CLIPS

Communication & Localization with Indoor Positioning Systems

Università di Padova

Specifica Tecnica v0.03

leaf.gruppo@gmail.com

Versione	
Data Redazione	
Redazione	
Verifica	
Approvazione	
Uso	Esterno
Distribuzione	Prof. Vardanega Tullio Prof. Cardin Riccardo Miriade S.p.A.

Diario delle modifiche

Versione	Data	Autore	Ruolo	Descrizione
0.13	2016-03-18	Marco Zanella	Progettista	correzioni
0.12	2016-03-18	Marco Zanella	Progettista	aggiunta immagini
0.11	2016-03-18	Marco Zanella	Progettista	Spiegazione database
0.10	2016-03-18	Marco Zanella	Progettista	Spiegazioni componenti model
0.09	2016-03-16	Eduard Bicego	Progettista	Stesura tecnologie utilizzate
0.08	2016-03-13	Federico Tavella	Progettista	Stesura descrizione pattern strategy
0.07	2016-03-13	Federico Tavella	Progettista	Stesura descrizione pattern facade
0.06	2016-03-13	Federico Tavella	Progettista	Stesura descrizione pattern singleton
0.05	2016-03-13	Federico Tavella	Progettista	Stesura descrizione pattern observer
0.04	2016-03-13	Federico Tavella	Progettista	Stesura descrizione pattern dependency injection
0.03	2016-03-13	Federico Tavella	Progettista	Stesura descrizione pattern MVP
0.02	2016-03-08	Oscar Elia Conti	Progettista	Stesura sezione tecnologia Android

Versione	Data	Autore	Ruolo	Descrizione
0.01	2016-03-08	Oscar Elia Conti	Progettista	Inizio stesura documento

Indice

Elenco delle figure

1 Introduzione

1.1 Scopo del documento

1.2 Glossario

Allo scopo di rendere più semplice e chiara la comprensione dei documenti viene allegato il *Glossario v1.00* nel quale verranno raccolte le spiegazioni di terminologia tecnica o ambigua, abbreviazioni ed acronimi. Per evidenziare un termine presente in tale documento, esso verrà marcato con il pedice $_g$.

1.3 Riferimenti utili

1.3.1 Riferimenti normativi

- rif

1.3.2 Riferimenti informativi

- Documentazione Android:
  http://developer.android.com/training/index.html;
- Documentazione Java:
  https://www.java.com/it/about/;
- Documentazione SQLite:
  http://developer.android.com/reference/android/database/sqlite/package-summary.html
  https://it.wikipedia.org/wiki/SQLite
  https://www.sqlite.org/about.html;
- Documentazione AltBeacon:
  https://altbeacon.github.io/android-beacon-library/index.html
  https://github.com/AltBeacon/spec;
- Documentazione JGraphT:
  http://jgrapht.org/;
- Documentazione PostgreSQL:
  https://wiki.postgresql.org/wiki/FAQ
  https://it.wikipedia.org/wiki/PostgreSQL.

2 Tecnologie utilizzate

In questa sezione vengono descritte le tecnologie sulle quali si basa lo sviluppo di BlueWhere seguite dai vantaggi e svantaggi riscontrati nel loro uso.

2.1 Android

2.1.1 Descrizione

Android$_g$ è un sistema operativo mobile sviluppato da Google$_g$ e basato su kernel$_g$ Linux$_g$. È stato progettato per essere eseguito principalmente su smartphone$_g$ e tablet$_g$ con interfacce utente specializzate per orologi e televisori. Le versioni di riferimento sono la 4.4 e superiori. L'utilizzo di questa tecnologia è stato richiesto dal proponente.

2.1.2 Vantaggi

- possiede una vasta fetta di mercato mobile;
- disponibile su un vasto numero di dispositivi;
- quasi totalmente gratuito ed Open Source$_g$.

2.1.3 Svantaggi

- essendoci un vasto numero di produttori di smartphone e tablet che non aggiornano la versione di Android che rilasciano all'interno dei loro dispositivi, Android risulta essere estremamente frammentato;
- necessità di sviluppare applicazioni per dispositivi che possono differire per:
  - prestazioni;
  - risoluzione dello schermo;
  - durata della batteria;
  - sensori disponibili.

2.2 Java

Java è uno dei più famosi linguaggi di programmazione orientato agli oggetti supportato da una moltitudine di librerie e documentazione. Viene utilizzato per la scrittura e lo sviluppo dell'applicazione.

2.2.1 Vantaggi

- linguaggio più conosciuto, diffuso e utilizzato nell'ambiente di sviluppo Android
- ampia documentazione disponibile;
- dispone di un gran numero di librerie;
- portabilità su diversi sistemi operativi.

2.2.2 Svantaggi

- linguaggio verboso.

2.3 SQLite

Libreria che implementa un database SQL transazionale senza la necessità di un server. Viene utilizzata per gestire le mappe scaricate e installate nel dispositivo e il relativo contenuto. Il suo utilizzo è stato consigliato dal proponente.

2.3.1 Vantaggi

- database transazionale leggerissimo e molto veloce;
- consigliato per i dispositivi mobile;
- supporta buona parte dello standard SQL92, già conosciuto dai membri del team;
- sistema diffuso e documentato;
- formato del database multipiattaforma;

2.3.2 Svantaggi

- non gestisce da sé la concorrenza;
- alcune funzionalità SQL sono limitate.

2.4 AltBeacon

Libreria che permette ai sistemi operativi mobile di interfacciarsi ai Beacon, offrendo molteplici funzionalità. Viene utilizzata per permettere la comunicazione tra l'applicativo Android e i Beacon. Il suo utilizzo è stato consigliato dal proponente.

2.4.1 Vantaggi

- pieno supporto Android;
- supporta le tipologie di beacon più popolari attualmente in commercio;
- supporta dispositivi Android con versione 4.3 o superiore;
- si propone come nuovo standard open source.

2.4.2 Svantaggi

- poca documentazione disponibile.

2.5 JGraphT

JGraphT è una libreria Java che fornisce funzionalità matematiche per modellare grafi. Viene utilizzata per la rappresentazione e l'uso delle mappe.

2.5.1 Vantaggi

- progettata per essere semplice e type-safe;
- fornisce la possibilità di visualizzare i grafi attraverso JGraph;
- ben documentata;

2.6 PostgreSQL

PostgreSQL è un DBMS ad oggetti open source. Viene utilizzato per la gestione del database da cui scaricare le mappe attraverso l'applicazione. Il suo utilizzo è stato consigliato dal proponente.

2.6.1 Vantaggi

- molto più robusto di MySQL, più stabile e performante;
- gestisce la conversione delle informazioni dal mondo SQL a quello della programmazione orientata agli oggetti.

2.6.2 Svantaggi

- più complesso rispetto al classico MySQL.

3 Descrizione dell'architettura

3.1 Metodo e formalismo di specifica

3.2 Architettura generale

4 Componenti e classi

4.1 **NomeApplicazione**

Namespace globale dell'applicazione. Le relazioni tra i package Model, View e Presenter rappresentano le relazioni tipiche del design pattern MVP.

4.1.1 Package contenuti

- **NomeApplicazione**::Model;
- **NomeApplicazione**::View;
- **NomeApplicazione**::Presenter.

4.2 **NomeApplicazione**::Model

4.2.1 Struttura del package

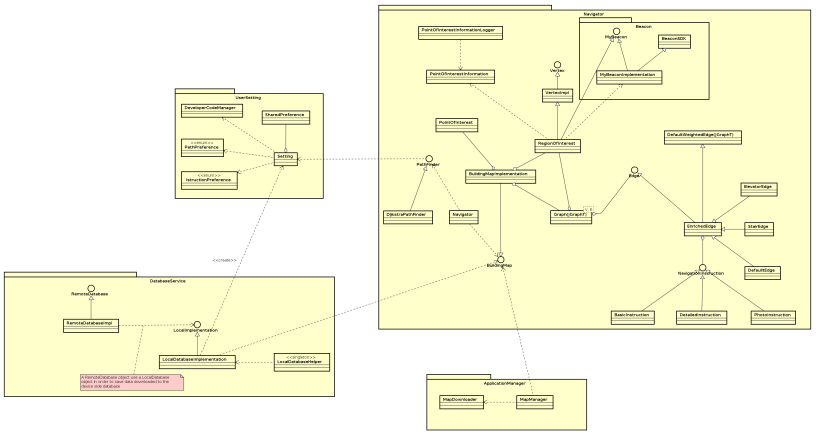

Figura 1: Struttura del pacchetto Model

4.2.2 Descrizione

Package che contiene tutte le classi e package appartenenti al Model dell'applicazione.

4.2.3 Package contenuti

- **NomeApplicazione**::Model::UserSetting;
- **NomeApplicazione**::Model::Navigator;
- **NomeApplicazione**::Model::Beacon;
- **NomeApplicazione**::Model...

4.2.4 Descrizione

4.2.5 Package Contenuti

- **NomeApplicazione**::Model::Beacon
- **NomeApplicazione**::Model::Setting
- **NomeApplicazione**::Model::DatabaseServices

4.3 **NomeApplicazione**::Model::UserSetting

4.3.1 Struttura del package

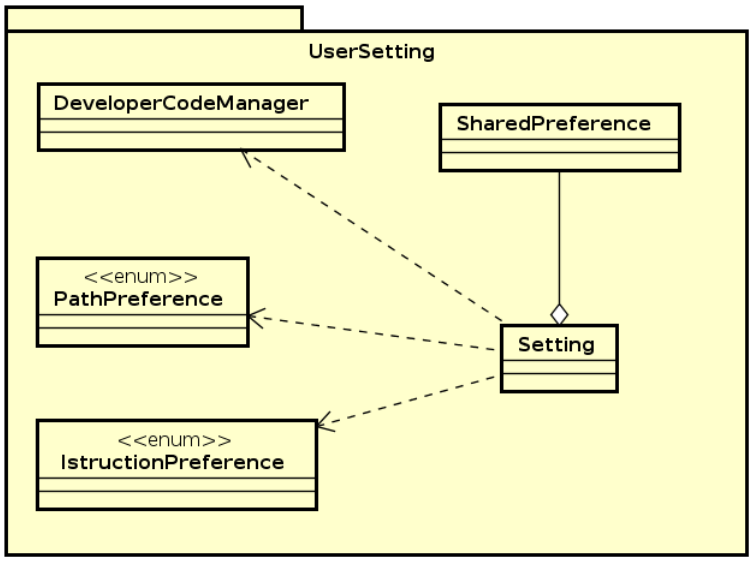

Figura 2: Struttura del pacchetto UserSetting

4.3.2 Descrizione

Componente del Model per la gestione delle preferenze riguardanti il percorso, le impostazioni di fruizione delle informazioni di navigazione e per gestire se un utente può accedere alle funzioni di sviluppatore oppure no.

4.3.3 Classi

4.3.3.1 **NomeApplicazione::Model::UserSetting::Setting** Classe per accedere alle impostazioni di un utente.
Viene utilizzata per mantenere un riferimento in memoria e accedere a queste informazioni quando serve(per esempio quando viene effettuato il calcolo del percorso). Le preferenze vengono salvate tramite la classe "SharedPreference" di Android.

4.3.3.2 **NomeApplicazione::Model::UserSetting::PathPreference**
Classe che definisce le preferenze impostabili riguardanti il percorso preferito dall'utente.

È un semplice enumeratore utile solamente per definire tutti e soli i valori che possono essere utilizzati in **NomeApplicazione**::Model::UserSetting::Setting.

4.3.3.3 **NomeApplicazione**::Model::UserSetting::IstructionPreference

Classe che definisce le preferenze impostabili riguardanti la fruizione delle istruzioni di navigazione.
È un semplice enumeratore utile solamente per definire tutti e soli i valori che possono essere utilizzati in **NomeApplicazione**::Model::UserSetting::Setting.

4.3.3.4 **NomeApplicazione**::Model::UserSetting::DeveloperCodeManager

Classe statica per la verifica dei codici sviluppatore.
Questa classe esporrà un metodo pubblico che ricevuta una stringa ritornerà un booleano che specificherà se il codice passato come parametro è valido oppure no. In prima battuta possiamo fare questo controllo hard-coded, con dei raffinamenti invece possiamo mettere sul server i codici sviluppatore.

4.4 **NomeApplicazione**::Model::Beacon

4.4.1 Struttura del package

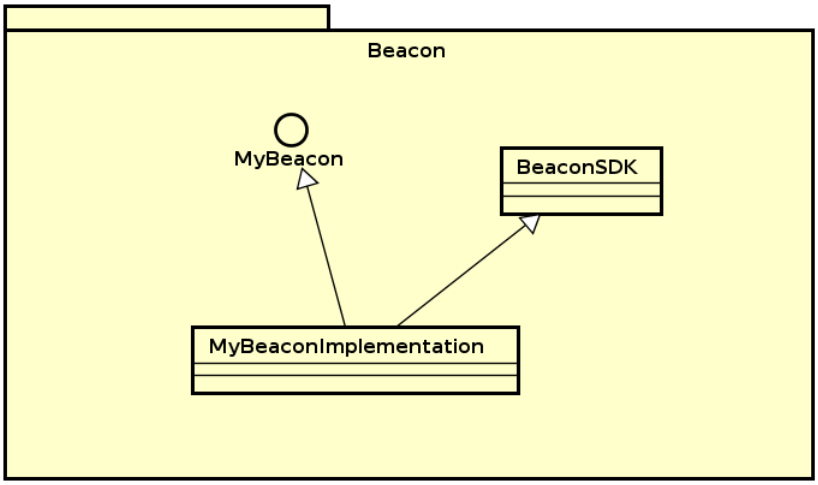

Figura 3: Struttura del pacchetto Beacon

4.4.2 Descrizione

Componente del Model per la gestione dei beacon.

4.4.3 Interfacce

4.4.3.1 **NomeApplicazione::Model::Beacon::MyBeacon** Interfaccia per accedere alle informazioni di un beacon.
Viene utilizzata per specificare il contratto delle classi che la implementano, definendo i metodi che devono essere implementati. Praticamente serve per offrire un livello di astrazione al posto di avere direttamente una classe concreta e quindi poter creare un "adapter" per il beacon della proposto da Altbeacon.

4.4.4 Classi

4.4.4.1 **NomeApplicazione::Model::Beacon::MyBeaconImp** Classe che implementa l'interfaccia **NomeApplicazione**::Model::Beacon::MyBeacon. Questa classe rappresenta un beacon e permette di accedere alle sue informazioni tramite i metodi definiti nell'interfaccia da cui deriva.

4.5 **********Manca la parte di rilevamento dei beacon**********

4.6 **NomeApplicazione**::Model::Navigator

4.6.1 Struttura del package Navigator

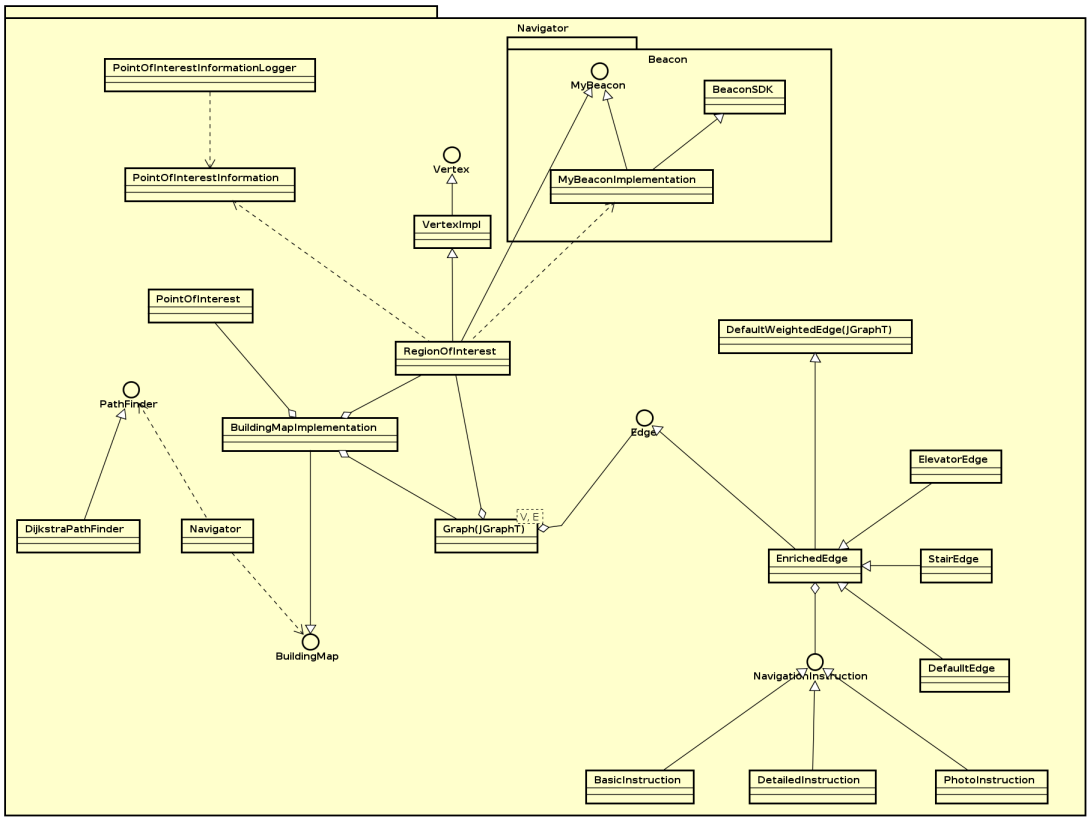

Figura 4: Struttura del pacchetto Navigator

4.6.2 Descrizione

Componente del Model che si occupa della parte di navigazione.
Questo pacchetto ha il compito di calcolare il percorso che l'utente deve seguire ed offrire le classi che permettono di rappresentare la mappa.

4.6.3 Interfacce

4.6.3.1 **NomeApplicazione::Model::Navigator::Vertex** Interfaccia che rappresenta un vertice di un grafo.
Viene utilizzata per dare un livello di astrazione molto alto rispetto a ciò che effettivamente utilizzeremo durante la navigazione. Definisce il contratto di

metodi molto semplici come metodi per accedere all'identificativo di un certo vertice.

4.6.3.2 **NomeApplicazione::Model::Navigator::Edge** Interfaccia che rappresenta un arco pesato di un grafo.
Viene utilizzata per dare un livello di astrazione molto alto rispetto a ciò che effettivamente utilizzeremo durante la navigazione. Definisce il contratto di metodi molto semplici come metodi per accedere al peso dell'arco, ai nodi iniziali e finali dell'arco, impostare i nodi dell'arco.

4.6.3.3 **NomeApplicazione::Model::Navigator::PathFinder** Interfaccia che espone il contratto della classe che calcola il percorso.
Questa interfaccia viene utilizzata per definire la firma dei metodi per gli algoritmi che possono calcolare il percorso definendo uno "Strategy" (Non ho idea se in modo corretto).

4.6.3.4 **NomeApplicazione::Model::Navigator::BuildingMap** Interfaccia che rappresenta la mappa di un edificio.
Viene utilizzata per definire il contratto che le classi devono seguire per poter rappresentare una mappa utilizzabile dal nostro Model.

4.6.3.5 **NomeApplicazione::Model::Navigator::NavigationInstruction** Interfaccia che rappresenta le informazioni di navigazione.

4.6.4 Classi

4.6.4.1 **NomeApplicazione::Model::Navigator::VertexImp** Classe che implementa l'interfaccia **NomeApplicazione**::Model::Navigator::Vertex.
Questa classe rappresenta un vertice e permette di accedere alle sue informazioni tramite i metodi definiti nell'interfaccia da cui deriva.

4.6.4.2 **NomeApplicazione::Model::Navigator::RegionOfInterest** Classe che estende **NomeApplicazione**::Model::Navigator::VertexImp e **NomeApplicazione**::Model::Beacon::MyBeaconImp.
Questa classe rappresenta un "punto di navigazione" nel grafo rappresentante l'edificio, ovvero rappresenta un beacon come un vertice nel grafo dell'edificio.

4.6.4.3 **NomeApplicazione**::Model::Navigator::EnrichedEdge

Classe che implementa l'interfaccia **NomeApplicazione**::Model::Navigator::Edge e deriva dalla classe org.jgrapht.graph.DefaultWeightedEdge della libreria JGraphT.
Viene utilizzata per la navigazione. Infatti questo tipo di arco contiene informazione sia sul peso dell'arco(utile per il calcolo del percorso), sia l'informazioni di attraversamento(informazioni di base, descrizione lunga, url delle photo).

4.6.4.4 **NomeApplicazione**::Model::Navigator::ElevatorEdge

Classe che estende la classe **NomeApplicazione**::Model::Navigator::EnrichedEdge.
Rappresenta un arco che prevede un ascensore e viene utilizzata per ridefinire il peso utilizzato per calcolare il percorso sulla base delle preferenze dell'utente in base agli ascensori.

4.6.4.5 **NomeApplicazione**::Model::Navigator::StairEdge

Classe che estende la classe **NomeApplicazione**::Model::Navigator::EnrichedEdge.
Rappresenta un arco che prevede delle scale e viene utilizzata per ridefinire il peso utilizzato per calcolare il percorso sulla base delle preferenze dell'utente in base alle scale.

4.6.4.6 **NomeApplicazione**::Model::Navigator::DefaultEdge

Classe che estende la classe **NomeApplicazione**::Model::Navigator::EnrichedEdge.
Rappresenta un arco che non contiene particolari ostacoli e viene utilizzata per ridefinire il peso utilizzato per calcolare il percorso nel caso in cui l'utente non imposti alcuna preferenza di percorso.

4.6.4.7 **NomeApplicazione**::Model::Navigator::BasicInstruction

Classe che implementa l'interfaccia **NomeApplicazione**::Model::Navigator::NavigationInstruct
Rappresenta le indicazioni di base che possono guidare un utente (Navigazione di primo livello).

4.6.4.8 **NomeApplicazione**::Model::Navigator::DetailedInstruction

Classe che implementa l'interfaccia **NomeApplicazione**::Model::Navigator::NavigationInstruct
Rappresenta delle informazioni aggiuntive che possono essere fornite ad un utente (descrizione lunga).

4.6.4.9 **NomeApplicazione**::Model::Navigator::PhotoInstruction

Classe che implementa l'interfaccia **NomeApplicazione**::Model::Navigator::NavigationInstruct

Rappresenta delle informazioni visuali del prossimo punto da raggiungere per procedere con la navigazione (fotografie).

4.6.4.10 **NomeApplicazione**::Model::Navigator::PointOfInterest

Classe che rappresenta un POI ovvero un area dell'edificio che può risultare interessante ad un utente a livello sia informativo che di navigazione (una possibile destinazione).
Questa classe offre la possibilità di accedere alle informazioni di un POI quali:

- nome del POI (esistente o dato da noi);
- descrizione del POI (qualcosa che almeno ne descriva le funzionalità);

4.6.4.11 **NomeApplicazione**::Model::Navigator::BuildingMapIml

Classe che implementa l'interfaccia **NomeApplicazione**::Model::Navigator::BuildingMapIml. Viene utilizzata per rappresentare la mappa dell'edificio, racchiudento le informazioni dell'edificio stesso. in particolare mantiene la corrispondenza tra RegionOfInterest e PointOfInterest (ad un ROI possono essere associati più POI ed un POI può appartenere a più ROI. Esempio 1C150 della Torre Archimede).

4.6.4.12 **NomeApplicazione**::Model::Navigator::DijktraPathFinder

Classe che implementa l'interfaccia **NomeApplicazione**::Model::Navigator::PathFinder. Viene utilizzata per calcolare il percorso per portare un utente in un certo POI (destinazione della navigazione). Questa classe sfrutta gli algoritmi messi a disposizione dalla libreria JGraphT. Essendo che un POI può essere associato a più ROI e la navigazione è possibile solamente tramite i ROI (punti in cui abbiamo i beacon e quindi possiamo controllare se l'utente va nel percorso giusto oppure no) allora sarà necessario calcolare un percorso per ogni ROI associato ad un POI e successivamente scegliere quello con peso minore (in base al percorso stesso e alle preferenze dell'utente). Utilizzando l'algritmo di Dijkstra non sarebbe necessario in generale ma l'implementazione fornita da JGraphT richiede un ricalcolo.

4.6.4.13 **NomeApplicazione**::Model::Navigator::Navigator

Classe che si occupa di gestire la navigazione.
Viene utilizzata per controllare se l'utente segue le indicazioni fornite per la navigazione e gestire in generale i percorsi restituiti da Dijkstra (liste di archi, nel nostro caso possibilmente sarebbe meglio EnrichedEdge).

4.7 Dubbi & ToDo

- manca la parte del model riguardante il rilevamento dei beacon;
- manca la parte del model riguardante database e server;
- manca la parte del model riguardante le fotografie e la loro gestione;
- verifica della parte del model riguardante la gestione del codice sviluppatore;
- manca la parte del presenter;
- manca la parte della view;
- da valutare se il package Beacon è interno o esterno al package Navigator;
- da valutare se le informazioni dell'edificio sono una classe separata da BuildingMapIml;
- da valutare la gestione dei valori dei beacon che cambiano velocemente.

5 Schema della base di dati

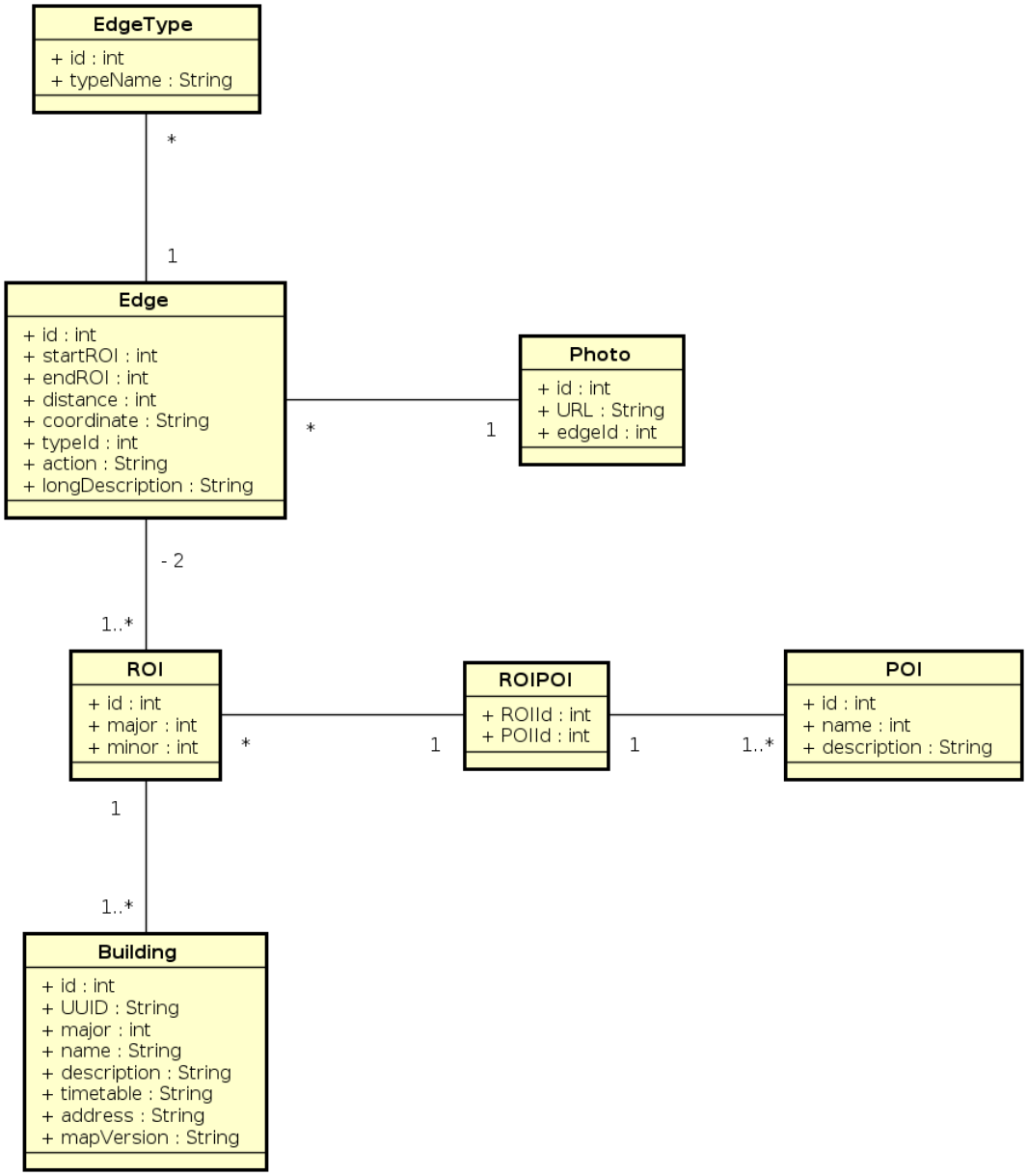

Figura 5: Struttura del database

5.1 Descrizione

Il database proposto serve per salvare mappe e informazioni associate ai luoghi in esse contenute.

5.2 Entità

5.2.1 Building

Rappresenta un edificio di cui è presente la mappa. I suoi dati sono:

- **id**: chiave primaria, autoincrementale, intero che identifica univocamente un edificio;
- **UUID**: stringa che rappresenta l'UUID dei beacon utilizzati dal nostro applicativo;
- **major**: intero che distigue i beacon della nostra applicazione appartenenti ad un edificio rispetto a quelli presenti in un altro edificio;
- **name**: stringa che rappresenta il nome dell'edificio;
- **description**: stringa che rappresenta la descrizione dell'edificio(funzione e quialcos'altro);
- **timetable**: stringa che rappresenta gli orari di un edificio;
- **address**: stringa che rappresenta l'indirizzo di un edificio;
- **mapVersion**: stringa che rappresenta la versione della mappa.

5.2.2 ROI

Rappresenta una ROI all'interno di un edificio. I suoi dati sono:

- **id**: chiave primaria, autoincrementale, intero che identifica univocamente una ROI;
- **major**: intero che distigue i beacon della nostra applicazione appartenenti ad un edificio rispetto a quelli presenti in un altro edificio;
- **minor**: intero che rappresenta univocamente un beacon della nostra applicazione in un edificio.

5.2.3 ROIPOI

Rappresenta l'associazione tra ROI e POI. I suoi dati sono:

- **idROI**: chiave esterna, intero che identifica univocamente una ROI;
- **idPOI**: chiave esterna, intero che identifica univocamente un POI;

5.2.4 POI

Rappresenta una POI all'interno di un edificio. I suoi dati sono:

- **id**: chiave primaria, autoincrementale, intero che identifica univocamente un POI;
- **name**: stringa che identifica in qualche modo un POI;
- **description**: string a che spiega le funzionalità fi un POI ed altro.

5.2.5 Edge

Rappresenta un collegamento tra due ROI. I suoi dati sono:

- **id**: chiave primaria, autoincrementale, intero che identifica univocamente un POI;
- **startROI**: chiave esterna, intero che identifica univocamente il ROI di partenza dell'edge;
- **endROI**: chiave esterna, intero che identifica univocamente il ROI di arrivo dell'edge;
- **coordinate**: ?intero? ?identifica la rotazione dell'utente per procedere secondo il persorso ideale?;
- **distance**: intero che identifica la distanza tra startROI e ednROI;
- **typeId**: chiave esterna, intero che identifica il tipo di arco;
- **action**: stringa che identifica qualche azione da compiere per attraversare l'arco;
- **longDescription**: stringa che fornisce le le indicazioni dettagliate di attraversamento di un arco.

5.2.6 Photo

Rappresenta una fotografia di utile per attraversare un edge. I suoi dati sono:

- **id**: chiave primaria, autoincrementale, intero che identifica univocamente una fotografia;
- **URL**: stringa che rappresenta l'URL al quale è possibile recuperare la fotografia;
- **edgeId**: chiave esterna, intero che identifica univocamente l'edge a cui la fotografia è associata.

5.2.7 EdgeType

Rappresenta un tipo di edge. I suoi dati sono:

- **id**: chiave primaria, autoincrementale, intero che identifica univocamente un tipo di edge;
- **typeName**: stringa che descrive il tipo di edge.

5.3 Relazioni

5.3.1 Building - ROI

- **Tipo di relazione**: 1 a N;
- ogni Building può essere associato a 1 o più ROI;
- ogni ROI è associato ad un unico Building.

5.3.2 ROI - ROIPOI

- **Tipo di relazione**: 1 a N (risoluzione della relazione N a N tra ROI e POI);
- ogni ROI può essere associato a 0 o più ROIPOI;
- ogni ROIPOI è associato ad un unico ROI.

5.3.3 ROIPOI - POI

- **Tipo di relazione**: 1 a N (risoluzione della relazione N a N tra ROI e POI);
- ogni POI può essere associato a 1 o più ROIPOI;
- ogni ROIPOI è associato ad un unico POI.

5.3.4 ROI - Edge

- **Tipo di relazione**: 2 a N;
- ogni ROI può essere associato a 1 o più Edge;
- ogni Edge è associato a 2 ROI.

5.3.5 Edge - Photo

- **Tipo di relazione**: 1 a N;
- ogni Photo può essere associata ad un unico Edge;
- ogni Edge può essere associato a 1 o più Photo.

5.3.6 Edge - EdgeType

- **Tipo di relazione**: 1 a N;
- ogni Edge può essere associata ad un unico EdgeType;
- ogni EdgeType può essere associato a 0 o più Edge.

5.4 SQL

```
CREATE TABLE IF NOT EXISTS Building{
        id PRIMARY KEY INT,
        UUID VARCHAR(255),
        major INT,
        name VARCHAR(255),
        description VARCHAR(2000),
        timetable VARCHAR(255),
        address VARCHAR(255),
        mapVersion VARCHAR(255)
```

```
}
CREATE TABLE IF NOT EXISTS ROI{
        id PRIMARY KEY INT,
        major INT REFERENCES Building(major),
        minor INT UNIQUE
}
CREATE TABLE IF NOT EXISTS ROIPOI{
        idROI INT REFERENCES ROI(id),
        idPOI INT REFERENCES POI(id),
        PRIMARY KEY (idROI, idPOI)
}
CREATE TABLE IF NOT EXISTS POI{
        id PRIMARY KEY INT,
        name VARCHAR(255),
        description VARCHAR(2000)
}
CREATE TABLE IF NOT EXISTS Edge{
        id PRIMARY KEY INT,
        startROI INT REFERENCES ROI(id),
        endROI INT REFERENCES ROI(id),
        distance INT,
        coordinate INT,
        typeId INT REFERENCES EdgeType(id),
        action VARCHAR(255),
        longDescription VARCHAR(2000)
}
CREATE TABLE IF NOT EXISTS Photo{
        id PRIMARY KEY INT,
        URL VARCHAR(2048),
        edgeId INT REFERENCES Edge(id)
}
CREATE TABLE IF NOT EXISTS EdgeType{
        id PRIMARY KEY INT,
        typeName VARCHAR(255)
}
```

6 Diagrammi delle attività

7 Design pattern

7.1 Design pattern architetturali

7.2 Design pattern creazionali

7.3 Design pattern strutturali

7.4 Design pattern comportamentali

8 Stime di fattibilità e bisogno di risorse

9 Tracciamento

A Descrizione design pattern

A.1 Design pattern architetturali

A.1.1 MVP

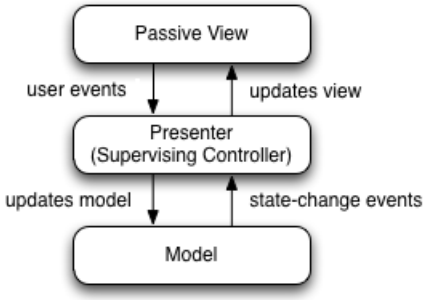

Figura 6: Struttura del pattern MVP

Model-View-Presenter (MVP) è un pattern architetturale derivato dal MVC (Model-View-Controller), utilizzato per dividere il codice in funzionalità distinte. Il suo principale ambito di utilizzo è nelle applicazioni un insieme di informazioni deve essere rappresentato mediante un'interfaccia grafica.

A.1.1.1 Componenti MVC è basato sul principio di disaccoppiamento di tre oggetti distinti, riducendo in questo modo le dipendenze reciproche; inoltre permette di fornire una maggiore modularità, manutenibilità e robustezza al software.

A.1.1.1.1 Model Il Model rappresenta il cuore dell'applicazione: esso definisce il modello dei dati definendo gli oggetti secondo la logica di utilizzo dell'applicazione, ossia la sua business logic. Inoltre, indica le possibili operazioni che si possono effettuare sui dati.

A.1.1.1.2 View Nel pattern MVP, il Model è un componente prevalentemente passivo, ma si occupa anche di notificare al Presenter eventuali modifiche del proprio stato. Nella struttura del pattern MVP, la View si occupa di prendere gli input dell'utente e passarli al Controller, affinché esegua operazioni sul Model.

A.1.1.1.3 Presenter Il Presenter è l'intermediario tra il Model e la View. Si occupa di implementare l'insieme di operazioni eseguibili sul modello dei dati attraverso una particolare vista, ossia l'application logic. Ad ogni View, deve corrispondere un diverso Controller.

A.1.1.2 Vantaggi Elenco vantaggi.

A.1.1.3 Svantaggi Elenco svantaggi.

A.1.2 Dependency injection

Dependency injection è un pattern architetturale utilizzato nella programmazione object-oriented al fine di separare il comportamento di una componente dalla risoluzione delle sue dipendenze. Di conseguenza, il pattern su basa su tre elementi:

- un componente dipendente;
- la dichiarazione delle dipendenze della componente;
- un injector, che crea su richiesta le istanze delle classi che implementano le dipendenze.

Il principio su cui si basa è l'inversione di controllo, secondo il quale il ciclo di vita degli oggetti viene gestito da un'entità esterna, detta container. Nella dependency injection implementata con inversione di controllo, le dipendenze vengono inserite nel container, mentre la componente si limita a dichiararle. In questo modo si limita la dipendenza fra classi. Esistono due tipi di dependency injection:

constructor injection: le dipendenze vengono dichiarate come parametro del costruttore. In questo modo un oggetto è valido appena viene istanziato;

setter injection: le dipendenze vengono dichiarate come metodi setter. In questo modo vengono evidenziate le dipendenze.

A.1.2.1 Vantaggi Elenco vantaggi.

A.1.2.2 Svantaggi Elenco svantaggi.

A.2 Design pattern creazionali

A.2.1 Singleton

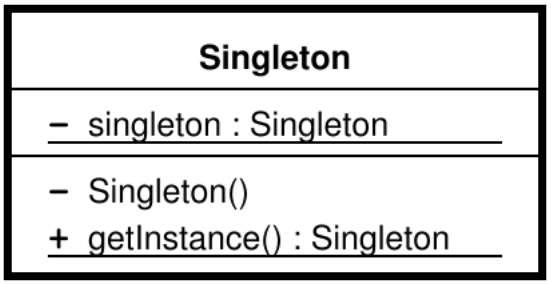

Figura 7: Struttura del pattern Singleton

Lo scopo del pattern Singleton è assicurare l'esistenza di un'unica istanza di una classe fornire un punto di accesso globale ad essa. Questo pattern è nate per rispondere alla necessità di non avere più istanza della stessa glasse, pur dando la possibilità alla classe di tener traccia di quella sua istanza. Il pattern Singleton è applicabile ogni volta in cui debba esistere una sola istanza di una determinata classe in tutta l'applicazione, prestando attenzione al fatto che l'istanza sia estendibile tramite ereditarietà.

A.2.1.1 Vantaggi

Elenco vantaggi.

A.2.1.2 Svantaggi

Elenco svantaggi.

A.2.2 Strategy

Lo scopo del pattern comportamentale Strategy è definire una famiglia di algoritmi, incapsularli e renderli intercambiabili, in modo da poter variare indipendentemente dal client che ne fa uso. Questo pattern soddisfa la necessità di poter applicare diversi algoritmi al medesimo problema senza dover modificare codice già scritto, aumentandone l'estendibilità e la riusabilità.

A.2.2.1 Vantaggi

Elenco vantaggi.

A.2.2.2 Svantaggi

Elenco svantaggi.

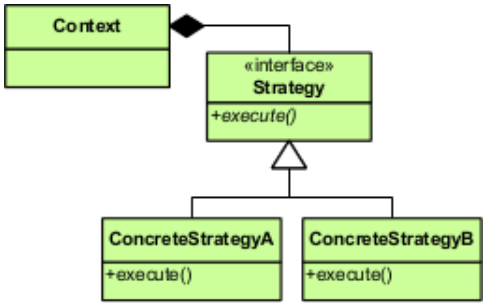

Figura 8: Struttura del pattern Strategy

A.3 Design pattern strutturali

A.3.1 Facade

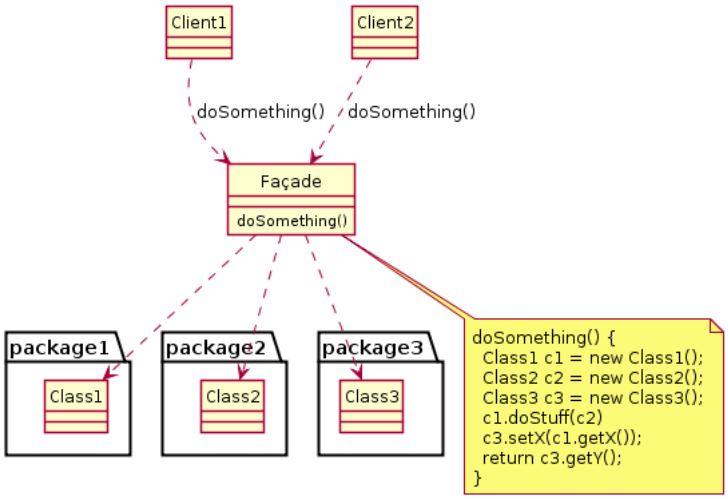

Figura 9: Struttura del pattern Facade

Lo scopo del pattern strutturale Facade è di fornire un'interfaccia unificata per un insieme di interfacce presenti in un sottosistema, definendo un'interfaccia di livello più alto che rende il sottosistema più semplice da utilizzare. Suddividendo un sistema in sottosistemi, si aiuta a ridurne la complessità e si minimizzano le comunicazioni e le dipendenze fra i diversi sottosistemi.

A.3.1.1 Vantaggi

Elenco vantaggi.

A.3.1.2 Svantaggi

Elenco svantaggi.

A.4 Design pattern comportamentali

A.4.1 Observer

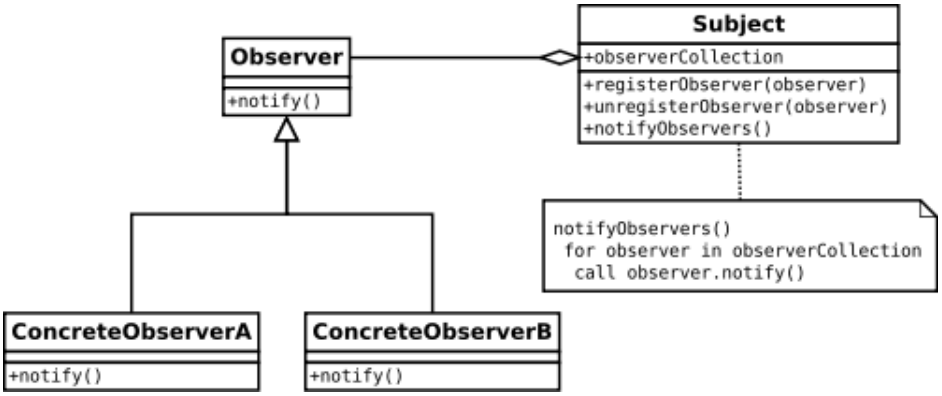

Figura 10: Struttura del pattern Observer

Observer è un pattern comportamentale il cui scopo è quello di tenere sotto controllo lo stato di diversi oggetti legati ad un soggetto (detta anche **dipendenza uno a molti**). Il paradigma su cui si basa corrisponde al modello **Publisher and Subscribe**: i sottoscrittori si registrano presso un pubblicatore e quest'ultimo li informa ogni volta che ci sono nuove notizie. Il pattern è composto da:

- una classe astratta **Subject** da cui eredita il soggetto concreto, che mantiene una lista di riferimenti agli oggetti dipendenti per poterli avvisare;
- un'interfaccia **Observer** implementata dagli osservatori concreti, che tengono il riferimento al soggetto per poterne leggero lo stato.

A.4.1.1 Vantaggi

Elenco vantaggi.

A.4.1.2 Svantaggi

Elenco svantaggi.

B Mockup dell'interfaccia grafica