

Object Design Document

Green Leaf

|  |  |
| --- | --- |
| Riferimento |  |
| Versione |  |
| Data |  |
| Destinatario | Prof.ssa F. Ferrucci,  Prof. F. Palomba |
| Presentato da | Alessandro Borrelli,  Vincenzo Cerciello,  Michela Faella,  Gerardo Napolitano,  Mirko Vitale |
| Approvato da |  |

# **Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Data** | **Versione** | **Descrizione** | **Autori** |
| 23/12/2022 | 0.1 | Stesura del punto 1 e 2 | Team |
| 27/12/2022 | 0.2 | Stesura dei rimanenti capitoli | Team |

**Sommario**

[Revision History 2](#_bookmark0)

1. [Introduzione 5](#_bookmark2)
   1. [Object design goals 5](#_bookmark3)
   2. [Linee guida per la documentazione dell’interfaccia 5](#_bookmark4)
   3. [Definizioni, acronimi, e abbreviazioni 5](#_bookmark5)
   4. [Riferimenti 6](#_bookmark6)
2. [Packages 6](#_bookmark7)
3. [Class Interfaces 12](#_bookmark8)
4. [Design Patterns 23](#_bookmark9)
5. [Glossario 27](#_bookmark10)

# **Introduzione**

### Object design trade-offs

Durante la fase di Object Design sono sorti diversi compromessi di progettazione, di seguito sono riportati i trade-off:

## **Prestazioni vs Costi**

Tenuto conto del budget stanziato, e dalla necessità di avere funzioni chiave perfettamente operative nei tempi prestabiliti, si preferisce dedicare il monte ore a disposizione all’implementazione e revisione di quest’ultime.

## **Prestazioni vs Affidabilità**

Dovendo il sistema gestire dati sensibili, si preferisce garantire un maggior controllo di input e consistenza a scapito della latenza.

### Componenti off-the-shelf

### Green Leaf farà affidamento su varie componenti off-the-shelf sia per il front-end che per il back-end. Per quanto riguarda il front-end verrà utilizzato HTML5, linguaggio di mark-up attualmente più diffuso, CSS3, per definire lo stile delle pagine Web, e Bootstrap, raccolta di strumenti per la creazione di applicazioni Web.

### Lato back-end saranno utilizzate le seguenti componenti off-the-shelf: MySQL, Tomcat, Maven. MySQL è un celebre RDBMS di Oracle. Il suo utilizzo sarà fondamentale per la realizzazione, la gestione e l'implementazione della base di dati di GreenLeaf. Tutti i dati che per necessità di business devono essere salvati persistentemente saranno affidati ad un database basato su MySQL. Sarà inoltre utilizzata la componente MySQL Connector/J, il driver JDBC ufficiale per MySQL necessario per la comunicazione con la base di dati da Java.

### Tomcat è un server web open-source sviluppato dalla Apache Software Foundation. Il suo impiego è necessario, poiché tutto il back-end di GreenLeaf (nucleo di elaborazione principale, comunicazione con la base di dati) sarà eseguito in ambiente Tomcat. Esso permetterà di utilizzare le Java Servlet per l'implementazione della logica di business e le Java Servlet Pages per la generazione dinamica delle pagine web utilizzate dal client. Maven è uno strumento di gestione dei progetti software basati su Java, anch'esso sviluppato dalla Apache Software Foundation. Agevolerà la gestione delle varie librerie utilizzate, delle loro versioni e delle dipendenze tra esse, permettendo agli sviluppatori di concentrarsi esclusivamente sulla scrittura del codice.

### 1.3 Linee guida per la documentazione dell’interfaccia

### È richiesto agli sviluppatori di seguire le seguenti linee guida al fine di essere consistenti nell’intero progetto e facilitare la comprensione delle funzionalità di ogni componente.

|  |  |  |
| --- | --- | --- |
| Tipi | Regole per la denominazione | Esempi |
| Package | Il prefisso di un nome di pacchetto univoco sempre scritto in minuscolo e tutte le lettere ASCII minuscole tranne le abbreviazioni come UI, in modo da non rendere il nome del pacchetto lungo | package greenleaf;package pluto; |
| Classi | I nomi delle classi dovrebbero essere sostantivi, scritti in CamelCase. I nomi delle classi semplici e descrittivi, utilizzando parole intere ed evitato per quanto possibile acronimi e abbreviazioni (a meno che l'abbreviazione non sia molto più utilizzata rispetto alla forma lunga, come DB per database e UI per interfaccia utente) | class UtenteDB;class Tracciamento; |
| Interfacce | I nomi delle interfacce devono essere scritti in CamelCase come i nomi delle classi. | interface Identificativo; |
| Metodi | I nostri metodi sono verbi, scritti nella forma camelCase (da notare la prima lettera minuscola). | insert();createReport(); |
| Variabili | I nomi delle variabili sono brevi ma significativi. La scelta del nome di una variabile è mnemonica, cioè finalizzata a indicare all'osservatore casuale l'intento del suo utilizzo. I nomi delle variabili composte da una sola lettera vengono evitati tranne che per le variabili "usa e getta" temporanee. Se i nomi delle varabili sono composti da due nomi allora verranno scritti nella forma camelCase (da notare la prima lettera minuscola). | int i;string nomeRegione;float costo; |
| Costanti | I nomi delle variabili dichiarate costanti di classe e delle costanti ANSI devono essere tutti maiuscoli con parole separate da trattini bassi ("\_"). (Le costanti ANSI dovrebbero essere evitate, per facilitare il debug.) | static finali int MAX = 5; |

### Riferimenti

Bernd Bruegge, Allen H. Dutoit - Object-Oriented Software Engineering

GL\_SDD\_V\_2

# **Packages**

In questa sezione viene mostrata la suddivisione del sistema in package, in base a quanto definito nel documento di System Design. Tale suddivisione è motivata dalle scelte architetturali prese e ricalca la struttura di directory standard definita da Maven.

##### .idea

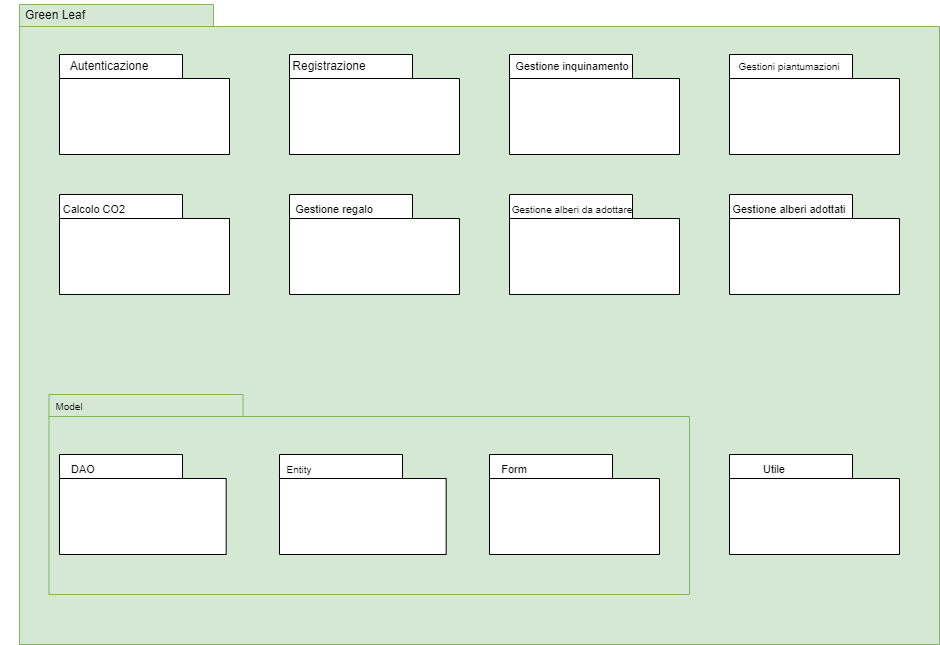
* **.mvn**, contiene tutti i file di configurazione per Maven
* **src**, contiene tutti i file sorgente
* main
* **java**, contiene tutte le classi java relative alle componenti Application e Storage
* **risorse**, contiene tutti i file relativi alle componenti Presentation
* **style**, contiene i fogli di stile CSS
* **js**, contiene gli script JS
* **static**, contiene i file HTML
* **test**, contiene tutto il necessario per il testing
* **java**, contiene le classi Java per l’implementazione del testing
* **target**, contiene tutti i file prodotti dal sistema di build di Maven

#### Package Green Leaf

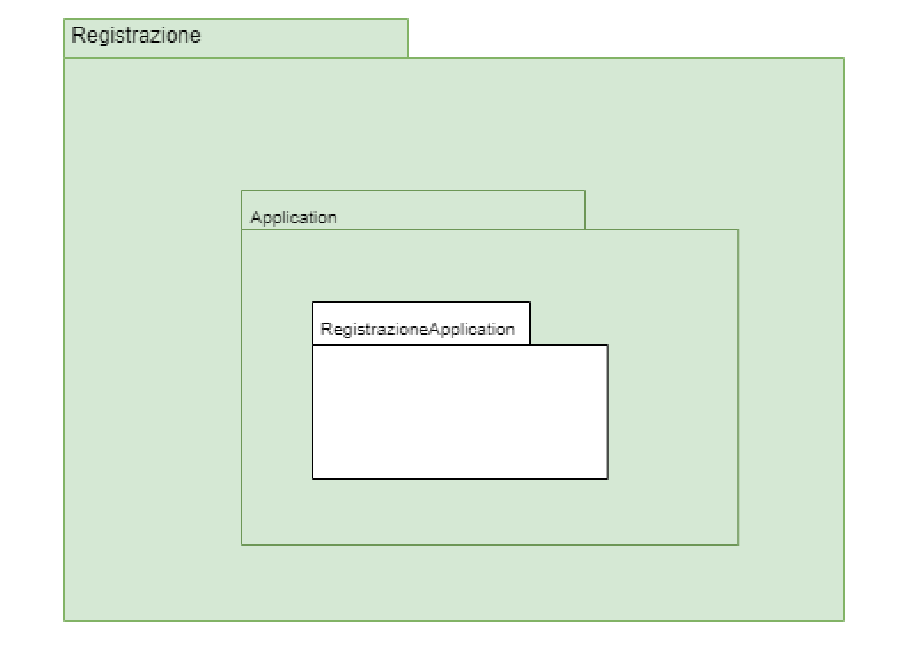
Nella presente sezione si mostra la struttura del package principale di Green Leaf. Questa struttura è stata ottenuta a partire da tre principali scelte:

1. Creare un package separato per ogni sottosistema, contenente le classi Application del suddetto, ed eventuali classi di utilità usate unicamente da esso.
2. Creare un package separato per le classi del *storage*, contenente le classi entity e i DAO per l’accesso al DB. Tale scelta è stata presa vista l’elevata complessità del database di Green Leaf che prevede numerose relazioni tra le entità. Si è quindi preferito tenere tutto in un package separato e collegato a tutti gli altri package dei sottosistemi.
3. Creare un package chiamato *utile* in cui inserire eventuali classi di utilità per il sistema e usabili da più sottosistemi.

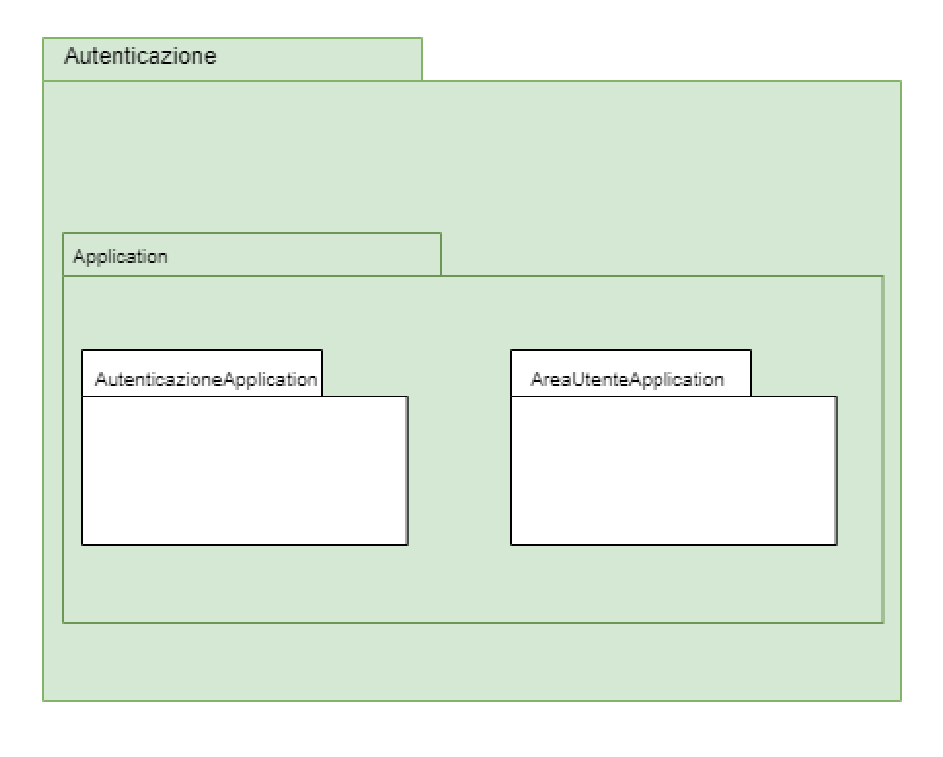
Per ciò che concerne la dipendenza tra i packages, la suddivisione precedentemente illustrata ha portato alla creazione di una relazione tra il package storage e tutti gli altri package del sistema.



#### Package Registrazione

****

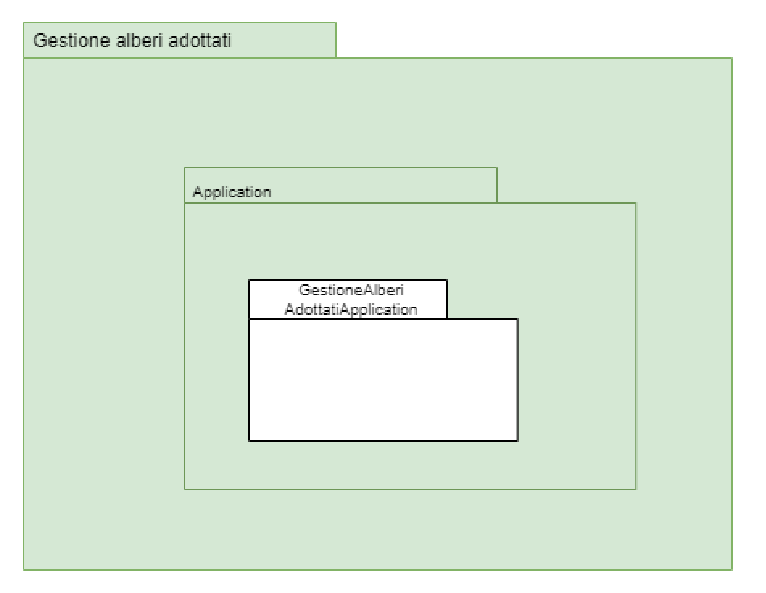
#### Package Autenticazione



#### Package Gestione piantumazioni

#### 

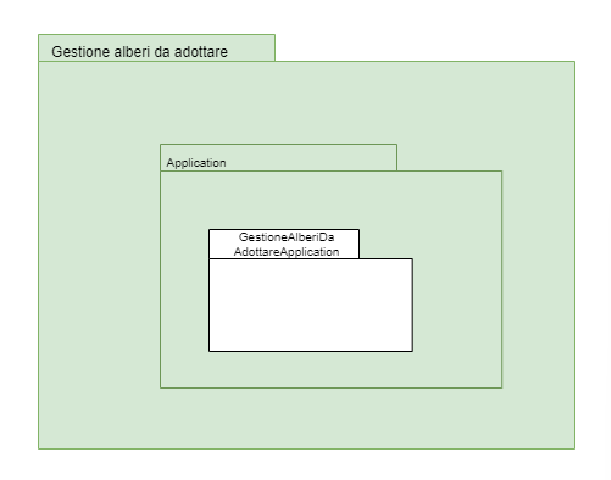
#### Package Alberi adottati



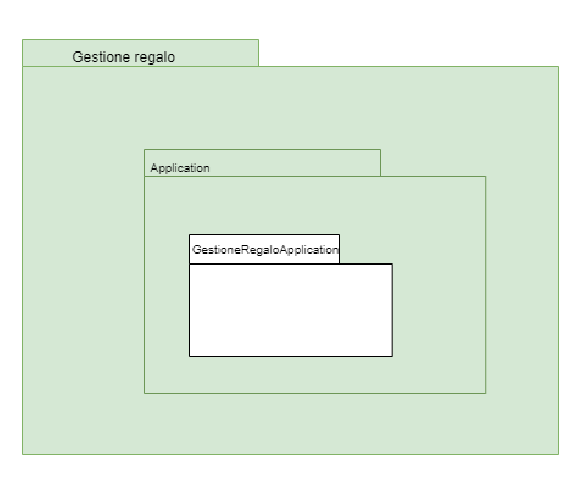
#### Package Gestione inquinamento

#### 

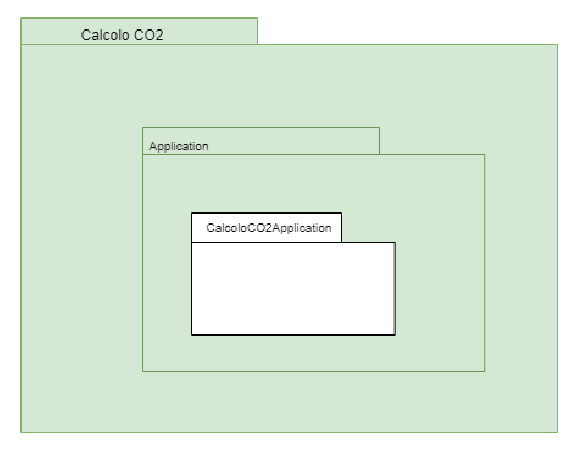
#### Package Gestione alberi da adottare



#### Package Gestione regalo

****

#### Package Calcolo CO2



# Class Interfaces

#### Package Registrazione

|  |  |
| --- | --- |
| Nome classe | Registrazioni |
| Descrizione | Questa classe permette di gestire le operazioni relative alla registrazione. |
| Metodi | +registrazione(Utente u): boolean  +creaOperatore(Operatore o): boolean |
| Invariante di classe | / |

|  |  |
| --- | --- |
| Nome metodo | +registrazione (Utente u) |
| Descrizione | Questo metodo consente di registrare un nuovo utente. |
| Pre-condizione | L’utente non è registrato |
| Post-condizione | context: Registrazione:: registrazione (Utente u)post: UtenteDAO.save(u) == true |

|  |  |
| --- | --- |
| Nome metodo | +creaOperatore(Operatore o) |
| Descrizione | Questo metodo consente di registrare un nuovo operatore. |
| Pre-condizione | Colui che effettua la registrazione è un adminL’operatore non è registrato |
| Post-condizione | context: Registrazione:: creaOperatore(Operatore o)post: OperatoreDAO.save(o) == true |

#### Package Autenticazione

|  |  |
| --- | --- |
| Nome classe | Autenticazione |
| Descrizione | Questa classe permette di gestire le operazioni relative all’autenticazione. |
| Metodi | +login (String email, String password): Object  +logout(): boolean  +eliminaOperatore(Operatore o): boolean  +eliminaAccount(): boolean  +recuperoPassword(String email): String |
| Invariante di classe | / |

|  |  |
| --- | --- |
| Nome metodo | + login (String email, String password) |
| Descrizione | Questo metodo permette autenticarsi sul sito Green Leaf. |
| Pre-condizione | L’utilizzatore non è loggato |
| Post-condizione | context: Autenticazione:: login (String email, String password) |

|  |  |
| --- | --- |
| Nome metodo | + eliminaOperatore(Operatore o) |
| Descrizione | Questo metodo permette all’admin di eliminare un operatore. |
| Pre-condizione | L’admin è loggatoL’operatore da eliminare è di sua competenza |
| Post-condizione | context: Autenticazione:: eliminaOperatore(Operatore o)post: OperatoreDAO.delete(o) |

|  |  |
| --- | --- |
| Nome metodo | + eliminaAccount() |
| Descrizione | Questo metodo permette di eliminare un utente. |
| Pre-condizione | L’utente è loggato |
| Post-condizione | context: Autenticazione:: eliminaAccount()post: UtenteDAO.delete(u) |

|  |  |
| --- | --- |
| Nome metodo | +recuperoPassword(String email) |
| Descrizione | Questo metodo permette di recuperare la password. |
| Pre-condizione | L’utente è loggato |
| Post-condizione | context: Autenticazione:: recuperaPassword(String email) |

#### Package Gestione piantumazioni

|  |  |
| --- | --- |
| Nome classe | Gestione piantumazioni |
| Descrizione | Questa classe permette di gestire le operazioni relative alla piantumazione. |
| Metodi | +inserisciPiantumazione(Albero a): void  +visualizzaPiantumazioni(): List<Albero> |
| Invariante di classe | / |

|  |  |
| --- | --- |
| Nome metodo | +inserisciPiantumazione(Albero a) |
| Descrizione | Questo metodo permette all’operatore modificare lo stato di un albero. |
| Pre-condizione | A effettuare questa operazione è un operatore |
| Post-condizione | context: GestionePiantumazioni:: inserisciPiantumazione(Albero a)post: AlberoDAO.update(a) |

|  |  |
| --- | --- |
| Nome metodo | +visualizzaPiantumazioni() |
| Descrizione | Questo metodo permette all’operatore di visualizzare tutti gli alberi della sua regione di competenza. |
| Pre-condizione | A effettuare questa operazione è un operatore |
| Post-condizione | context: GestionePiantumazioni:: visualizzaPiantumazioni() |

#### Package Gestione alberi adottati

|  |  |
| --- | --- |
| Nome classe | Gestione alberi adottati |
| Descrizione | Questa classe permette di gestire le operazioni relative agli alberi già adottati. |
| Metodi | +Mappa (da controllare)  +visualizzaAlberiAdottati: List<Albero> |
| Invariante di classe | / |

|  |  |
| --- | --- |
| Nome metodo | + visualizzaAlberiAdottati () |
| Descrizione | Questo metodo permette all’utente di visualizzare tutti gli alberi da lui adottati. |
| Pre-condizione | A effettuare questa operazione è un utente |
| Post-condizione | context: GestioneAlberiAdottati:: visualizzaAlberiAdottati () |

#### Package Gestione inquinamento

|  |  |
| --- | --- |
| Nome classe | Gestione inquinamento |
| Descrizione | Questa classe permette di gestire le operazioni relative agli alberi già adottati. |
| Metodi | / |
| Invariante di classe | / |

#### Package Gestione alberi da adottare

|  |  |
| --- | --- |
| Nome classe | Gestione alberi da adottare |
| Descrizione | Questa classe permette di gestire le operazioni relative agli alberi già adottati. |
| Metodi | +checkout(List<Alberi>): String |
| Invariante di classe | / |

|  |  |
| --- | --- |
| Nome metodo | + checkout(List<Alberi>) |
| Descrizione | Questo metodo permette all’utente di visualizzare tutti gli alberi da lui adottati. |
| Pre-condizione | A effettuare questa operazione è un utente |
| Post-condizione | context: GestioneAlberiDaAdottare:: checkout(List<Alberi>) |

#### Package Gestioni regalo

|  |  |
| --- | --- |
| Nome classe | Gestione regalo |
| Descrizione | Questa classe permette di gestire le operazioni relative ai buoni regalo. |
| Metodi | +generaRegalo(): String  +riscattaRegalo(): float (void) |
| Invariante di classe | / |

|  |  |
| --- | --- |
| Nome metodo | + generaRegalo() |
| Descrizione | Questo metodo permette all’utente di generare un buono regalo. |
| Pre-condizione | A effettuare questa operazione è un utente |
| Post-condizione | context: GestioneRegalo:: generaRegalo()post: RegaloDAO.save(r) |

|  |  |
| --- | --- |
| Nome metodo | + riscattaRegalo() |
| Descrizione | Questo metodo permette all’utente di riscattare un buono regalo. |
| Pre-condizione | A effettuare questa operazione è un utente |
| Post-condizione | context: GestioneRegalo:: riscattaRegalo()post: RegaloDAO.update(r) |

# Design Patterns

Nella presente sezione si andranno a descrivere e dettagliare i design patterns utilizzati nello sviluppo

dell’applicativo BiblioNet. Per ogni pattern si darà:

* + Una brevissima introduzione teorica.
  + Il problema che doveva risolvere all’interno di BiblioNet.
  + Una brevissima spiegazione di come si è risolto il problema in BiblioNet.
  + Un grafico della struttura delle classi che implementano il pattern.

#### Singleton e Observer

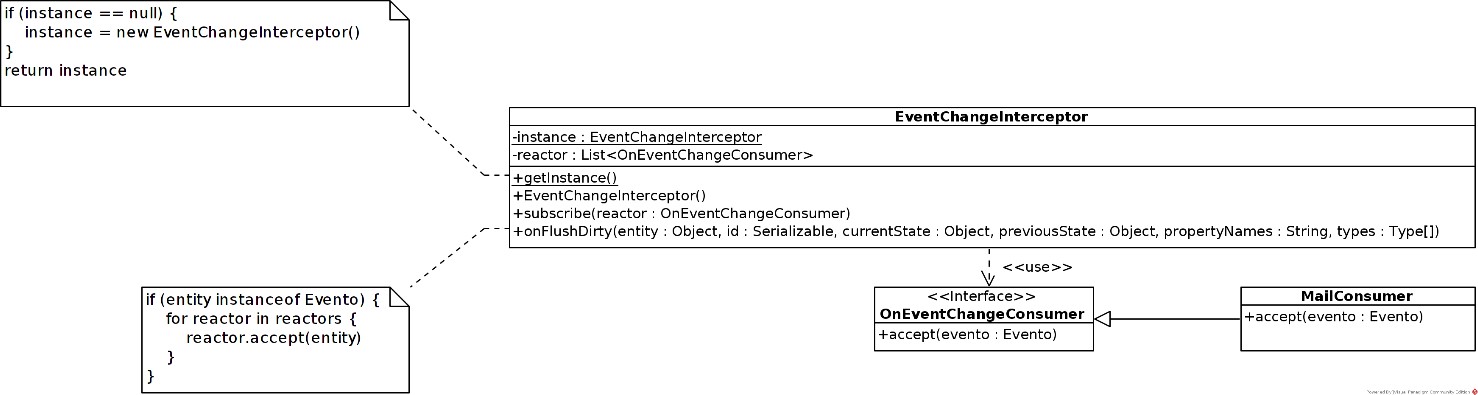
L’Observer è un design pattern comportamentale, ossia un design pattern che fornisce una soluzione a un problema di interazione tra più oggetti. Observer, nello specifico, permette a un oggetto “osservato” di generare (pubblicare) eventi ai quali una lista di oggetti “osservatori” (iscritti) possa poi reagire.

Singleton è un design pattern creazionale, ossia un design pattern che si occupa dell’istanziazione degli oggetti, che ha lo scopo di garantire che di una determinata classe venga strutturata una sola istanza e di fornire un punto di accesso globale a tale istanza.

BiblioNet prevede un sistema di notifica via email per la modifica degli eventi.

Per gestire ciò a livello implementativo senza accoppiare strettamente il sistema con un dato servizio di mailing o sistema di notifiche, viene creato un oggetto Singleton che conserva una propria lista di osservatori dotati di un metodo “accept” che accetti un singolo Evento come argomento. Ogni volta che viene modificato un evento sul database, l’oggetto, che è anche un intercettore di hibernate, “notifica” così la lista degli osservatori tramite il metodo onFlushDirty.

Ciò rende possibile la creazione in un momento successivo di multipli osservatori legati agli eventi (ad esempio per le notifiche push per un eventuale applicazione mobile).

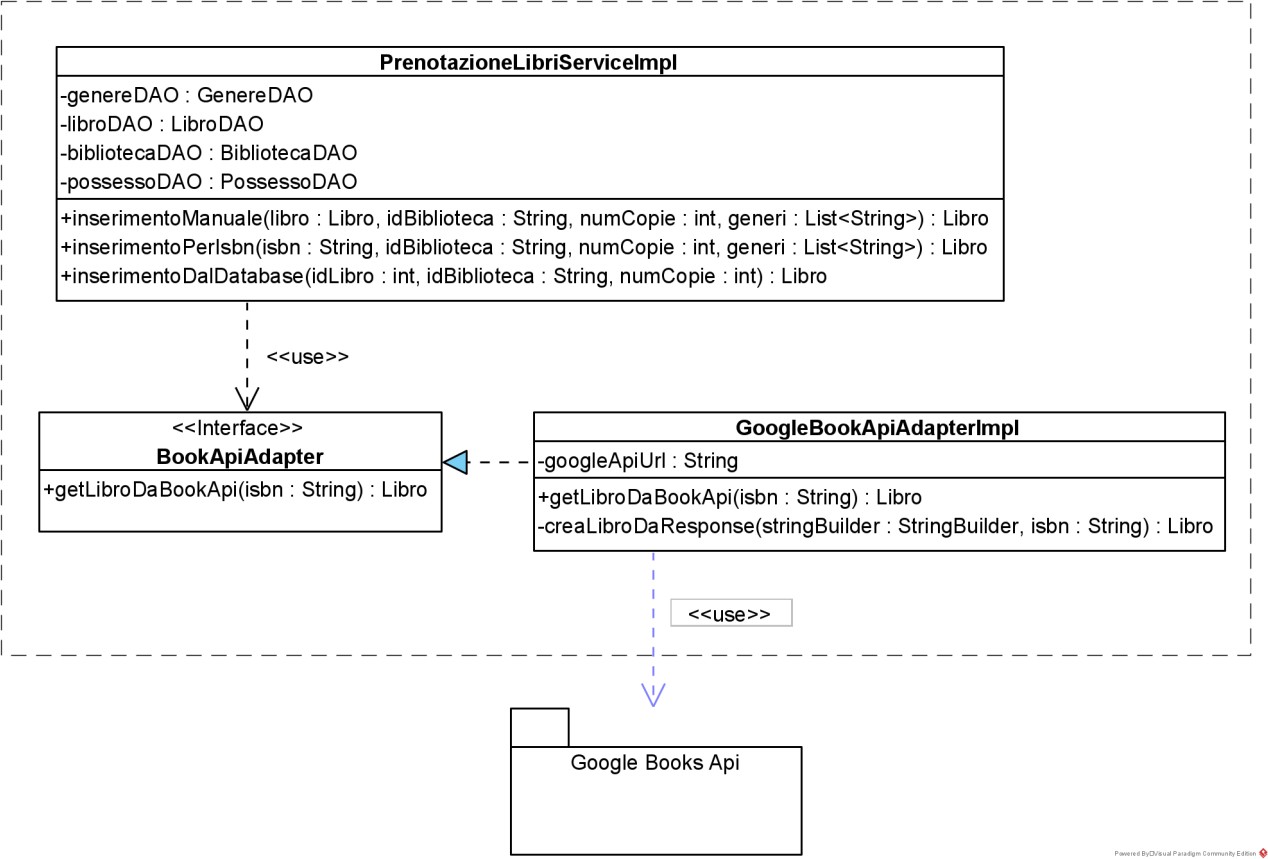


#### Adapter

L’Adapter è un design pattern strutturale, ovvero quei design pattern che facilitano la progettazione attraverso la semplificazione delle relazioni tra entità. Adapter, in particolare, permette ad oggetti con differenti interfacce di collaborare. Si implementa attraverso una classe “adapter”, che si occupa di convertire i dati in oggetti comprensibili dal sistema.

Biblionet si pone l’obiettivo di semplificare e digitalizzare l’interazione tra biblioteche e lettori. In questo ambito, offre anche la funzionalità di prenotazione di libri attraverso la piattaforma, il che presuppone che le biblioteche debbano rendere disponibile una lista di libri prenotabili dal sito stesso. Per l’inserimento dei libri all’interno di questo catalogo virtuale, oltre all’inserimento manuale, viene offerta ai gestori delle biblioteche la possibilità di inserire un libro tramite l’ISBN, senza doversi preoccupare di compilarne tutti i campi. Questa ricerca viene effettuata tramite un servizio esterno al sistema, in particolare il database di Google Books, attraverso una API offerta al pubblico sul web. Ovviamente l’API di ricerca dei libri, in quanto accessibile da chiunque, risponde alle query in un formato standard ed ampiamente utilizzato, ovvero il JSON.

Il design pattern Adapter si occupa della conversione dei dati di un libro da formato JSON ad oggetto della classe Libro, in modo da rendere le informazioni comprensibili dal sistema. La ricerca dei libri avviene in due fasi: la query all’API passando l’ISBN, e la traduzione del risultato grazie all’adapter.

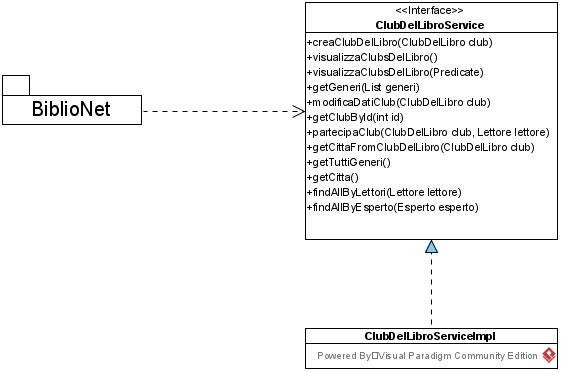


#### Facade

Il Facade è un design pattern che permette, implementando una interfaccia semplificata di accedere a sottosistemi più complessi. In questo modo si può nascondere al sistema la complessità delle librerie, dei framework o dei set di classi che si stanno usando. Si garantisce così un alto disaccoppiamento e si rende la piattaforma più manutenibile e più aggiornabile, poiché basterà cambiare l’implementazione dei metodi dell’interfaccia per implementare le modifiche.

BiblioNet, essendo un sistema molto complesso, sfrutta il design pattern Facade per implementare tutta la sua logica di business e rendere più facile l’interfacciarsi con essa. Nello specifico BiblioNet utilizza il Facade per ogni suo sottosistema, implementandolo attraverso delle interfacce che sono usate per accedere ai metodi interni.

Di seguito un esempio di Facade nel sistema BiblioNet:



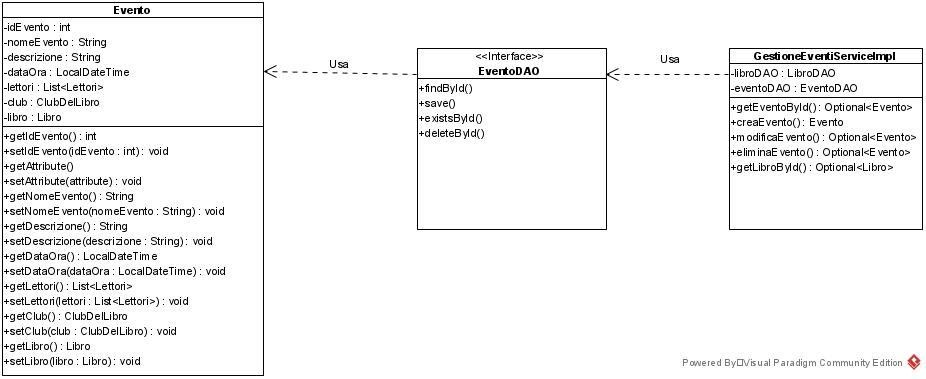
#### DAO

Un DAO (Data Access Object) è un pattern che offre un’interfaccia astratta per alcuni tipi di database. Mappando le chiamate dell’applicazione allo stato persistente, il DAO fornisce alcune operazioni specifiche sui dati senza esporre i dettagli del database. I DAO sono utilizzabili nella maggior parte dei linguaggi e la maggior parte dei software con bisogni di persistenza, principalmente viene associato con applicazioni JavaEE che utilizzano database relazionali.

Essendo biblionet una web application che punta di gestire sia prenotazione di libri che numerosi club del libro presenta un database molto vasto, quindi ha bisogno di poter interagire con database in modo rapido e sicuro con numerosi query per quella che è la moltitudine di dati da gestire. Per questo motivo abbiamo usato varie interfacce DAO all’interno del nostro sistema le quali, grazie all’utilizzo del framework di Spring (SpringJPARepository), vengono implementate in maniera del tutto automatica e trasparente per il programmatore. Attraverso l’utilizzo di Spring JPA Repository viene quindi rimossa la necessità di avere una classe che implementi tutte le eventuali query continuando a poter usufruire a pieno dei DAO per accedere al database nel resto dell’applicazione.

Qui è riportato un esempio di DAO utilizzato in Biblionet e delle relazioni che ha con altre classi

dell’applicazione.



# Glossario

|  |  |
| --- | --- |
| **Sigla/Termine** | **Definizione** |
| **Package** | Raggruppamento di classi ed interfacce. |
| **DAO** | Data Access Object, implementazione dell’omonimo pattern architetturale che si occupa di fornire un accesso in modo astratto ai dati persistenti. |
| **Controller** | Classe che si occupa di gestire le richieste effettuate dal client. |
| **Service** | Classe che implementa la logica di business, viene utilizzata dal controller o da un altro sottosistema. |
| **Model** | Parte del design architetturale MVC che fornisce al sistema i metodi per accedere ai dati utili al sistema. |
| **MVC** | Model-View-Controller: design architetturale che permette di separare la logica di presentazione dalla logica di business alla base del sistema. |
| **Facade** | Un oggetto che permette, attraverso un’interfaccia più semplice, l’accesso a sottosistemi che espongono interfacce complesse e molto diverse tra loro. |

|  |  |
| --- | --- |
| **Adapter** | È un pattern strutturale che può essere basato sia su classi che su oggetti il cui fine è fornire una soluzione astratta al problema dell’interoperabilità tra interfacce differenti. |
| **Singleton** | È un design pattern creazionale che ha lo scopo di garantire che di una determinata classe venga struttura una solo istanza e di fornire un punto di accesso globale a tale istanza. |
| **Observer Pattern** | Design pattern usato come base per la gestione di eventi. |
| **Hibernate** | Piattaforma open source per la gestione della persistenza dei dati sul database. |
| **SRP** | Principio di singola responsabilià: ogni element del programma deve avere una singola responsabilità |