# Introduzione

Una volta redatti i documenti relativi all’analisi dei requisiti e all’identificazione degli obiettivi di design passiamo alla realizzazione di un ulteriore documento, L’Object Design Document.

Tale documento integrerà tutte le informazioni già evidenziate nei precedenti documenti in modo chiaro e preciso, servirà come base per l’implementazione e conterrà:

* Le specifiche legate alle interfacce per ogni sottosistema;
* Le librerie per le classi;
* I design pattern al fine di risolvere problemi e proteggere le classi da futuri cambiamenti.

## Object Design Trade-offs

In questa sezione verranno definiti i trade-offs relativi al sistema “Youth Club”

* **Prestazioni vs. Costi,** considerando il budget a disposizione, è stato di utilizzare componenti open source, con l’obiettivo di minimizzare i costi in rapporto alla qualità delle funzione rilasciate;
* **Interfaccia vs Tempo di risposta,** per garantire tempi di risposta sufficientemente brevi è stato deciso di focalizzare l’attenzione sulla realizzazione di una parte di back-end il più lineare possibile in modo da ridurre i tempi di risposta del sistema;
* **Interfaccia vs. Usabilità,** grazie all’utilizzo delle componenti scelte si garantisce un design gradevole e facilmente comprensibile; e conseguentemente un’interfaccia facilmente usabile;
* **Costi vs. Mantenimento,** grazie all’utilizzo delle componenti e i linguaggi di implementazione scelti, il sistema risulterà essere facilmente mantenibile a costi ridotti.

## Linee guida per l’implementazione

Le linee guide per lo sviluppo del progetto software sono dettate da “ ” documento web contenente regole ben precise sullo standard da seguire durante l’implementazione.

Linee guida:

le linee guida sono molto importanti in quanto vanno a stabilire i Naming Convention dei componenti utilizzati.

**Nome variabili:**

I nomi delle variabili devono essere brevi e significativi. L’identificatore di una variabile è una sequenza di lettere e cifre il cui primo elemento deve essere una lettera. I nomi delle variabili dovranno iniziare con una lettere minuscola e qualora formati da più parole concatenati tutte le parole successive alla prima dovranno essere capitalizzate.(Ex. nomeVariabile)

Le variabili locali saranno dichiarate all’interno dei metodi, quindi saranno create solo quando il metodo verrà invocato e cancellate alla terminazione del metodo.

*void nomeMetodo(){*

*int nomeVariabile=valore;*

*….*

*})*

Le variabili devono essere inizializzate prima del loro utilizzo.

Nelle variabili costanti dove il valore assunto dalle variabili non può essere modificato è possibile

utilizzare “\_”.

(Ex NOME\_VARIABILE)

Le variabili di istanza saranno dichiarate esternamente alla dichiarazione di metodi.

*Class NomeClasse{*

*int nomeVariabile=valore;*

*void nomeMetodo(){..}*

*}*

**Nome metodi:**

I metodi definiscono i comportamenti degli oggetti.

I nome dei metodi inizieranno con la *lettera minuscola* e, qualora formati da più parole le successive dovranno essere capitalizzate secondo il CamelCase (“nomeMetodo()”).

Il nome di un metodo va a specificare l’azione che andrà ad eseguire, quindi sarà presente almeno un verbo all’interno di esso.

Ai metodi viene aggiunto un commento in JavaDoc, il quale verrà posizionato prima della dichiarazione del metodo e serve a descriverne lo scopo.

Nel commento sono anche presenti delle informazioni riguardanti i paramenti, il valore di ritorno e le possibili eccezioni che potrebbero essere lanciate.

**Nome classi:**

I nomi delle classi dovranno iniziare con la lettera maiuscola, anche la prima lettera delle parole

concatenate dovrà essere in maiuscolo.

(EX NomeClasse)

Il nome della classe deve essere intuitivo e comprensibile.

Ogni file sorgente(.class) ha al suo interno una sola classe l’introduzione alla classe che conterrà lo scopo di quest’ultima.

/\*

\*scopo della classe

\*/

La classe ha una struttura da seguire: dichiarazione classe pubblica,dichiarazione delle costanti,dichiarazione delle variabili di classe e d’istanza,costruttore ,metodi e i relativi commenti



**Nomi package:**

I nome dei pacchetti dovranno essere scritti in minuscolo concatenando un insieme di sostantivi separati dal punto.

In generale, un nome comincia con il dominio di primo livello dell’organizzazione che lo produce, seguito dal dominio e da altri eventuali sottodomini, elencati in ordine inverso.

L’organizzazione può infine scegliere un nome specifico per quel particolare package.

Non sono ammessi caratteri speciali.

## Definizioni, acronomi e abbreviazioni

**Acronomi:**

RAD: Requirements Analysis Document

SDD: System Design Document  
ODD: Object Design Document  
DAO: Data Access Object

# Riferimeti

Documento YouthClub\_RAD\_V\_2.0.pdf

Documento YouthClub\_SDD\_V\_3.0.pdf

**COTS (Components Off The Shelf)**

In questa sezione definiamo le Component Off The Shelf, ovvero le componenti hardware e software disponibili sul mercato che andremo ad utilizzare nel nostro progetto software:

* JDBC: connettore (driver) per database che consente l’accesso e la gestione della

persistenza dei dati sulle basi di dati da qualsiasi programma scritto con il linguaggio di

programmazione Java, indipendentemente dal tipo di DBMS utilizzato.

Design Pattern

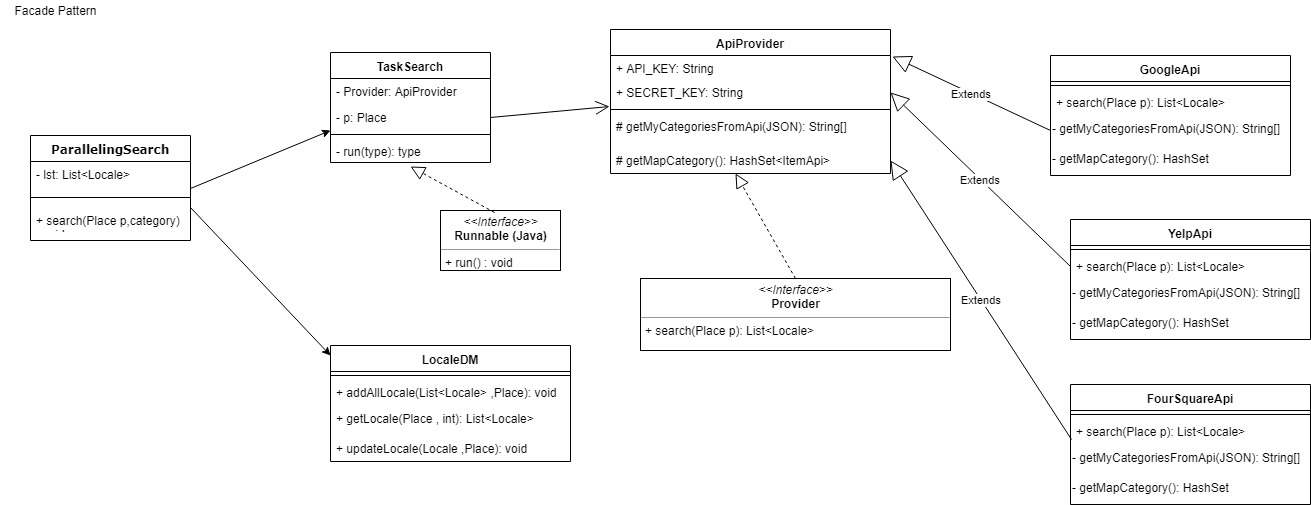
2.1 Facade Design Pattern

Il Sistema di ricerca sfrutta più sottosistemi quali model e api ma ci si interfaccia solo tramite la classe **ParallelingSearch ,** nascondendo il tutto.La classe si occupa di valutare se deve accedere al database o richiedere i dati ai servizi esterni, questo pattern è noto con il nome di **Facade Pattern**.

Il design pattern architetturale Facade fornisce un’unica interfaccia per accedere ad un insieme di oggetti che compongono un sottosistema.

L’applicazione di questo design pattern comporta vari vantaggi:

1. Nascondere la complessità dei servizi offerti dal sottosistema, fornendo un’architettura chiusa;
2. Rendere indipendenti l'implementazione di chi fa uso dei servizi del sottosistema dall'implementazione delle componenti;
3. Rendere il sottosistema più facile da usare, astraendone i dettagli.



2.2 Adapter Design Pattern

Il sottosistema delle api è sviluppato in modo da far comunicare 3 sistemi differenti, trovandone punti in comune sfruttando quindi l’**Adapter Pattern**.

Il fine dell'adapter è di fornire una soluzione astratta al problema dell'interoperabilità tra interfacce differenti. Questo problema si presenta ogni qual volta nel progetto di un software si debbano utilizzare dei sistemi di supporto quali librerie o servizi esterni, possiamo distinguere tra due adapter pattern: *Class adapter*, che usa ereditarietà multipla per adattare un’interfaccia ad un’altra e *Object adapter*, che usa ereditarietà singola e delegazione.

Nel caso della classe ApiProvider sfrutta l’Object adapater (Inserire immagine)

2.3 DAO Design Pattern

Il design pattern architetturale DAO (Data Access Object) permette di evitare una comunicazione diretta e strettamente accoppiata con il database prevedendo l’utilizzo di un’interfaccia preposta a tale scopo, consentendo dunque di astrarre i dettagli implementativi dalla persistenza dei dati e favorendo manutenibilità e gestione dei dati.

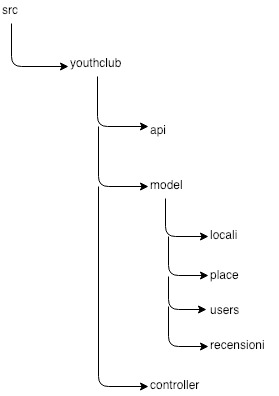
Tale design pattern ha come scopo quello di separare le operazioni di business dalle operazioni di accesso ai dati quali aggiornamento, rimozione e caricamento, che avvengono attraverso un controller che interagisce unicamente con un’interfaccia comune.

L’applicazione di questo design pattern comporta vari vantaggi:

* Facilità di utilizzo;
* Manutenibilità;
* Riutilizzo del codice.

Questo Design Pattern si colloca tra i design pattern usati in ambito di sviluppo web e appartiene alla classe dei pattern architetturali. (Inserire 1 immagine di dao)

3 Packaging



Il package “YouthClub“ conterrà i controller.

Il package “Api” conterrà le classi riguardanti le api e il geocoder.

Il package “Model” conterrà “locali” “place” “users” “recensioni”.

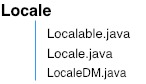
**//MANCA PACKAGE YOUTH CLUB**



|  |  |
| --- | --- |
| **Classe** | **Descrizione** |
| ApiProvider.java | Classe astratta che rappresenta il modello di interazione con le tre Api |
| FourSquare.java | Classe che permette di eseguire l’interrogazione al servizio offerto da FourSquare |
| Geocoding.java | Classe che permette in base ai dati in input di effettuare il geocoding o il reverse geocoding; e gestire i luoghi per l’applicativo |
| GoogleApi.java | Classe che permette di eseguire l’interrogazione al servizio offerto da Google |
| ItemApi.java | Classe di supporto che mette in correlazione le nostre categorie con quelle dei servizi esterni. |
| Provider.java | Interfaccia che implementa i metodi esterni per la ricerca. |
| YelpApi.javael | Classe che permette di eseguire l’interrogazione al servizio offerto da Yelp |



|  |  |
| --- | --- |
| **Classe** | **Descrizione** |
| Categoria.java | Classe che rappresenta l’astrazione delle categorie |
| DriverManagerConnectionPool.java | Classe che gestisce la creazione di una serie di connessioni al database e il mantenimento delle stesse per la realizzazione di una connection pool. |



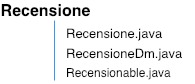
|  |  |
| --- | --- |
| **Classe** | **Descrizione** |
| Localable | Un'interfaccia che astrae il concetto di un locale. |
| Locale | Classe che rappresenta il concetto di locale. |
| LocaleDM | Classe che si occupa di aggiungere, modificare o restituire un locale. |



|  |  |
| --- | --- |
| **Classe** | **Descrizione** |
| Place.java | Classe che rappresenta il concetto di luogo. |
| PlaceDM.java | Classe che si occupa di aggiungere, modificare e controllare se esiste un luogo |
| Placing | Interfaccia che astrae il concetto di luogo |



|  |  |
| --- | --- |
| **Classe** | **Descrizione** |
| Utente.java | Classe che rappresenta il concetto di utente |
| UserDM.java | Classe che crea e restituisce un utente |
| Utentable.java | Interfaccia che astrae il concetto di utente |



|  |  |
| --- | --- |
| **Classe** | **Descrizione** |
| Recensione.java | Classe che rappresenta il concetto di recensione |
| RecensioneDM.java | Classe che aggiunge, modifica e restituisce una recensione |
| Recensionable.java | Interfaccia che astrae il concetto di recensione |