CLIPS

Communication & Localization with Indoor Positioning Systems

Università di Padova

Specifica Tecnica v0.02

leaf.gruppo@gmail.com

Versione	
Data Redazione	
Redazione	
Verifica	
Approvazione	
Uso	
Distribuzione	

Diario delle modifiche

Versione	Data	Autore	Ruolo	Descrizione
0.02	2016-03-08	Oscar Elia Conti	Progettista	Stesura sezione tecnologia Android
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

- rif

2 Tecnologie utilizzate

In questa sezione vengono descritte le tecnologie sulle quali si basa lo sviluppo di BlueWhere.

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

I principali vantaggi del sistema operativo Android sono:

- 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

I principali svantaggi del sistema operativo Android sono:

- 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.

3 Descrizione dell'architettura

3.1 Metodo e formalismo di specifica

3.2 Architettura generale

4 Componenti e classi

5 Schema della base di dati

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

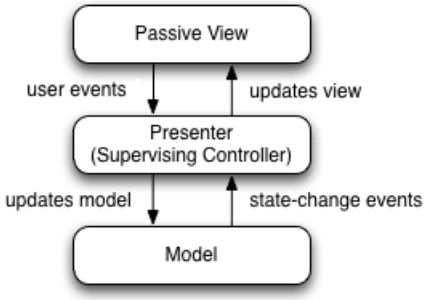

Figura 1: Struttura del pattern MVP

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

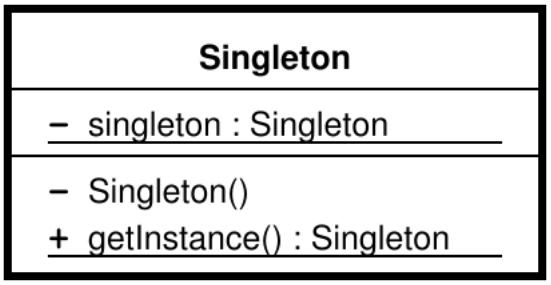

Figura 2: Struttura del pattern Singleton

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.

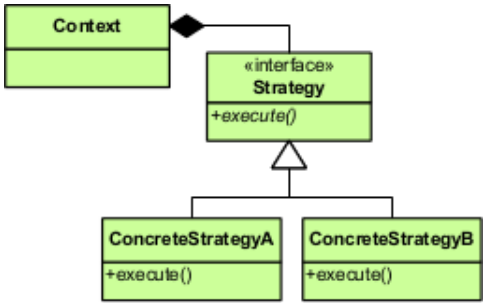

Figura 3: Struttura del pattern Strategy

A.3 Design pattern strutturali

A.3.1 Facade

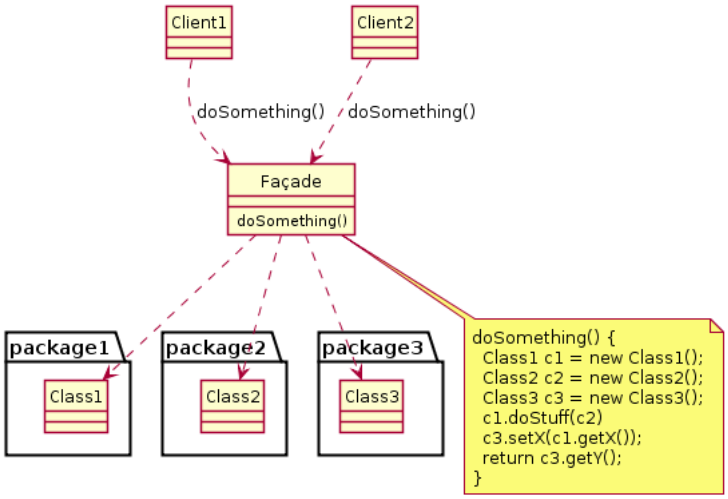

Figura 4: Struttura del pattern Facade

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

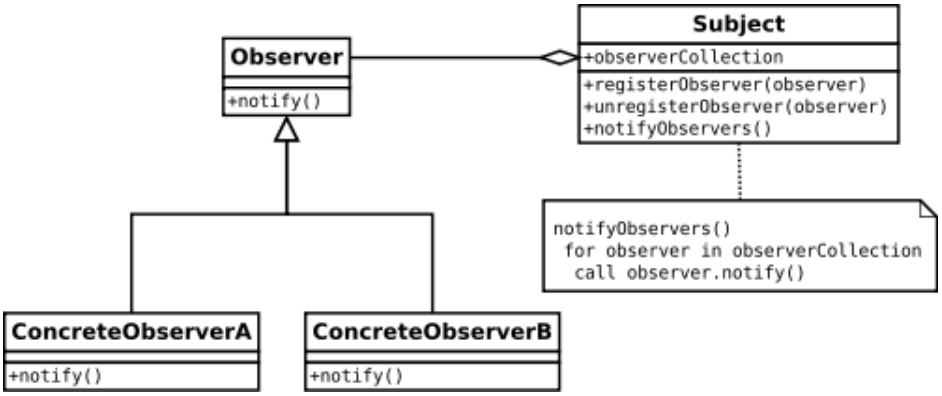

Figura 5: Struttura del pattern Observer

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