (beans) dichiarati nel progetto e di reperire e iniettare tutte le dipendenze ad essi associate. Tali dipendenze possono essere componenti del framework o altri bean dichiarati nel contesto applicativo.

2.1.4 Spring Boot Project

Spring Boot è un progetto Spring che ha lo scopo di rendere più semplice lo sviluppo e l'esecuzione di applicazioni Spring e che è stato adottato nel corso dello sviluppo del sistema software poiché offre i seguenti vantaggi:

1. **Un'applicazione Spring può richiedere una gran quantità di metadati** di configurazione, anche se si utilizzano componenti e autowiring, mentre **Spring Boot semplifica lo sviluppo** delle applicazioni, poiché effettua una **configurazione automatica** (ove possibile), sulla base di valori di default "intelligenti". Un'applicazione Spring Boot richiede, di solito, solo **una configurazione minima** (anche se le scelte di default possono essere comunque sovrascritte mediante configurazioni esplicite).

2. Spring Boot fornisce delle **opzioni per la costruzione** (build) **e il dispiegamento** (deploy) delle applicazioni in produzione. Il risultato del build dell'applicazione Spring Boot sarà un standalone JAR file con al suo interno un embedded server. Questo vuol dire che non sarà necessario configurare un servlet container come Tomcat per eseguire l'applicazione, basterà eseguire il jar risultato della build e Spring boot genererà un embedded server che fungerà da container per la web application. Questo comporta un enorme vantaggio in termini di tempo e scalabilità dell'applicazione, per mezzo di diverse istanze dello stesso file Jar, istanziate eventualmente dietro un proxy.

Analizzando gli elementi di un'applicazione Spring Boot troviamo:

- La classe **SpringApplication** (di Spring Boot), che viene utilizzata per effettuare il "bootstrap", l'avvio la creazione di un **application context** per l'applicazione.
- L'annotazione **@SpringBootApplication** usata per la classe principale dell'applicazione indica una combinazione di:

- **@Configuration**: etichetta la classe come una classe di configurazione Java.
- **@ComponentScan**: abilita la scansione e **l'identificazione automatica dei componenti/bean.**
- **@EnableAutoConfiguration**: si occupa della **creazione automatica dei componenti/bean mancanti o necessari** (secondo SpringBoot), sulla base dell'application context e delle dipendenze specificate per l'applicazione.

Le applicazioni Spring Boot possono essere configurate mediante dei **file di proprietà** in cui è possibile specificare sia le proprietà comuni di Spring Boot (ad es. server.port) che delle proprietà specifiche dell'applicazione.
Le proprietà dell'applicazione vanno specificate nel file **Application properties** (oppure nel file **application.yml**).
In effetti, Spring Boot, definisce dei valori di default per le proprietà comuni e dunque non sempre è necessario configurare tutte le proprietà dell'applicazione (ad esempio il valore di default della proprietà server.port è proprio 8080 ma, volendo, può essere modificato).
Possono essere anche definiti anche degli **handler** associati a degli URL specifici e dei path.
In alternativa, l'applicazione può essere assemblata come un WAR e rilasciata in un application server separato.

2.1.5 Gestione dei dati

Spring Data è un progetto Spring (composto a sua volta da altri progetti) di supporto alla gestione di oggetti persistenti e all'accesso alle basi di dati.
In particolare, il progetto **Spring Data JPA** supporta l'implementazione di repository basati su JPA.
In JPA, **un'entità** è un tipo di oggetto persistente che va etichettato con l'annotazione **@Entity.**
Un **repository** è un oggetto che **fornisce un'interfaccia CRUD** per l'accesso a un'entità nella

base di dati.
Spring Data fornisce repository dinamici: **lo sviluppatore ne definisce solo l'interfaccia** e l'implementazione viene realizzata automaticamente da Spring Data.
Per utilizzare Spring Data JPA va utilizzata la dipendenza starter spring-boot-starter-data-jpa.
Questa dipendenza implica transitivamente **l'uso di Hibernate come provider JPA**.
Anche questa è una scelta convenzionale di Spring Boot, che può essere sovrascritta e modificata. Va invece aggiunta separatamente una dipendenza per il driver per il database.
In genere è utile definire una **classe Service** per esporre le funzionalità dell'applicazione nei confronti dei controller.

2.2 Maven

Maven è uno strumento di build automation utilizzato prevalentemente per la **gestione di progetti Java**. Simile per certi versi a strumenti precedenti, come ad esempio **Apache Ant**, si differenzia però da quest'ultimo per quanto concerne la compilazione del codice.
Con questo strumento, infatti, non è più necessaria la compilazione totale del codice, ma viene fatto uso di una struttura di progetto standardizzata su template definita **archetype**.
Il vantaggio derivante dall'utilizzo di questo tool è da subito evidente: **con la build automation l'intero processo viene automatizzato**, riducendo il carico di lavoro del programmatore e diminuendo le possibilità di errore da parte dello stesso.
Alcune delle fasi fondamentali che vengono automatizzate dal tool sono la compilazione in codice binario, il packaging dei binari, l'esecuzione di test per garantire il funzionamento del software, il deployment sui sistemi ed infine la documentazione relativa al progetto portato a termine.
Esistono poi alcuni tratti comuni ai prodotti di build automation, a partire dal build file, che contiene tutte le operazioni svolte.

2.2.1 Componenti

La prima caratteristica dello strumento è la presenza del file **pom.xml**, acronimo di **Project Object Model**, che ha il compito di definire identità e struttura del progetto.
Il file pom.xml è diviso a sua volta in cinque parti:

- relazione tra diversi file pom.xml;
- build settings, la project information;
- il build environment;
- configurazione dell'ambiente Maven.

Un'altra componente importante di Maven è rappresentata dai **goal**, ovvero l'insieme delle funzioni che possono essere eseguite sui diversi progetti. Tramite la cartella repository, inoltre l'utente è in grado di gestire il sistema delle **librerie**.
Ultima componente fondamentale del tool è il file **settings.xml**, usato per la configurazione di repository, proxy e profili.

La particolarità di questa componente è che può essere specificata su due livelli, ovvero:

- **user.home**: per il singolo utente
- **maven.home** : per più utenti.

In sintesi, i principali componenti di maven sono:

- **pom.xml** (POM, Project Object Model): file di configurazione che contiene tutte le informazioni su un progetto (dipendenze, test, documentazione).
- **Goal**: singola funzione che può essere eseguita sul progetto. I goal possono essere sia specifici per il progetto dove sono inclusi, sia riutilizzabili.
- **Jelly script**: linguaggio XML con il quale vengono definiti i goal.
- **Plug-in**: goal riutilizzabili in tutti i progetti.
- **Repository**: directory strutturata destinata alla gestione delle librerie. Un repository può essere locale o remoto.

2.3 Room

Room è una libreria che fornisce un layer aggiuntivo (astratto) a SQLite che consente da un lato un accesso fluido e robusto al database, e dall'altro di sfruttare tutti i punti di forza di SQLite.
La libreria può assolvere a molteplici scopi:

- per costruire un **in-memory database**, utile a creare una cache dei dati dell'applicazione.
- per costruire un **database persistente** e consistente in cui memorizzare dati che necessitano di sopravvivere allo spegnimento del dispositivo.

2.3.1 Componenti

In Room ci sono tre componenti principali:

1. **Database**: contiene il database holder e serve da **punto di accesso principale** ai dati che si è deciso di rendere persistenti. La classe che viene annotata con **@Database** dovrebbe soddisfare le seguenti condizioni:

- **Essere una classe astratta** che estende RoomDatabase.
- **Includere la lista delle entità associate** al database all'interno dell'annotazione stessa (esempio: @Database (entities = {User.class}, version = 1))
- Contenere un metodo astratto che ha 0 argomenti e che ritorna la classe che è annotata con @Dao.

A Runtime è possibile dunque acquisire un'istanza del Database chiamando il metodo

Room.databaseBuilder() (nel caso di Database persistente) oppure **Room.inMemoryDatabaseBuilder()** (nel caso di un database in-memory).
Vale la pena notare che nel caso in cui il database non sia di tipo in-memory, tutte le operazioni che vengono effettuate su di esso devono essere obbligatoriamente di natura asincrona, in maniera tale da evitare di incorrere in rischi legati alla dilatazione dei tempi di attesa dell'utente che, in caso di errori gravi, potrebbero causare il freezing dell'applicazione.

2. **Entity**: ciascuna entity rappresenta una **tabella all'interno del database**. Una classe di questo tipo viene annotata con **@Entity**.
3. DAO: una classe annotata con @Dao contiene i metodi utilizzati per accedere al database.

Il **flusso di esecuzione** di un'applicazione che sfrutta Room come framework per la persistenza dei può essere così schematizzato:

1. l'applicazione utilizza il database Room per ottenere i DAO associati al database.
2. L'applicazione, quindi, usa ciascun DAO per ottenere le entità dal database ed eventualmente salvare ogni cambiamento apportato alle suddette entità sul database.
3. l'applicazione utilizza un'entità per ottenere e/o impostare valori che corrispondono alle colonne della tabella all'interno del database.

2.4 Retrofit

Retrofit è un client REST type-safe per Android, Java e Kotlin. La libreria **fornisce un framework potente per l'autenticazione e l'interazione con delle API** e permette di inviare richieste http con OkHttp. È possibile utilizzare Retrofit per ricevere dati in vari formati quali Json, XML etc.

2.4.1 Componenti

Definiamo quindi **due elementi** concettuali **fondamentali**:

1. **Rest Client**: tutta le porzioni di codice usate lato Client (Android) per generare ed inviare richieste http, oltre che per processare la relativa risposta.
2. **Rest Api**: Una Rest Api definisce un set di funzioni attraverso cui gli sviluppatori possono performare richieste e ricevere risposte attraverso il protocollo http (richieste GET, POST). In buona sostanza è possibile definire una RESTful Api come un'application Program Interface che usa richieste http di tipo GET, PUT, POST e DELETE.

Affinché sia possibile utilizzare retrofit, sono necessarie **tre classi fondamentali**:

1. **Un'interfaccia** che definisce le operazioni http (funzioni e metodi). Retrofit trasforma l'API http in una interfaccia Java. Ogni metodo all'interno dell'interfaccia rappresenta una

possibile richiesta all'API. Essa deve possedere un'opportuna annotazione (@GET, @POST, etc.) per specificare il tipo di richiesta ed il relativo URL. Il valore di ritorno incapsula la risposta in un oggetto di tipo Call con il tipo del risultato previsto. Inoltre, possono essere aggiunti al metodo parametri aggiuntivi alla richiesta a seconda delle esigenze del programmatore. Con l'ausilio dell'annotazione @Path al parametro del metodo, ad esempio, è possibile specificare il path relativo alla richiesta mediante diverse metodologie.

2. **Una classe di tipo Retrofit** che genera un'implementazione concreta dell'interfaccia definita per le richieste http, permettendo eventualmente l'abilitazione e la personalizzazione di una serie di opzioni aggiuntive (tipo di crittografia utilizzata, tipo di connessione, etc.)
3. **Una classe POJO** i cui attributi corrispondono a ciascun campo dell'oggetto JSON response (oggetto che modella la risposta http) ottenuto dall'interrogazione di un'API. Essa è una classe contenente gli attributi sopra menzionati ed i relativi metodi "getters and setters".

2.4.2 Converters

I **convertitori Retrofit** costituiscono concettualmente un vero e proprio **contratto tra client e server relativo al formato con cui i dati verranno rappresentati**. Ciascuna delle parti concorda, dunque, il formato di comunicazione relativo al trasferimento dati.
Tra questi ricordiamo:

1. **Gson:** Gson è utilizzato per il mapping e la gestione del **formato JSON** e può essere aggiunto con la seguente dipendenza:

   "compile 'com.squareup.retrofit2:converter-gson:2.2.0'"

2. **SimpleXML:** SimpleXML è utilizzato per il mapping e la gestione del **formato XML** e può essere aggiunto con la seguente dipendenza**:**

   "compile 'com.squareup.retrofit2:converter-simplexml:2.2.0'"

3. **Jackson:** Jackson è un'alternativa a GSON che offre un maggiore grado di personalizzazione e, plausibilmente, offre una **maggiore velocità di mapping dei dati in formato JSON.**
   Esso può essere aggiunto con la seguente dipendenza:

   "compile 'com.squareup.retrofit2:converter-jackson:2.2.0'"

2.5 Postman

Postman è un ambiente di sviluppo API che ha lo scopo di aiutare lo sviluppatore a **creare, testare, documentare, monitorare e pubblicare** documentazione per le loro API.

È possibile creare ogni richiesta specificando tutti i parametri possibili, e conoscere ogni informazione della suddetta *request* e della *response*: *headers*, *body* etc.
Le caratteristiche principali di Postman sono:

- **Invio di richieste** (con supporto per diversi schemi di autenticazione, cookie, certificati, intestazioni, parametri di query, corpo della richiesta ed eventualmente SOAP con / senza WSDL) e debugging e salvataggio delle risposte.
- **Organizzazione delle API** in gruppi denominati Raccolte.
- **Test di scrittura**: Gli script di test possono eseguire pre-richiesta, dopo che è stata ricevuta una risposta e possono avere concetti di looping e ramificazione.
- **Test di automazione utilizzando le sequenze di raccolta**: le raccolte possono anche essere esportate ed eseguite nella riga di comando usando Newman come parte del processo di compilazione.
- **Un modo per generare e personalizzare automaticamente la documentazione** dell'API direttamente dalle raccolte. Può essere privato, condiviso con il tuo team, pubblico e può anche essere impostato sul tuo dominio personalizzato.

2.6 XAMPP

XAMPP, acronimo di **C**ross **A**pache **M**ySQL **P**HP e **P**erl, è una piattaforma software che **facilita l'installazione e la gestione degli strumenti più comuni per lo sviluppo di applicazioni web**, raggruppandoli in un unico luogo. Infatti, attraverso un'unica installazione è possibile avere in una sola cartella Apache, cioè il web server ovvero il programma che gestisce le richieste che arrivano da un qualsiasi client attraverso il protocollo HTTP, MySQL cioè il DBMS, PHP e Perl, cioè due linguaggi utili per lo sviluppo di applicazioni web.
Di questa suite è stato utilizzato esclusivamente mySQL per generare un'istanza di database utile in fase di testing.

2.7 Beacon Scanner

Beacon Scanner è un'applicazione per ambiente Android in grado di **rilevare dispositivi bluetooth a bassa energia** nei dintorni. È stata utilizzata principalmente nella fase di testing di modulo per verificare il corretto funzionamento dell'unità di trasmissione.

2.8 Beacon Simulator

Beacon Simulator è un programma che simula la trasmissione di un segnale Bluetooth a bassa energia personalizzandone i parametri (Name, UUID, Major, Minor, Power Range).

3. Specifica dei requisiti

Alla luce dello stato pandemico, si desidera realizzare un'applicazione i cui scopi sono:

- **Monitoraggio** di luoghi aperti al pubblico (spiagge, bar, ecc.).
- **Tracciabilità** della catena di contatti intrattenuti dall'utente.

Le **parti interessate** allo sviluppo ed alla produzione di questo sistema software possono essere raggruppati nelle seguenti categorie:

- **Utenti**: ciascuna persona che disponga di uno smartphone e che desideri essere informata nel caso in cui abbia avuto un contatto con individui che sono risultati positivi al virus SARS-CoV-2 o che, in caso di propria positività, desideri segnalarla alla propria catena di contatti. Egli inoltre potrebbe essere interessato a reperire informazioni commerciali e relative al numero di persone presenti in un determinato luogo, disponendo della possibilità di prenotare nel suddetto in totale sicurezza.
- **Proprietari**: utenti che oltre alle funzionalità di monitoraggio, sono interessati a registrare la propria attività e a condividere informazioni riguardanti quest'ultima a fini commerciali.

Essendo il sistema pensato principalmente per dispositivi mobili, questo sarà adoperabile esclusivamente su quelli che soddisfino i **requisiti tecnologici e software** necessari, di seguito elencati:

- **Requisiti minimi Hardware:** Bluetooth 4.0.
- **Requisiti minimi Software:** sistema operativo Android 8+

3.1 Funzioni

Il sistema software che si vuole realizzare verrà indicato con “CovidSafe”.
CovidSafe deve fornire le seguenti funzionalità:

1. provvedere a fornire un meccanismo di registrazione al sistema.
2. provvedere **all'autenticazione dell'utente**, consentendogli di accedere solo alle funzionalità di sua competenza.
3. deve tenere **traccia dei contatti** intrattenuti dall’utente per un tempo definito di 15 giorni, oltre cui il rischio di contagio è da considerarsi nullo.
4. fornire a qualsiasi utente la possibilità di **segnalare la propria positività**.
5. deve **notificare la positività** di un utente alla sua catena di contatto.
6. deve fornire a qualsiasi utente la possibilità di **creare una white list.**
7. deve **elaborare statistiche** con i dati raccolti.
8. deve fornire a qualsiasi utente un **meccanismo di ricerca** di luoghi pubblici e relative statistiche.
9. deve **monitorare** il numero di accessi ai luoghi pubblici.
10. deve fornire ad un gestore la possibilità di modificare i parametri di un luogo pubblico di sua competenza.
11. deve fornire ad un amministratore la possibilità di aggiungere ed eliminare luoghi pubblici e/o proprietari.

Inoltre, il sistema software dovrà rispettare i seguenti vincoli:

- Il sistema dovrà rispettare le norme vigenti sulla **privacy**.
- Il sistema dovrà essere **portabile**.
- Il sistema dovrà essere **facilmente adoperabile dagli utenti**.
- Il sistema dovrà essere **scalabile**.
- Il sistema dovrà essere **evolvibile**.
- Il sistema dovrà provvedere a fornire un meccanismo di **protezione dei dati**.

3.2 Attori e Casi d’uso

Gli attori coinvolti nel funzionamento del sistema software sono:

- **Utente:** egli può soltanto registrarsi al sistema al fine di usufruire dei suoi servizi ed è la figura identificata con il primo avvio.
- **Utente Registrato:** è l’utente sottoscritto al sistema; egli, dopo aver effettuato il login tramite le credenziali personali, può attivamente usufruire dei servizi di statistiche dei

luoghi pubblici, dichiarare la propria positività e creare una personale white list. Dal momento della registrazione, CovidSafe monitorerà l'accesso dell'utente registrato a luoghi pubblici e traccerà i suoi contatti.

- **Amministratore di sistema:** egli provvederà ad aggiungere ed eliminare luoghi pubblici dal sistema.
- **Proprietario:** provvederà ad impostare/modificare i parametri relativi al proprio/ai propri luoghi.
- **Tempo:** Attore fittizio, definito al fine di modellare la cancellazione dei contatti obsoleti.

I casi d'uso sono:

- **U1** - creare il profilo
- **UR1** - effettuare l'accesso
- **UR2** - dichiarare la propria positività
- **UR3** – aggiungi dispositivo alla white list
- **UR4** - cercare un luogo pubblico
- **UR5 –** accedere al luogo pubblico
- **UR6 –** uscire dal luogo pubblico
- **AS1 -** aggiungere un luogo pubblico
- **AS2** - eliminare un luogo pubblico
- **AS3** – modificare un luogo pubblico
- **P1 -** modificare un luogo pubblico
- **T1 –** cancellare contatti obsoleti

Attori e relativi casi d'uso:

Attore	**Obiettivi**
UTENTE (U)	• Creare Profilo
UTENTE REGISTRATO (UR)	• Effettuare l'accesso • Dichiarare la propria positività • Aggiungere dispositivo alla white list • Cercare un luogo pubblico • Entrare in un luogo pubblico • Uscire da un luogo pubblico
AMMINISTRATORE DI SISTEMA (AS)	• Aggiungere luogo pubblico • Eliminare luogo pubblico
PROPRIETARIO (P)	• Modificare luogo pubblico
TEMPO (T)	• Cancellare contatti obsoleti

ID	U1
NOME DEL CASO D'USO	Creare Profilo
ATTORI PRIMARI COINVOLTI	Utente
BREVE DESCRIZIONE	Consente di registrare un utente nel sistema.
PRECONDIZIONI	Primo avvio dell'applicazione.
POSTCONDIZIONI	Un utente è stato aggiunto al sistema.
SCENARIO DI SUCCESSO	
AZIONE DELL'ATTORE	**RESPONSABILITÀ DEL SISTEMA**
1. L'utente inserisce email e password	
	2. IL Sistema verifica la validità dei dati immessi
	3. Il Sistema registra il nuovo utente
4. L'utente viene portato alla pagina principale	
SCENARIO ALTERNATIVO 1	Dati non validi
	2.a Il sistema chiede all'utente di verificare i dati ed inserirne di validi.
SCENARIO ALTERNATIVO 2	Utente già presente
	2.b Il sistema segnala la presenza di un account già collegato a quella email e chiede di inserire una email valida.

ID	UR1
NOME CASO D'USO	Effettuare l'accesso
ATTORI PRIMARI COINVOLTI	Utente Registrato
BREVE DESCRIZIONE	L'utente si autentica al sistema con i dati d'accesso
PRECONDIZIONI	L'utente si è già registrato al sistema.
POSTCONDIZIONI	Accesso alla pagina principale
SCENARIO DI SUCCESSO	
AZIONE DELL'ATTORE	**RESPONSABILITÀ DI SISTEMA**
1. L'utente si autentica	
	2. Il sistema verifica i dati di accesso
3. L'utente viene portato alla pagina principale	
SCENARIO ALTERNATIVO	Dati non validi
	2.a Il sistema chiede all'utente di verificare e reinserire i dati di accesso o di registrarsi.

ID	UR2
NOME CASO D'USO	Dichiarare positività
ATTORI PRIMARI COINVOLTI	Utente Registrato
BREVE DESCRIZIONE	L'utente registrato dichiara la propria positività al sistema.
PRECONDIZIONI	L'utente non si è dichiarato positivo nella settimana precedente
POSTCONDIZIONI	Viene notificata la positività ai contatti.
SCENARIO DI SUCCESSO	
AZIONE DELL'ATTORE	**RESPONSABILITÀ DI SISTEMA**
1. L'utente sceglie di dichiarare la propria positività	
	2. Il sistema verifica la validità dell'azione
	3. Il sistema notifica alla catena di contatti l'avvenuta positività
SCENARIO ALTERNATIVO	L'utente si è già dichiarato positivo nell'ultima settimana
	2.a Il sistema rileva l'inconsistenza della dichiarazione e avverte l'utente con un messaggio di errore

ID	UR3
NOME CASO D'USO	Aggiungere dispositivo alla white list
ATTORI PRIMARI COINVOLTI	Utente Registrato
BREVE DESCRIZIONE	L'utente registrato decide di creare una lista di dispositivi alla quale non notificare la propria positività
PRECONDIZIONI	Gli utenti della white list sono registrati.
POSTCONDIZIONI	La white list associata all'utente è creata
SCENARIO DI SUCCESSO	
AZIONE DELL'ATTORE	**RESPONSABILITÀ DEL SISTEMA**
1. L'utente inserisce l'email associata al dispositivo da aggiungere e clicca su "Aggiungi"	
	2. Il sistema verifica che l'email indicata sia presente tra gli utenti registrati
	3. Il sistema aggiunge il contatto alla white list
SCENARIO ALTERNATIVO	L'email indicata non è associata a nessun contatto
	2.a Il sistema verifica la mancata corrispondenza e mostra un messaggio di errore

ID	UR4
NOME CASO D'USO	Cercare un luogo pubblico
ATTORI PRIMARI COINVOLTI	Utente Registrato
BREVE DESCRIZIONE	L'utente cerca un luogo pubblico per poterne visualizzare le statistiche in tempo reale
PRECONDIZIONI	Il luogo deve essere presente nel sistema
POSTCONDIZIONI	Visualizzazione del luogo con le relative statistiche
SCENARIO DI SUCCESSO	
AZIONE DELL'ATTORE	**RESPONSABILITÀ DI SISTEMA**
1. L'utente digita il nome luogo e clicca su "Cerca"	
	2. Il sistema ricerca nei luoghi le corrispondenze e le mostra
3. L'utente clicca sulla corrispondenza desiderata	
	4. Il sistema mostra le statistiche relative al luogo e la disponibilità/indisponibilità di posti
SCENARIO ALTERNATIVO	Luogo inesistente
	2.a Il sistema mostra un messaggio di errore per l'assenza del luogo

ID	UR5
NOME CASO D'USO	Accedere al luogo pubblico
ATTORI PRIMARI COINVOLTI	Utente Registrato
BREVE DESCRIZIONE	L'utente accede ad un luogo pubblico
PRECONDIZIONI	Il luogo pubblico è accessibile
POSTCONDIZIONI	L'utente ha acceduto al luogo
SCENARIO DI SUCCESSO	
AZIONE DELL'ATTORE	**RESPONSABILITÀ DI SISTEMA**
1. L'utente passa il proprio dispositivo NFC vicino al sensore	
	2. Il sistema verifica la disponibilità di posti
	3. Il sistema permette l'accesso
SCENARIO ALTERNATIVO	Posti non disponibili
	2.a Il sistema blocca l'accesso

ID	UR6
NOME CASO D'USO	Uscire dal luogo pubblico
ATTORI PRIMARI COINVOLTI	Utente Registrato
BREVE DESCRIZIONE	L'utente esce dal luogo pubblico
PRECONDIZIONI	L'utente si trova nel luogo pubblico
POSTCONDIZIONI	L'utente non è più nel luogo pubblico
SCENARIO DI SUCCESSO	
AZIONE DELL'ATTORE	**RESPONSABILITÀ DI SISTEMA**
1. L'utente passa il proprio dispositivo NFC vicino al sensore	
	2. Il sistema permette l'uscita

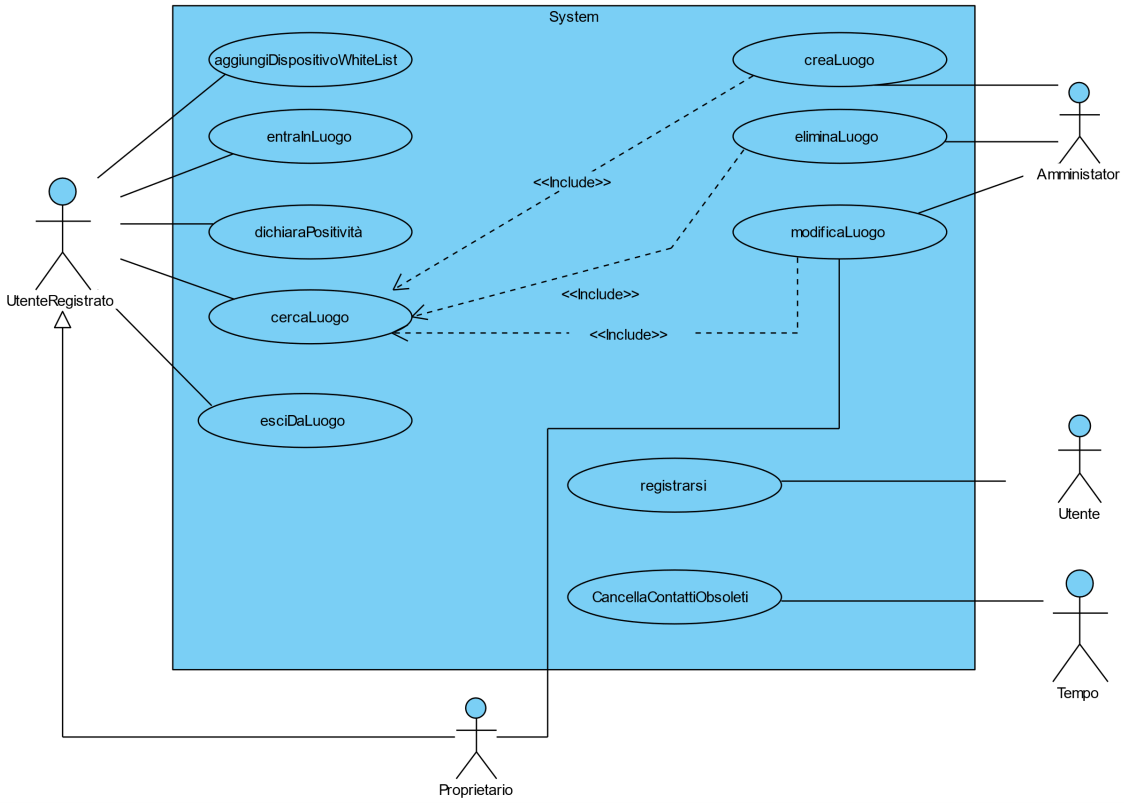
ID	AS1
NOME CASO D'USO	Aggiungere luogo pubblico
ATTORI PRIMARI COINVOLTI	Amministratore di sistema
BREVE DESCRIZIONE	L'amministratore aggiunge un nuovo luogo al sistema
PRECONDIZIONI	Il luogo non deve essere già presente nel sistema
POSTCONDIZIONI	Il luogo è aggiunto al sistema
SCENARIO DI SUCCESSO	
AZIONE DELL'ATTORE	**RESPONSABILITÀ DI SISTEMA**
1. L'amministratore seleziona la funzionalità "Aggiungi luogo"	
	2. Il sistema richiede i dati all'amministratore
3. L'amministratore aggiunge i dati	
	4. Il sistema verifica i dati
	5. Il sistema aggiunge il luogo
SCENARIO ALTERNATIVO 1	Luogo già presente
	4.a Il sistema mostra un messaggio di errore per il luogo già presente
SCENARIO ALTERNATIVO 2	Dati non validi
	4.b Il sistema mostra un messaggio di errore e chiede di inserire dati validi
SCENARIO ALTERNATIVO 3	Dati incompleti
	4.c Il sistema chiede di inserire i restanti dati

ID	AS2
NOME CASO D'USO	Eliminare luogo pubblico
ATTORI PRIMARI COINVOLTI	Amministratore di sistema
BREVE DESCRIZIONE	L'amministratore elimina un luogo dal sistema
PRECONDIZIONI	Il luogo da eliminare è presente nel sistema L'amministratore è associato a quel luogo
POSTCONDIZIONI	Il luogo è eliminato dal sistema
SCENARIO DI SUCCESSO	
AZIONE DELL'ATTORE	**RESPONSABILITÀ DEL SISTEMA**
1. L'amministratore elimina il luogo	
	2. Il sistema elimina il luogo

ID	AS3
NOME CASO D'USO	Modificare luogo pubblico
ATTORI PRIMARI COINVOLTI	Amministratore di sistema
BREVE DESCRIZIONE	L'amministratore modifica i parametri del luogo
PRECONDIZIONI	Il luogo da modificare è presente nel sistema
POSTCONDIZIONI	I parametri del luogo sono modificati
SCENARIO DI SUCCESSO	
AZIONE DELL'ATTORE	**RESPONSABILITÀ DEL SISTEMA**
1. L'amministratore inserisce i nuovi parametri	
	2. Il sistema verifica i dati inseriti (consistenza)
	3. Il sistema modifica i parametri del luogo
SCENARIO ALTERNATIVO	Dati non validi
	2.a Il sistema mostra un messaggio di errore relativo ai dati inseriti

ID	P1
NOME CASO D'USO	Modificare luogo pubblico
ATTORI PRIMARI COINVOLTI	Proprietario
BREVE DESCRIZIONE	Il proprietario modifica i parametri del luogo
PRECONDIZIONI	Il luogo da modificare è presente nel sistema Il proprietario è associato a quel luogo
POSTCONDIZIONI	I parametri del luogo sono modificati
SCENARIO DI SUCCESSO	
AZIONE DELL'ATTORE	**RESPONSABILITÀ DEL SISTEMA**
1. L'amministratore inserisce i nuovi parametri	
	2. Il sistema verifica i dati inseriti (consistenza)
	3. Il sistema modifica i parametri del luogo
SCENARIO ALTERNATIVO	Dati non validi
	2.a Il sistema mostra un messaggio di errore relativo ai dati inseriti
SCENARIO ALTERNATIVO 2	Mancata corrispondenza tra luogo e proprietario
	2.b Il sistema mostra un messaggio di errore per la mancata corrispondenza

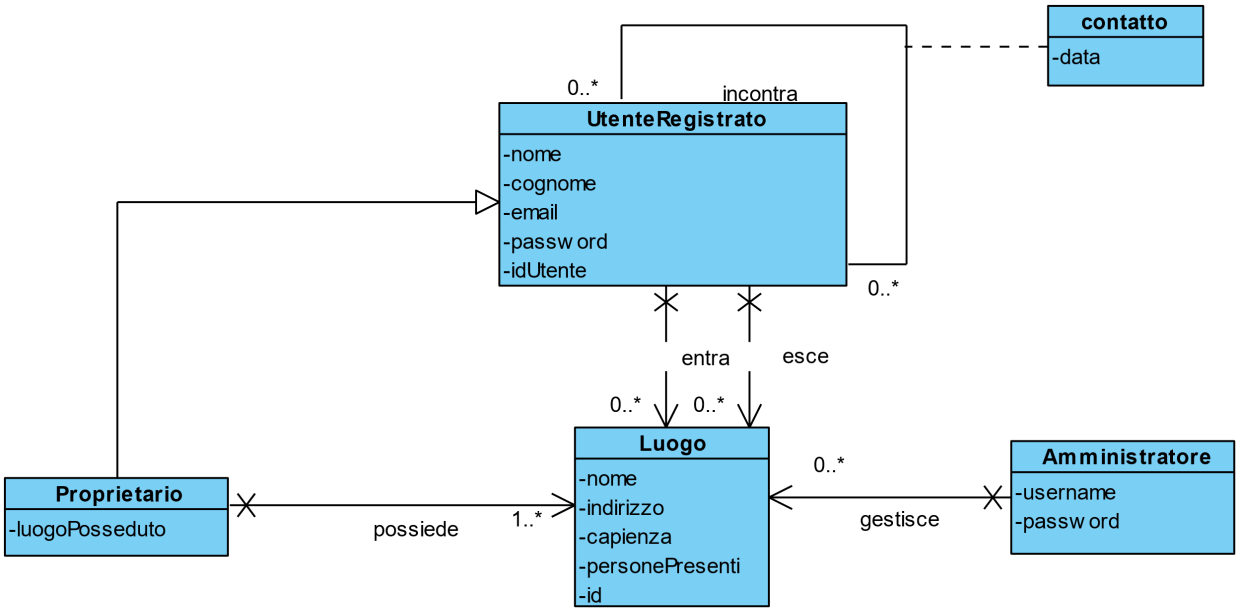
ID	T1
NOME CASO D'USO	Cancellare contatti obsoleti
ATTORI PRIMARI COINVOLTI	Tempo
BREVE DESCRIZIONE	Ogni giorno il sistema cancella i contatti risiedenti in memoria da più di quindici giorni.
PRECONDIZIONI	Nessuna
POSTCONDIZIONI	I contatti rimasti in memoria hanno meno di 15 giorni.
SCENARIO DI SUCCESSO	
AZIONE DELL'ATTORE	**RESPONSABILITÀ DEL SISTEMA**
1. Sono passate 24 ore. L'attore Tempo invoca il sistema.	
	2. Il sistema verifica la presenza di contatti obsoleti.
	3. Il sistema elimina i contatti obsoleti.



4. Documenti di Analisi

Di seguito sono riportati tutti i diagrammi usati in fase di analisi del sistema oltre che le scelte architetturali definite in fase di progettazione.
confronti dei controller.

4.1 System Domain Model



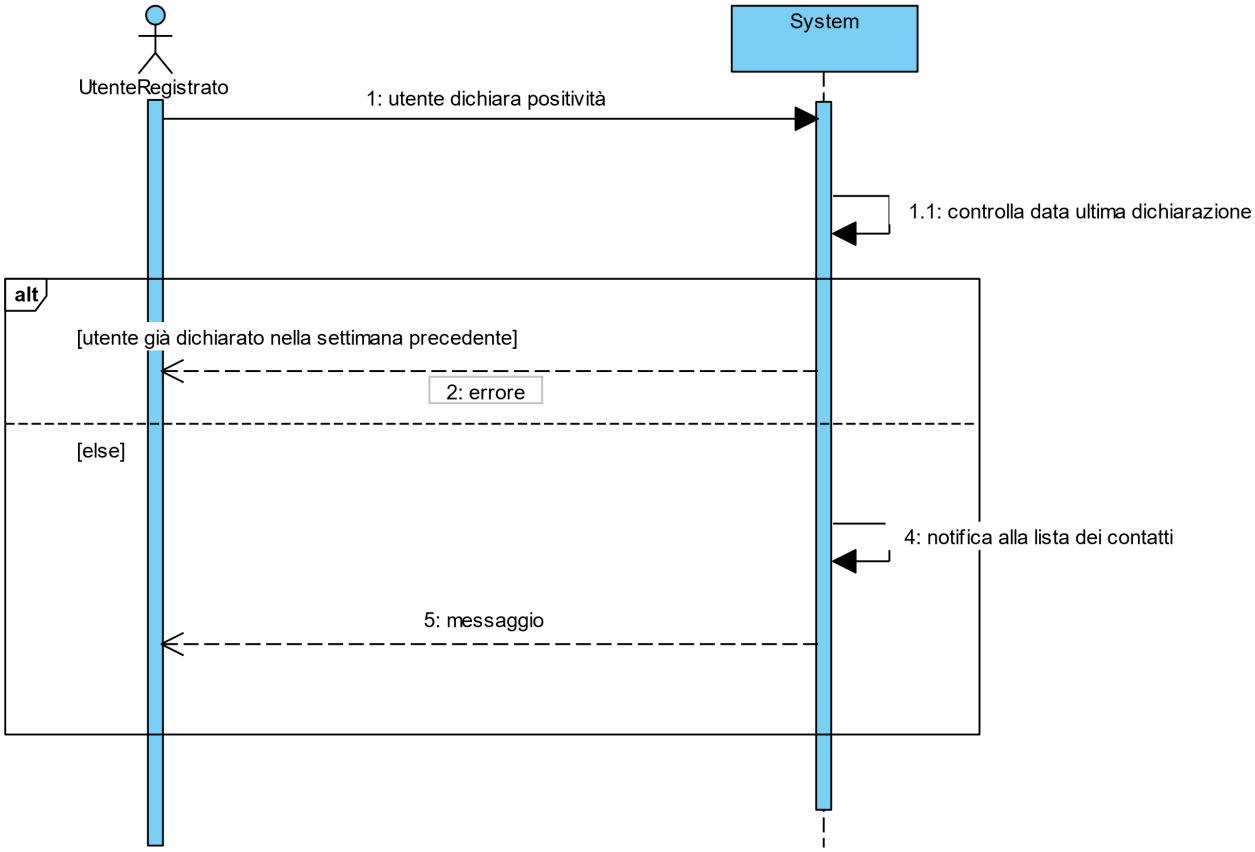
Il primo artefatto di analisi Object Oriented prodotto è il **Modello di Dominio** ovvero **una visualizzazione degli oggetti o classi di dominio** di interesse.
In un modello di dominio non possono essere inserite funzionalità ed operazioni ma solo attributi ed associazioni tra classi concettuali, poiché lo scopo principale del dominio è quello di focalizzare l'attenzione sulle entità di interesse, che sono:

Proprietario: ciascun proprietario possiede uno o più luoghi.
Luogo: questa entità conterrà sicuramente le informazioni necessarie per identificarlo, il numero di persone presenti all'interno ed informazioni di carattere generale da mostrare all'utente.
Amministratore: questa entità rappresenta l'attore che **agisce in backend** per gestire l'intero

sistema.
Utente Registrato: attore principale del sistema, interagisce con il sistema attraverso il proprio smartphone**. Le relazioni di entrata ed uscita da un luogo sono state esplicitate poiché già dalla fase di analisi è parso evidente che tali azioni avrebbero dovuto essere gestite digitalmente** (per monitorare un luogo fisico dal punto di vista digitale è necessario utilizzare qualche tipo di tecnologia come l'**NFC**) e che fossero pertanto rilevanti dal punto di vista del sistema.

4.1.1 Sequence diagram

Il **Sequence Diagram è stato utilizzato per definire chiaramente**, nella prima fase di analisi, **gli input e gli output del sistema** da realizzare in riferimento al caso d'uso scelto. In questo diagramma gli attori interagiscono con il **sistema schematizzato come una black box**, la quale risponde agli stimoli prodotti dagli attori con operazioni di sistema.

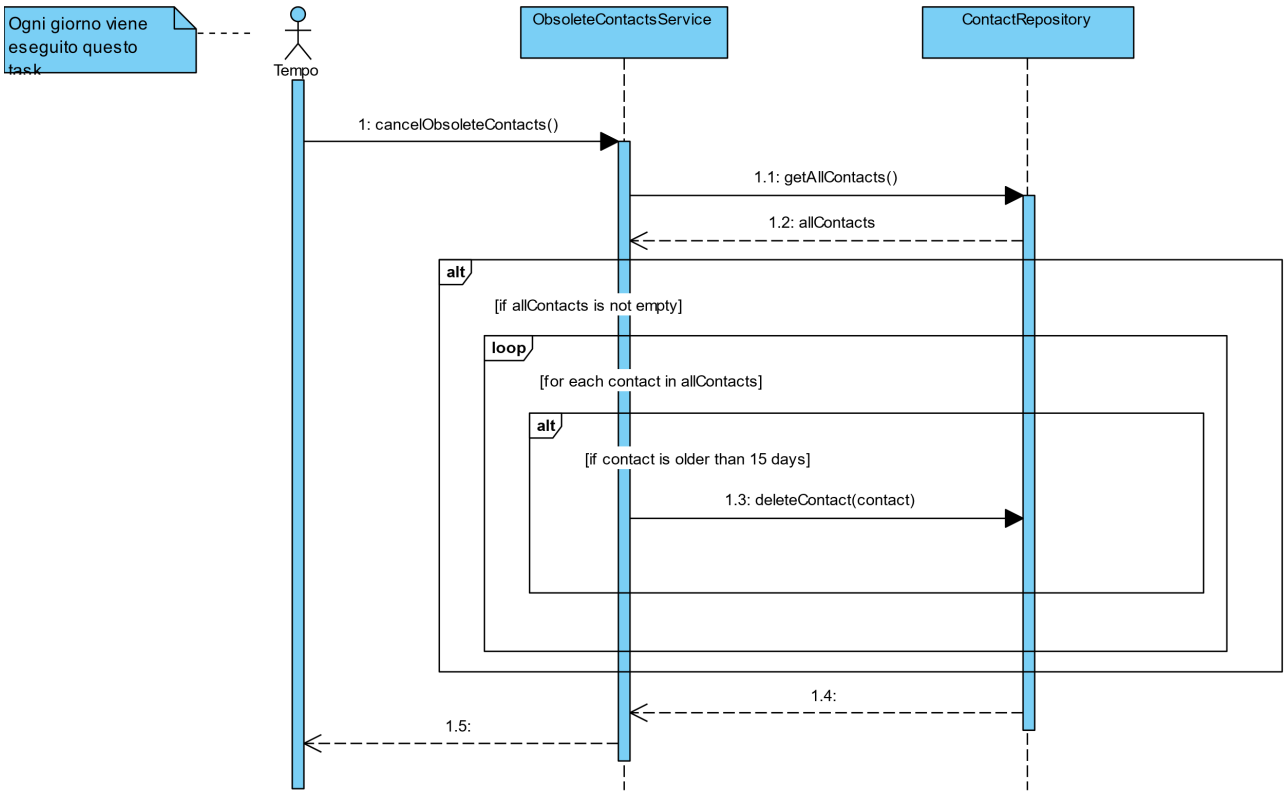


La dichiarazione, come evidenziato dal diagramma, procede nel seguente modo:

1. **L'utente registrato** si dichiara positivo (in questa fase è stato ipotizzato un tasto apposito per procedere alla dichiarazione in una activity dedicata).
2. **Il sistema** prevede un meccanismo di prevenzione da dichiarazioni mendaci, controllando che non siano state effettuate dichiarazioni nei 7 giorni precedenti, limitando quindi il tempo che intercorre tra una dichiarazione avvenuta con successo e l'altra.
3. **Se il sistema rileva una dichiarazione** avvenuta nei 7 giorni precedenti informa l'utente della dichiarazione già avvenuta, **altrimenti** procede ad informare la propria catena di

contatti.

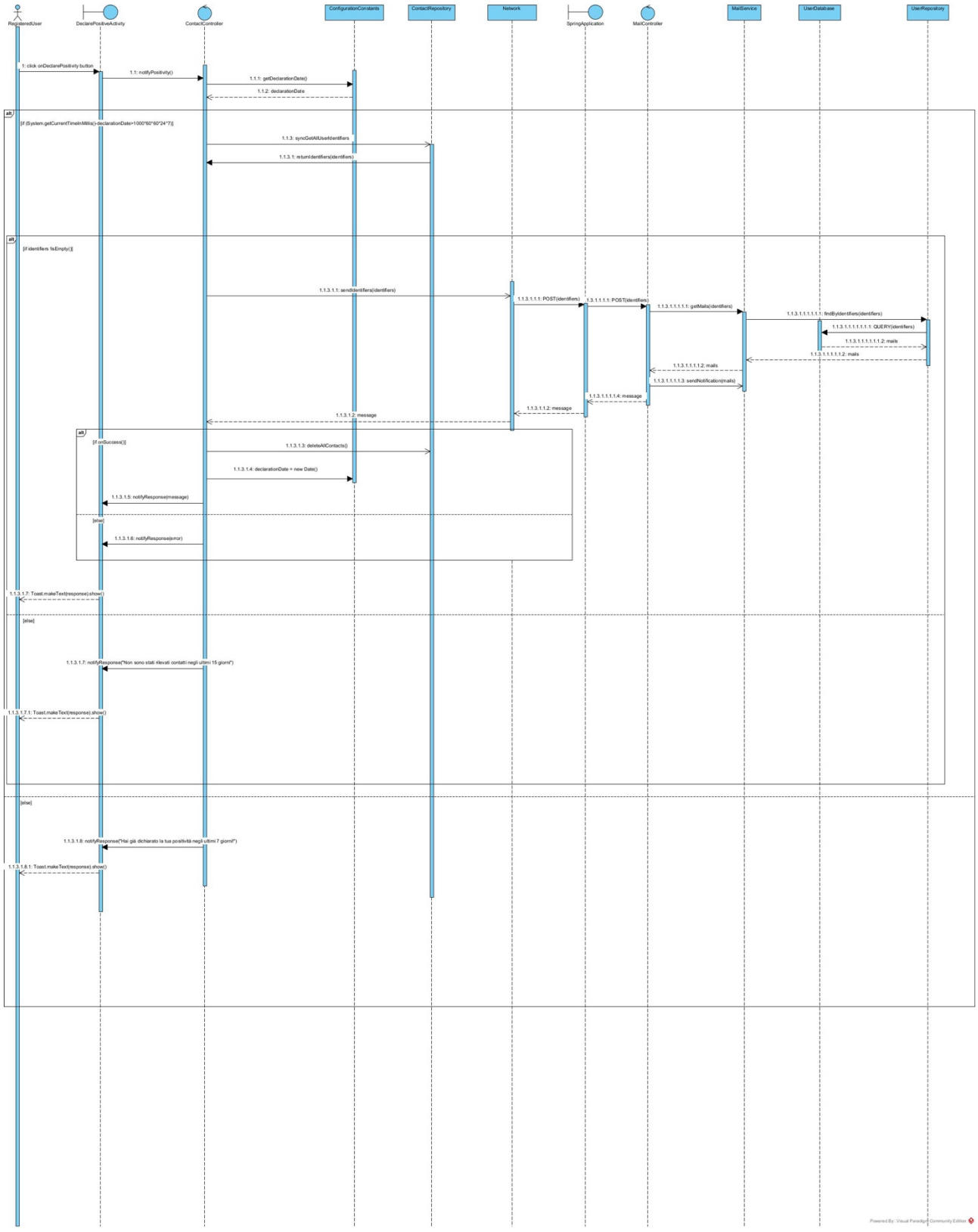
4. **Il sistema invia un messaggio all'utente** informandolo dell'avvenuta dichiarazione oppure, in caso di errore, di riprovare poiché ci sono stati problemi.



In questo diagramma di analisi è stato descritto in forma embrionale **l'algoritmo elaborato per cancellare i contatti obsoleti**, avendo cura di **descriverne il comportamento in termini di scambio di messaggi** tra attori e classi di sistema. In particolare, si può notare che:
Tempo: è un **attore fittizio utilizzato per rendere l'idea di un'azione ciclica**, effettuata con cadenza giornaliera, **necessario per evidenziare il trigger scatenante il processo** descritto.
ObsoleteContactService: istanza di un **oggetto generico a cui si è assegnata la responsabilità in fase di analisi di interagire con il Model** per cancellare i contatti obsoleti.
ContactRepository: classe che **astrae il model** e ne semplifica l'utilizzo.
Già dalla fase di analisi ci è stato chiaro che avremmo utilizzato un Repository Pattern per **due ragioni**:

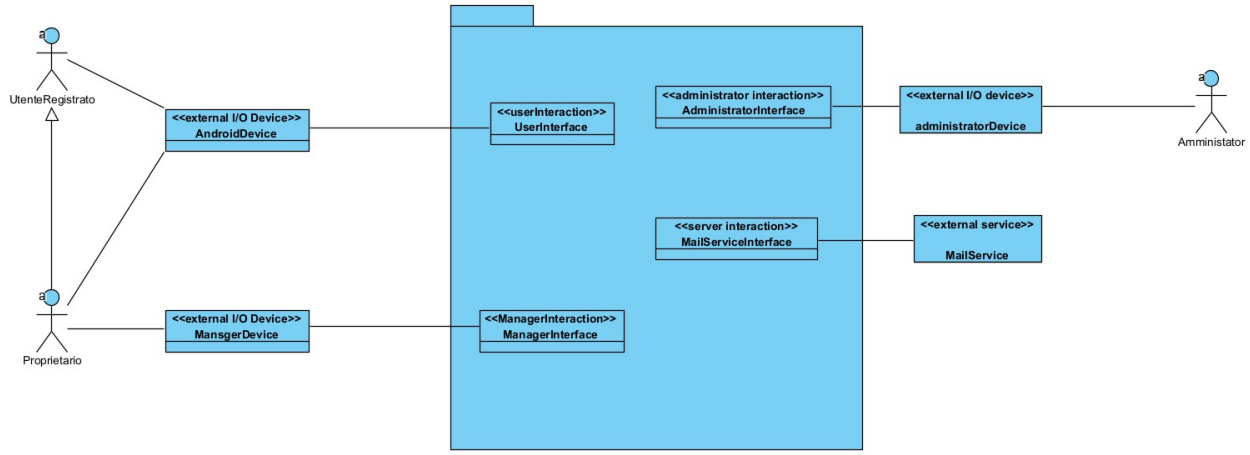
1. Sono presenti **innumerevoli vantaggi** di questo design pattern.
2. Studiando ed analizzando i principali pattern architetturali consigliati (in termini di buone pratiche di programmazione) da Google, azienda proprietaria del sistema operativo su cui si è operatto. **Ciascuno di questi pattern prevede l'utilizzo di una repository**. La volontà di uniformarsi alle **linee guida dell'azienda** ed alle **buone pratiche consigliate** ha spinto l'adozione tale pattern.

Sequence diagram di dettaglio che illustra il **comportamento finale dell'intero sistema software** dal momento in cui l'utente scatena il caso d'uso.


RegisteredUser
DeclarePositiveActivity
ContactController
ConfigurationConstants
ContactRepository
Network
SpringApplication
MailController
MailService
UserDatabase
UserRepository
1: click onDeclarePositivity button
1.1: notifyPositivity()
1.1.1: getDeclarationDate()
1.1.2: declarationDate
alt
[if (System.getCurrentTimeInMillis()-declarationDate>1000*60*60*24*7)]
1.1.3: syncGetAllUserIdentifiers
1.1.3.1: returnIdentifiers(identifiers)
alt
[if identifiers !isEmpty()]
1.1.3.1.1: sendIdentifiers(identifiers)
1.1.3.1.1.1: POST(identifiers)
1.1.3.1.1.1.1: POST(identifiers)
1.1.3.1.1.1.1.1: getMails(identifiers)
1.1.3.1.1.1.1.1.1: findByIdentifiers(identifiers)
1.1.3.1.1.1.1.1.1.1: QUERY(identifiers)
1.1.3.1.1.1.1.1.1.2: mails
1.1.3.1.1.1.1.1.2: mails
1.1.3.1.1.1.1.2: mails
1.1.3.1.1.1.1.3: sendNotification(mails)
1.1.3.1.1.1.1.4: message
1.1.3.1.1.2: message
1.1.3.1.2: message
alt
[if onSuccess()]
1.1.3.1.3: deleteAllContacts()
1.1.3.1.4: declarationDate = new Date()
1.1.3.1.5: notifyResponse(message)
[else]
1.1.3.1.6: notifyResponse(error)
1.1.3.1.7: Toast.makeText(response).show()
[else]
1.1.3.1.7: notifyResponse("Non sono stati rilevati contatti negli ultimi 15 giorni")
1.1.3.1.7.1: Toast.makeText(response).show()
[else]
1.1.3.1.8: notifyResponse("Hai già dichiarato la tua positività negli ultimi 7 giorni!")
1.1.3.1.8.1: Toast.makeText(response).show()
Powered By: Visual Paradigm Community Edition

4.1.2 Context diagram

Il **Context Diagram** è utile per definire i confini del sistema o di una parte di esso rispetto all'environment esterno, mostrando le entità che interagiscono con esso.

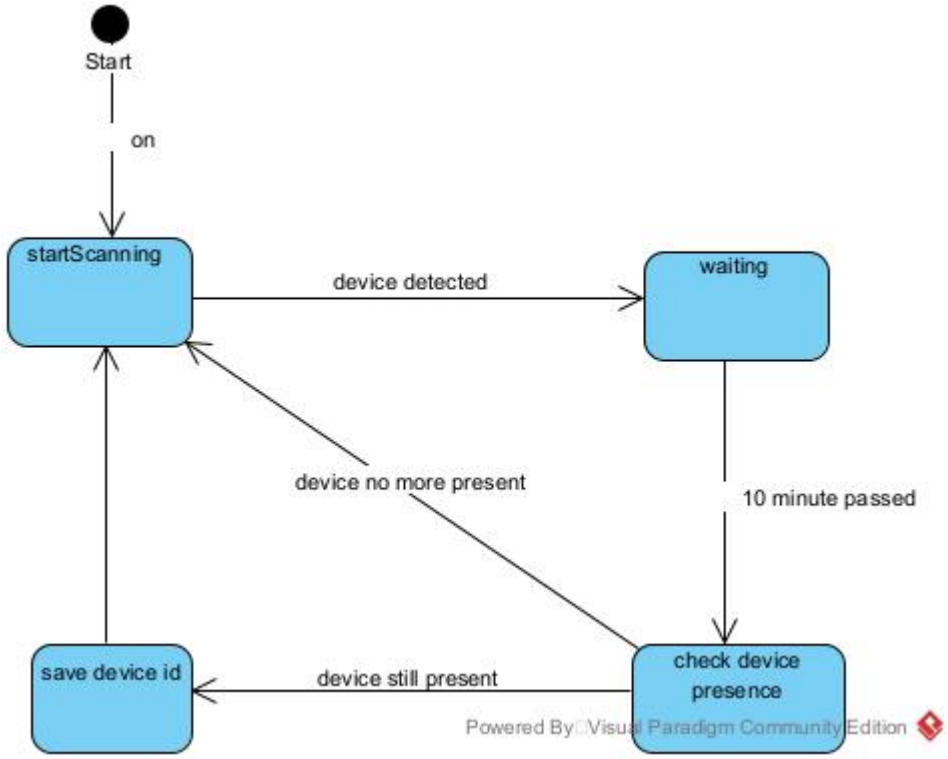


Le **classi di interfacciamento** sono definite come segue:

1. **Android device:** rappresenta il dispositivo utilizzato da proprietario e utente registrato per poter interfacciarsi con la UserInterface.
2. **UserInterface**: rappresenta l'interfaccia d'entrata al sistema per il lato Android client.
3. **Manager Device**: rappresenta il dispositivo utilizzato dal proprietario per poter interfacciarsi con la ManagerInterface
4. **ManagerInterface**: rappresente l'interfaccia d'entrata al sistema per il lato Manager Client.
5. **AdministrationDevice:** rappresenta il dispositivo utilizzato dall'amministratore di sistema per poter accedere ai metodi esposti dalla AdministratorInterface.
6. **AdministratorInterface:** rappresenta l'interfaccia d'entrata al sistema per il lato Administrator client.
7. **MailService:** rappresenta il sistema esterno capace di inviare emails.
8. **MailServiceInterface:** rappresenta la sezione del sistema che si occupa di interfacciarsi con il servizio esterno di invio emails.

4.1.3 State chart diagram

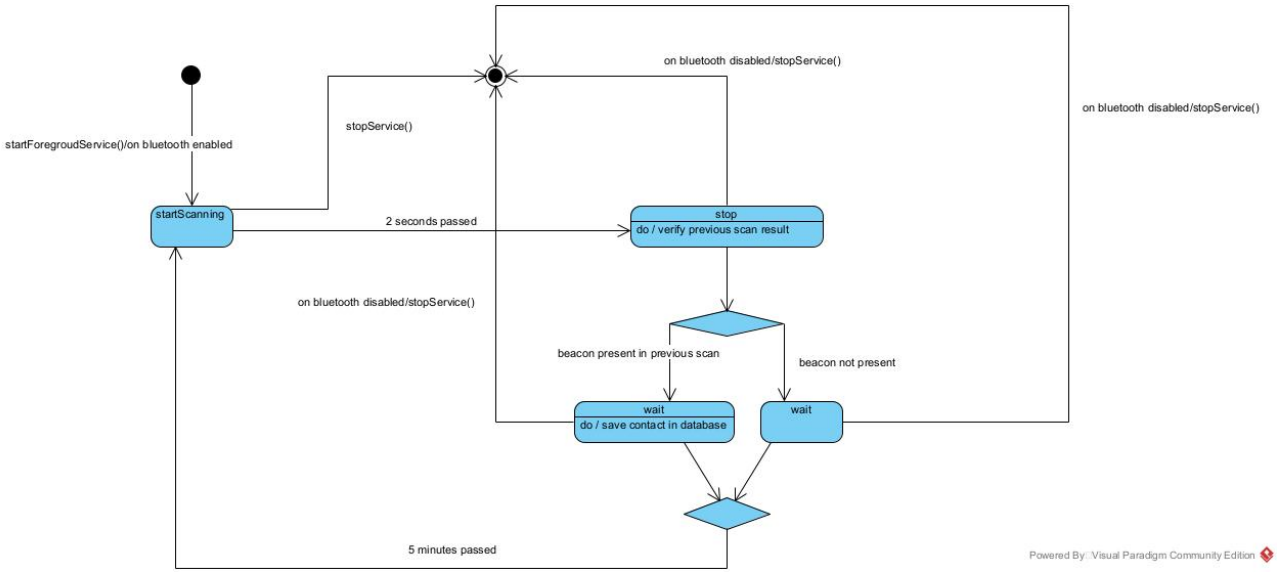
Uno **statechart diagram** descrive il flusso di controllo da uno stato ad un altro. Gli stati sono definiti come una condizione nella quale un oggetto esiste e cambia quando qualche evento viene scatenato. Lo scopo principale di uno statechart diagram è di modellare il ciclo di vita di un oggetto dalla sua creazione alla terminazione. In questa prima fase di analisi, si è abbozzato il **meccanismo di funzionamento del sottosistema dedicato alla rilevazione ed eventualmente il salvataggio di un contatto** attraverso la descrizione degli stati che esso assume.



Il diagramma di cui sopra mostra una prima analisi dei diversi stati del sottosistema dedicato alla rilevazione (ed eventualmente il salvataggio) di un contatto e sono stati delineati i trigger che avrebbero scatenato il passaggio da uno stato ad un altro.
Sono stati individuati **quattro stati** concettuali differenti:

1. **startScanning**: è lo stato scatenato all'atto dell'avvio del servizio e rappresenta la fase di scansione vera e propria. Quando un device viene rilevato nelle vicinanze il modulo passa nello stato di "waiting".
2. **waiting**: è uno stato di attesa; dopo il modulo passa allo stato di "check device presence".
3. **check device presence**: è lo stato che identifica la seconda ricerca di device nelle vicinanze necessaria a stabilire se la rilevazione di un dispositivo è fortuita o da considerarsi un contatto: nel caso il medesimo dispositivo non fosse rilevato nella seconda scansione, il modulo passa allo stato di "startScanning". In caso contrario il modulo passa allo stato di "save device id" poiché deve considerarsi contatto.
4. **save device id**: è lo stato che rappresenta il salvataggio del device rilevato per due volte consecutive; dopo il salvataggio il modulo si riporta nello stato di "startScanning" senza alcun evento.

È stato scelto di **basare l'algoritmo su due scansioni consecutive**: sostanzialmente tra due scansioni intercorre un **tempo di cinque minuti** e **se un dispositivo viene rilevato due volte consecutive allora viene considerato contatto**. Le disposizioni governative considerano un contatto di durata pari a quindici minuti continuativi ed utilizzano scansioni da cinque minuti. In questo caso è stata fatta una **scelta più conservativa** scegliendo un tempo pari a dieci minuti in modo tale da ridurre il numero di falsi negativi.



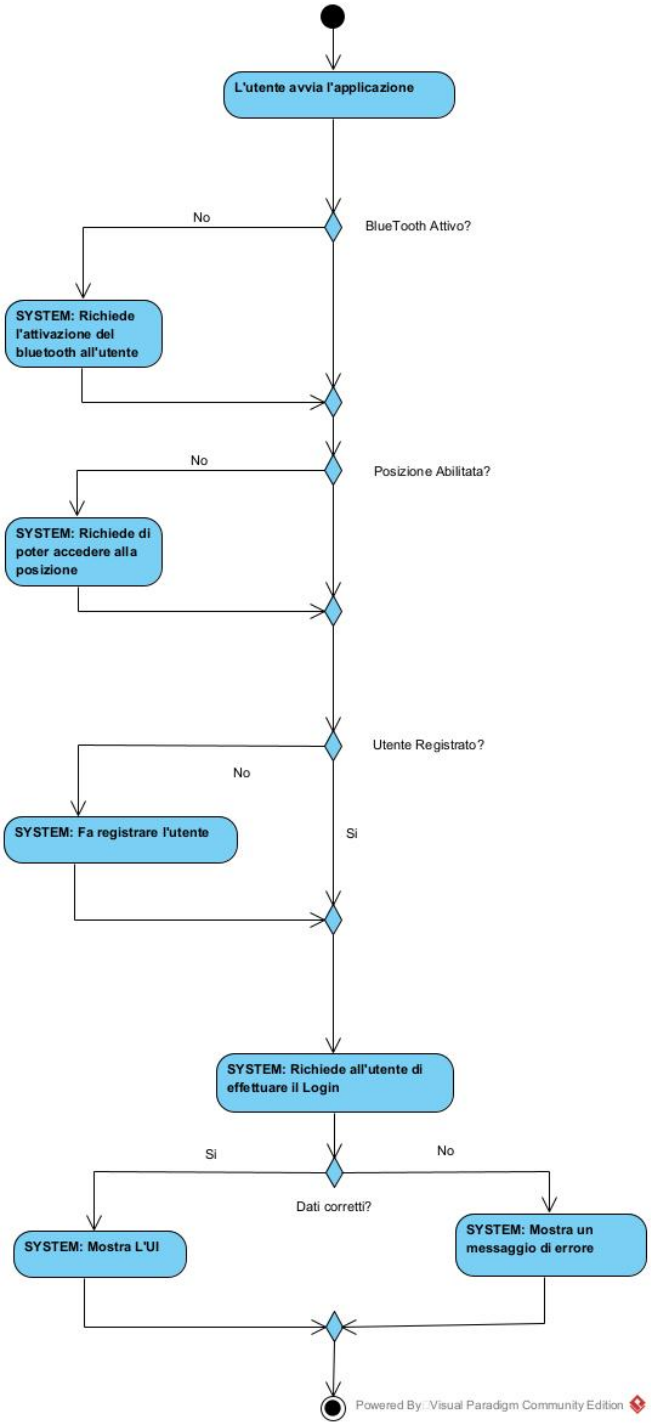
In questo state chart diagram è rappresentato nel dettaglio **l'algoritmo di scansione dei dispositivi ed il salvataggio dei contatti**.

È possibile distinguere i seguenti stati:
startScanning: rappresenta la **fase di scansione** vera e propria, dura due secondi. In questa fase **vengono ricercati tutti i dispositivi nel raggio di due metri**. Passati due secondi si passa automaticamente alla fase di stop.
stop: è la fase in cui vengono confrontati i beacon presenti nell'attuale scansione con quella precedente. Per ogni beacon:
- **se questo è presente in entrambi le scansioni** allora è stato rilevato un contatto, quindi si passa allo stato di "wait" con successivo **salvataggio del contatto in locale**.
- **se il beacon è presente solo nell'attuale scansione** allora si passa allo stato di "wait" senza compiere alcuna azione.

wait: è lo stato di attesa tra una scansione ed un'altra. In questo stato possono essere o meno salvati contatti nel database a seconda del risultato della funzione di "verify previous scan result" del precedente stato. Dopo 5 minuti, il modulo si riporta automaticamente nello stato di "startScanning".
In qualunque stato il modulo si trovi **è sempre possibile interrompere il servizio** di scansione attraverso l'azione di "**stopService()**".
Tale azione viene scatenata in tutta una serie di circostanze quali:

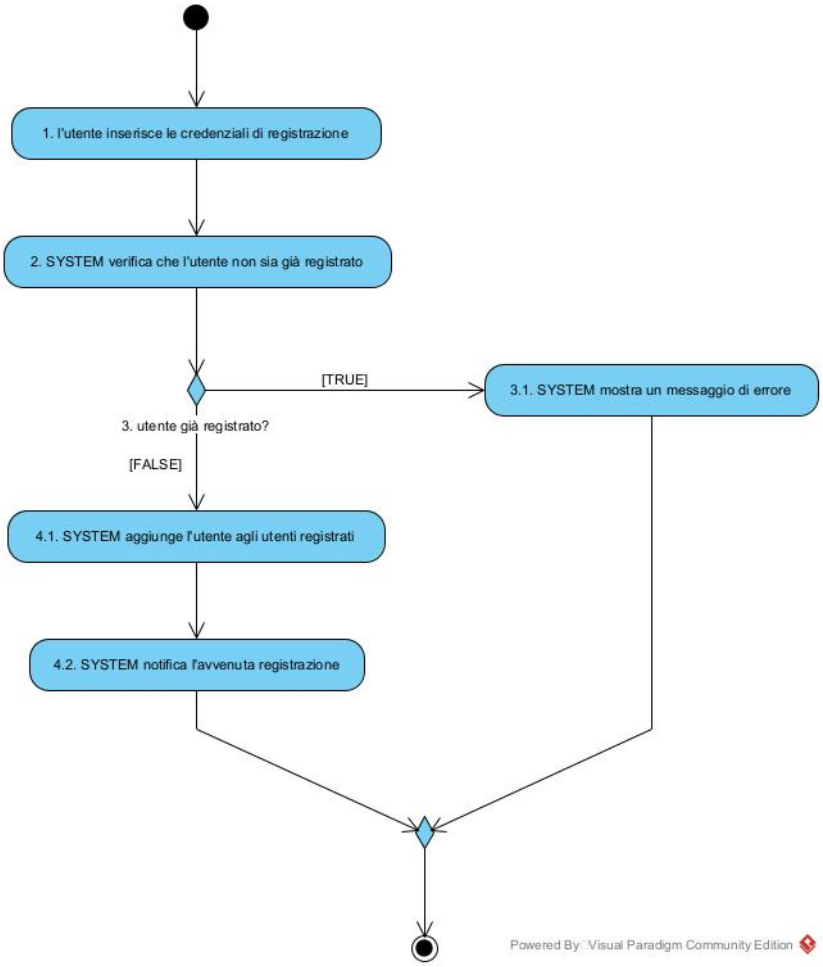
1. **spegnimento improvviso** del bluetooth.
2. **spegnimento** del dispositivo.
3. **Chiusura forzata dell'applicazione**.
4. **Stop del servizio deciso dall'utente** tramite l'applicazione.

4.1.4 Activity diagram



Questo Activity Diagram è stato utilizzato per modellare l'avvio dell'applicazione. Entrando più nel dettaglio nel momento in cui l'utente apre l'app, vi è un controllo sia sull'abilitazione del Bluetooth che sulla posizione, nel caso in cui non fossero attive, il sistema richiede all'utente di abilitarle per poter usufruire a pieno dei servizi offerti dell'applicazione.

A questo punto il sistema verifica se l'utente sia registrato o meno, nel caso in cui l'utente non sia registrato (e quindi apre l'app per la prima volta) si scatena il caso d'uso della registrazione, modellato nel seguente Activity Diagram :



In Alternativa, se l'utente è già registrato sull'applicazione il sistema fa loggare l'utente e mostra l'Interfaccia Utente nel caso in cui i dati inseriti da quest'ultimo siano corretti, altrimenti si limita a mostrare un messaggio d'errore.

Nei seguenti diagrammi invece viene modellata l'entrata e l'uscita in/da un luogo da parte dell'utente.

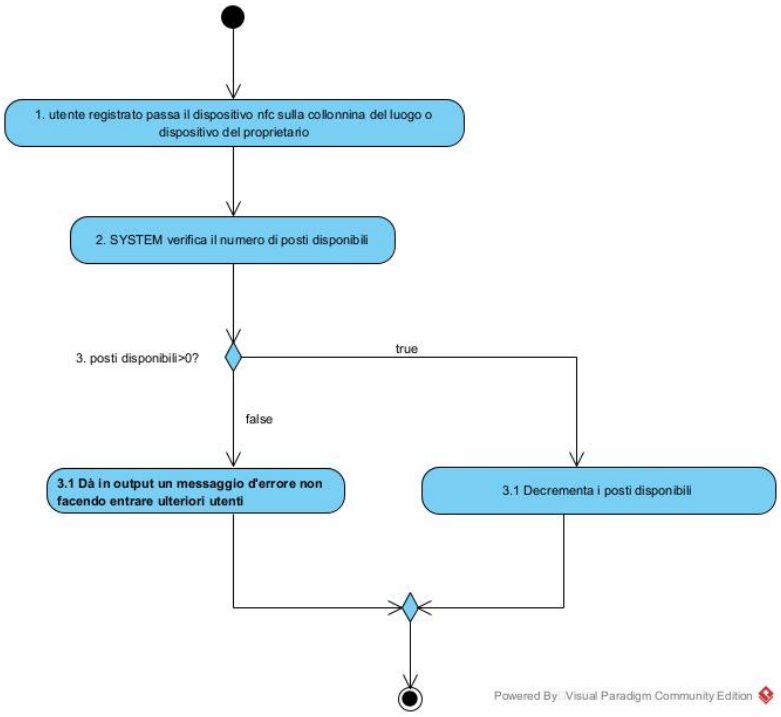


Figura 1: Utente entra in luogo

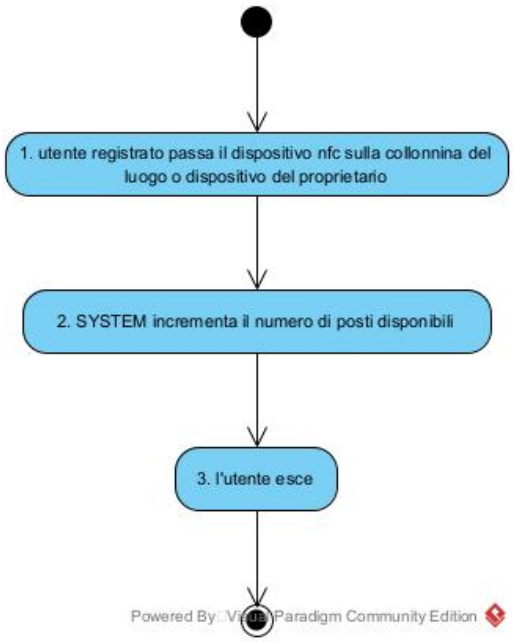
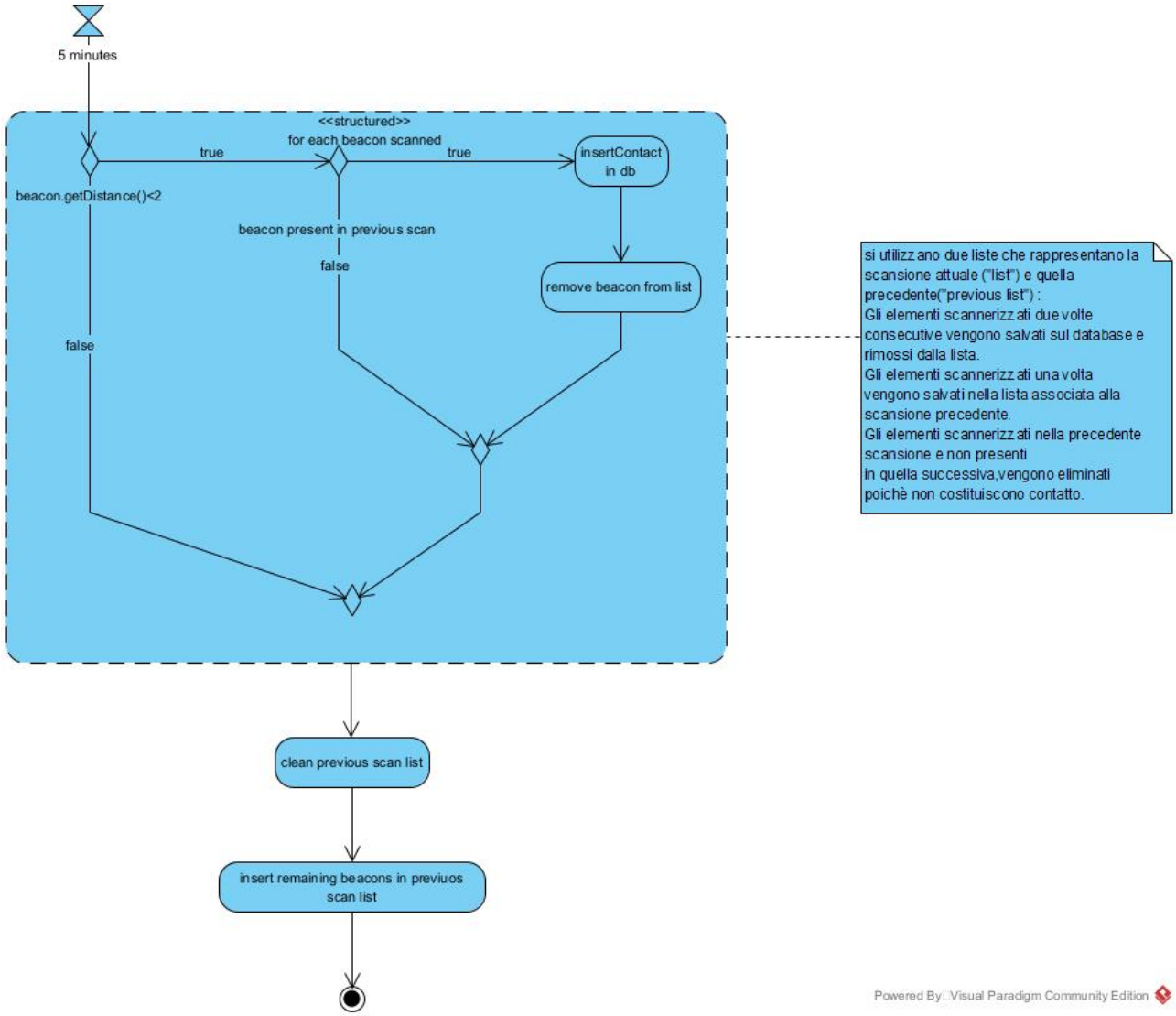
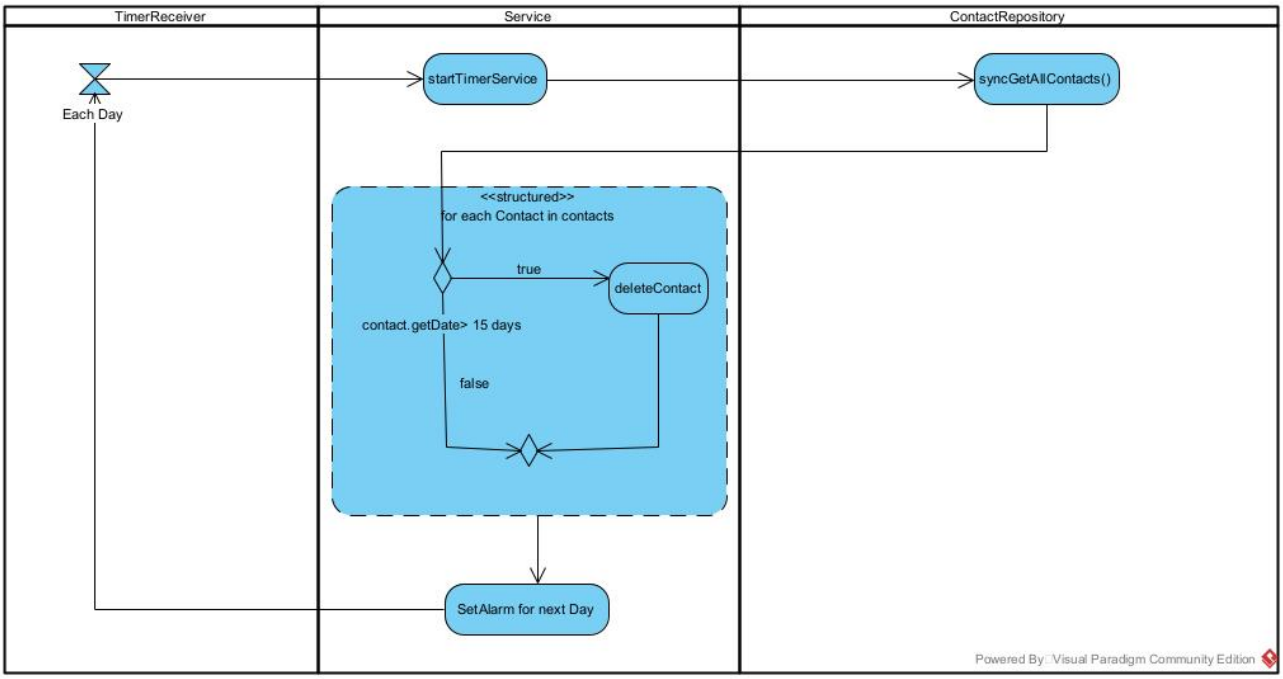


Figura 2: Utente esce da un luogo

Il seguente activity è stato utilizzato invece per modellare un contatto. Ogni 5 minuti si controlla, per ogni beacon presente nella scansione se la distanza sia minore di 2 metri, in caso affermativo si verifica se il beacon preso in questione sia presente o meno nella scansione precedente:



In caso affermativo, il beacon viene salvato sul database e rimosso dalla lista. Infine vi è una "pulizia" della lista con i beacon scansionati alla precedente scansione, inserendo quelli nuovi all'interno di quest'ultima.



In questo diagramma è stato messo in evidenza il **workflow della funzione di cancellazione contatti**, consistente in un'attività giornaliera indipendente che avviene lato client. Ogni giorno, **a prescindere dallo stato dell'applicazione**, un service viene richiamato per cancellare i contatti residenti sul database locale da più di 15 giorni. Nello specifico:
Al primo avvio dell'applicazione viene lanciato un TimerService che provvede ad impostare un Alarm per il giorno successivo affinché possa essere avviato il servizio stesso. In particolare, ogni giorno:

1. **viene aperto un nuovo thread** per avviare il servizio attraverso la funzione "**startTimerService()**"
2. viene chiamata la funzione "**syncGetAllContacts()**" in modo da **ottenere tutti i contatti salvati sul dispositivo in locale**.
3. **Viene controllata la data di ciascun contatto** in modo tale da cancellare i contatti considerati obsoleti.
4. Infine, **il servizio imposta un Alarm per richiamare se stesso il giorno successivo** al medesimo orario.

La scelta di reimpostare ogni giorno un Alarm (sostanzialmente una sveglia per effettuare certe azioni, definite in un opportuno oggetto di tipo Receiver) è stata dettata dalla necessità di **effettuare le operazioni sopra elencate giornalmente sempre allo stesso orario**. Infatti, ritardi accumulati su tali operazioni avrebbero comportato la compromissione del meccanismo di notifica.

4.2 Architettura software

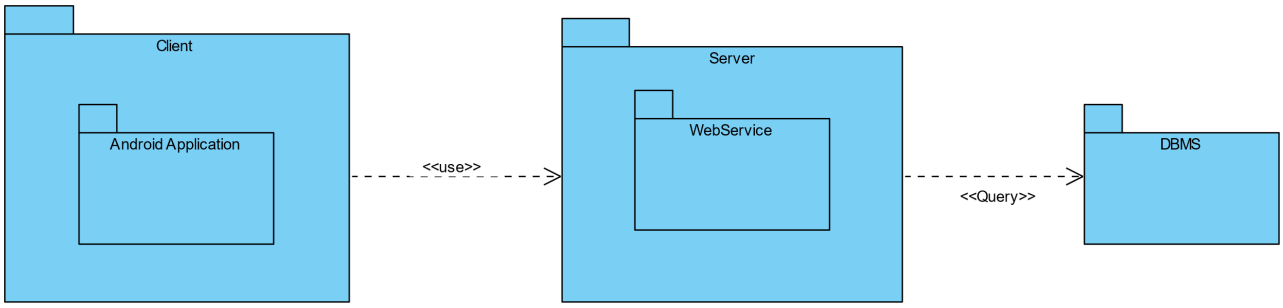
In questo paragrafo sarà descritta l'architettura software relativa al caso d'uso scelto procedendo con una **strategia top-down**. Nello specifico, nel seguente ordine:

1. Descrizione dell'**architettura ad un alto livello di astrazione**, illustrando lo **stile architetturale utilizzato** e le ragioni che ci hanno portato a adottarlo, i suoi vantaggi e svantaggi.
2. Elenco dei **pattern architetturali** scelti nella fase iniziale di progettazione, **esplicitando la ratio** che ha condotto a tali decisioni.
3. **Analisi di ogni sottosistema del software a diversi livelli di dettaglio** in modo tale da:

    - **Ripercorrere il processo logico e pratico** (dal punto di vista dell'implementazione) che ci ha portato alla versione definitiva del software.
    - **Rendere più chiara l'intera struttura del software**.

4.2.1 Architettura generale

Descrizione **a livello globale** dell'intera architettura del sistema software:



Il sistema è stato progettato facendo riferimento allo stile Client-Server.
Lo **stile client-server** è uno stile architetturale a livelli che ne prevede **due soli** e nasce nell'ambito dei sistemi distribuiti per risolvere problemi di comunicazione tra parti diverse.
L'idea alla base di questo stile architetturale è quello di avere due componenti diversi dell'architettura su macchine diverse, ciascuno con il proprio ruolo:

- **Client**: componente che **conosce l'identità del server** e può essere **thin** (se non implementa application logic) o **fat**.
- **Server**: fornisce dei servizi, non conosce numero e identità del client.

Generalmente i **connettori** sono basati su RPC e su protocolli di interazione di rete.
È anche possibile distinguere due tipologie di server:

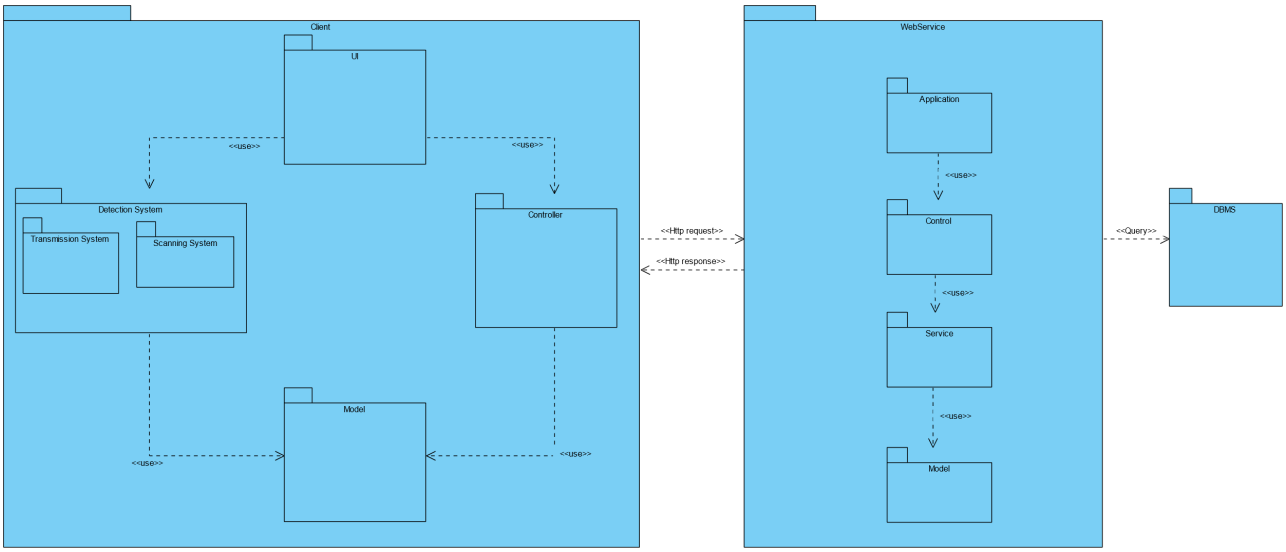
- **Stateless**: **non ha necessità di memorizzare informazioni** necessarie allo svolgimento del servizio richiesto del client. Si limita a rispondere inviando il dato richiesto.
- **Statefull**: ha **necessità di mantenere informazioni** per l'elaborazione del servizio richiesto per ogni client connesso.

Il sistema è stato progettato utilizzando questo stile architetturale. Le motivazioni che hanno portato a scegliere questo stile sono molteplici:

- **Privacy**: per tutelare la riservatezza degli utenti, come **prescritto dai vincoli precedentemente elencati**. Infatti, per questo motivo non era contemplabile fornire ai dispostivi la capacità di ottenere direttamente informazioni private quali le e-mail (motivo per cui è stato elaborato il sistema di identificazione esposto nel capitolo "Meccanismo di identificazione del sistema software").
- **Sicurezza**: il WebService è l'unico a poter accedere al database con le proprie credenziali e presenta la possibilità di incrementare facilmente la sicurezza della comunicazione.
- **Scalabilità**: è semplice duplicare le istanze di WebService dispiegandole su più nodi fisici. È inoltre possibile migliorare le performance dispiegando le istanze dietro un proxy con funzioni di Load Balancing.

Il **lato client nel sistema software** progettato è dato dall'applicazione Android. All'interno del sistema l'applicazione ha il compito di rilevare i dispositivi in un raggio di due metri e, in caso di contatto, salvarne gli identificativi ed il giorno del contatto. In caso di positività, sarà compito dell'Android Client inviare una richiesta di servizio al Server (che offre un servizio di notifica).
Il lato server nel sistema software progettato, stateless, è stato pensato come un WebService che, una volta ricevuto dal client la lista di identificativi con cui è entrato in contatto, provvede ad interrogare un database esterno sul quale risiedono le corrispondenze tra identificativi ed e-mail, al fine di trovare quest'ultime per poter inviare a questi un messaggio di posta elettronica che notifichi all'utente corrispondente l'avvenuto contatto con un positivo.

Si procede con l'analisi più nel dettaglio della struttura del sistema:



Questo diagramma rappresenta l'organizzazione in package del sistema ancora ad alto livello, evidenziando i pattern architetturali adottati in fase di analisi.
Innanzitutto, si analizza come è stato concepito **il Client**:

Esso presenta **un'architettura strictly layered**:

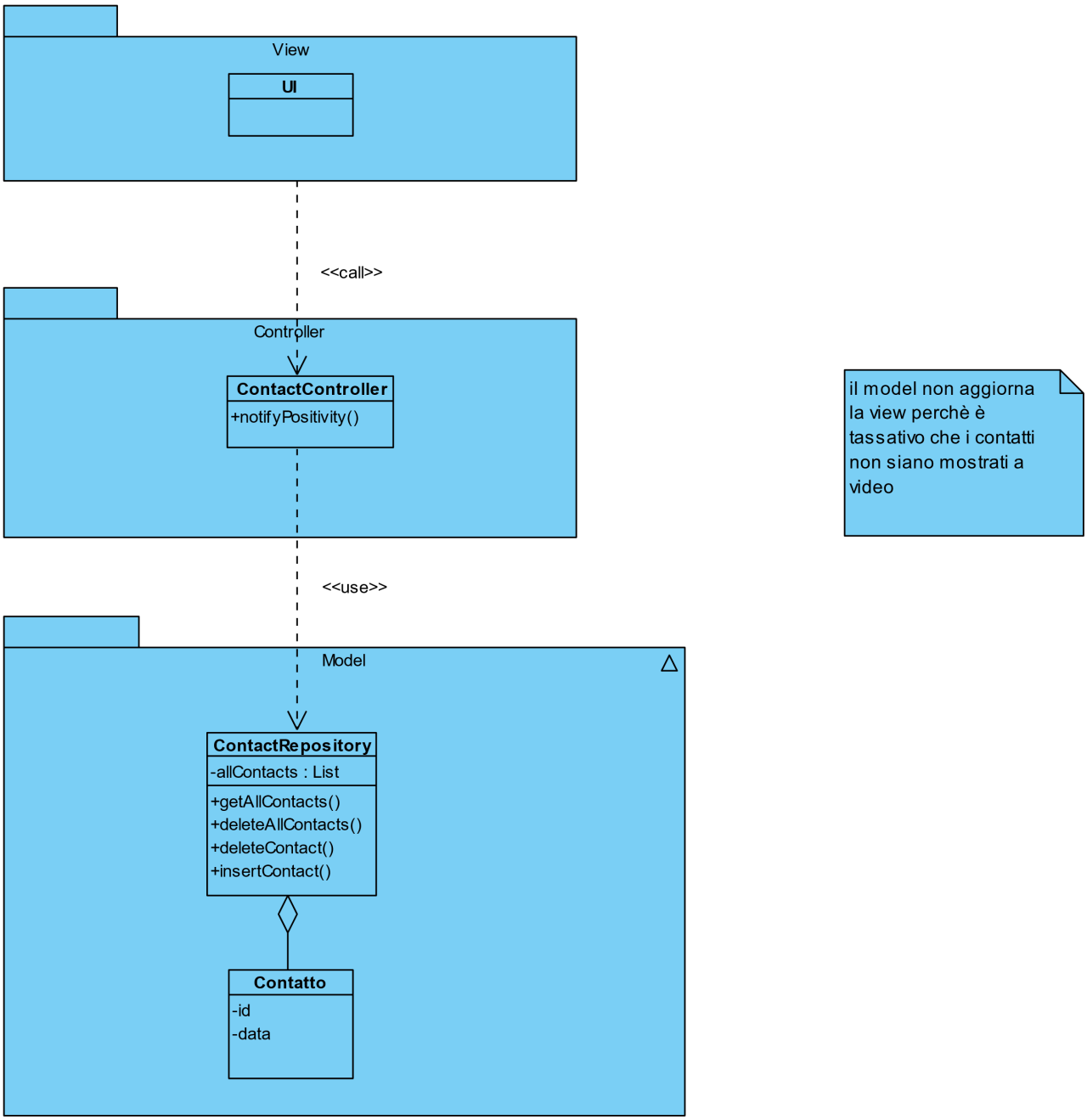
- Al livello più alto troviamo la User Interface, attraverso cui l'utente interagisce con il sistema.
- Al livello intermedio troviamo due **moduli indipendenti**:

1. **Sottosistema di rilevamento e trasmissione dei segnali**, necessari rispettivamente a trovare e a rendersi visibili agli altri dispositivi, del tutto indipendente dal Controller.
2. **Un Controller** necessario a svolgere le funzioni di manipolazione ed elaborazione dei dati del model in modalità utili al caso d'uso scelto (Il controller gestisce i contatti, elimina quelli obsoleti, si interfaccia con l'esterno etc.).

- Al livello più basso troviamo il **Model**, che rappresenta ed astrae i dati presenti su dispositivo.

4.2.2 Pattern architetturale Model-View-Controller

Analisi a livello di dettaglio intermedio la struttura statica del sottosistema che gestisce il caso d'uso lato client e le ragioni che hanno portato alla scelta di questo pattern architetturale.



Il **pattern MVC** nasce dalla necessità di **disaccoppiare i componenti** di presentazione dai componenti che gestiscono i dati di interesse.

Model: insieme di classi le cui istanze rappresentano i **dati da visualizzare e manipolare**, definisce le regole di Business ed **espone funzionalità per l'accesso e l'aggiornamento**. Notifica alla View aggiornamenti verificatisi in seguito alle richieste del controller.
È stato adottato un **Repository pattern** attraverso l'implementazione della classe ContactRepository per una serie di motivi:

- Una volta implementato il Repository Pattern, **è più semplice utilizzare il Model** secondo le modalità definite in fase di progettazione senza preoccuparsi dei dettagli implementativi.

- **È possibile cambiare il layer di persistenza senza incidere sul funzionamento della logica** (cambiare meccanismo di persistenza, libreria, etc.).
- Diventa **più semplice testare il Model** una volta implementato il Repository.

Controller: insieme di classi che funge da punto di raccordo, le cui istanze **gestiranno l'interazione tra la View ed il Model**. Il controller è il componente responsabile di trasformare le interazioni dell'utente sulla View in azioni eseguite dal model e quindi di implementare la **Business Logic dell'applicazione**.
Il controller utilizzato è un **controllore di caso d'uso**, un oggetto creato ad hoc che ha come unico ruolo quello di **coordinare l'esecuzione di una sequenza di operazioni atte ad implementare il caso d'uso scelto**.

View: insieme delle classi che contiene oggetti usati per rendere in output nella UI i dati contenuti nel model ed ha la responsabilità di **delegare le interazioni utente al controller**.
Vantaggi:

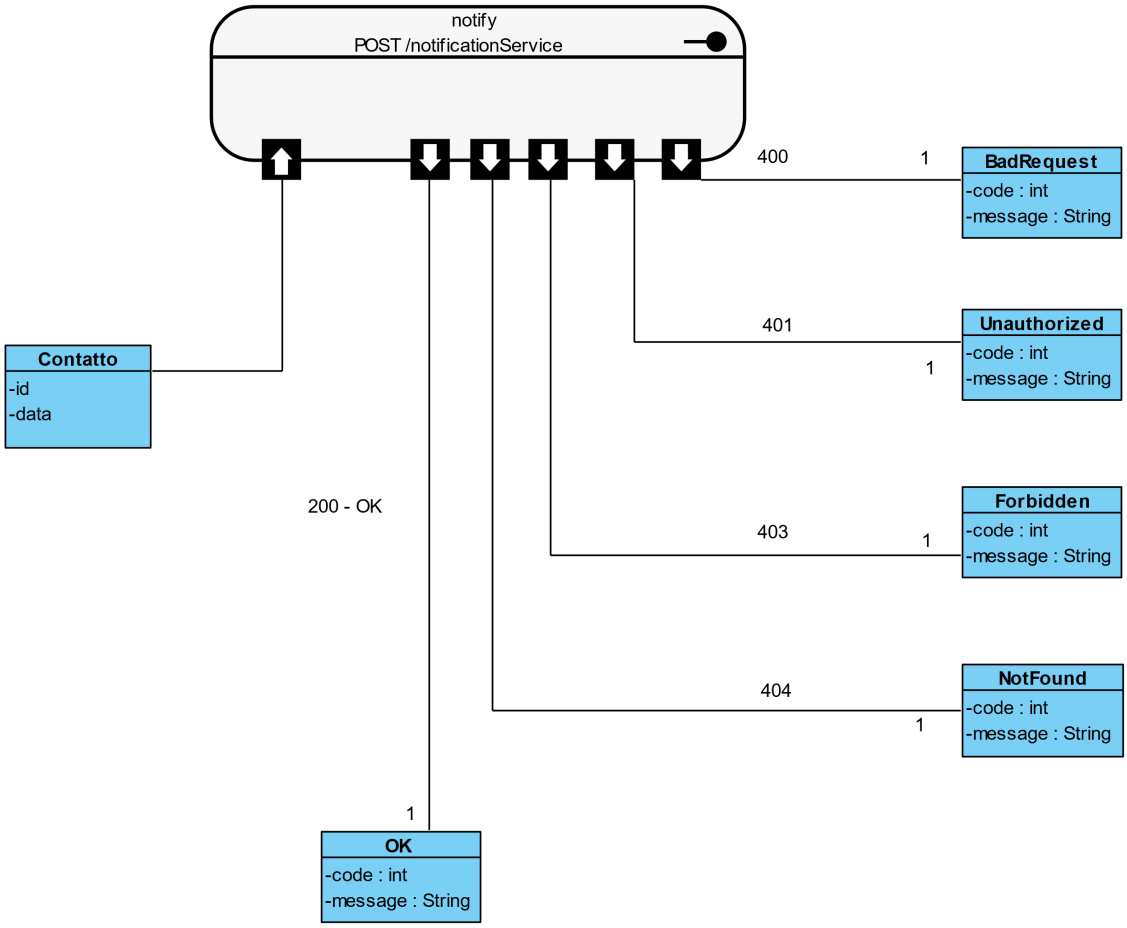
- Alta coesione
- Basso accoppiamento
- Alta modificabilità

Svantaggi:

- Navigabilità del codice
- Possibilità di rendere il Controller una classe con troppe responsabilità, troppo complessa (Bloated Controller).

4.2.3 Rest

Per quanto riguarda la comunicazione tra lato client e server è stato adoperato uno stile architetturale REST.



REST, come stile architetturale, è stato definito sulla base del protocollo **http** per la comunicazione tra Client e WebService (Restful).
Rest è infatti uno **stile architetturale** basato sul trasferimento di rappresentazioni di risorse da un server ad un client ed è alla base del web. I servizi restful comportano un sovraccarico inferiore rispetto ai così detti grandi servizi web.
I sistemi REST seguono i seguenti principi:

Stateless: non prevedono il concetto di sessione.

La comunicazione tra utente del servizio (client) e servizio (server) deve essere senza stato tra le richieste. Ciò significa che ogni richiesta da parte di un client dovrebbe contenere tutte le informazioni necessarie al servizio per comprendere il significato della richiesta.
Se il server è senza stato può **scalare** molto più **facilmente**, infatti ciò potrà accadere semplicemente dispiegando più istanze del servizio ed utilizzando un Load Balancer che si ponga davanti alle istanze dei server e passi le richieste seguendo opportune regole.
Rappresentazione della risorsa: **la risorsa** (un elemento di dati**) esiste indipendentemente**

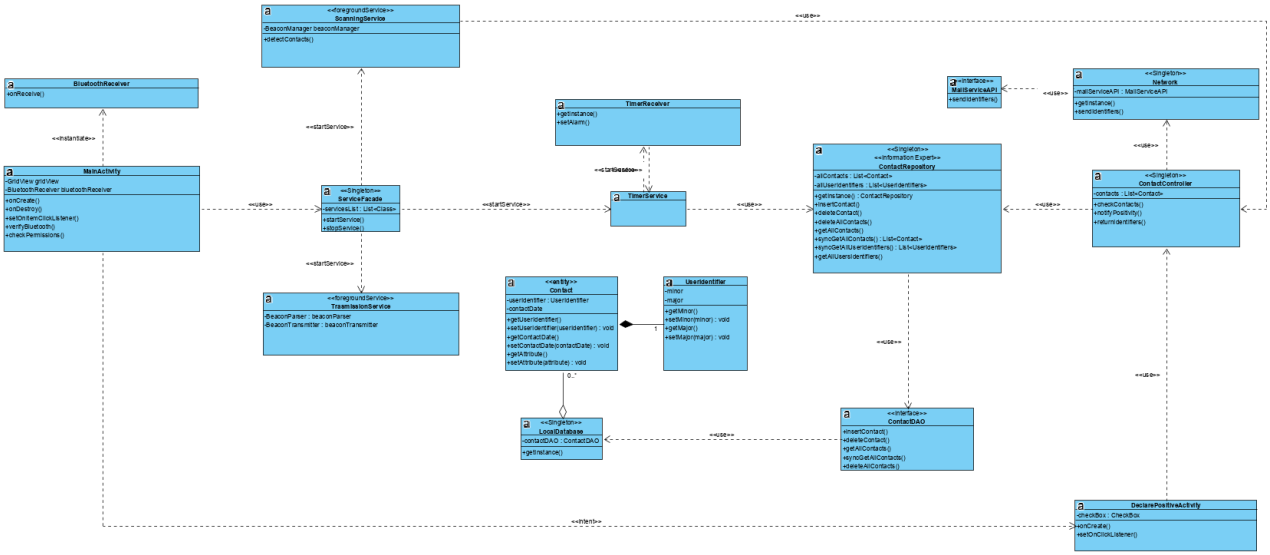
dalla sua rappresentazione.
I servizi RESTful non sono vincolati ad usare la rappresentazione XML dei dati, come avviene per SOAP, ma possono anche usarne altre, come ad es. JSON (Javascript Object Notation), maggiormente ottimizzate per un minore overhead. In REST **ogni risorsa ha un identificatore unico**: l'URL. Ad ogni risorsa sono associate quattro possibili operazioni: creare, leggere, aggiornare ed eliminare (POST, GET, SET, DELETE). **I clienti interagiscono con le risorse tramite le loro rappresentazioni**, il che è un modo potente per mantenere i concetti di risorse astratti dalle loro interazioni.
Nel diagramma è descritta la **POST request**:

- **Formato dati**: JSON.
- **Parametri input**: è stata definita una classe POJO "contatto", che modellasse l'avvenuto contatto incapsulando i dati di interesse (data, identificativo), i quali sono inviati, serializzando opportunamente l'oggetto, al servizio che provvede a notificare l'avvenuta positività agli utenti.
- **Output:** messaggio di risposta tra quelli elencati in figura, costituito da un codice di stato standard del protocollo http e un breve messaggio esplicativo.
- **Identificazione della risorsa**: viene specificato un **URL specifico** ("**/notificationService**") oltre che ad un base URL univoco.

4.3 Documentazione di raffinamento

In questa sezione è raffinata l'architettura delineata in fase di analisi attraverso i diagrammi che seguono, sia dal punto di vista strutturale che comportamentale. Tali diagrammi, sebbene **non coincidenti perfettamente rispetto all'implementazione effettiva**, costituiscono il più valido strumento di **comprensione** del progetto. Essi, infatti, costituiscono il punto di **congiunzione tra la fase di analisi**, con la quale condivide la chiarezza e la semplicità di interpretazione, **e la fase di implementazione**, con la quale condivide un certo livello di dettaglio. In questo modo **è possibile focalizzare l'attenzione sulle componenti fondamentali del software** evitando le complicazioni di leggibilità causate da elementi utili all'implementazione effettiva ma supplementari.

4.3.1 Class diagram

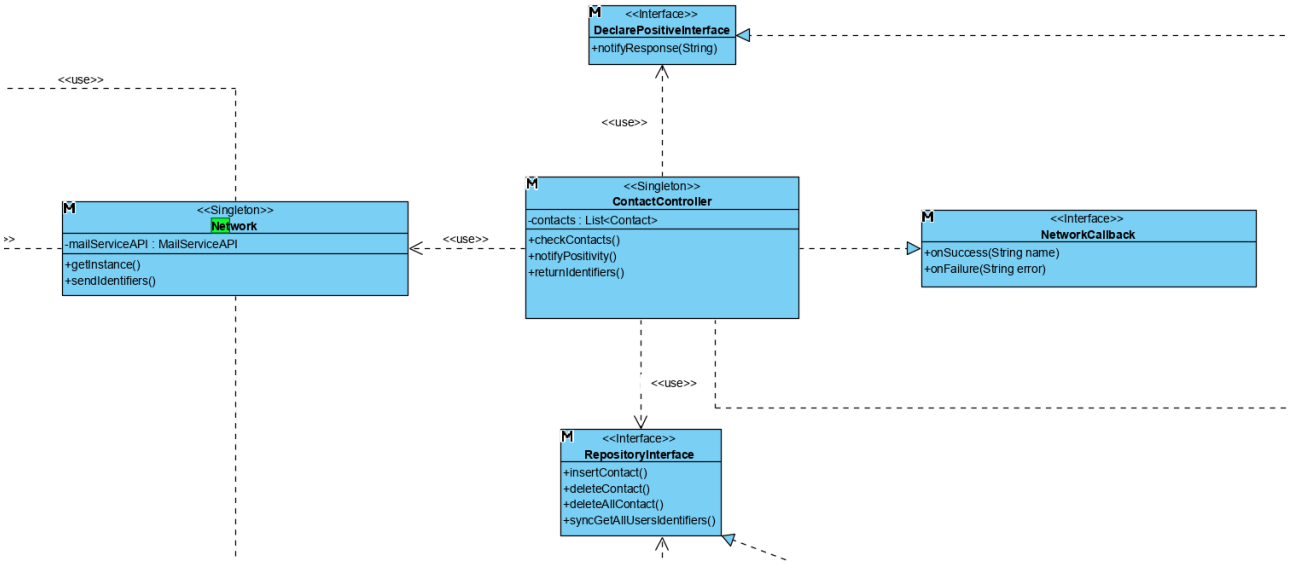


Questo Class Diagram è un **diagramma intermedio** che offre una vista statica del sistema più chiara. Segue al diagramma una **breve descrizione delle classi** presenti nel diagramma ed il loro scopo:

mailServiceAPI: è l'interfaccia usata per la comunicazione con il WebService ed espone un solo metodo, sendIdentifiers(), per l'invio degli identificativi.
network: è la classe responsabile della gestione della comunicazione verso l'esterno ed espone un solo metodo, sendIdentifiers(), per l'invio degli identificativi. La classe è implementata come Singleton per prevenire problemi di inconsistenza generati dalla creazione di più istanze della stessa.
ContactController: è la classe che si occupa della manipolazione dei contatti e degli identificativi. Espone il metodo di checkContacts() per la rilevazione dei contatti, il metodo notifyPositivity() per il recupero di tutti i contatti già residenti in locale ed il metodo returnIdentifiers() per notificare la propria positività e quindi inviare i contatti ricevuti al WebService.
DeclarePositiveActivity: è l'activity che permette la dichiarazione della propria positività tramite la pressione di una check Box.
Contact: è la classe che modella il contatto. Contiene gli attributi di "userIdentifier" per l'identificativo e una "contactDate" utile per l'individuazione e la successiva cancellazione dei contatti obsoleti.
LocalDatabase: è la classe responsabile della creazione e della gestione dell'istanza del Room Database. Anche questa classe è dichiarata "Singleton", ancora una volta per evitare problemi di inconsistenza.
UserIdentifier: è la classe che racchiude i due campi di tipo intero utili a costruire l'identificativo di uno specifico utente registrato.
ContactRepository: è la classe responsabile della manipolazione, l'estrazione e la gestione dei dati su database locale (contatti) ed astrae completamente il Model.
ScanningService: è la classe, modellata come servizio, che si occupa della funzione di

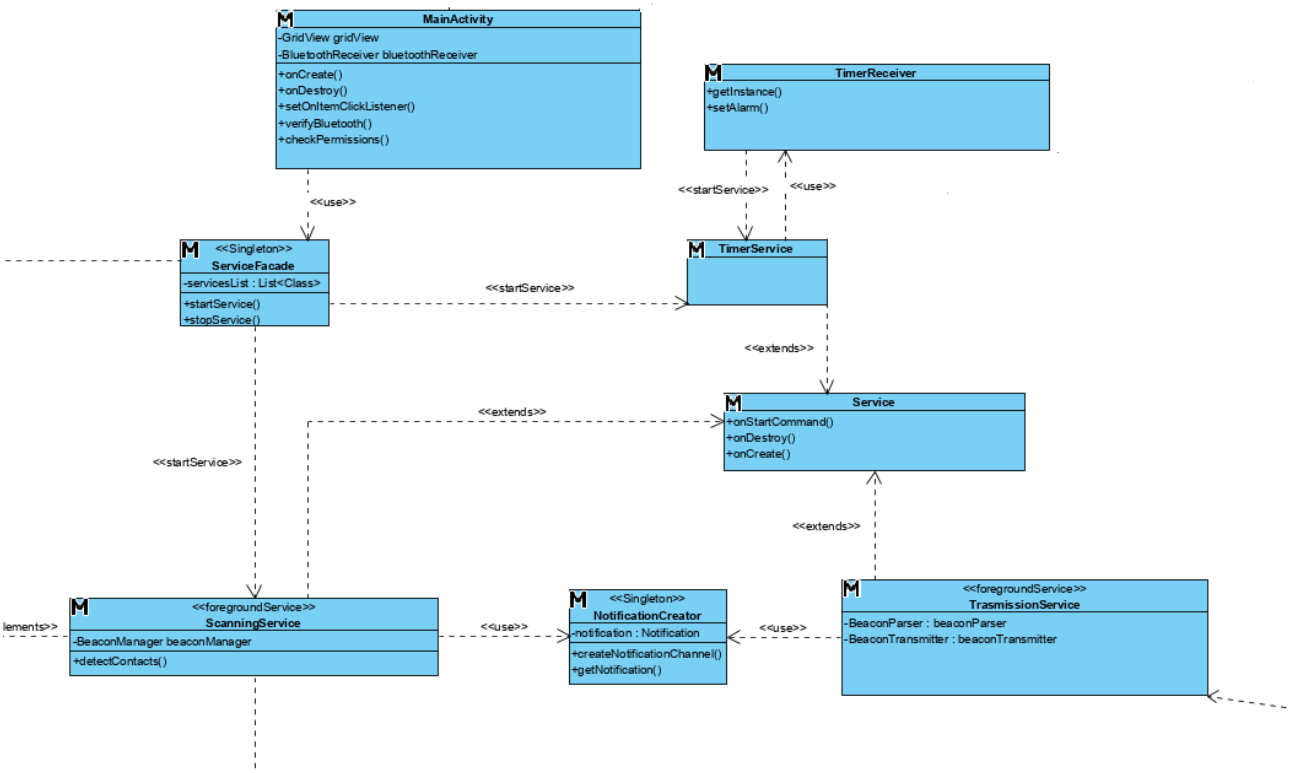
rilevamento di contatti e presenta un metodo “detectContacts()”.
BluetoothReceiver: attraverso questo oggetto è possibile gestire la pressione del tasto di accensione/spegnimento del bluetooth attraverso il metodo “onReceive()”.
MainActivity: è l’activity principale dell’applicazione ed attraverso essa vengono inizialmente richiesti i permessi necessari al corretto funzionamento del sistema con il metodo “checkPermissions()”. Garantiti tali permessi, l’activity gestisce il layout ed i contenuti della home page dell’applicazione.
ContactDAO: è l’interfaccia utilizzata per interfacciarsi con il database locale per il salvataggio e l’eliminazione dei contatti.
ServiceFacade: è la classe facade responsabile dell’avvio e dell’arresto dei servizi dell’applicazione attraverso i metodi startServices() e stopServices().
TrasmissionService: è la classe, modellata come servizio, responsabile della gestione delle funzioni di trasmissione di beacon.
TimerService: è la classe responsabile della cancellazione dei contatti obsoleti. Questa funzione viene richiamata ogni giorno al fine di non creare notifiche fallaci in fase di dichiarazione di positività
TimerReceiver: è la classe receiver responsabile dell’impostazione di un Alarm attraverso il metodo “setAlarm” necessario a chiamare la funzione di cancellazione dei contatti obsoleti.

Scompattando l’intero class diagram si trovano varie sezioni, che giustificano le principali scelte di progetto.



Il primo problema che si è palesato è stato **l’alto accoppiamento tra User Interface e Domain Model**, a causa della peculiare struttura dell’architettura Android.
È sorta dunque la consapevolezza di dover adottare un **pattern architetturale che ci consentisse di attenerci al principio della Separation of Concerns**, dividendo in maniera quanto più netta possibile le responsabilità delle classi, **cercando di ottenere alta coesione all’interno delle stesse** ed un basso accoppiamento tra View e model.
Quindi si è cercato di tenere la View lontana da qualunque responsabilità che non sia quella di mostrare dati all’utente, **delegando la gestione dell’evento di interesse ad un controller**, il

quale coordina ed interagisce con il model.
Nel dettaglio è stato utilizzato un **Session Controller**, ovvero un oggetto creato ad hoc che ha come unico ruolo quello di **coordinare l'esecuzione di una sequenza di operazioni per implementare un caso d'uso** o uno scenario.

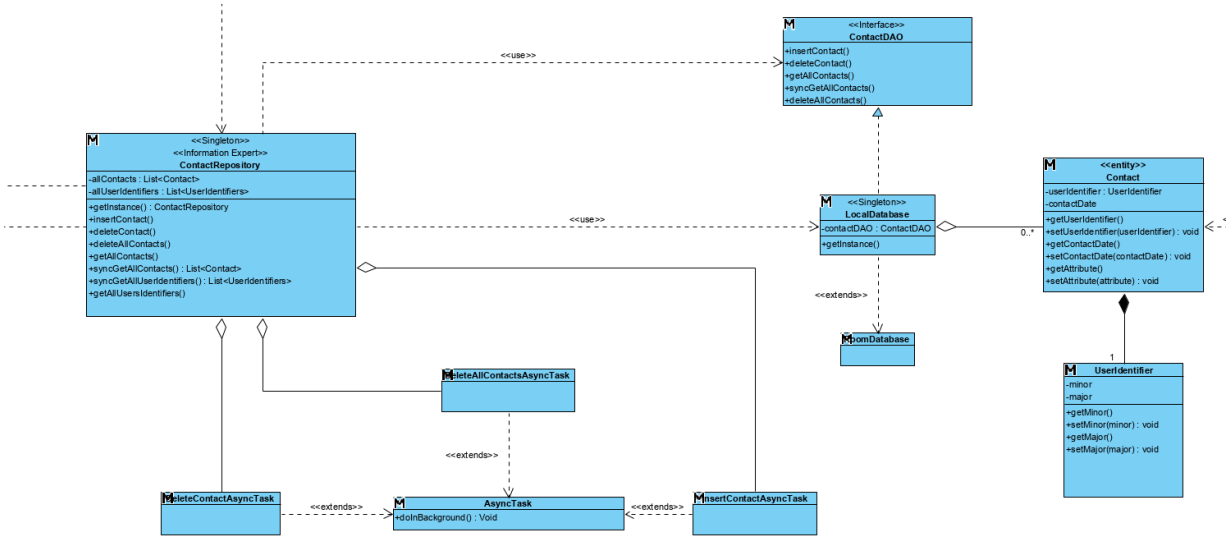


È stato scelto di utilizzare, e quindi, implementare un **Facade Pattern per incapsulare i servizi** in maniera tale da:

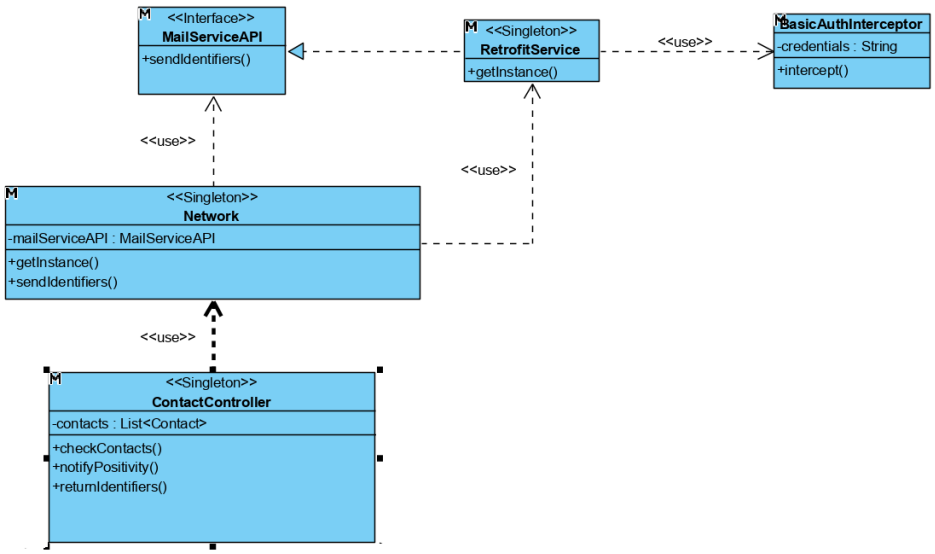
1. **semplificarne l'utilizzo da parte del sistema**, esponendo un unico metodo che si occupasse della **fase di boot** ed un altro che si occupasse della **fase di stop** (con conseguente distruzione) dei servizi.
2. **disaccoppiare l'activity dai servizi**.

Le funzioni di scansione, trasmissione ed eliminazione di contatti obsoleti sono state implementate come **Foreground Service**.
Questa scelta è stata preferita ad altre (come quella del Background Service) **poiché tale tipo di servizio è l'unico a poter sopravvivere al ciclo di vita dell'applicazione**.
Questi Servizi sono stati associati ad una singola Notification. A tal scopo è stata creata una classe Singleton**, NotificationCreator**, che restituisse **l'unica istanza di notifica**, affinchè venisse mostrata una singola notifica a video.
Le classi di **BroadcastReicever** del progetto sono state utilizzate con **due scopi differenti**: **l'una (BluetoothReceiver) che gestisse l'improvviso spegnimento del Bluetooth**, con il conseguente stop dei servizi di Scansione e Trasmissione**, l'altra (TimerReceiver) capace di rischedulare il servizio di cancellazione dei contatti considerati obsoleti ogni 24 ore**. Ogni

cancellazione viene rischedulata di giorno in giorno in quanto il sistema operativo non permette la pianificazione di un alarm (e quindi di un task) al di fuori del contesto dell'applicazione.



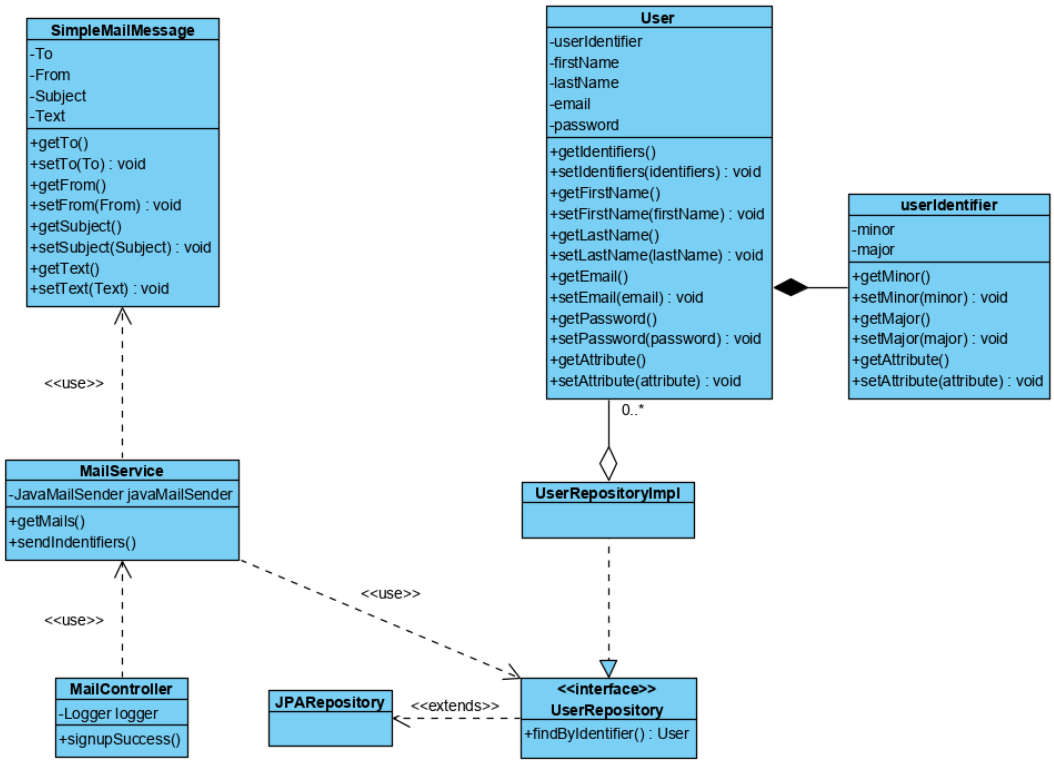
In generale, ogni classe dovrebbe avere responsabilità correlate alle informazioni che essa possiede e di cui è Information Expert, ma in questo caso **è stato preferito utilizzare classi logicamente separate**, i DAO, la cui unica responsabilità è quella di interfacciarsi con il Database.
Il framework utilizzato per gestire la persistenza dei contatti sul dispositivo (Room) ha consentito di definire un'interfaccia (**ContactDAO**) attraverso cui interagire con il Database interno, sollevando la responsabilità di implementare la classe concreta.
L'oggetto Repository è l'Information Expert dei contatti, ed è l'unica classe a poter accedere, attraverso i DAO al database. Pertanto**, è l'unica classe che possiede tutte le informazioni riguardanti i contatti**. Tale classe, infatti, rappresenta un'astrazione del Model ed ogni classe che avrà la necessità di interagire con esso si rivolgerà al Repository affinché le proprie richieste siano soddisfatte.



La classe Network è nata con l'intento di accorpare in un unico oggetto le responsabilità relative all'interfacciamento con la rete evitando così di assegnare responsabilità al controller fuori dalla sua competenza.
È stato utilizzato il pattern Observer affinché venisse notificato alla Repository il cambiamento dei dati all'interno del database, che in tal modo è costantemente aggiornata.
Generalmente il pattern Observer nel pattern architetturale MVC è utilizzato per fare in modo che il Model aggiorni la View sulle recenti modifiche su di esso operate. Nel contesto del nostro progetto però non avendo la necessità di mostrare le informazioni del model all'utente, gli aggiornamenti di quest'ultimo sono stati notificati al controller, il quale ha la necessità di fare delle operazioni sulla base del risultato delle azioni effettuate sul model.
Le chiamate su database devono essere asincrone per due motivi:

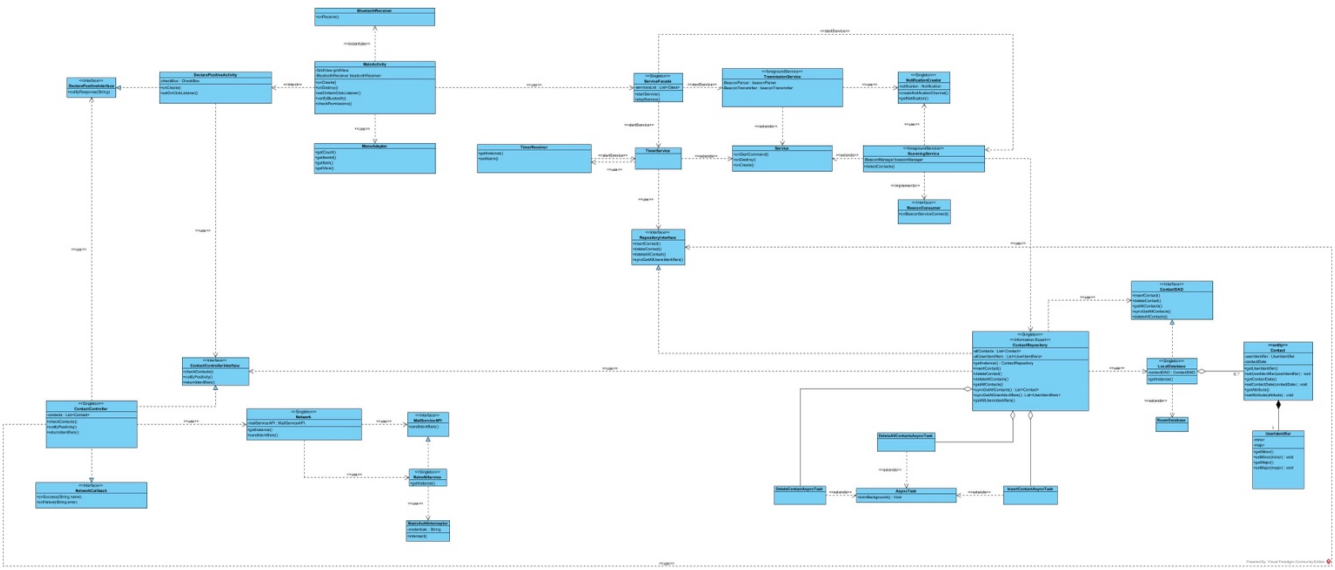
1. **Le chiamate a database non hanno un ritorno immediatamente disponibile**.
2. **Il sistema operativo**, per la ragione al punto 1**, forza ad utilizzare chiamate asincrone per le operazioni su database**, dal momento che non sono consentite chiamate sincrone dal Main Thread per problemi legati alla possibilità di bloccare quest'ultimo (e quindi la user experience) per un lungo periodo.

Pertanto, per risolvere il problema al punto 1 è stato utilizzato un **wrapper (<<LiveData>>) nativo del sistema Android**, che definisce un Subject, su cui è necessario porre un Observer (Observer Pattern).
Per risolvere il problema al punto 2 si è fatto uso di una classe nota come **AsyncTask**, che consentisse di svolgere chiamate a funzioni in modo asincrono aprendo e gestendo automaticamente nuovi thread.
Dal momento che l'Information Expert dei contatti è la repository, è stato deciso di porre l'Observer in questa classe. **È sorta l'esigenza**, tuttavia, **di imporre una certa sincronia** affinché la classe chiamante (il controller) potesse leggere le informazioni presenti nella repository:
per questo motivo **è stata utilizzata una lista di appoggio** (che si aggiornasse, tramite Observer, ad ogni cambiamento nel Model**) in modo tale da rendere disponibile al controller un metodo sincrono e sicuro di accesso ai dati**.



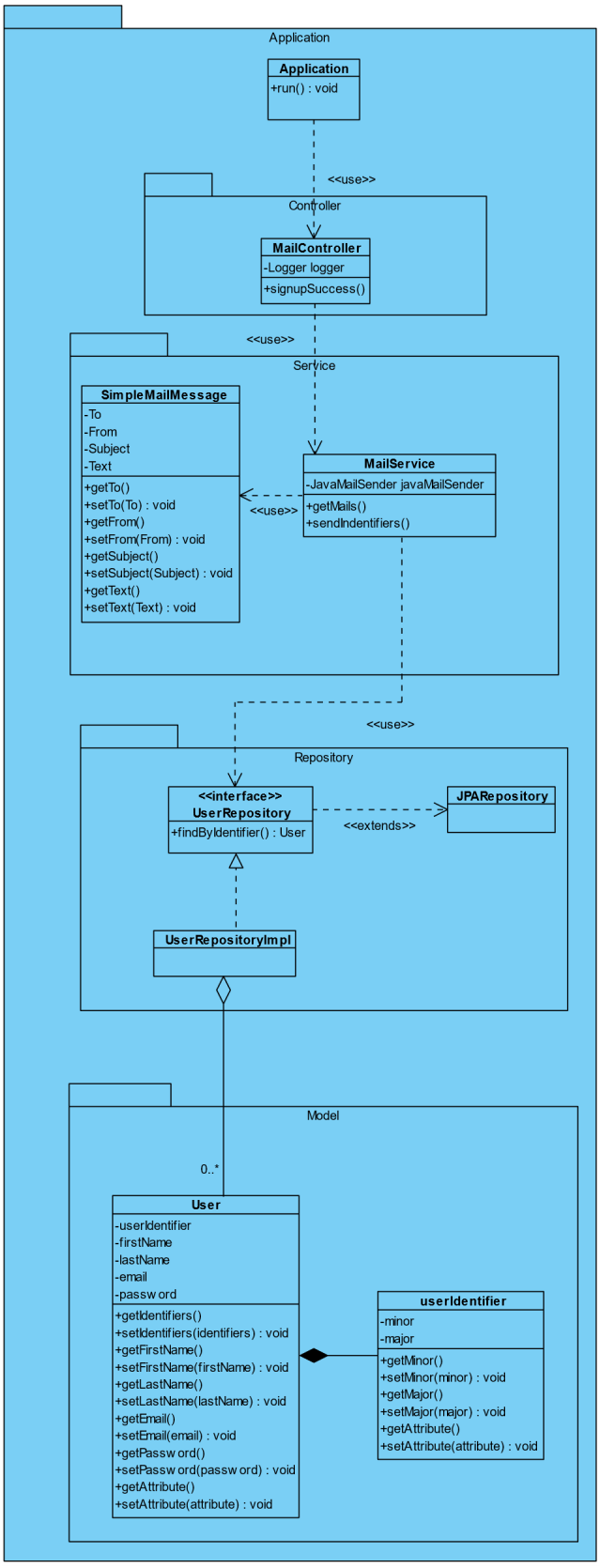
In questo Class Diagram sono mostrate le classi e le relazioni tra di esse presenti nel Web Service.
Il web service è stato implementato utilizzando il framework Spring e pertanto la sua struttura e la sua progettazione sono state vincolate dal framework stesso.

Di seguito è rappresentato il class diagram completo lato client:



4.3.3 Package Diagram

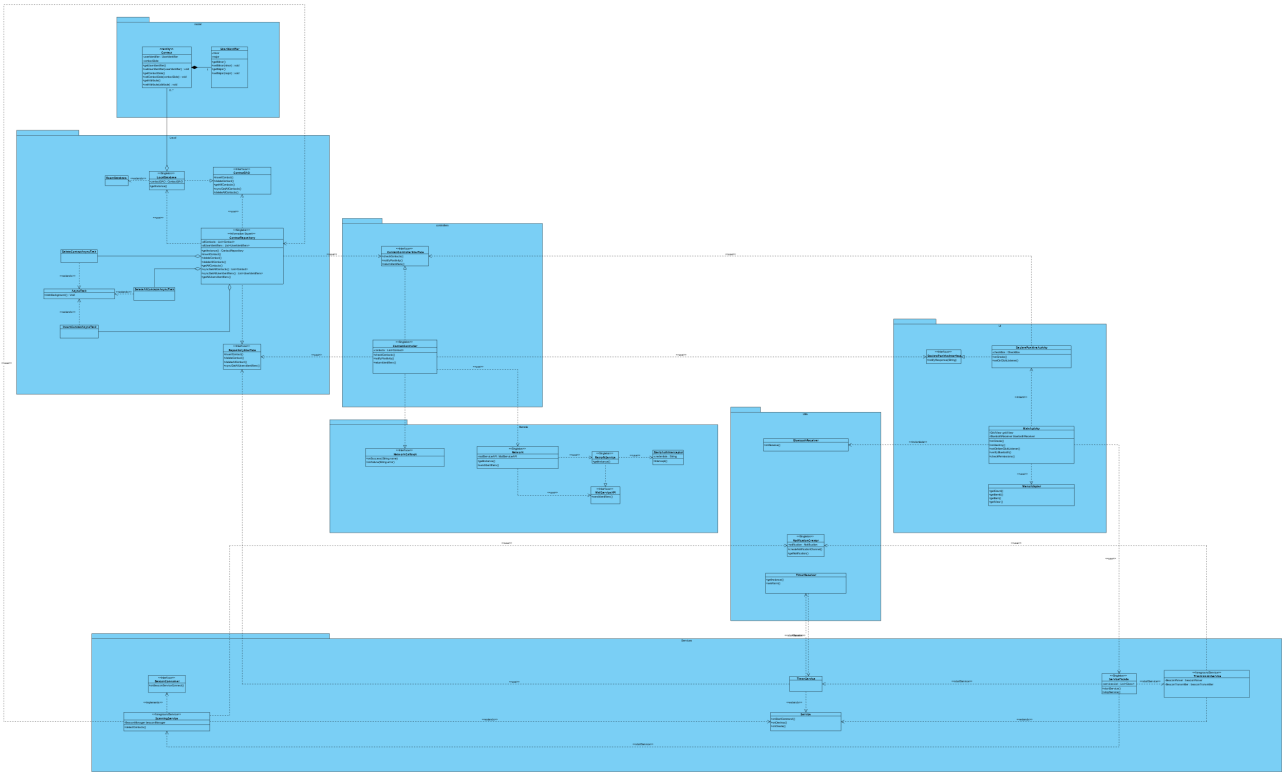
In questo **package diagram**, che descrive l'**organizzazione in package del WebService**, è riflesso lo stile architetturale adottato.



Sono presenti **quattro packages**:

- **Model**: Contiene le classi che modellano gli oggetti di dominio (User).
- **Repository**: il package contiene le **classi utilizzate per implementare il "repository pattern"**.
- **Controller**: package che contiene il controller, **responsabile esclusivamente del coordinamento delle operazioni previste,** utilizza un Service per poter assolvere ai suoi compiti.
- **Service**: **classe "servizio"** prevista dal framework utilizzato (SpringBoot) **utile ad esporre le funzionalità ad altri componenti**, in questo caso al controller.
- **Application**: package che **contiene la classe principale per l'avvio del webService**.

La struttura in package è fissata, prescritta dal framework utilizzato, che possiede **meccanismi di scansione dell'intero progetto** alla ricerca di componenti denotati da annotazioni specifiche con particolari proprietà, come descritto nel capitolo dedicato agli strumenti utilizzati, alla sezione "SpringBoot", motivo per cui tutti i package sono contenuti nel package "application".



Il **Package Diagram** (lato client) riflette l'organizzazione del software e la sua decomposizione funzionale:

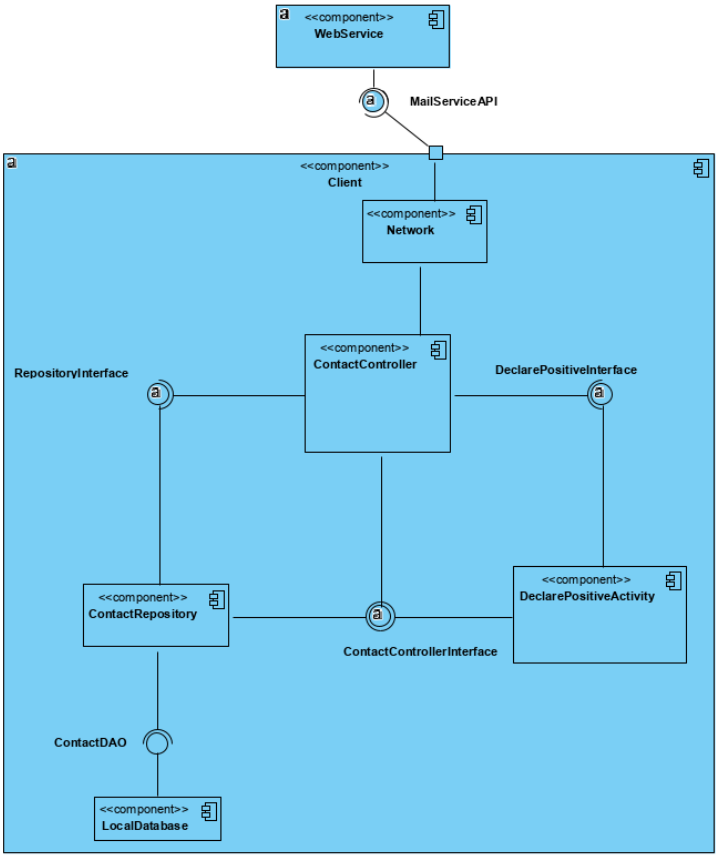
- **Model**: è il package contenente le classi che modellano gli oggetti di dominio, "Contatto" e "UserIdentifier".
- **Controllers**: è il package contenente la classe "Controller".
- **UI**: package contenente le classi correlate all'interfaccia grafica: MainActivity,

DeclarePositiveActivity, DeclarePositiveInterface, MenuAdapter.

- **Local**: package contenente classi per la gestione e la persistenza dei dati su database locale.
- **Remote**: package contenente classi ed interfacce correlate alla comunicazione remota con il RESTFul WebService.
- **Services**: package contenente le classi responsabili delle funzioni di trasmissione e ricezione di beacons, cancellazione dei contatti obsoleti (modellate come servizi) nonché la classe façade che li incapsula.
- **Utils**: package contenente le classi responsabili di funzioni di utilità come la creazione delle notifiche, e la gestione del funzionamento del modulo di trasmissione e ricezione rispetto al funzionamento del modulo Bluetooth del dispositivo.

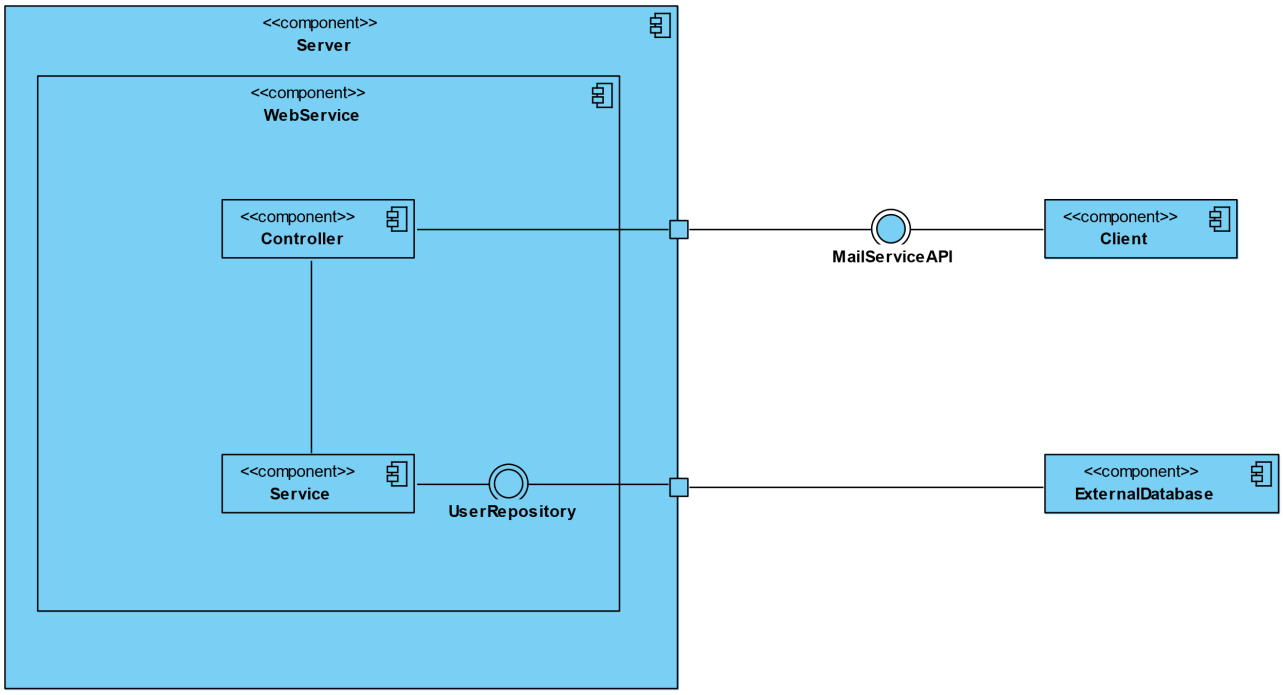
4.3.4 Component Diagram

Questo diagramma **serve ad evidenziare chiaramente le interfacce** tra le varie componenti del client. Ogni interfaccia infatti **definisce un contratto** di comunicazione tra le parti: ciascun componente è quindi tenuto a rispettare tale contratto, vincolando le modalità di interazione. Un'interfaccia **consente quindi di nascondere i dettagli implementativi ed incrementa la modificabilità/modularità del codice** (purchè vengano rispettati i contratti tra le classi, è possibile, modificare completamente l'implementazione delle classi retrostanti).



Oltre ai componenti principali che costituiscono lo scheletro dell'applicazione e delle interfacce di comunicazione, è stato definito un porto che, insieme con l'interfaccia

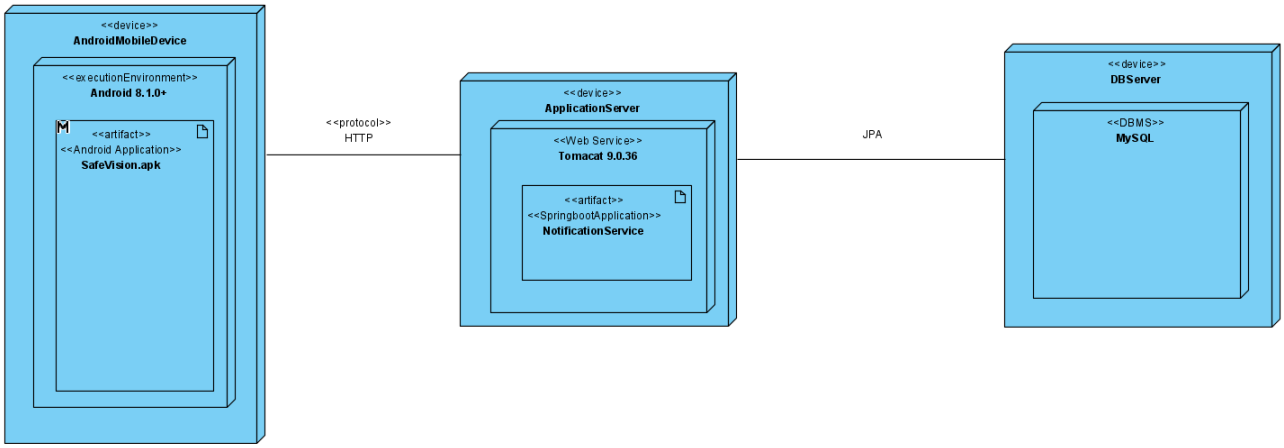
MailServiceApi, costituisce l'unico punto di uscita e di comunicazione verso il webService.



Il secondo diagramma per le stesse ragioni mostra i principali componenti ed i porti di comunicazione relativi al web Service.

4.3.5 Deployment Diagram

Un **deployment diagram** mostra l'architettura di esecuzione di un sistema software e **rappresenta l'assegnazione/dispiegamento (deployment) di software artifacts a deployment targets** (tipicamente sono dei nodes).
I **nodes** rappresentano o **hardware devices** oppure **software execution environments**, connessi attraverso **communication paths** su cui sono dispiegati degli artifacts che rappresentano elementi concreti risultanti da un processo di sviluppo. Il primo diagramma descrive il deployment degli artifacts di tutto il sistema mentre il secondo focalizza l'attenzione sugli elementi costitutive dell'apk.



Per quanto riguarda il primo diagramma, esso può essere diviso idealmente in tre blocchi:

1. **Lato client.**
2. **Lato Server** (Web service).
3. **Database esterno.**

Lato client possiamo osservare quindi:

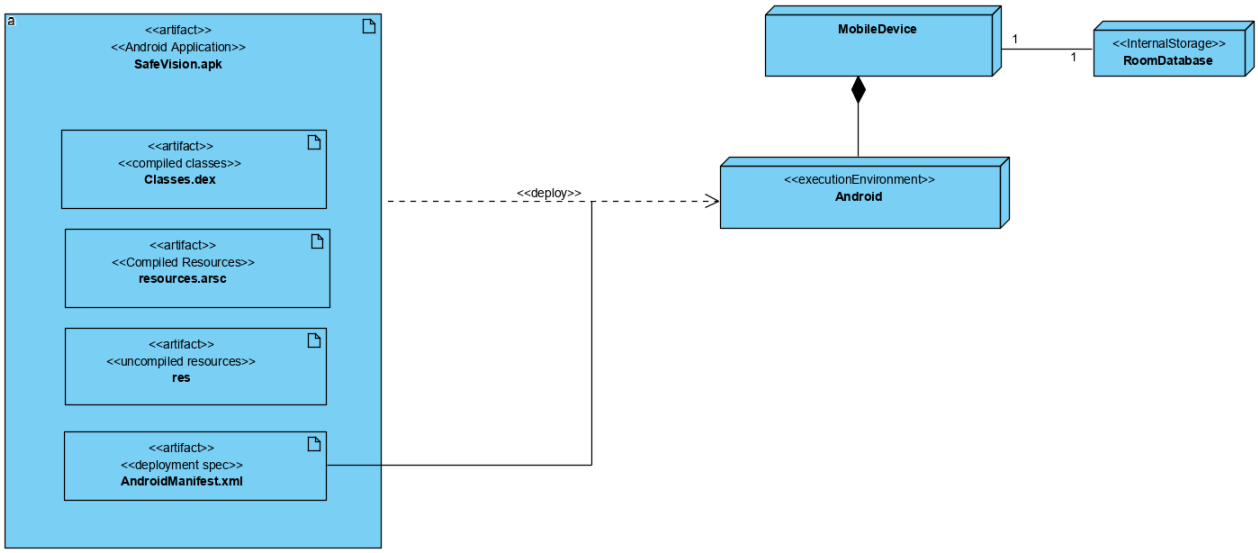
1. **Un nodo** che rappresenta il dispositivo fisico (smartphone).
2. **Un Execution Environment** specifico, ovvero Android 8.1.0 +, dispiegato sul dispositivo.
3. **Un artifact** che rappresenta un'istanza dell'apk che viene eseguita sul sistema.

Lato Server:

1. **Un nodo** che rappresenta il server.
2. **Un Execution Environment** specifico (Tomcat 9.0.36).
3. **Un artifact** che rappresenta il file .jar della SpringBoot Application dispiegato sul nodo.

Database Esterno:

1. **Un nodo** che ospita l'istanza del database.
2. **Un Execution Environment** presente sul nodo, ovvero il DBMS presente (MySql).



Il secondo deployment diagram rappresenta nel dettaglio l'apk, costituito da più artefatti:

- **classes.dex**: Il file classes.dex non è altro che l'insieme dei file .class di cui è composta l'applicazione, in un formato tradotto per Dalvik (dex = Dalvik Executable), la virtual machine di Android.
- **resources.arsc**: resource.arsc contiene tutte le meta-informazioni sulle risorse, i nodi xml

(ad es. LinearLayout, RelativeLayout, etc), i loro attributi (ad esempio Android:layout_width) etc. Gli id di tali risorse si riferiscono alle risorse reali nel file apk. Gli attributi sono risolti in un valore in fase di esecuzione.

- **res**: cartella che contiene le risorse non compilate.
- **AndroidManifest.xml**: ogni applicazione Android dev'essere accompagnata da un file chiamato **AndroidManifest.xml** nella sua cartella principale. Il **Manifest** raccoglie informazioni basilari sull'app, informazioni necessarie al sistema per far girare qualsiasi porzione di codice della stessa. Tra le altre cose il Manifest presente in ciascuna applicazione:

    a. **Da un nome al package Java dell'applicazione, che** è anche un identificatore univoco della stessa.
    b. **Descrive le componenti dell'applicazione** (attività, servizi, receiver, provider, etc.), nomina le classi e pubblica le loro "competenze".
    c. Determina quali processi ospiteranno componenti dell'applicazione.
    d. **Dichiara i permessi dell'app**, e i permessi necessari alle altre app per interagire con la stessa.
    e. **Dichiara il livello minimo di API** Android che l'app richiede.
    f. **Elenca le librerie necessarie** all'app.