UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Ingegneria

Corso di Laurea in Ingegneria Informatica

Sistemi di navigazione outdoor e indoor su dispositivi mobili per la fruizione di beni museali e artistici

Relatore
Prof A. Del Bimbo

Candidato
Riccardo Del Chiaro, Franco Yang

Anno Accademico 2013/2014

OBIETTIVI

L'obiettivo principale dell'applicazione è quello di rendere fruibili, in maniera più naturale possibile, le opere d'arte situate nei vari musei, attraverso la navigazione sia outdoor che indoor. La filosofia seguita è appunto quella della "interazione naturale", che cerca di rendere più semplice e immediato possibile il dialogo con l'utente, ottenuta attraverso una sensorizzazione dell'ambiente d'interesse. In tal modo lo sforzo richiesto all'utente è ridotto al minimo e l'azione per ricercare le informazioni rilevanti è delegata al dispositivo piuttosto che all'utente, che, di fatto, non deve far altro che avvicinarsi ad un'opera o, nel peggiore dei casi, inquadrare un QR code con la fotocamera del dispositivo mobile.

Lo scopo di tutto questo è riuscire a mantenere il più alto possibile grado di piacere nel visionare l'arte, mantenendo comunque un utile e poco invasivo contatto con la macchina, in grado di esaltare l'esperienza comunicando all'utente le giuste informazioni nel giusto momento.

Inoltre l'applicazione si prende carico di profilare le preferenze di un utente, attraverso un'analisi statistica sui musei e/o opere maggiormente visitate. Il tutto per eventuali suggerimenti sulle possibili mete da fruire in futuro, in ottica di un progetto più ampio basato su una idea di "Smart Art".

STATE OF ART

La parte che presenta più problematiche e che sicuramente è più aperta a futuri sviluppi è la navigazione indoor, problema non ancora del tutto risolto a livello mondiale (come riscontrabile da altri progetti e da ricerche online). Esistono, infatti, molte soluzioni che variano per costi e complessità di realizzazione.

Una possibile soluzione potrebbe essere quella di tentare di creare un nuovo standard delle telecomunicazioni per la triangolazione indoor, rendendo possibile una localizzazione accurata tramite antenne poste internamente all'edificio e di una adeguata tecnologia in ogni dispositivo.

È chiaro come una soluzione del genere sia costosa sia in ambito di ricerca che di sviluppo. Tutti i vecchi dispositivi non sarebbero probabilmente compatibili e tuttavia non sarebbe completamente esente da problematica quali interferenze ed altre problematiche generali legati ai segnali radio.

Soluzioni più economiche ed immediate consistono nella sensorizzazione degli edifici con tecnologie già disponibili, in modo da sfruttare più elementi ed arrivare ad una localizzazione più accurata possibile. La nostra soluzione fa parte proprio di questa categoria. In particolare abbiamo deciso di utilizzare beacon e QR code. La motivazione principale di questa scelta sta nei costi relativamente ridotti, nell'ampia diffusione di queste tecnologie lato client. Infatti la gran parte dei dispositivi mobili moderni ha ormai una radio bluetooth capace di connettersi ai beacon, ed una fotocamera con la quale è possibile scannerizzare QR code.

La principale problematica della localizzazione con triangolazione bluetooth è la calibrazione e la definizione della relazione tra i valori RSSI con la distanza in metri dal dispositivo trasmettente. Per stimare, infatti, la distanza in metri da un dispositivo trasmettente bluetooth si utilizzano infatti 3 valori fondamentali:

- Broadcasting power: è la potenza di uscita dell'apparecchio trasmettente (beacon).
- RSSI: è la potenza del segnale vista dal dispositivo ricevente (dispositivo mobile).
- Measured Power: è un valore calibrato di fabbrica, indica quale dovrebbe essere la potenza ricevuta del segnale ad un metro di distanza dal dispositivo trasmettente.

Questi tre valori, purtroppo, da soli non bastano per garantire in generale una buona accuratezza sulla distanza. Infatti, nonostante la formula utilizzata suggerita dalla teoria dei segnali, una migliore stima della distanza basata sulla potenza del segnale ricevuto (RSSI) può essere ottenuta effettuando una ri-calibrazione della potenza basata su una tabella distanze/valori RSSI. [1]

[1] http://altbeacon.github.io/android-beacon-library/distance-calculations.html

Questo perché ogni dispositivo mette in campo molte variabili non facilmente predicibili, e, di fatto, riceve i segnali in modo molto differente dagli altri. Ogni dispositivo può infatti avere un chipset bluetooth differente, oltre che una antenna diversa, una posizione diversa della antenna stessa all'interno del dispositivo e dei materiali differenti (alluminio, policarbonato, vetro ecc..) che possono alterare la ricezione dei segnali.

Pensiamo che sarebbe senz'altro necessario uno standard ed una sorta di certificazione che assicuri la calibrazione di un terminale certificato per questo tipo di tecnologia. In questo modo, se lo standard si diffondesse, i produttori di telefoni potrebbero fare le calibrazioni in fabbrica, facendole certificare da un ente addetto in modo da poter esporre pubblicamente la certificazione ed i parametri necessari alla calibrazione delle formule che legano la distanza con i valori RSSI per ogni preciso dispositivo certificato.

Un'altra problematica importante, oltre alla corretta calibrazione, riguarda l'uso dei segnali radio in generale e alla loro suscettibilità a interferenze, quindi a errori dovuti all'ambiente circostante. Problematiche che Apple ha in parte risolto (per i propri dispositivi) attraverso lo sviluppo della libreria software iBeacons e probabilmente con una più evoluta gestione dello stack bluetooth.

Si può infatti provare sperimentalmente la notevole accuratezza dei dati ricevuti su un dispositivo iOS in particolar modo sulla distanza. Accuratezza che Apple sembra essere riuscita a raggiungere attraverso l'elaborazione dei dati "raw" ricevuti dai beacon e dall'uso di algoritmi efficienti per l'attenuazione delle interferenze fra beacon e ambiente circostante (comprendente anche altri beacon, rifrazioni dei segnali ecc.). Chiaramente tale accuratezza è fondamentale per funzioni quali localizzazione attraverso triangolazione e meno rilevante qualora si necessiti solamente della prossimità.

Sfortunatamente per la piattaforma Android non esiste ancora niente del genere e le conseguenze sono immediatamente riscontrabili: le misurazioni delle distanze oltre ad 1 metro risultano ben poco accurate e, per i dispositivi non calibrati, quasi casuali.

Per sperimentare in maniera immediata la differenza del comportamento dei beacon sulle varie piattaforme è possibile scaricare da AppStore o da GooglePlay l'applicazione demo di estimote, che comprende le varie funzioni dimostrative di prossimità e distanza.

Date queste problematiche ci siamo limitati ad utilizzare i beacons bluetooth come sensori di prossimità. Ovviamente i beacon non sono l'unico strumento utilizzabile per questo tipo di lavoro, vi sono molti altri sensori di prossimità più o meno accurati che andrebbero testati (come per esempio la tecnologia NFC). Tuttavia i beacon risultano semplici da usare poiché non richiedono alcuno sforzo da parte dell'utente (con NFC si deve avvicinare il sensore posto sul proprio dispositivo ad almeno 20cm di distanza con il sensore trasmettente), i costi sono relativamente ridotti e la tecnologia è già fruibile con la quasi totalità dei dispositivi, al contrario di NFC che, ancora oggi, è una tecnologia che spesso non viene inserita nei dispositivi mobili.

CONTESTO

L'applicazione è stata pensata per funzionare in un contesto ben preciso:

- L'utente finale possiede un dispositivo mobile (smartphone, tablet) con sistema operativo Android.

- L'utente che offre il servizio, sensorizza un edificio con Beacon (Estimote) e/o QR code.

- L'utente che offre il servizio si appoggia su un server con le seguenti caratteristiche:
    - Database per il retrieving delle informazioni, sia sui musei che sulle opere.
    - Possiede uno strato di software che effettua query sul database e che risponde all'applicazione con un formato JSON interpretabile.

In questo caso si è utilizzato uno strato di software lato server in php per compiere query su un database MySQL, traducendo poi le risposte del database in JSON.

UTENTE FINALE

Data la grande diffusione su un'ampia fascia sociale dei dispositivi mobili basati su sistema operativo ANDROID, abbiamo deciso di focalizzare i nostri sforzi su questa precisa piattaforma.

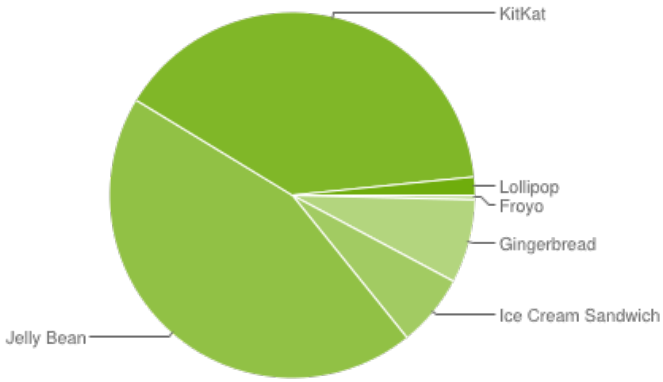
Per offrire un'esperienza di utilizzo più coinvolgente ed allo stesso tempo semplice ed immediata abbiamo pensato di utilizzare le ultime versioni dell'SDK di Android fornito da Google, che consente di utilizzare strumenti più avanzati sia per la programmazione che per lo sfruttamento di risorse dei dispositivi moderni, come l'accelerazione dell'interfaccia grafica utilizzando le risorse di calcolo grafico della GPU.

Non tutte le features fornite dagli aggiornamenti dell'SDK sono retrocompatibili con le vecchie versioni del sistema operativo Android, per questo siamo stati costretti a scegliere in anticipo quali versioni di android escludere dal supporto del nostro software e quali invece integrare.

Giudicando le analisi ufficiali fornite da google e confrontandole con i reali vantaggi nell'utilizzare una versione minima richiesta più aggiornata, siamo giunti alla conclusione che avremmo potuto escludere le versioni di android precedenti alla 4.

La versione 4 di android offre infatti una serie di nuove feature che rivoluzionano completamente il sistema rispetto alle precedenti release [2] offrendo una nuova modalitá più efficace e dinamica per la gestione della interfaccia grafica (i fragment, disponibili fino a quel momento solo su android 3, che è però riservato a dispositivi con grossi schermi, come tablet) oltre che un nuovo tema di base più moderno ed il supporto all'accelerazione hardware tramite GPU per il disegno dell'interfaccia grafica.

Tutte queste feature (e probabilmente molte altre) si sono rivelate fondamentali nella costruzione della nostra applicazione, che fa largo utilizzo di fragment e della accelerazione hardware per disegnare le mappe degli edifici tramite le librerie di disegno standard di Android.



Version	Codename	API	Distribution
2.2	Froyo	8	0.4%
2.3.3 - 2.3.7	Gingerbread	10	7.4%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	6.4%
4.1.x	Jelly Bean	16	18.4%
4.2.x		17	19.8%
4.3		18	6.3%
4.4	KitKat	19	39.7%
5.0	Lollipop	21	1.6%

Stato attuale della distribuzione di agggiornamenti nell'ecosistema Android.

[2] https://developer.android.com/about/versions/android-4.0.html

Inoltre con android 4 viene riunificato l'ecosistema che era stato diviso in due con l'annuncio di android 3 (disponibile ed ottimizzato solo per tablet).

Android 4, infatti, supporta pienamente sia dispositivi con piccoli schermi (smartphone) che dispositivi con schermi più grossi (tablet o mini-pc). Abbiamo così rinunciato, al momento dell'analisi descritta, a dare il supporto per il nostro software a circa il 7.8% del totale degli utenti dell'ecosistema android [[3]]. Numero che è comunque destinato scendere col tempo.

UTENTE EROGATORE DEL SERVIZIO

L'utente erogatore del servizio è colui che decide di utilizzare la tecnologia descritta per offrire un servizio ai propri client/utenti. Dovrà quindi farsi carico dell'operazione di sensorizzazione del proprio edificio, della mappatura dello stesso, fornendo i dati in un formato adeguato al server.

I beacons Bluetooth iniziano ad essere fabbricati da varie aziende in tutto il mondo, e la crescente concorrenza dovuta sta già dando i suoi frutti nell'abbassamento dei prezzi di mercato.

Per abbassare i prezzi iniziali di installazione, che siamo convinti caleranno negli anni successivi, abbiamo previsto l'utilizzo di QR code, tecnologia a costo zero e fruibile da qualsiasi utente con uno smartphone. È possibile stampare codici QR su qualsiasi supporto cartaceo, in modo da essere posti in luoghi visibili per essere scannerizzati dagli utenti che otterranno informazioni utili, una volta tradotte dall'applicazione, in modo simile all'interazione che si ha con dei beacons bluetooth.

I costi dei beacons bluetooth, allo stato attule, non consentono forse una sensorizzazione completa di un museo. Si parla infatti di cifre che oscillano tra 10 e 35 dollari per sensore, a seconda del tipo e del produttore.

Probabilmente però, per questo tipo di applicazione, i sensori low cost sono più che sufficienti, poiché non ci interessa che abbiano una forte potenza di uscita ed un ampio raggio di azione (la potenza dovrà anzi essere limitata via software per evitare interferenze con i beacons vicini). In tal senso probabilmente potrebbero essere sufficienti beacons più piccoli ed economici (definiti Sticker Beacon sul sito ufficiale di Estimote, noto produttore di Beacons bluetooth).

Prodotti simili si riescono ad acquistare per circa 10 dollari a pezzo, cifre che già adesso possono essere considerate concorrenziali.

I costi d'installazione e mantenimento sono quasi nulli, poiché possono essere semplicemente incollati vicino all'oggetto da sensorizzare, hanno una batteria autonoma che dura circa un anno, e la sostituzione della batteria è una operazione possibile nella maggior parte dei prodotti.

Inoltre l'impatto visivo può essere nullo, poiché, date le ridottissime dimensioni, possono essere facilmente nascosti dentro la mobilia, teche o altro.

La nostra convinzione sull'abbassamento dei prezzi dei beacons deriva da due considerazioni:

1) Un beacon bluetooth non è molto di più di un semplice sistema basato su un SOC con architettura ARM (System On a Chip, cioè CPU, RAM e ROM in un unico chip) che deve gestire una radio bluetooth. Una produzione di massa in serie abbatterebbe senz'altro i costi di produzione.

2) Apple spinge su questa tecnologia, pubblicizzandola con il nome di iBeacon, ed i produttori stanno via via aumentando. L'aumento della concorrenza porterà ad una diminuzione dei costi per l'utente finale.

[3] https://developer.android.com/about/dashboards/index.html

SOLUZIONI TECNICHE ADOTTATE

Per produrre questo applicativo abbiamo proceduto con un approccio TOP-DOWN, focalizzando la nostra attenzione su come avrebbe dovuto essere l'esperienza di utilizzo del prodotto per l'utente finale sul proprio dispositivo mobile. L'obiettivo, come abbiamo detto, è proprio offrire un'esperienza meno invasiva ma comunque efficace da parte del supporto elettronico verso l'utente, in modo che la propria esperienza con l'arte sia il meno possibile disturbata da esso.

L'interfaccia grafica è quindi fondamentale per ottenere questo risultato, poiché deve essere semplice, familiare e intuitiva.

Siamo così partiti dalla sua progettazione, immaginando il prodotto finito nella sua interessa, disegnando come sarebbero dovuti apparire tutti gli elementi grafici. Abbiamo quindi prodotto un mockup dell'applicazione che abbiamo utilizzato poi come linea guida durante tutto il processo di creazione del software.

Le ultime linee guida di Google per lo sviluppo d'interfacce grafiche ci sono venute in aiuto. Esprimono proprio quel bisogno di semplicità e immediatezza che stavamo cercando.

MATERIAL DESIGN

Il Material Design vuole essere un linguaggio visivo che sintetizzi, con pochi concetti, i principi classici per un buon design[4]. Può quindi essere una chiave intuitiva di lettura per un'interfaccia grafica di un software. Semplice ed immediata per un nuovo utente, oltre che familiare ed intuitiva per un utente che ha già avuto modo di utilizzare le ultime versioni dei software Google e Android.

Abbiamo così deciso di seguire queste linee guida per uniformarci ad uno standard che riteniamo comodo e quanto mai efficace. Comodo per l'utente finale… ma purtroppo non possiamo dire lo stesso per lo sviluppatore, che, se vuole utilizzare questo standard allo stato attuale, supportando terminali precedenti alla ultima release di Android (Lollipop, 5.0), non avrà alcuno strumento o aiuto da parte di Google nell'SDK.

Il Material Design è infatti una idea nuova e gli aiuti per implementare i suoi principi in modo semplice nelle applicazioni Android sono stati introdotti con le ultime API (level 21). Non è possibile quindi utilizzare la maggior parte di questo supporto sui terminali con versione Android installata precedente alla 5.0, cioè più del 98% dei dispositivi!

Essendo stati affascinati da questa filosofia di design, abbiamo ricercato sulla rete strumenti che ci potessero aiutare ad implementare il Material Design sulla nostra applicazione mantenendo la retrocompatibilitá con android 4. Ci siamo così imbattuti in alcuni progetti open source che ci hanno aperto la possibilitá di utilizzare alcuni componenti di base del design ideato da Google senza dover reinventare la ruota partendo da zero. Convinti della possibilitá di poterlo quindi implementare in qualche modo con le nostre mani e con l'aiuto della comunità open source, dopo alcuni test tecnici, abbiamo iniziato a progettare e disegnare il mockup seguendo le linee guida di Google.

Per la concretizzazione del nostro mockup su applicazione reale, sono stati fondamentali gli aiuti fornitici indirettamente dalla comunità open source. In particolare abbiamo basato la nostra interfaccia su due componenti grafici personalizzati (custom view) completamente open source e reperibili sulla rete:

- Sliding Up Panel library: https://github.com/umano/AndroidSlidingUpPanel.
- Getbase Floating Action Button: https://github.com/futuresimple/android-floating-action-button.

[4] http://www.google.com/design/spec/material-design/introduction.html

Sliding Up Panel è stato utilizzato come scheletro dell'interfaccia della nostra applicazione, mantenendola sempre attiva sulla finestra principale e modificando i "fragment" contenuti da essa a seconda degli eventi scaturiti dall'interazione con l'utente e con l'ambiente sensorizzato.

I componenti fondamentali delle GUIs (Graphic User Interface, interfaccia grafica) in Android 4.0 e successivi, sono proprio le Activity ed i Fragment. Le Activity sono molto simili al concetto di finestra, che ormai qualsiasi utente tecnologico conosce, il Fragment è invece un concetto un po' più tecnico. Sono "pezzi" di interfaccia grafica con un loro proprio ciclo vitale che possono essere "incollati" uno alla volta o più di uno in contemporanea su una o più Activity, come adesivi che possono essere staccati e riattaccati in vari punti della finestra in qualsiasi momento.

Abbiamo quindi basato l'applicazione su un'unica Activity nella quale andiamo a scambiare i fragment attivi con quelli non attivi secondo cosa richiede l'utente.

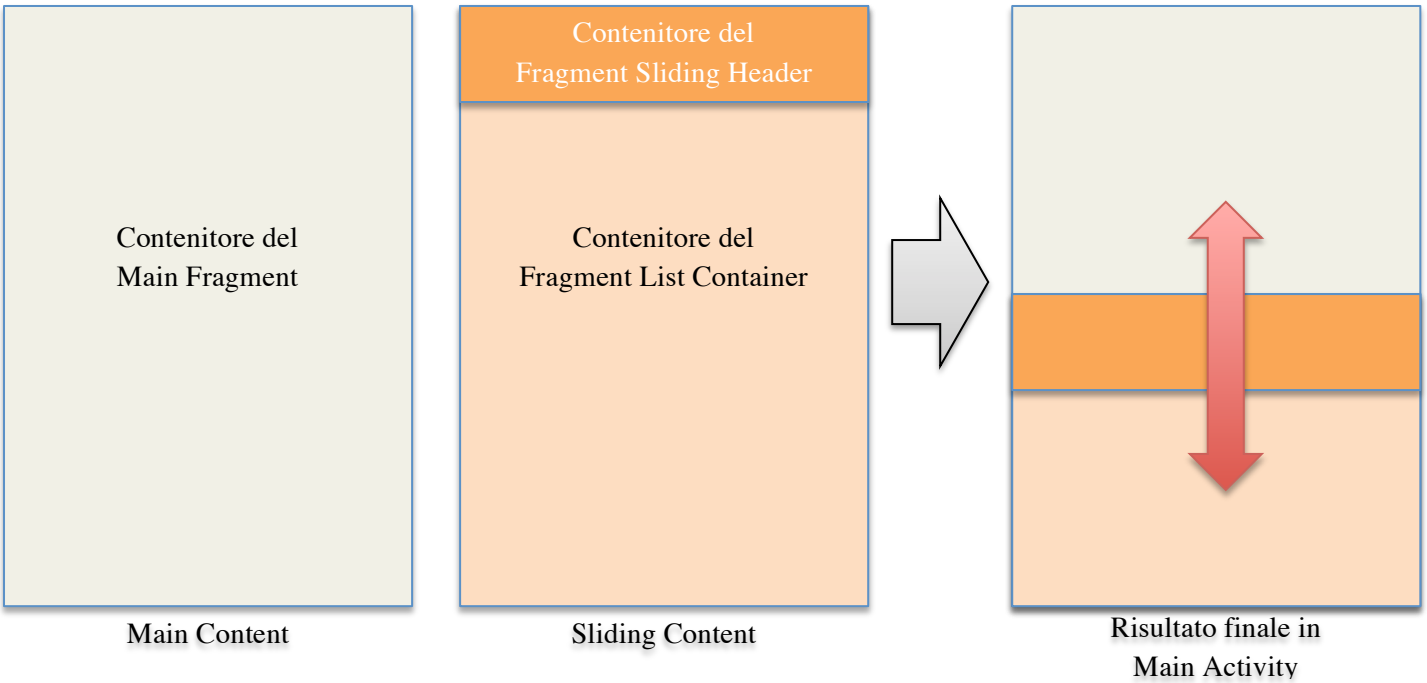
L'Activity contiene così solo lo scheletro dell'interfaccia, con lo SlidingUpPanel che fa da contenitore per 3 fragment. Deve quindi solo preoccuparsi di gestire i componenti grafici comuni a tutta l'interfaccia grafica, lasciando che siano i singoli fragment a gestire i propri peculiari componenti. Lo SlidingUpPanel, infatti, gestisce 2 contenuti dei quali il secondo può scorrere sopra il primo partendo dal basso fino a coprirlo completamente:

1. **Main Content**: il contenuto principale, lo sfondo. Contiene il fragment principale della nostra applicazione. In particolare contiene la vista della mappa, sia che ci troviamo all'esterno di un edificio sia che ci troviamo all'interno (visualizzando ovviamente in questo caso la mappa dell'edificio, se disponibile).

2. **Sliding Content**: il contenuto scorrevole. La custom view SlidingUpPanel gestisce lo scorrimento di questo contenuto quando l'utente effettua un trascinamento della sua parte più alta (barra dello slider) che è sempre visibile. Lo abbiamo gestito in modo che contenga due fragment:
    a. Fragment sliding header: contiene il contenuto della barra dello slider.
    b. Fragment List Container: contiene il content secondario, nella pratica saranno delle liste o delle descrizioni dei contenuti selezionati.

Quindi per passare visivamente, ad esempio, dalla lista dei musei alla descrizione di un museo, rimuoveremo il fragment della lista dal fragment_list_container, inserendo al suo posto un fragment dedicato alla visualizzazione dei dettagli dei musei in generale, al quale diremo di visualizzare i dettagli di uno specifico museo.

Con questo tipo di gestione evitiamo cambi d'interfaccia, l'utente non si preoccuperá di dove si trova poiché i fragment verranno posti nel posto giusto al momento giusto in modo automatico, e con un semplice trascinamento dello Sliding Header sará possibile passare dalla visualizzazione della mappa alla visualizzazione dei dettagli su ciò che abbiamo vicino (opera o museo).



PROGETTAZIONE DEL SOFTWARE: PRELIMINARI

Completato il mockup siamo passati alla progettazione del software. Per la scrittura del codice abbiamo utilizzato Android Studio suddividendo inizialmente il contesto in:

1. Lato Client
    a. Gestione dell'interfaccia grafica e interazione con l'utente.
    b. Gestione del motore outdoor (mappe, localizzazione e prossimitá).
    c. Gestione del motore indoor (mappe indoor, localizzazione e prossimitá indoor).
    d. Interfacciamento con un generico database che risponde con formato JSON.

2. Lato Server
    a. Progettazione base di dati.
    b. Gestione del database con query e traduzione delle risposte in JSON.

Per la parte lato client, data la nostra totale ignoranza in merito all'SDK Android, abbiamo iniziato a studiare l'ecosistema sulla documentazione ufficiale Google, in modo da comprendere i principi fondamentali.

Una volta completato questo primo ciclo siamo passati ad un approccio empirico, dato che le conoscenze acquisite non sarebbero bastate data la quasi totale inesperienza sul campo: abbiamo iniziato a fare dei test per ogni singola area che sapevamo avremmo dovuto sviluppare, per capire quali fossero le strade percorribili, oltre che per fare esperienza ed ottenere manualitá con questi, per noi, nuovi strumenti.

I test fondamentali che abbiamo condotto sono:

- **Test sulla gestione dei Fragment e dell'interfaccia grafica**: inoltre test sull'integrazione dei componenti grafici open source (in particolare in questa fase abbiamo scelto di integrare SlidingUpPanel).

- **Test sulla visualizzazione delle mappe outdoor**: siamo partiti con l'idea di utilizzare Open Street Map, ma abbiamo riscontrato i grossi limiti di questa piattaforma dovuti anche ad una quasi inesistente documentazione. Siamo così ricaduti sull'utilizzo delle Google Maps Api v2 per Android.

- **Test sulla visualizzazione delle mappe indoor**: non avendo trovato alcun componente che potesse aiutarci nella visualizzazione di una mappa indoor personalizzata, siamo partiti con l'idea di costruire un nostro motore proprietario per la visualizzazione di mappe indoor. Google Maps fornisce infatti un motore per mappe indoor, ma obbliga che le nostre mappe siano fornite al server centrale Google, obbligo che non vogliamo noi stessi dover dare ai clienti erogatori del servizio. Abbiamo quindi iniziato a fare test con librerie grafiche (come openGL ES) ricadendo su un soluzione più consona ai nostri obiettivi, ovvero le librerie di disegno grafico di Android, che sono accelerate dalla GPU dalla versione 4.0 presentando quindi sia il vantaggio della semplicitá sia quello delle performance e leggerezza.

- **Test su localizzazione GPS, beacon bluetooth Estimote e QR code.**

Una volta completata questa fase preliminare di test, abbiamo potuto procedere per la progettazione vera e propria del software con la sicurezza di poter integrare tutte le tecnologie richieste.

PROGETTAZIONE DEL SOFTWARE: INTERFACCIA GRAFICA

I due fondamentali moduli per la gestione dell'interfaccia grafica sono:

Main Activity: classe java, activity principale del programma, ed anche il suo punto d'ingresso per l'esecuzione. Il suo compito principale è quello di inizializzare tutti i componenti grafici tramite file esterni xml, inizializzare il FragmentHelper e tutti i gestori di eventi dello slidingBarPanel. Si può quindi dire che il suo ruolo sia quello di gestire lo SlidingBarPanel e quello di inizializzare il programma delegando esternamente ad altri componenti il compito di gestire il resto dell'interfaccia grafica (il primo delegato è il FragmentHelper). Inoltre gestisce le animazioni ed i cambi di tema e colore.

FragmentHelper: classe java il cui compito è quello di facilitare la gestione dell'interfaccia grafica e dei fragment, prendendosi carico di tutti i dettagli tecnici riguardanti le trasformazioni dell'interfaccia da un punto ad un altro dell'applicativo, fornendo dei semplici metodi di accesso che possono essere richiamati ovunque. Il FragmentHelper interpreta infatti il ruolo di un mediatore, si interpone tra l'interfaccia grafica (mainActivity e fragments) ed il resto dei componenti specifici del software. Il FragmentHelper implementa il design pattern Singletone, in modo che sia uno strumento utilizzabile da chiunque e ovunque all'interno del codice java.

Oltre a questi due componenti principali, abbiamo delegato le funzionalitá secondarie (come animazioni) e le funzionalitá specifiche di ogni fragment ad un modulo esterno (package **gui**).

In particolare il package gui.customSlidingHeader contiene i fragment che possono essere posti nel contenitore del FragmentSlidingHeader di cui abbiamo parlato, mentre in gui.customList ci sono i fragment che possono essere posti nel FragmentListContainer, oltre che gli adapter necessari per riempire i dati delle liste e delle descrizioni che devono comparire nel fragment.

PROGETTAZIONE DEL SOFTWARE: OUTDOOR ENGINE E LOCALIZZAZIONE OUTDOR.

Avendo deciso di utilizzare le API di Google Maps V2, il lavoro per l'outdoor engine è stato per lo più di integrazione. Abbiamo utilizzato le API per visualizzare la mappa sul nostro MapFragment, delegando ad una classe esterna Map la gestione della mappa e delle API di google. In questo modo abbiamo incapsulato in Map tutte le funzionalitá offerte da google maps, per poi costruire dei semplici metodi di accesso per le funzionalitá che ci erano utili, interconnessi al resto della nostra applicazione.

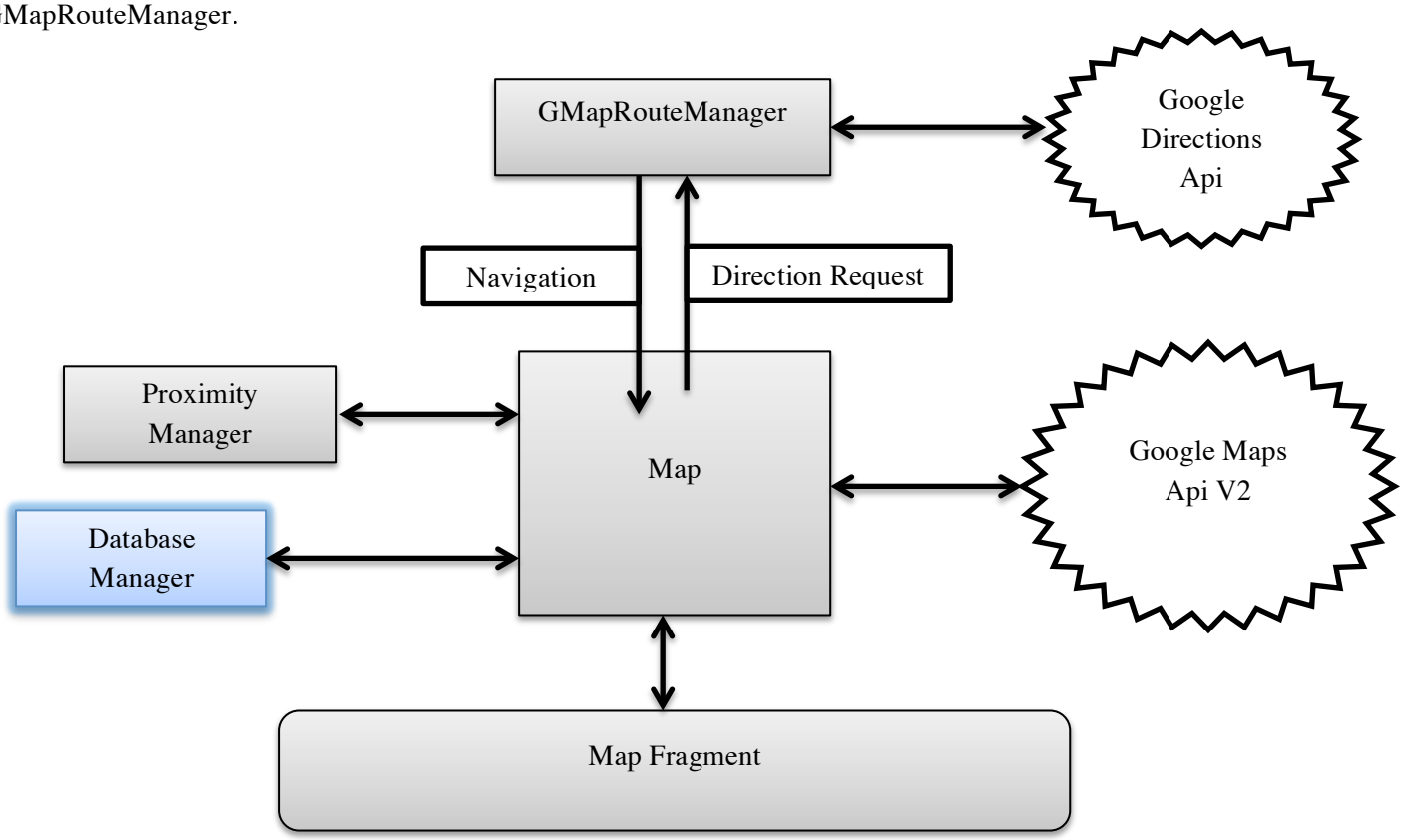
In particolare Map integra e traduce per google maps le informazioni sulla posizione dei musei derivate dal database, inoltre gestisce la geolocalizzazione e la prossimitá con un museo con l'aiuto delle classi esterne contenute nel package OutdoorProximity.

Per ottenere le indicazioni stradali dalla posizione GPS corrente fino ad un museo, abbiamo utilizzato le Google Directions Api. Per farlo eseguiamo una query sul server google in cui specifichiamo partenza e arrivo in coordinate geografiche, google ci risponde con un oggetto JSON complesso costituito da:

- Route: contiene una serie di leg che uniscono tutti i waypoints (punti attraverso i quali si impone il passaggio del percorso) partendo dal punto di partenza arrivando alla destinazione. Se non ci sono waypoints ci sará una sola Leg.
- Leg: singolo percorso tra 2 waypoints, o tra origine e primo waypoints, o tra ultimo waypoints e destinazione, o tra origine e destinazione in assenza di waypoints.
- Step: singolo tratto rettilineo compreso tra un possibile bivio ed il successivo. Ogni step contiene una polyline, cioè un insieme di punti sulla mappa che approssimano il percorso stradale.

Per gestire tutto questo abbiamo utilizzato una gerarchia di classi che ricalcano gli oggetti JSON, ed una classe GMapRouteManager che ci aiuta ad utilizzare questi oggetti con un unico punto di accesso per gestire questa funzionalitá.
Il Manager effettuerá la query verso i server google, e provvederá a restituire al chiamante un oggetto Navigation, che contiene tutti gli oggetti sopra elencati giá tradotti da JSON a Java.

Nel nostro caso specifico sará la classe Map a richiamare il GMapRouteManager al momento in cui l'utente richiede di navigare verso un museo grazie all'interazione col MapFragment. Il GMapRouteManager effettuerá la richiesta ai server google traducendo i dati e restituendoli all'oggetto della classe Map, il quale provvederá a dialogare con le Google Maps Api per disegnare, linea per linea, il percorso contenuto nell'oggetto Navigation fornito dal GMapRouteManager.



Degno di nota è il Proximity Manager. Questa classe è il punto di accesso per la funzionalitá di prossimitá outdoor. Il suo funzionamento è dipendente da altre classi ed interfacce satelliti:

ProximityObject: interfaccia java che rappresenta un generico oggetto localizzabile tramite latitudine e longitudine. Qualsiasi classe i cui oggetti vogliono essere processati dal ProximityManager per una analisi di prossimitá devono implementare questa interfaccia.

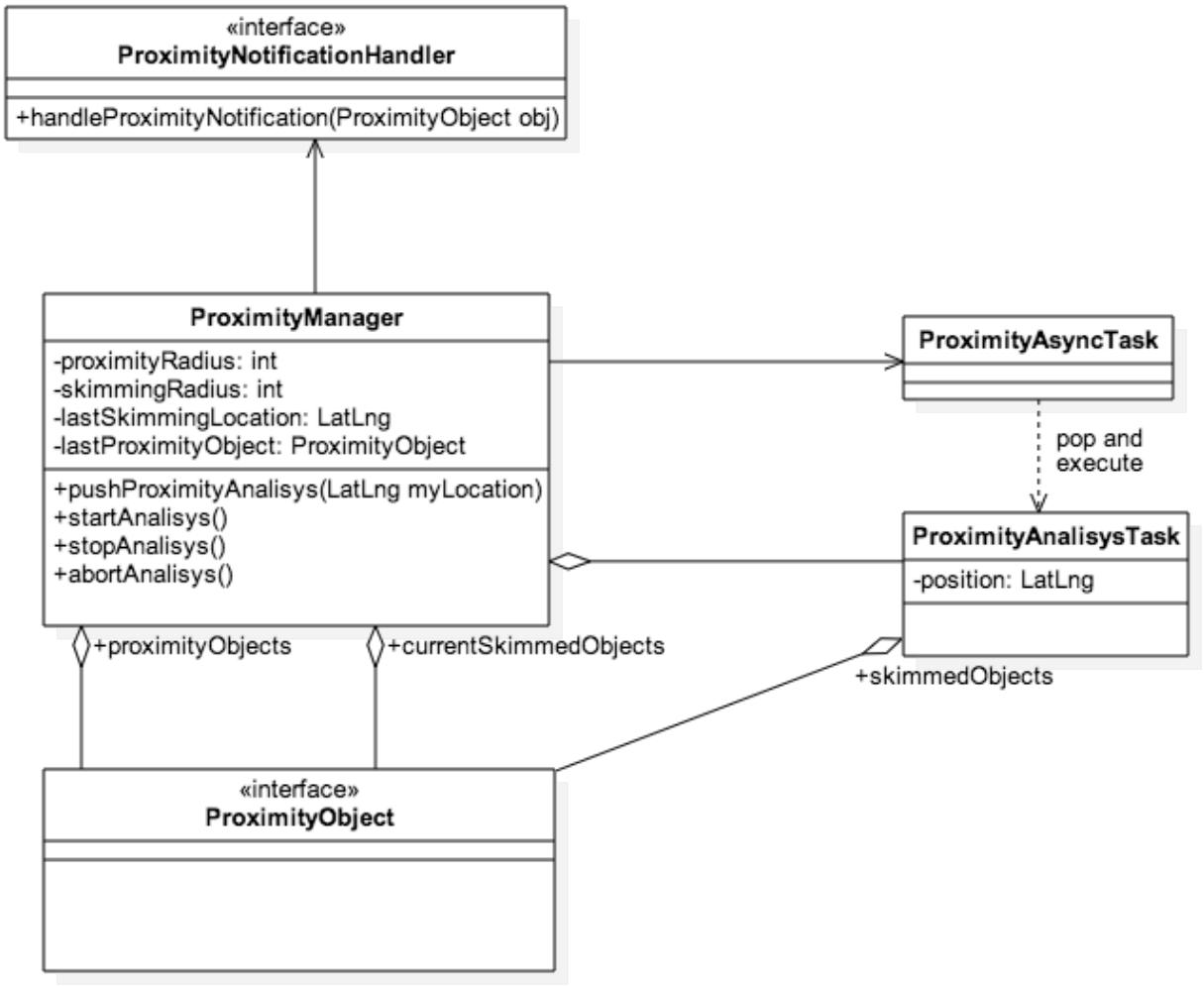
ProximityNotificationHandler: interfaccia java che rappresenta l'oggetto che ha interesse a ricevere gli esiti delle analisi eseguite dal manager. Ogniqualvolta un'analisi di prossimitá da un risultato degno di nota (ovvero quando c'è un cambiamento, cioè quando viene rilevato un nuovo oggetto vicino oppure viene rilevato che non c'è più un oggetto nelle vicinanze) il ProximityManager richiamerá i relativi gestori di eventi del ProximityNotificationHandler che è in ascolto.

ProximityAnalisysTask: rappresenta un unico task che ha il compito di analizzare le distanze tra la posizione corrente ed i musei, in modo da avvertire il manager se esiste un ProximityObject nelle vicinanze. Con "posizione corrente" si intende la posizione al momento in cui il ProximityAnalisysTask è stato inviato al manager. Infatti un ProximityAnalisysTask viene messo in una coda dal ProximityManager ogni qualvolta che un oggetto esterno richieda di eseguire una analisi su una nuova posizione corrente aggiornata, e viene fatta una sorta di screenshot della situazione corrente nel manager (tutti i riferimenti ai ProximityObject da analizzare al momento in cui si richiede la nuova analisi vengono memorizzati nel nuovo ProximityAnalisysTask stesso). Nella pratica verrá creato un task nuovo ogni volta che il GPS rileverà una nuova posizione corrente.

ProximityAsyncTask: il ProximityManager lancerá questo task su un thread separato, il suo compito è quello di prendere il prossimo elemento dalla lista di AnalisysTask contenuta nel manager, eseguirla (sul thread separato in modo da non bloccare la graphic user interface) e restituire al manager il risultato. Quando non ci sono AnalisysTasks in coda, il proximityAsyncTask starà inattesa per 1 secondo senza utilizzare le risorse, per poi ripartire e controllare se ci sono nuovi task in coda.

Il ProximityManager implementa inoltre un meccanismo di "scrematura" degli oggetti prima di passarli al nuovo AnalisysTask, in modo da velocizzare l'esecuzione: se ci fossero centinaia o migliaia di musei sul database sarebbe inutile analizzarli tutti ogni qual volta la posizione del GPS varia anche di poco. Abbiamo quindi pensato di mantenere su una lista separata i riferimenti a tutti i musei entro un certo raggio (sull'ordine dei kilometri) in modo che, fino a che la nostra posizione non esce da questo cerchio, compiremo la ricerca dei musei vicini solo tra quelli contenuti nel cerchio, cioè quelli preventivamente scremati. Ogni volta che ci avviciniamo al bordo di questo cerchio, sarà eseguita una nuova scrematura, prendendo come origine del cerchio il punto delle coordinate geografiche in cui è stata rilevata l'uscita dal precedente cerchio di scrematura.

Questo è stato fatto perché non è possibile essere certi che in un futuro il numero di musei contenuti su un database sia piccolo, e senza questa modalità di ricerca probabilmente le ricerche da effettuare si ammasserebbero nello stack e non sarebbero mai eseguite tutte in tempi ragionevoli, andando a rallentare la reattività della notifica di avvicinamento ad un museo. Inoltre, in casi estremi, la massa di Task in coda, potrebbe riempire la memoria heap del dispositivo mobile, causando un errore con conseguente chiusura forzata dell'applicazione.



PROGETTAZIONE DEL SOFTWARE: INDOOR ENGINE

La progettazione del motore per la gestione delle mappe indoor è stata ben più impegnativa, poiché non abbiamo potuto utilizzare nessuna libreria esistente.

Siamo partiti con dei Test sui Canvas (classe di Android SDK per gestire il disegno vettoriale) per assicurarci della fattibilitá della cosa, implementando dei gestori di eventi per gestire le gestures sulle ImageView (come il classico pinch-to-zoom per ingrandire ed il drag per spostare la visuale).

Lavorando sulle matrici di trasformazione per oggetti 2D siamo riusciti a costruire due tipologie di oggetti disegnabili differenti:

- Oggetti che reagiscono allo zoom scalando la propria dimensione in modo proporzionale ad uno scale factor. Questi oggetti rappresentano il background, la mappa.

- Oggetti che reagiscono allo zoom senza scalare la propria dimensione, ma traslando il loro centro geometrico in modo che risulti sempre posizionato nello stesso punto relativo sul background che nel contempo si è ingrandito o rimpicciolito. Questi oggetti rappresentano gli Spot: indicatori posti in punto preciso della mappa che rappresentano un punto di interesse per l'utente o per lo sviluppatore.

Abbiamo cioè simulato ciò che Google Maps fa con le sue mappe, i markers (indicatori sulla mappa) non cambiano mai dimensione quando compiamo lo zoom, mentre lo sfondo (la mappa) sì. Naturalmente i markers indicheranno sempre lo stesso punto nonostante che esso sia stato traslato a causa dello zoom.

Su questa base abbiamo costruito il package **building**, che contiene le classi per la gestione e per il disegno di una generica mappa indoor di un edificio, ed il package **spot**, che contiene la gerarchia di classi per la gestione di ciò che deve essere disegnato sopra alla mappa avendo quel comportamento traslatorio agli eventi di zoom.

Questi due package sono strettamente legati e non possono esistere indipendentemente. Data la maggiore complessità della gerarchia degli **Spot**, partiremo descrivendo prima com'è costituito un edificio indoor per il disegno della mappa.

Building: è il contenitore dell'edificio, contiene tutti gli oggetti e le informazioni necessarie per disegnare l'edificio. In particolare è un'aggregazione di Floor (cioè di "piani" dell'edificio), inoltre contiene uno SpotManager e può gestire il disegno degli spot e del cammino tra due spot, ma approfondiremo questo argomento in un secondo momento.

Floor: è un contenitore per le stanze dell'edificio, inoltre contiene anche questo una serie di SpotManager che vedremo più tardi. Ogni edificio può avere un solo Floor attivo contemporaneamente, e sullo schermo verrà disegnato un unico Floor alla volta.

Room: è l'entitá più complessa e ricca di funzionalitá. Identifica il concetto di stanza di un edificio. Deve gestire il disegno dei muri, delle porte e dei pavimenti. Per farlo deve essere definito un perimetro della stanza utilizzando i Vertex. La Room contiene una lista ordinata di Vertex e disegnerá il perimetro della stanza seguendo le coordinate indicate da questa lista. A seconda del tipo di Vertex in cui la procedura di disegno si imbatte, essa disegnerá un muro piuttosto che una porta o una apertura (spazio aperto).

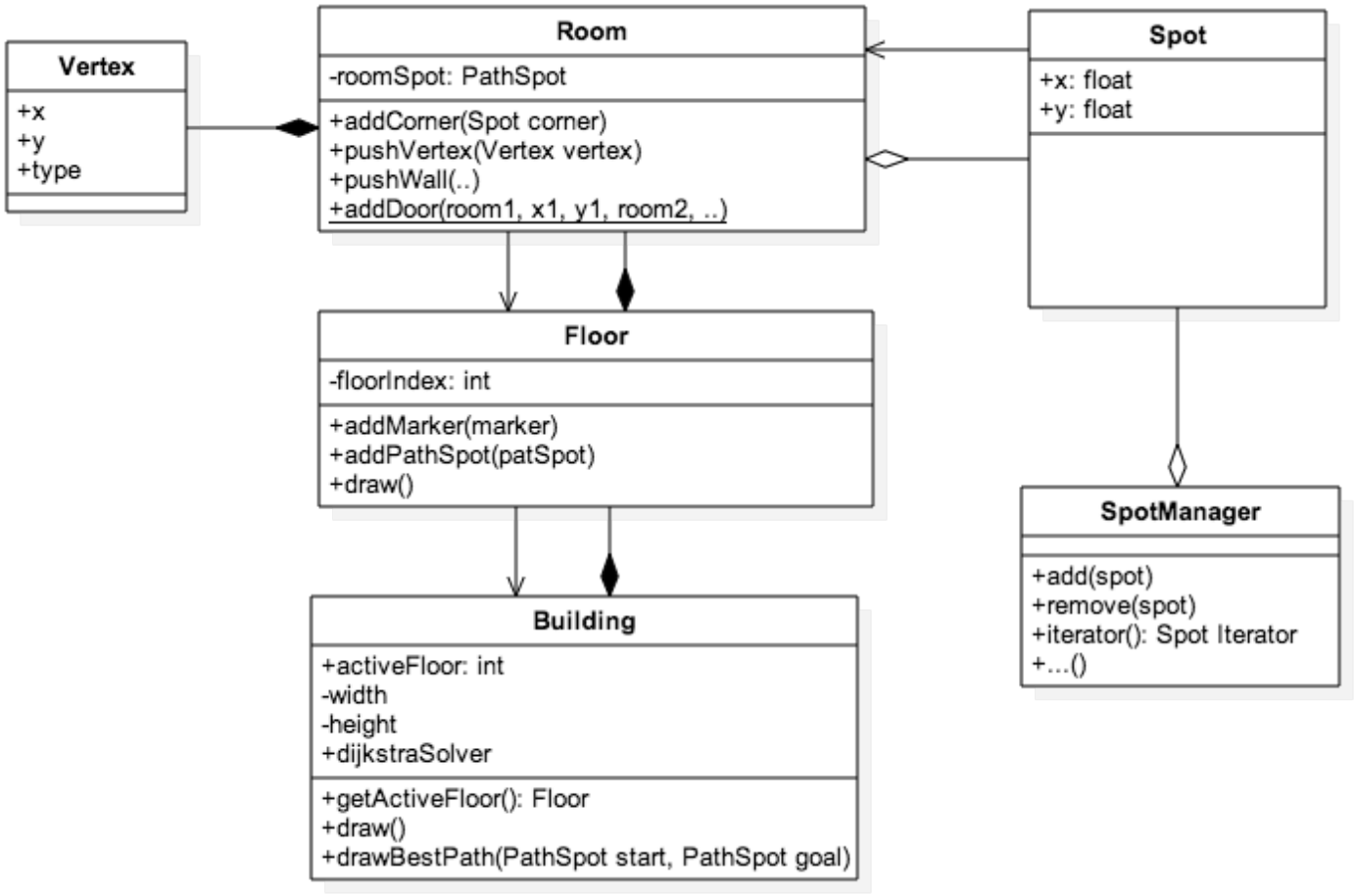
Room inoltre contiene una lista di Spot. Ogni spot infatti è localizzato in una Room. In questo modo possiamo a priori sapere dove si trovano gli spot (in che stanza) e possiamo ricavare una lista di spot a partire da ogni stanza comunicandoci che cosa è contenuto, funzionalità molto interessante poiché le estensioni della classe spot identificheranno i concetti di marker e punti di interesse. Sarà quindi possibile, a partire da ogni stanza, avere una lista dei marker o punti di interesse contenuti.

Ogni stanza ha inoltre uno Spot speciale, chiamato RoomSpot, che deve identificare approssimativamente il centro della stanza. Sarà utilizzato per semplificare la costruzione dei cammini all'interno della stanza.

Vertex: rappresenta un vertice di una stanza con delle coordinate spaziali x y in metri. Ogni edificio è disegnato dentro ad una cornice delle dimensioni specificate dall'edificio (width e height), per cui si prende come punto (0, 0) l'origine degli assi, ovvero il primo punto a partire da sinistra in alto dell'immagine. Ogni Vertex ha un tipo che specifica come deve essere disegnato il poligono tra il vertex corrente ed il successivo.

Spot: classe base della gerarchia di Spot, identifica un punto nello spazio definito in metri (come per i Vertex). Al contrario del Vertex, uno spot sa sempre in che stanza è contenuto.

SpotManager: contenitore e gestore per Spot generici. Uno Spot non deve obbligatoriamente essere gestito da uno SpotManager, cioè uno Spot può esistere anche se nessuno SpotManager lo contiene. Oltretutto uno Spot non ha coscienza di dove sia contenuto (non ha un riferimento allo SpotManager che lo contiene, ma solo alla stanza). Quindi uno Spot può essere contenuto da una sola stanza, ma può essere contenuto da più SpotManager.



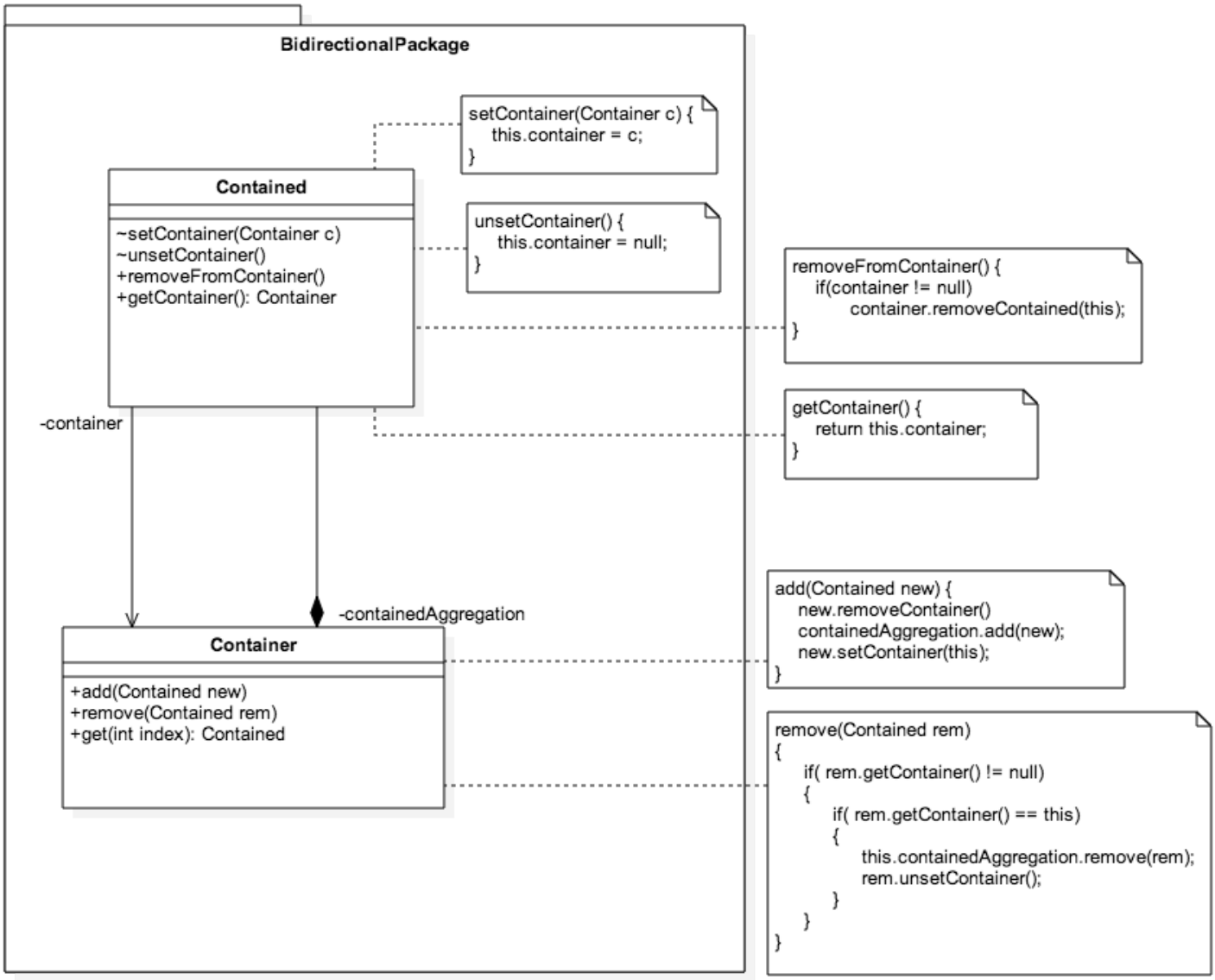
Per disegnare la mappa sarà quindi sufficiente richiamare il metodo draw() di Building, il quale provvederà a delegare la richiesta a tutti i suoi contenuti. Ovviamente non è sufficiente richiamare draw() per fare apparire sullo schermo la mappa, infatti questo metodo riceve in ingresso un oggetto della classe Canvas di Android, e farà in modo che tutto ciò che verrà disegnato sarà diesgnato proprio sopra questo Canvas.

L'utilizzatore della classe Building dovrà quindi gestire la parte di proiezione del canvas sullo schermo. Nel nostro caso abbiamo delegato questa operazione alla classe IndoorMap e IndoorMapFragment, che provvedono a proiettare su una ImageView di android il canvas su cui è stato disegnato l'edificio. Queste classi provvedono anche ad implementare i gestori degli eventi della ImageView in modo da gestire lo zoom e il drag della mappa.

Per gestire la bidirezionalitá delle relazioni *contenitore – contenuto* presenti tra Building – Floor, Floor – Room e Room – Spot, si è utilizzato sempre lo stesso modello. Poiché il modello è risultato funzionale ed efficace è stato creato una gerarchia di classi per implementare automaticamente questo modello. Le entitá fondamentali sono Container e Contained, e la loro unione (ContainerContained) fanno si che sia possibile gestire catene di Contenitori e Contenuti senza dover reimplementare lo stesso pattern.

Spieghiamo brevemente il funzionamento di tale pattern: il contained offre imetodi setContainer ed unsetContainer che, senza effettuare alcun controllo, modificano il riferimento privato del contained verso un container a piacimento (nullo nel caso di unsetContainer). Questi metodi sono package protected, ovvero accessibili solo al package. Al contrario, i metodi Add e Remove del Container sono pubblici ed aggiungono un oggetto Contained all'elenco interno degli oggetti contenuti. Allo stesso tempo effettuano un controllo sull'oggetto da aggiungere e modificano in modo appropriato il loro contenutore. Questo è possibile perché il Container ed il Contained devono stare nello stesso package, in modo che il Container possa accedere ad i metodi package protected.

Nonostante i dibattiti sul concetto di classe friend e metodo friend, siamo dell'idea che in questo caso sarebbe stato più consono l'utilizzo di tali concetti per poter offrire al solo Container l'accesso ai metodi del Contained che adesso hanno visibilitá package protected.



Tornando ai dattagli del motore indoor, possiamo adesso analizzare la gerarchia di classi degli Spot e SpotManager. Le principali classi della gerarchia sono: Spot, DrawableSpot, MarkerSpot e PathSpot. Ad ogni classe è delegata la definizione di un ben preciso comportamento.

Spot: è la classe base, definisce le funzionalitá basilari e comuni a tutti gli Spot, che sono:

- Il sistema di posizionamento assoluto (in coordinate (x,y) espresse in metri)
- Il comportamento da "contenuto" seguendo il pattern descritto poco fa, ogni spot è contenuto infatti in una ed una sola stanza (ed ha un riferimento alla stanza nel quale è contenuto).

DrawableSpot: estende lo Spot per implementare le funzionalitá di disegno e di coordinate in pixel relative allo schermo, alla densitá dello schermo e allo scale factor del background. È una classe astratta, poiché lascia alle sottoclassi il compito di definire che cosa si vuole disegnare, prendendosi carico solo di definire le linee guida di come gestire un generico disegno di uno Spot.

La gestione del disegno avviene nel seguente modo:

1) Un oggetto privato di tipo Drawable (classe di Android SDK, android.graphics.drawable.Drawable) si prende carico di memorizzare un generico oggetto disegnabile per adesso indefinito.
2) È fornito un metodo getter pubblico per ottenere tale Drawable e poterlo disegnare presumibilmente su un Canvas (classe di Android SDK per il disegno vettoriale).
3) Viene dichiarato (ma non definito) il metodo astratto protected generateDrawable(), che si prende carico di generare e restituire un generico Drawable.
4) Il costruttore della classe Drawable costruisce l'oggetto privato di cui abbiamo parlato al punto (1) richiamando il metodo astratto (che verrá definito in una sottoclasse).

Chiaramente un DrawableSpot di per se non può essere istanziato (è astratto) ma saranno le sottoclassi a dover definire che cosa vogliono effettivamente disegnare, implementando il metodo generateDrawable(). Una volta fatto sará comunque il codice della superclasse DrawableSpot a gestire il disegno, offrendo una interfaccia univoca per la gestione e lasciando che la sottoclasse si possa focalizzare su altro.

La gestione fondamentale imposta dal DrawableSpot è quella di trasformare il sistema di coordinate assoluto in metri in un sistema di coordinate in pixel. Deve quindi gestire tutte le possibili trasformazioni a cui una immagine interattiva è soggetta, come il rescale del background, la traslazione, il rescale dello spot stesso.

Come abbiamo giá detto è fondamentale che, facendo uno zoom del background, gli Spot possano comunque rimanere visivamente della stessa dimensione, continuando a "puntare" la propria coordinata assoluta (x,y) nello stesso punto del background dilatato dallo zoom (esattamente il comportamento che hanno i Markers di google maps). Per fare quest devono poter reagire ad uno zoom del background con una traslazione nelle coordinate in pixel nelle direzioni x ed y.

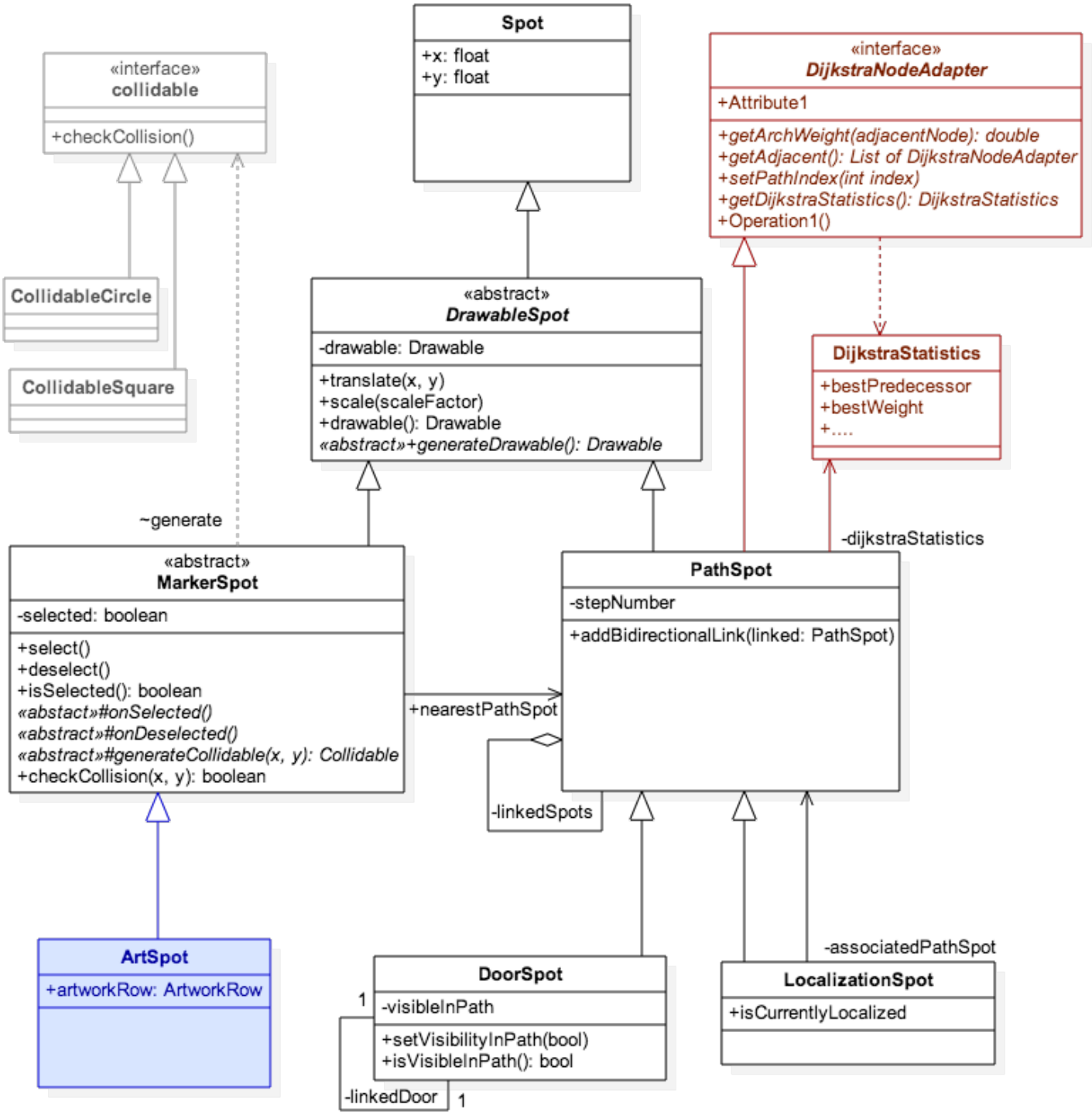
Per poter eseguire correttamente questo compito è stato necessario memorizzare nell'oggetto nuovi valori:

- **translation (x & y):** traslazione dell'oggetto rispetto alle coordinate assolute dovuta da una traslazione del background.
- **scaleTranslaton(x & y):** translazione dell'oggetto rispetto alle coordinate assolute dovuta al rescale del background.
- **realtime_scaleTranslation_factor:** variabile a virgola mobile in cui memorizziamo in tempo reale lo scale factor istantaneo del resize corrente. Se eseguiamo un pinch to zoom infatti, ad ogni istante avremo un nuovo fattore di scala indipendente da quello dell'istante precedente (i gestori di eventi effettuano comunque una discretizzazione del tempo, ogni unitá di tempo viene gestito un evento che impone un fattore di scala nuovo ed indipendente dal fattore di scala dell'evento precedente).
- **last_final_scaleTranslation_factor:** variabile a virgola mobile incui memorizziamo l'ultimo valore del realtime_scaleTranslation_factor, al termine di una serie di continui resize dovuti ad un pinch to zoom o simili.

Nel momento in cui le dita lasciano lo schermo, il fattore realtime viene memorizzato nel last_final e il realtime settato a 1.

Tutti questi parametri sono interni alla classe (privati) e non accessibili direttamente. Saranno i metodi pubblici messi a disposizione a gestire automaticamente questi parametri in modo da ottenere i risultati desiderati. I metodi a disposizione sono:

- **setRealtimeScaleTranslationFactor(float newRealtimeScaleFactor)** : viene passato a questo metodo lo scaleFactor int tempo reale durante la gestione del pinch to zoom.
- **setFinalScaleFactor()** : richiameremo questa procedura quando il touch event pinch to zoom è terminato.
- **setTranslation(flaot x, float y)**: consente di indicare una traslazione del background/foreground a cui gli spot sono legati.
- **x / y _forDrawing()**: getter che ritornano i valori delle coordinate x/y in pixel giá scalati e pronti per essere correttamente disegnati sullo schermo.



PathSpot: estende il DrawableSpot per fornire le funzionalitá di “tappa intermedia” di un percorso indoor. La principale caratteristica di un PathSpot è che possiede una lista di archi uscenti verso altri PathSpot. Questo consente di creare dei grafi, e quindi dei percorsi.

Il PathSpot implementa l’interfaccia offerta da un nostro package esterno, costruito per la risoluzione di un generico problema di cammino minimo, cioè l’interfaccia DijkstraNodeAdapter.

È quindi obbligata ad implementare i metodi astratti offerti dall’interfaccia:

- **getAdjacent():** restituirá tutti i PathSpot collegati al PatSpot corrente
- **getArchWeight(adjacent):** restituisce il peso dell’arco tra il PathSpot corrente e quello adiacente passiato come argomento. PathSpot implementa questo metodo restituendo la distanza euclidea (utilizzando le coordinate assolute in metri).
- **getDijkstraStatistics():** deve restituire un oggetto di tipo DijkstraStatistic associato al PathSpot corrente. Per questo viene mantenuta una istanza di tale classe memorizzata come oggetto privato, per poi restituirlo su richiesta.
  Sará il package DijkstraSolver a gestire questo oggetto ai fini di risolvere il problema del percorso minimo.
- **setPathIndex(index):** serve per comunicare all’oggetto generico, che implementa il DijkstraNodeAdapter, in che posizione del percorso ottimo si trova.

Oltre ad implementare queste funzionalitá derivate, il PathSpot implementa dei metodi per la gestione dei collegamenti tra PathSpot come addBidirectionalLink(patSpot) e altri. Inoltre implementa il metodo generateDrawable(canvas) della superclasse DrawableSpot, definendo così una modalitá standard di disegno dei PathSpot, modalitá che potrá comunque essere sovrascritta da eventuali sottoclassi che vogliano utilizzare una differente modalidtá di disegno.

Questi Spot sono quindi completamente compatibili con la classe DijkstraSolver, che abbiamo creato per risolvere un generico problema di cammino minimo, e sono quindi utilizzati come punti intermedi/partenza/destinazione nella navigazione indoor di un generico edificio.

Esistono due sottoclassi importanti che ridefiniscono un comportamento speciale di un generico PathSpot:

- **DoorSpot**: deve essere collegata obbligatoriamente ad un altro DoorSpot, ognuno dei quali deve stare in due stanze adiacenti, con una relazione strettamente 1 ad 1. Estende inoltre la modalitá di disegno, poiché i DoorSpot di default non vengono disegnati nel percorso (sono nascoste), ma con un setter è possibile specificare che vengano disegnate. Oltre all’associazione 1 ad 1 con l’altra “estremitá” della porta, continuano ad essere dei normali PathSpot, con il proprio insieme di collegamenti verso altri N generici PathSpot.

- **LocalizationSpot:** un LocalizationSpot è un PathSpot speciale che ha un collegamento con il PathSpot più vicino. Il ruolo fondamentale è quello di disegnare l’indicatore della posizione corrente dell’utente sullo schermo, e generalmente questo è fatto ponendo il LocalizationSpot esattamente nelle stesse coordinate del PathSpot in cui è localizzato. Ridefinisce quindi i metodi per il disegno del PathSpot ereditati dal DrawableSpot in modo che venga visualizzato l’indicatore di posizione: blu se l’utente è in quel momento localizzato in tempo reale da un sensore, grigio se non si è più localizzati, ad indicare l’ultima posizione rilevata.

MarkerSpot: estendono DrawableSpot per offrire una funzionalitá aggiuntive, come la reazione agli eventi di selezione. Un marker è infatti uno Spot disegnabile che può essere selezionato dall'utente.

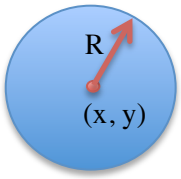
È ancora una classe astratta, poiché definisce un marker astratto disegnabile senza andare a definire che cosa verrá disegnato, delegando tale compito ad eventuali sottoclassi concrete. Nel nostro caso utilizzeremo la classe ArtSpot, che estende MarkerSpot, fornendo due bitmap disegnabili che si scambiano a seconda che il marker sia stato selezionato o meno.

La classe MarkerSpot ha la responsabilitá di definire dei metodi per il controllo delle collisioni, in modo che si possa capire se un generico MarkerSpot è in collisione con un punto qualsiasi dello spazio definito dallo schermo del dispositivo.

Per fare ciò abbiamo definito una interfaccia Collidable, che rappresenta un oggetto che può collidere con un punto, che quindi è in grado di controllare tali collisioni e restituire un valore di veritá. Il MarkerSpot ha un metodo protetto **generateCollidable(x, y)** che restituisce un oggetto generico della gerarchia Collidable. Questo metodo viene utilizzato dagli altri metodi pubblici come checkCollision(x, y) per controllare le collisioni. Per questo motivo, le sottoclassi di MarkerSpot (ovvero i Marker non astratti, concreti), dovranno anche ridefinire dei Collidable concreti, poiché uno di essi dovrá essere ritornato dal metodo astratto generateCollidable(x, y).

Con **generateCollidable** andiamo cioè a definire la forma fisica del marker. Abbiamo implementato due Collidable di esempio che possono essere riutilizzati in qualsiasi MarkerSpot:

- CollidableCircle: definisce un collidable circolare a partire dalle coordinate (x,y) del centro e da un raggio R.



- CollidableRect: definisce un collidable rettangolare con lati ortogonali agli assi cartesiani, a partire da due punti definiti ognuno con due coordinate:

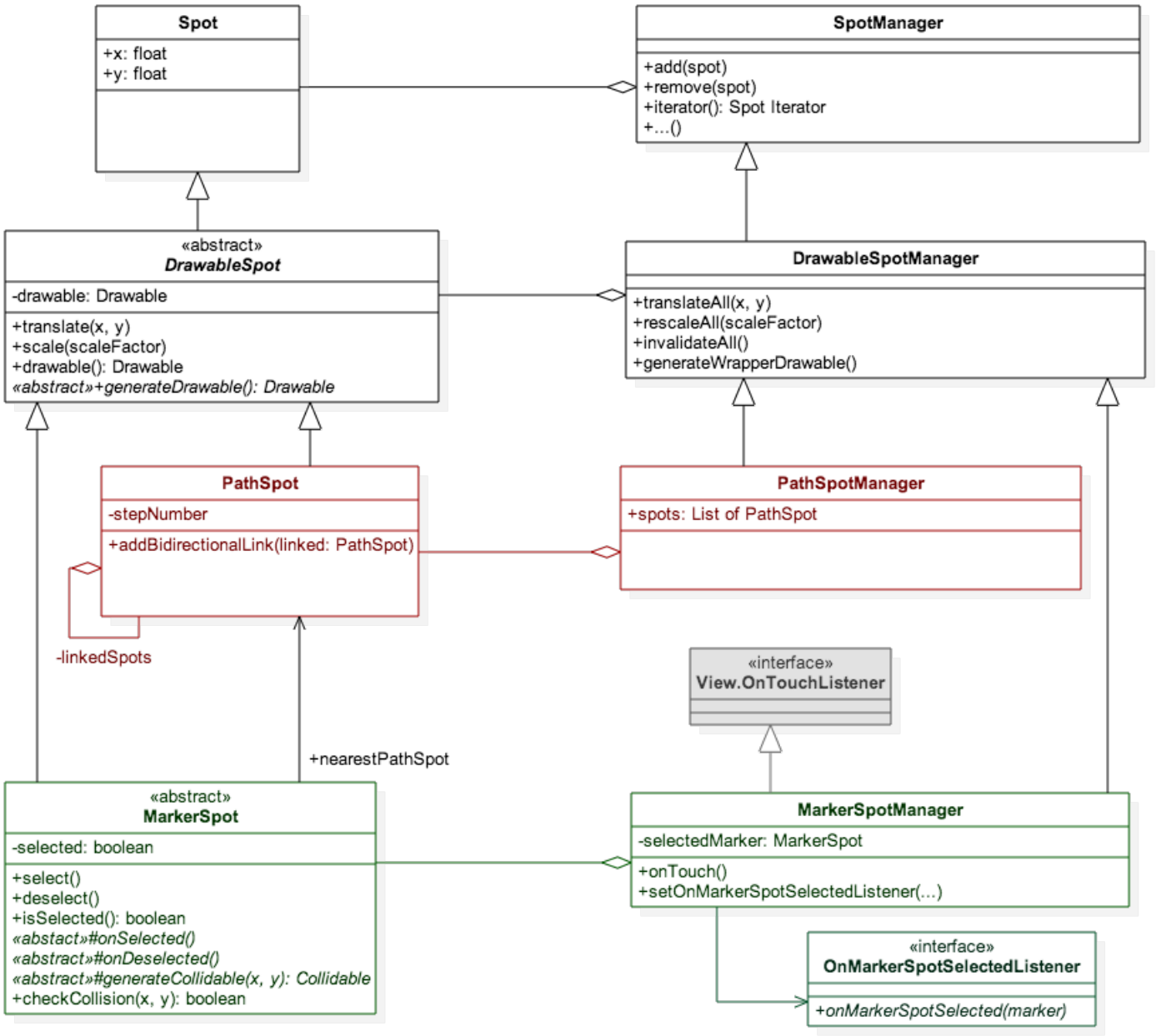
Il CollidableObject concreto dovrá implementare il metodo checkCollision(x, y), che deve ritornare il valore di veritá a seconda che il punto (x, y) appartenga o meno all'oggetto geometrico descritto dalle sue stesse proprietá.

Per applicazioni in cui non è necessaria una grande precisione, ma anzi i dispositivi di input sono inaccurati, queste due implementazioni di Collidable basteranno. Spesso, sui diffusissimi dispositivi touch screen capacitivi, si può preferire addirittura creare aree di collisioni più grandi della dimensione reale dell'oggetto disegnato sullo schermo, data la scarsitá di precisione della tecnologia e del supporto di puntamento, che spesso è proprio il dito dell'utente.

Oltre alla gestione delle collisioni, MarkerSpot gestisce anche la proprietá interna di selezione: un marker può essere selezionato o meno. Mette quindi a disposizione dei metodi getter e setter per tale proprietá (select(), deselect(), isSelected()) ed inoltre mette a disposizione dei gestori di eventi implementabili dalle sottoclassi, eventi che verranno scaturiti ogniqualvolta l'utente selezionerá o deselezionerá il Marker: onSelected(), onDeselected().

Le sottoclassi potranno definire il comportamento da assumere in questi due determinati eventi.

Per gestire in modo semplice grosse quantitá di Spot con comportamento simile, si è pensato di creare una seconda gerarchia di classi parallela a quella degli Spot: gli **SpotManager**.



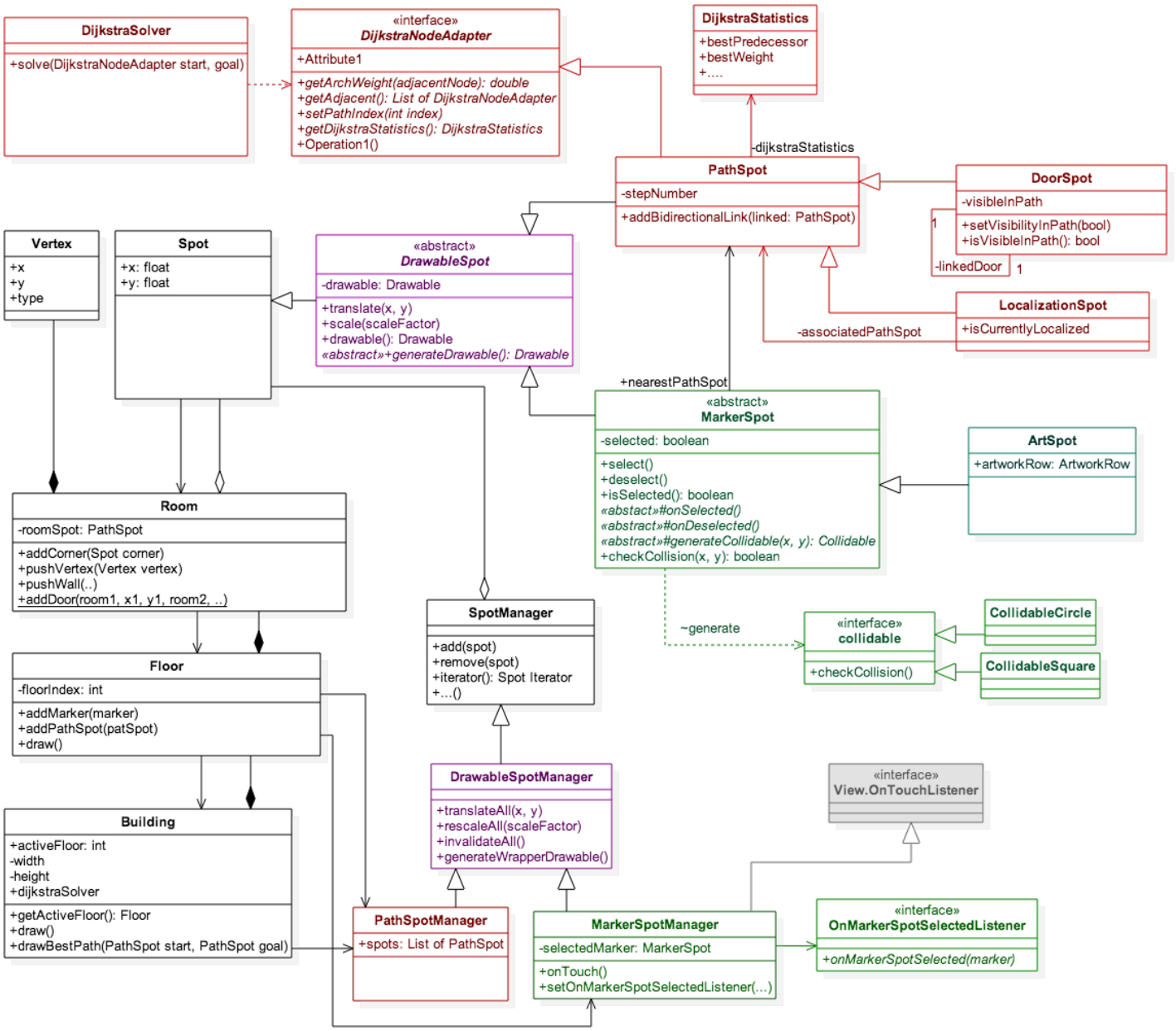
SpotManager: si occupa semplicemente di gestire l'aggregazione SpotManager – Spot, con metodi per aggiungere, rimuovere e recuperare gli spot contenuti.

DrawableSpotManager: aggiunge funzionalitá di disegno collettive, come metodi per traslare e scalare tutti i DrawableSpot contenuti. Inoltre offre il metodo per generare un "wrapper" per i Drawable, cioè un Drawable che a sua volta contiene tutti gli altri, in modo che disegnando questo unico Drawable contenitore si disegnino tutti gli altri. Con lo stesso principio, il metodo invalidateAll() servirá per invalidare tutti i Drawable contenuti in modo che siano ridisegnati il prima possibile.

PathSpotManager: ridefinisce semplicemente le funzionalitá del DrawableSpotManager, il wrapperDrawable infatti in questo caso non conterrá solo una serie di Drawable da disegnare, ma avrá anche dei disegni propri non contenuti in alcun Spot: quando si disegna il cammino tra due PathSpot, le linee che uniscono i singoli spot vengono disegnate dal PathSpotManager.

MarkerSpotManager: implementa l'interfaccia OnTouchListener di Android SDK, in questo modo potrá gestire direttamente gli eventi touch generati dal sistema operativo Android, per poi delegare ad ogni MarkerSpot la verifica delle collisioni. Ad ogni evento touch gestito, verrá richiamato un gestore di eventi di un appropriato listener: OnMarkerSpotSelectedListener. Ogni MarkerSpotManager può avere solo un listener al quale invierá i messaggi.

Adesso che sono stati spiegati singolarmente i componenti del motore, con una visione di insieme dell'indoor engine, è possibile comprenderne il funzionamento generale e l'interazione tra le classi.



Visione completa di Indoor Engine

Come possiamo vedere dal diagramma UML, il Floor contiene un PathSpotManager ed un MarkerSpotManager. Infatti verrá visualizzato sullo schermo un solo Floor alla volta, quindi con questi manager abbiamo un accesso comune ed unico per tutti gli spot contenuti in un singolo piano dell'edificio. Sará quindi il Floor che si occuperá di disegnare tutti i DrawableSpot (e altri oggetti della gerarchia Spot) contenuti nel PathSpotManager e MarkerSpotManager.

Il Building invece offre una interfaccia per il DijkstraSolver. Utilizzerá una istanza di tale classe per generareun PathSpotManager disegnabile con il percorso dal punti iniziale al punto finale, percorso calcolato dal DijkstraSolver con l'aiuto dei DijkstraNodeAdapter che, in questo caso, sono concretizzati dai PathSpot.

Un accenno sul IndoorMap e IndoorMapFragment: queste due classi incapsulano il building e quindi tutto il resto, e gestiscono la costruzione dell'edificio a partire dai dati forniti dal database oltre che la visualizzazione della mappa su ImageView e la gestione dei TouchEvent legati al pinch to zoom e al drag per spostare la visuale della mappa. Poiché anche il MarkerSpotManager vuole gestire gli eventi touch, il gestore presente in IndoorMap, una volta

analizzato e gestito l'evento, richiamerá il gestore di eventi del MarkerSpotManager del piano dell'edificio attivo, per lasciare che esso possa gestire i propri personali touch events.

PROGETTAZIONE DEL SOFTWARE: SENSORI INDOOR.

Per la gestione dei sensori beacon bluetooth abbiamo utilizzato le librere per Android fornite da Estimote, noto produttore di beacon bluetooth. Dopo vari esperimenti e confronti con diversi dispositivi e diverse piattaforme, siamo giunti alle conclusioni già discusse riguardo all'inaffidabilità generale dei valori delle distanze riportati sui terminali basati sul sistema Android, confrontandoli invece con l'accuratezza dei valori riscontrati con i terminali di Apple.

Abbiamo quindi abbandonato del tutto la possibilità di compiere una triangolazione, concentrando piuttosto le nostre attenzioni per cercare di regolarizzare i valori in ingresso sulla distanza, per ottenere una buona stabilità nella rilevazione della prossimità con i sensori.

La classe **BeaconHelper** è pensata per offrire un'interfaccia semplice verso la libreria di Estimote, per la gestione dei beacons.

Sulla base di questo Helper, abbiamo costruito un BeaconProximityManager astratto, per poi implementare un Manager concreto che abbiamo chiamato **GoodBadoBeaconProximityManager.**

Questo manager utilizza una semplice idea per tentare di regolarizzare i valori ottenuti dai beacon: se siamo vicini ad un beacon e lontani da un altro, con grande probabilità i segnali provenienti dal beacon vicino avranno mediamente una potenza maggiore dei segnali provenienti dal beacon lontano (e quindi i relativi valori di distanza convertiti in metri).

Non potendoci basare ciecamente sui valori delle distanze fornite dai beacon, a causa della mancata calibrazione dei terminali basati su sistema operativo Android, abbiamo pensato che fosse più affidabile effettuare in tempo reale una sorta di "gara a punti" tra i beacon considerati abbastanza vicini. Il Beacon che "vince" la gara è considerato il più vicino. Ogni istante (step) viene rimesso in discussione il vincitore, mantenendo però in memoria i punti, per fornire una sorta di inerzia che impedisca cambi repentini del beacon eletto come più vicino. L'idea alla base del funzionamento è la seguente:

1. Se il beacon considerato vincitore nei passi precedenti, al passo corrente non vince (non ha il relativo valore di potenza ricevuta più alta) esso deve effettuare un **bad step**, perdendo punti proporzionali al suo punteggio corrente e ad un coefficiente moltiplicativo costante.
2. Se un qualsiasi beacon al passo corrente vince, effettua un **good step**, incrementando il proprio punteggio in base ad un secondo coefficiente.
3. In ogni caso, tutti beacon effettuano ad ogni step uno **standard step**, che andrà a riequilibrare la situazione (se un beacon ha un punteggio alto tenderà a scendere, se un beacon ha pochi punti o punti negativi tenderà a risalire, fino a stabilizzarsi intorno al valore iniziale, anch'esso registrato in un altro coefficiente costante)

Tutti i parametri costanti sono modificabili per cambiare il bilanciamento con cui i beacons non vicini riescono a guadagnare punti per essere considerati vicini, e viceversa. Bilanciando i parametri si riesce a modificare il tempo di risposta e la accuratezza.

Naturalmente anche i parametri dei beacon faranno la differenza (potenza in uscita e numero di pacchetti in uscita al secondo).

Con i nostri test abbiamo trovato un buon bilanciamento per la nostra applicazione, ma non è escluso che in altri ambienti e con tipologie di beacons differenti sia necessario ricalibrare i parametri.

Per quanto riguarda l'utilizzo dei QR code, la libreria ZXing rende l'integrazione semplice. Utilizzando un applicativo esterno scaricabile dal Google Play Store, si effettua la scansione del QR code ritornando i parametri di interesse.

Abbiamo quindi collegato con delle mappe hash i codici dei beacons (coppia di due indirizzi modificabili chiamati Minor e Major) e quelli dei QR code ai relativi oggetti PathSpot e ArtSpot.

In questo modo, nell'istante in cui si riceve per esempio una notifica di un nuovo beacon in prossimità, il software cercherà lo spot associato nella mappa e gestirà l'evento in modo appropriato modificando gli oggetti in gioco e avvisando l'utente.

PROGETTAZIONE DEL SOFTWARE: DB JSON MANAGER

Questo modulo del software gestisce il download di un generico file json costruito a partire da una risposta di un database generico, e lo traduce in oggetti JAVA.

I componenti fondamentali sono: TableSchema, TableRow, ColumnSchema, ColumnRow.

TableSchema: classe astratta generica, definisce uno schema per una tabella di un database generico e genera il TableRow generico associato (ovvero una riga del database, una tupla). Tramite i getter è possibile ottenere tutti gli schemi di tutte le colonne, oltre al nome della tabella stessa. I metodi astratti che devono essere definiti per generare una TableSchema concreta sono:

- **generateTableName()** : deve ritornare semplicemente una stringa statica col nome della tabella (o della vista) così come è nominata nell'oggetto JSON (abbiamo assunto che sia lo stesso nome utilizzato sul database).
- **generateTableColumns()**: deve ritornare un array di TableColumns che vanno a definire tutte le colonne (i campi) della tabella.
- **generateRow()**: deve ritornare una TableRow associata a questo TableSchema. Per avere una corrispondenza biunivoca abbiamo utilizzato i generics: ogni TableSchema è definito su una classe generica TR che estende TableRow, ed ogni TableRow (quindi anche TR, quando sarà concretizzata) è definito su una classe TS generica che estende TableSchema. TR avrà un riferimento al TS che lo ha generato, TS invece genererà TableRow di tipo TR.

TableRow: come appena accennato, è una classe astratta generica che interpreta il ruolo di tupla. È possibile risalire al TableSchema che lo ha generato ed ottenere i campi contenuti sotto forma di ColumnField (che contengono il valore ed il tipo, identificato dal ColumnSchema).

ColumnSchema: anche questa classe è astratta generica, interpreta il tipo di dato per la colonna e gestisce il nome della colonna stesso (l'identificatore). Oltre a ciò è in grado di generare dei ColumnField tramite un metodo astratto, che dovrà quindi essere ridefinito per ogni coppia concreta Column Schema - Field. È definita mediante un tipo generico T che sarà, una volta fissato in una sottoclasse, il tipo di dato rappresentato dal ColumnSchema. La ColumnField generabile dovrà infatti avere il medesimo tipo di dato T.

ColumnField: classe astratta generica, ha lo stesso tipo di dato T generico e memorizza al suo interno una variabile di questo tipo. Dal ColumnField è possibile risalire al ColumnSchema che lo ha generato, è possibile ottenere il valore con dei getter:

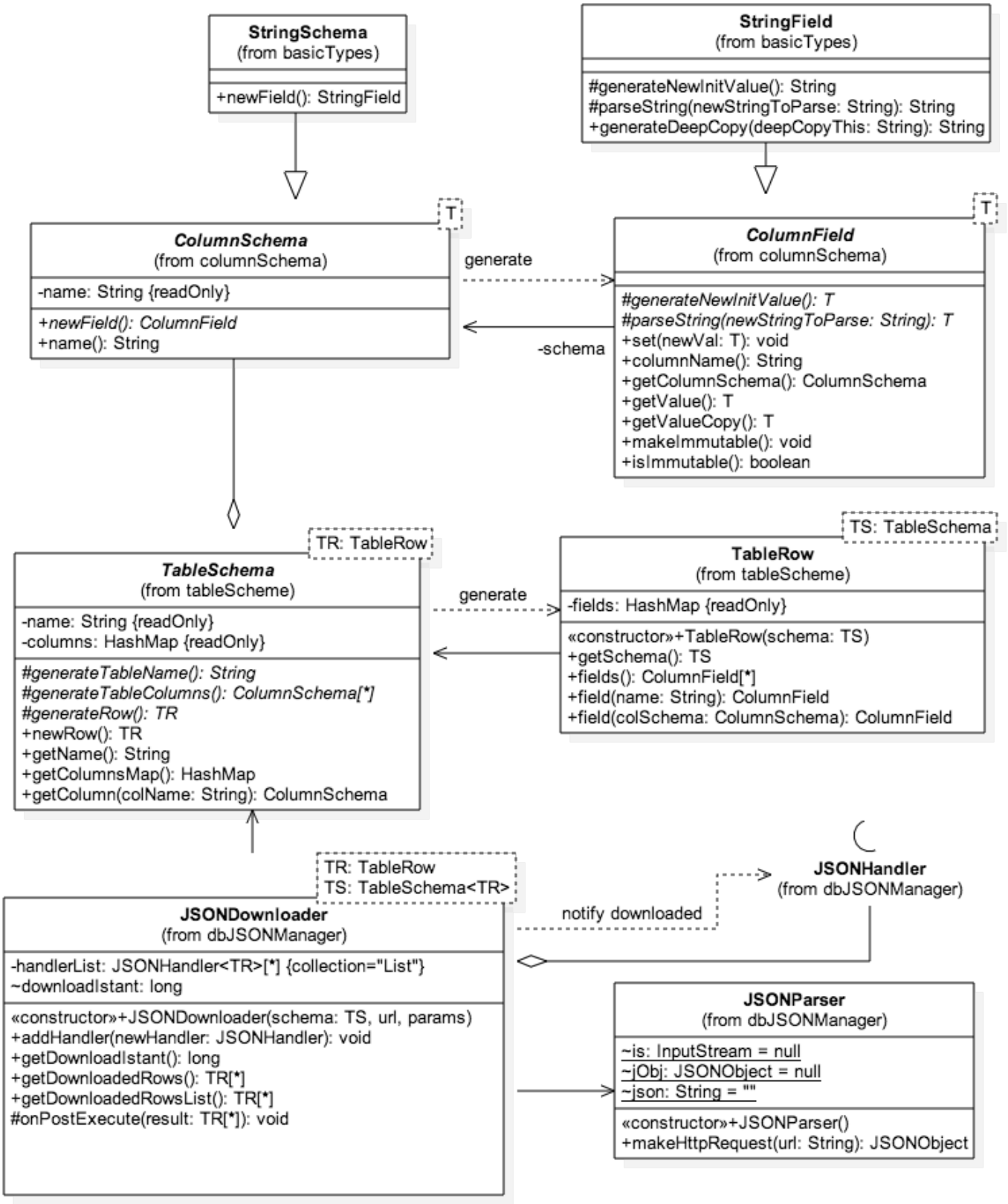
- **getValue():** ritorna il valore per riferimento, comportamento di default per java.
- **getValueCopy():** tenta di ritornare il valore per copia, richiamando il metodo protetto astratto generateDeepCopy() che deve essere ridefinito dalla classe che estende e definisce un ColumnField concreto. Il successo di una effettiva deep copy del valore dipenderà da come è definito nella sottoclasse questo metodo, non si può quindi a priori considerare una deep copy, per farlo dovremo assicurarci che la sottoclasse gestisca correttamente questa funzionalità.

Tramite setter è possibile modificare il valore (di tipo generico T) contenuto all'interno del field. Inoltre sono disponibili altri due metodi astratti che delegano alla sottoclasse il loro corretto funzionamento:

- **generateNewInitValue():** deve generare e ritornare un nuovo valore di tipo T, che a questo punto sarà un tipo concreto e quindi istanziabile. Servirà per inizializzare il valore del field nel costruttore.
- **parseString(string):** deve generare e ritornare un nuovo valore di tipo T in base alla stringa passata come argomento. Dovrebbe servire a convertire un valore di questo generico tipo T memorizzato come stringa, in un valore binario. Questo è molto utile in quanto, nella pratica, ogni oggetto JSON è memorizzato come stringa e deve essere convertito in valore binario per essere utilizzato e manipolato in java.

ColumnField offre anche la possibilitá di diventare immutabile. Richiamando il metodo makeImmutable() verrá automaticamente fatta una deepCopy (basata sul metodo generateDeepCopy()) che viene memorizzata su una variabile interna che, inizialmente, è inizializzata ad un valore nullo. Da quel momento in poi, i setter non funzioneranno più ed i getter ritorneranno il valore della nuova variabile generata tramite DeepCopy.

StringSchema e StringField sono una coppia di esempio che estendono ColumnSchema e ColumnField. Nel codice abbiamo esteso tutti i tipi base più usati (boolean, int, long, String, float, double).



TableSchema e TableRow possono essere ridefinite per creare degli schemi e delle rispettive tuple di una tabella concreta, utilizzando i ColumnSchema e ColumnField concreti messi a disposizione o creandone di nuovi.

La classe principale per l'interazione con JSON è **JSONDownloader**, che sfrutta la classe esterna JSONParser per scaricare e tradurre i file json in una lista di TableRow.

Per ogni tabella (o vista) che vogliamo scaricare, dobbiamo definire un oggetto di tipo JSONDownloader dedicato, poiché esso è un generics che richiede un TableRow concreto ed un TableSchema concreto associato ad esso.

Il download verrá fatto partire su un thread separato, una volta completato saranno avvertiti tutti gli oggetti **JSONHandler** che sono stati registrati nel downloader come oggetti in ascolto. **JSONHandler** è una interfaccia, quasliasi classe può estenderla in modo tale da implementare il metodo gestore di eventi e mettersi in coda al JSONDownloader. Se un JSONHandler prova a mettersi in coda ad un downloader che ha giá terminato il download, automaticamente ed istantaneamente verrá richiamato il gestore di eventi dell'handler.

È sempre buona prassi mettersi in coda per il termine del download e gestire l'evento di download completato piuttosto che tentare di ottenere i dati con i getter dal JSONDownloader. Infatti, non si può avere la certezza che in un certo istante qualsiasi il download sia stato portato a termine. Attendere il termine del download, per esempio con un ciclo, potrebbe portare al blocco del programma e dell'interfaccia grafica, rendendo vano l'aver gestito il download su un thread separato.

Per gestire il nostro database abbiamo quindi esteso varie volte TableSchema e TableRow ricalcando le viste costruite dalle query sql, e poi creato un downloader per ogni vista, che verrà fatto partire al momento opportuno con i giusti parametri per ottenere i dati che ci interessano.

Lato server abbiamo utilizzato uno strato di software in PHP per effettuare le connessioni e le query al database MySQL, ed un ulteriore strato per la traduzione delle risposte in JSON.

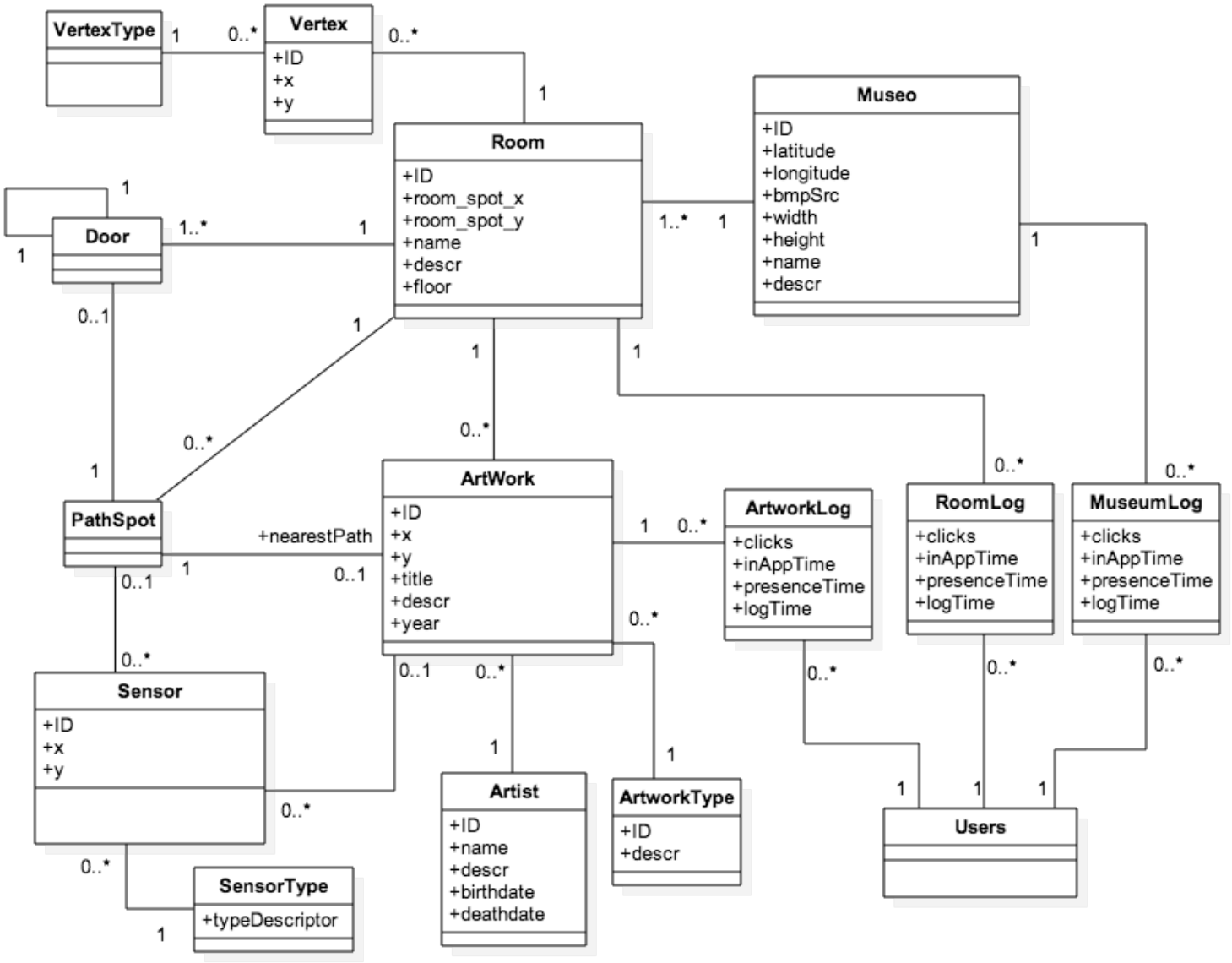
L'applicazione Android si collega quindi al server richiedendo un determinato file PHP che si occuperà di effettuare la query al database e di presentarla in uscita tradotta in formato JSON.

Di seguito viene esposta brevemente la modellazione della base di dati.

PROGETTAZIONE DEL SOFTWARE: MODELLO ER DEL DATABASE

Per progettare il modello Entity Relationship con cui memorizzare i dati sul server, abbiamo ricalcato la gestione dell'indoorEngine semplificandola opportunamente dove possibile. È stato necessario introdurre nuove entità per la gestione degli utenti e delle statistiche sulle loro scelte, ed una entità che rappresentante i sensori associati alle opere o ai PathSpot (associazione che nel codice java viene fatta semplicemente con un HashMap).

Inoltre sono state aggiunte entitá e campi per memorizzare i dettagli su opere, artisti, tipologie di opere, descrizioni dei musei.



RISULTATI

Il prodotto finale è un prodotto perfettamente funzionante ed estendibile, in grado di risolvere intelligentemente gli obiettivi che ci siamo preposti, sia ad alto livello (interfaccia ed usabilitá per utente finale) sia a livello progettuale (il progetto è estendibile e riusabile).

Sicuramente il lato client ed il motore dell'applicativo sono stati gli aspetti più curati, mentre si delega parte del lato server a futuri sviluppi. Per esempio è ancora complicata la creazione delle mappe, non essendoci un editor visuale per effettuare tale operazione. L'utente erogatore del servizio non avrá quindi ancora un prodotto finale, ma con poco lavoro sará possibile realizzare un simile editor basandosi sul motore indoor engine giá sviluppato, per generare mappe storabili su file json o su database.

Il nostro obiettivo era di proporre una ottima esperienza di utilizzo per l'utente finale utilizzatore del servizio, e siamo sicuri di esserci riusciti. Un obiettivo secondario è quello di mantenere bassi i costi di installazione dell'impianto, in modo che un utente erogatore del servizio possa scegliere la nostra tecnologia. Considerata la facilità di estendibilità garantita sotto ogni aspetto, crediamo che anche questo obiettivo possa essere considerato soddisfatto completamente con un altro minimo sforzo, ovvero la creazione di un editor di mappe per consentire ad un utente erogatore del servizio di generare mappe a basso costo ed in tempi rapidi.

Supponiamo che a breve i prezzi dei beacon bluetooth subiranno un forte calo, data la concorrenza crescente in questo campo, ma già adesso siamo in grado di trovare prezzi concorrenziali per beacon meno evoluti che supportano solo la tecnologia di prossimità, l'unica che consideriamo utilizzabile sotto l'ecosistema Android.

Abbiamo infatti scartato l'idea di fare una triagolazione, che garantirebbe una indicazione più precisa della posizione dell'utente all'interno dell'edificio in tempo reale, date le difficoltà riscontrate sotto questo ambiente di sviluppo. Inoltre pensiamo che la feature di prossimità sia più che sufficiente e molto più pratica, precisa, ed esente da problemi.

Un'altra utile tecnologia per lo scopo che ci siamo dati, implementabile in futuro, è quella NFC: con essa è possibile diminuire leggermente il grado di interazione uomo macchina in confronto all'utilizzo dei QR code, perdendo però il vantaggio dei costi nulli di questi ultimi oltre al vantaggio della grossa quantità di utenza che potrebbe sfruttare tale feature.

Riteniamo, in fine, che la tecnolgia software svuiluppata con questo applicativo possa essere un buon punto di partenza per altri applicativi che hanno la necessita di gestire mappe personalizzate, con il pieno controllo da parte dei progettisti del software, su piattaforma Android.