|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ODD  Progetto  STRAGAME   |  |  | | --- | --- | | Riferimento |  | | Versione | 0.1 | | Data | 16/11/2020 | | Destinatario | Prof.ssa F. Ferrucci | | Presentato da | STRANIVERSITY (Umberto Mauro, Vincenzo Iovino, Mario Maffettone, Carmine Laudato) | | Approvato da |  | |

RevisionHistory

|  |  |  |  |
| --- | --- | --- | --- |
| **Data** | **Versione** | **Descrizione** | **Autori** |
| 12/12/2020 | 0.1 | Primo capitolo e inizio del secondo capitolo | Vincenzo Iovino  Carmine Laudato |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Indice

1. Introduzione 4
   1. Object Design trade-off 4
   2. Interface Documentation Guidelines 4
      1. Package 5
      2. Pagine HTML 5
      3. Fogli di stile CSS 5
      4. Script JavaScript e JSP Indentazione 5
      5. Nomi 5
      6. Posizione dichiarazione variabili 5
      7. Database SQL 6
      8. Organizzazione dei file 6
   3. Definizioni, Acronimi e Abbreviazioni 6
   4. Riferimenti 7
2. Packages 7
   1. View 7
   2. Model 9
   3. Controller 10
3. Class Interfaces 11
4. Design Pattern con Class Diagram 19
   1. Data Access Object Pattern 19
   2. Observer Pattern 20
5. Glossario 21
6. **Introduzione**
   1. **Object Design trade-off**

Dopo la realizzazione del documento RAD (Requirement Analysis Document) e SDD (System Design Document), abbiamo descritto in linea di massima, quello che sarà il nostro sistema e quindi i nostri obiettivi, tralasciando gli aspetti implementativi.

* **Prestazioni VS Costi**

Considerando il sistema che stiamo realizzando, possiamo dire che il non eccessivo budget a nostra disposizione ci ha consentito di realizzare il prodotto utilizzando materiale open source partendo da zero minimizzando i costi e rendendo il sistema più che soddisfacente.

* **Interfaccia VS Tempo di risposta**

Il tempo di risposta tra server e interfaccia sono più che sufficienti(rapidi), a soddisfare le esigenze dei vari videogiocatori collegati al sistema.

* **Interfaccia VS Easy-use**

L’interfaccia, grazie all’utilizzo delle form e di una impostazione semplice e intuitiva, permette un suo facile (Easy-Use) della gestione del sistema di database, permettendo l’immediata attività anche ai meno esperti col computer.

* **Costi VS Mantenimento**

Grazie a un uso di materiale open source e l’utilizzo del linguaggio javadoc, il sistema può essere facilmente modificato, aggiungendo nuove funzioni o corregendo gli errori in loro presenza.

* 1. **Interface Documentation Guidelines**

Per contenere gli eventuali costi di manutenzione e, allo stesso tempo, fruire di un supporto già pronto all’uso ed open source, ci avverremo di Bootstrap. In tal modo non sarà necessario impiegare tempo per le molteplici configurazioni ma sarà possibile implementare facilmente tutti gli elementi caratteristici di un front end e tra gli altri:

* + 1. **Package**

I nomi dei package sono tutti “lowerCamelCase”. Per esempio, my.exampleCode.Stragame, ma non my.examplecode.stragame o my.example\_code.stra\_game.

* + 1. **Pagine HTML**

Le pagine HTML, sia in forma statica che dinamica, devono essere conformi allo standard HTML. Inoltre, il codice HTML statico deve utilizzare l'indentazione, per facilitare la lettura, secondo le seguenti regole:

* Un’indentazione consiste in una tabulazione;
* Ogni tag deve avere un'indentazione maggiore del tag che lo contiene;
* Ogni tag di chiusura deve avere lo stesso livello di indentazione del corrispondente tag di apertura;
* I tag di commento devono seguire le stesse regole che si applicano ai tag normali.
  + 1. **Fogli di stile CSS**I fogli di stile devono essere raggruppati in opportune cartelle. Lo stile non deve essere unito con il contenuto della pagina, scritto in HTML.
    2. **Script JavaScript e JSP Indentazione**

Indentare, utilizzando un TAB, opportunamente le istruzioni.

* + 1. **Nomi**

La notazione da usare per i nomi delle variabili e dei metodi è la nota Camel Notation. I nomi dei file, delle operazioni e delle variabili devono essere evocabili.

* + 1. **Posizione dichiarazione variabili**

Posizionare le dichiarazioni all’inizio dei blocchi. Non bisogna dichiarare le variabili al loro primo uso: può confondere il programmatore inesperto e impedire la portabilità del codice dentro lo scope. L’unica eccezione a questa regola sono gli indici dei cicli for che possono essere dichiarati a più alto livello. Ad esempio, non dichiarare una variabile con lo stesso nome in un blocco interno.

* + 1. **Database SQL**

I nomi delle tabelle e dei campi devono essere: evocativi, leggibili.

* + 1. **Organizzazione dei file**

Ogni file deve essere:

* Sviluppato e diviso in base alla categoria di appartenenza, ovvero deve essere correlato ad un’unica funzionalità che persegue. Ogni pagina deve essere implementata in file separati;
* Diviso in più file se raggiunge una lunghezza tale da divenire difficile da leggere e comprendere. Organizzare in una cartella i file della libreria usate e le altre risorse scaricate necessarie per lo sviluppo del progetto.
  1. **Definizioni, Acronimi e Abbreviazioni**

Acronimi:

* RAD: Requirements Analysis Document
* SDD: System Design Document
* ODD: Object Design Document

Definizioni:

* upperCamelCase: è una convenzione di denominazione in cui un nome è formato da più parole unite insieme come una singola parola con la prima lettera di ciascuna delle parole in maiuscolo all'interno della nuova parola che forma il nome.
* lowerCamelCase: è una variante dell’upperCamelCase, in cui la prima lettera della nuova parola è minuscola,consentendo di distinguerla facilmente da un nome UpperCamelCase.
* HTML: linguaggio di programmazione utilizzato per lo sviluppo di pagine Web.
* CSS: acronimo di Cascading Style Sheets, è un linguaggio usato per definire la formattazione delle pagine Web.
* MySQL: è un DBMS relazionale, composto da varie tabelle e relazioni tra di esse.
* Bootstrap: è una raccolta di strumenti liberi per la creazione di siti e applicazioni per il web.
* Off-The-Shelf: Servizi esterni al sistema di cui viene fatto utilizzo.
  1. **Riferimenti**
* RAD\_V.0.7.2
* SDD\_V.1.0

1. **Packages**
   1. **View**

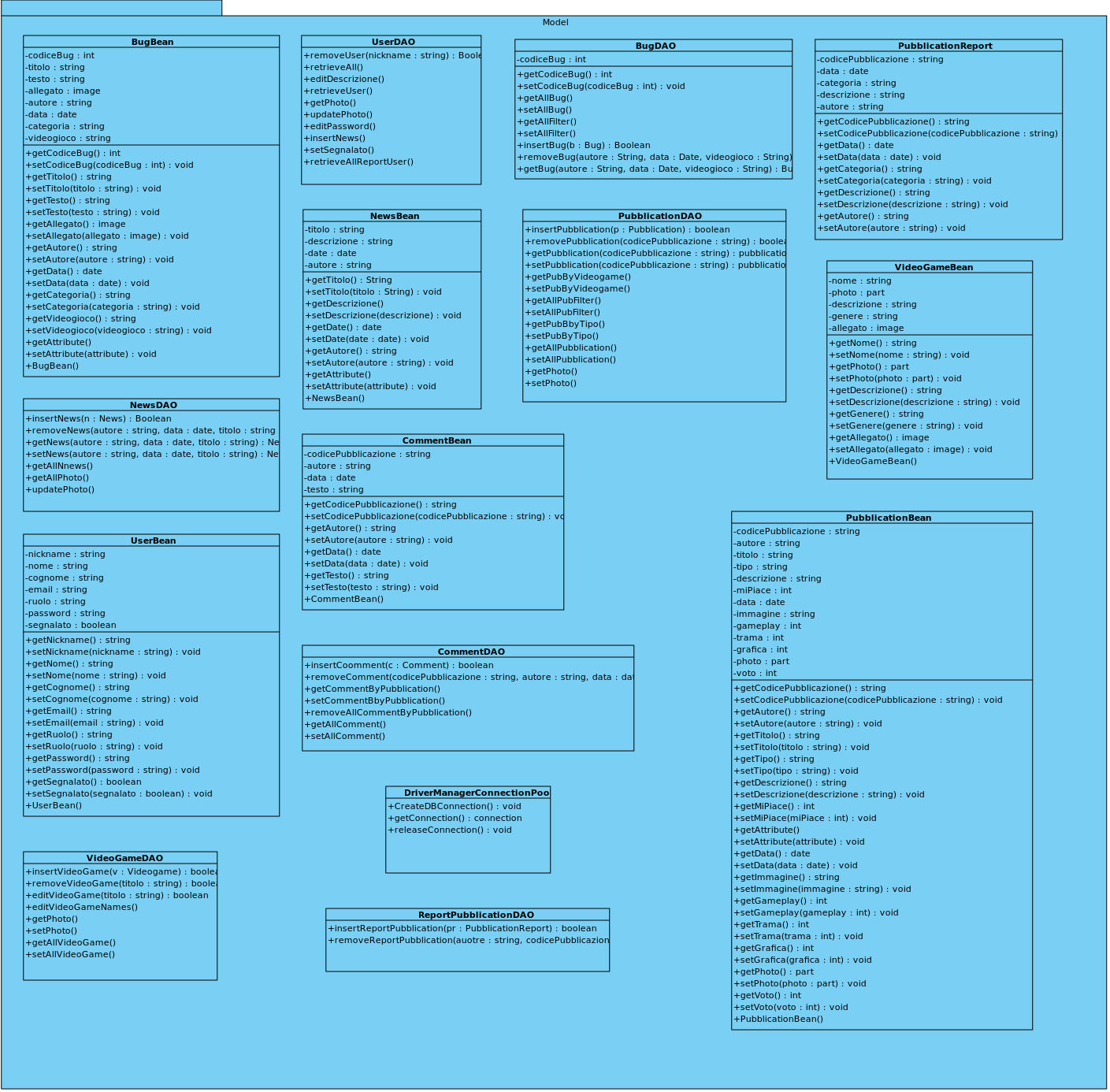
Il package View contiene le JSP per la visualizzazione delle pagine,esse possono essere globali, relative all’utente loggato e relative all’admin.

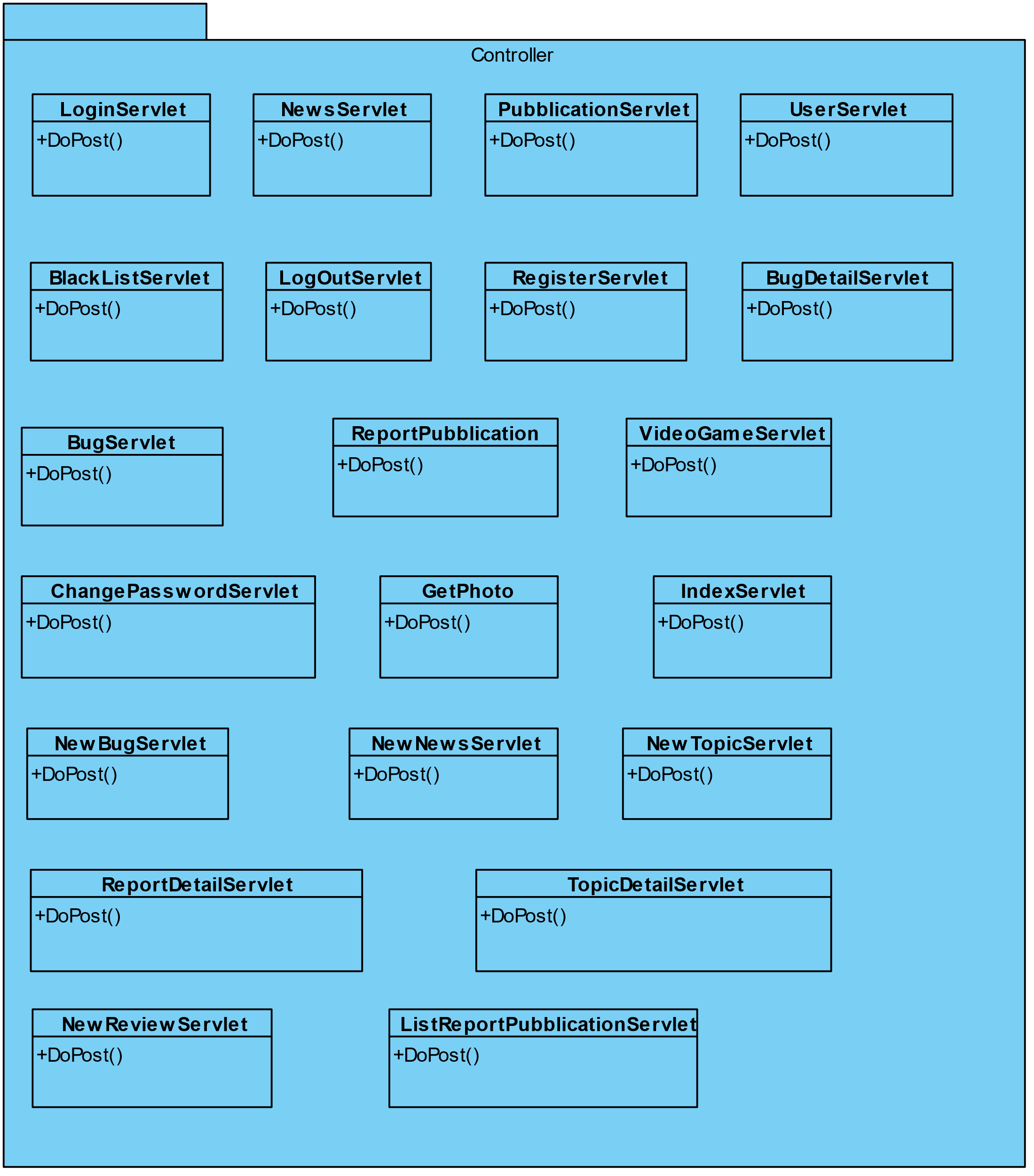
|  |  |
| --- | --- |
| Globali | |
| index.jsp | pagina relativa all’homepage della piattaforma, dove è possibile effettuare il login, ed è possibile visionare i prodotti videoludici presenti nel forum. |
| signUp.jsp | permette la registrazione di un nuovo utente. |
| topic.jsp | permette di visualizzare la lista dei topic ordinati in base alla loro pubblicazione. |
| review.jsp | permette di visualizzare la lista delle recensioni ordinati in base alla loro pubblicazione. |
| news.jsp | permette di visualizzare la lista delle news ordinati in base alla loro pubblicazione. |
| profile.jsp | permette la visualizzazione del profilo dell’utente loggato, e di modificare la descrizione, l’immagine del profilo e la password. |
| changePassword.jsp | permette di modificare la password dell’utente loggato. |
| topicDetails.jsp | permette di visionare i dettagli del topic selezionato. |
| reviewDetails.jsp | permette di visionare i dettagli della recensione selezionata. |
| videogamePubblication.jsp | permette di visionare la lista dei topic e delle recensioni relative ad un videogioco selezionato. |

|  |  |
| --- | --- |
| Utente | |
| newTopic.jsp | premette l’inserimento di un nuovo topic. |
| newReview.jsp | premette di inserire una nuova recensione. |
| newBug.jsp | premette di segnalare un bug relativo ad un prodotto videoludico. |
| reportPubblication.jsp | permette di segnalare una pubblicazione di un utente. |

|  |  |
| --- | --- |
| Admin | |
| newNews.jsp | premette di creare una nuova news. |
| listReportPubblication.jsp | permette, all’admin, di visualizzare la lista di pubblicazioni segnalate. Inoltre permette all’admin di rimuovere la pubblicazione oppure rimuovere la segnalazione. |
| reportDetails.jsp | permette di visionare i dettagli di una segnalazione di una pubblicazione. |
| blackList.jsp | permette, all’admin, di visionare e rimuovere un utente che ha qualche pubblicazione segnalata. |
| newGame.jsp | permette, all’admin, di inserire un nuovo prodotto videoludico. |
| bug.jsp | permette, all’admin, di visionare la lista delle segnalazioni dei bug. |
| bugDetails.jsp | permette di visualizzare i dettagli di un bug. |

* 1. **Model**

Il package Model si occupa di fare da tramite tra l’applicazione e il database sottostante. Ogni classe contenuta all’interno del pacchetto fornisce i metodi per accedere ai dati utili all’applicazione. I moduli contenuti all’interno del package sono: User, Admin, Notification, Bug, Pubblication, News.

1. ****
   1. **Controller**
2. **Class Interfaces**

In questa sezione sono presentate le interfacce delle classi del package model e sono assenti:

* Le classi Bean del model, in quanto dispongono di soli costruttori getter e setter;
* Le classi controller in quanto usufruiscono dei metodi offerti dai service con i quali interagiscono.

|  |  |
| --- | --- |
| Nome Classe | **UserDAO** |
| Descrizione | Questa classe rappresenta l’entità tabellare User contenuta all’interno del nostro DBMS relazionale. Ci permette di interfacciarci con il DBMS relazionale e di interrogarlo tramite dei metodi specifici. |
| Pre-condizione | **context** UserDAO**:** retrieveAll();  **pre:** User != null  **context** UserDAO**:** retrieveUser(String nickname);  **pre:** User != null  **context** UserDAO**:** editDescription(String descrizione, String nickname);  **pre:** login =True  **context** UserDAO**:** getPhoto(String nickname);  **pre:** photo!= null  **context** UserDAO**:** updatePhoto(String username, Part photo);  **pre:** register=True  **context** UserDAO**:** editPassword(String password, String nickname);  **pre:** login =True  **context** UserDAO**:** insertUser(String nickname);  **pre:** User != null  **context** UserDAO: removeUser(String nickname);  **pre:** User != null;  **context** UserDAO**:** setSegnalato();  **pre: l**ogin =True  **context** UserDAO**:** retrieveAllReportUser();  **pre:** User != null; |
| Post-condizione | **context** UserDAO**:** removeUser(String nickname);  **post:** removeUser=True  **context** UserDAO**:** retrieveAll();  **post:** return All  **context** UserDAO**:** retrieveUser(String nickname);  **post:** retrieveUser =True  **context** UserDAO**:** editDescription(String descrizione, String nickname);  **post:** editDescription =True  **context** UserDAO**:** getPhoto(String nickname);  **post:** return Part photo  **context** UserDAO**:** updatePhoto(String username, Part photo);  **post:** updatePhoto =True  **context** UserDAO**:** editPassword(String password, String nickname);  **post:** editPassword =True  **context** UserDAO**:** insertUser(String nickname);  **post:** insertUser =True  **context** UserDAO**:** setSegnalato();  **post:** setSegnalato =True  **context** UserDAO**:** retrieveAllReportUser();  **post:** return ArrauList <Report> |
| Invarianti |  |

|  |  |
| --- | --- |
| Nome Classe | **PublicationDAO** |
| Descrizione | Questa classe rappresenta l’entità tabellare Publication contenuta all’interno del nostro DBMS relazionale. Ci permette di interfacciarci con il DBMS relazionale e di interrogarlo tramite dei metodi specifici. |
| Pre-condizione | **context** PublicationDAO**:** insertPublication(Publication p);  **pre: if p.getTipo != null && p.getTipo == Topic:**  p.getCodicepubblicazione != null &&  p.getAutore != null &&  p.getTitolo != null &&  p.getDescrizione != null &&  p.getCategoria != null &&  p.getData != null &&  p.getVoto != null &&  p.getPhoto != null  **context** PublicationDAO**:** getPublication(String codicePubblicazione);  **pre:** codicePubblicazione != null  **context** PublicationDAO**:** removePublication(String codicePubblicazione);  **pre:** codicePubblicazione != null  **context** PublicationDAO**:** getPubsByVideogame(String tipo);  **pre:** tipo != null  **context** PublicationDAO**:** getAllPubFilter(String videogioco, String tipo);  **pre:** String videogioco != null && String tipo != null  **context** PublicationDAO**:** getPubsByTipo(int id);  **pre:** id!= null  **context** PublicationDAO**:** getAllPublicationr();  **pre:** Publication != null  **context** PublicationDAO**:** getPhoto(String nickname);  **pre:** photo!= null |
| Post-condizione | **context** PublicationDAO**:** insertPublication(Publication p);  **post:** insertPublication=True  **context** PublicationDAO**:** getPublication(String codicePubblicazione);  **post:** return Publication p  **context** PublicationDAO**:** removePublication(String codicePubblicazione);  **post:** removePublication=True  **context** PublicationDAO**:** getPubsByVideogame(String tipo);  **post:** return Publication pv  **context** PublicationDAO**:** getAllPubFilter(String videogioco, String tipo);  **post:** return <List> Publication pf  **context** PublicationDAO**:** getPubsByTipo(int id);  **post:** return Publication pt  **context** PublicationDAO**:** getAllPublicationr();  **post:** return ArrauList <Publication>  **context** PublicationDAO**:** getPhoto(String nickname);  **post:** return Part photo |
| Invarianti |  |

|  |  |
| --- | --- |
| Nome Classe | **NewsDAO** |
| Descrizione | Questa classe rappresenta l’entità tabellare News contenuta all’interno del nostro DBMS relazionale. Ci permette di interfacciarci con il DBMS relazionale e di interrogarlo tramite dei metodi specifici. |
| Pre-condizione | **context** NewsDAO**:** insertNews(News n);  **pre:** n.getAutore != null &&  n.getTitolo != null &&  n.getDescrizione != null &&  n.getData != null  **context** NewsDAO**:** getNews(String autore, String titolo, Date data);  **pre:** autore != null && titolo != null && data != null  **context** NewsDAO**:** removeNews(String autore, String titolo, Date data);  **pre:** autore != null && titolo != null && data != null  **context** NewsDAO**:** getAllNews();  **pre:** <List> != null  **context** UserDAO**:** getPhoto(String nickname);  **pre:** photo!= null  **context** UserDAO**:** updatePhoto(String username, Part photo);  **pre:** register=True |
| Post-condizione | **context** NewsDAO**:** insertNews(News n);  **post:** return True  **context** NewsDAO**:** getNews(String autore, String titolo, Date data);  **post:** return News n  **context** NewsDAO**:** removeNews(String autore, String titolo, Date data);  **post:** return True  **context** NewsDAO**:** getAllNews();  **post:** return ArrauList <Part>  **context** UserDAO**:** getPhoto(String nickname);  **post:** return Part photo  **context** UserDAO**:** updatePhoto(String username, Part photo);  **post:** return true |
| Invarianti |  |

|  |  |
| --- | --- |
| Nome Classe | **VideoGameDAO** |
| Descrizione | Questa classe rappresenta l’entità tabellare VideoGame contenuta all’interno del nostro DBMS relazionale. Ci permette di interfacciarci con il DBMS relazionale e di interrogarlo tramite dei metodi specifici. |
| Pre-condizione | **context** VideoGameDAO**:** insertVideoGame(VideoGame v);  **pre:** v.getGenere != null &&  v.getTitolo != null &&  v.getDescrizione != null  **context** VideoGameDAO**:** getVideoGame(String titolo);  **pre:** titolo != null  **context** VideoGameDAO**:** removeVideoGame(String titolo);  **pre:** titolo != null  **context** VideoGameDAO**:** getVideoGameNames();  **pre:** Videogame != null  **context** VideoGameDAO**:** getAllVideoGame(t);  **pre:** Videogame != null  **context** UserDAO**:** getPhoto(String nickname);  **pre:** photo!= null |
| Post-condizione | **context** VideoGameDAO**:** insertVideoGame(VideoGame v);  **post:** insertVideoGame=True  **context** VideoGameDAO**:** getVideoGame(String titolo);  **post:** return Videogame v  **context** VideoGameDAO**:** removeVideoGame(String titolo);  **post:** removeVideoGame=True  **context** VideoGameDAO**:** getVideoGameNames();  **post:** return String name  **context** VideoGameDAO**:** getAllVideoGame(t);  **post:** return ArrauList <Videogame>  **context** UserDAO**:** getPhoto(String nickname);  **post:** return Part photo |
| Invarianti |  |

|  |  |
| --- | --- |
| Nome Classe | **BugDAO** |
| Descrizione | Questa classe rappresenta l’entità tabellare Bug contenuta all’interno del nostro DBMS relazionale. Ci permette di interfacciarci con il DBMS relazionale e di interrogarlo tramite dei metodi specifici. |
| Pre-condizione | **context** BugDAO**:** insertBug(Bug b);  **pre:** b.getTitolo != null &&  b.getCategoria != null &&  b.getData != null &&  b.getTesto != null &&  b.getVideogioco != null &&  b.getAutore != null  **context** BugDAO**:** removeBug(String autore, String videogioco, Date data);  **pre:** autore != null && videogioco != null && data != null  **context** BugDAO**:** getBug(String autore, String videogioco, Date data);  **pre:** autore != null && videogioco != null && data != null  **context** BugDAO**:** getAllBug();  **pre:** Bug != null  **context** BugDAO**:** getAllBugFilter(String videogioco, String categotia);  **pre:** videogioco != null && categotia != null |
| Post-condizione | **context** BugDAO**:** insertBug(Bug b);  **post:** insertBug=True  **context** BugDAO**:** removeBug(String autore, String videogioco, Date data);  **post:** removeBug=True  **context** BugDAO**:** getBug(String autore, String videogioco, Date data);  **post:** return Bug b  **context** BugDAO**:** getAllBug();  **post:** return ArrauList <Bug>  **context** BugDAO**:** getAllBugFilter(String videogioco, String categotia);  **post:** return ArrauList <Bug> |
| Invarianti |  |

|  |  |
| --- | --- |
| Nome Classe | **CommentDAO** |
| Descrizione | Questa classe rappresenta l’entità tabellare Comment contenuta all’interno del nostro DBMS relazionale. Ci permette di interfacciarci con il DBMS relazionale e di interrogarlo tramite dei metodi specifici. |
| Pre-condizione | **context** commentDAO**:** insertComment(Comment c);  **pre:** c.getCodicepubblicazione != null &&  c.getAutore != null &&  c.getData != null &&  c.getTesto != null  **context** commentDAO**:** removeComment(int codicePubblicazione, String autore, Date data);  **pre:** codicePubblicazione != null && autore != null && data != null  **context** commentDAO**:** getAllCommentByPublication(int codicePubblicazione);  **pre:** codicePubblicazione!= null  **context** commentDAO**:** removeAllCommentByPub(int codicePubblicazione);  **pre:** codicePubblicazione != null  **context** commentDAO**:** getAllComment();  **pre:** Comment != null |
| Post-condizione | **context** commentDAO**:** insertComment(Comment c);  **post:** insertComment=True  **context** commentDAO**:** removeComment(String codicePubblicazione, String autore, Date data);  **post:** removeComment=True  **context** commentDAO**:** getListComment(String codicePubblicazione);  **post:** return ArrayList<Comment>  **context** commentDAO**:** getAllCommentByPublication(int codicePubblicazione);  **post:** return ArrayList<Comment>  **context** commentDAO**:** removeAllCommentByPub(int codicePubblicazione);  **post:** return ArrayList<Comment>  **context** commentDAO**:** getAllComment();  **post:** return ArrayList<Comment> |
| Invarianti |  |

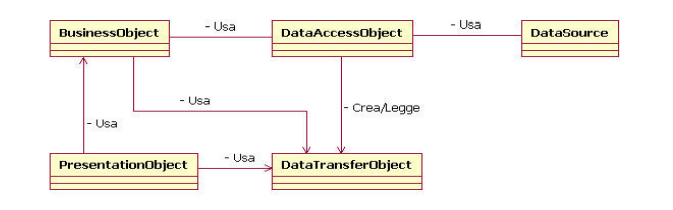
|  |  |
| --- | --- |
| Nome Classe | **PublicationReportDAO** |
| Descrizione | Questa classe rappresenta l’entità tabellare PublicationReport contenuta all’interno del nostro DBMS relazionale. Ci permette di interfacciarci con il DBMS relazionale e di interrogarlo tramite dei metodi specifici. |
| Pre-condizione | **context** PublicationReportDAO**:** insertReportPublication(PubblicationReport pr);  **pre:** pr.getCodicePubblicazione != null &&  pr.getAutoreSegnalazione != null &&  pr.getData != null &&  pr.getCategoria != null &&  pr.getDescrizione != null  **context** PublicationReportDAO**:** removeReportPublication (String autoreSegnalazione, String codicePubblicazione);  **pre:** autoreSegnalazione != null && codicePubblicazione != null  **context** PublicationReportDAO**:** getAllPublicationReport ();  **pre:** Publication != null  **context** PublicationReportDAO**:** getReportPublication (String autore, int codicePubblicazione);  **pre:** autore != null && codicePubblicazione != null  **context** PublicationReportDAO**:** removeAllReportByPub (int codicePubblicazione);  **pre:** codicePubblicazione != null |
| Post-condizione | **context** PublicationReportDAO**:** insertPublicationReport(PubblicationReport pr);  **post:** insertPublicationReport=True  **context** PublicationReportDAO**:** removePublicationReport(String autoreSegnalazione, String codicePubblicazione);  **post:** removePublicationReport=True  **context** PublicationReportDAO**:** getAllPublicationReport ();  **post:** return ArrayList <ReportPublication>  **context** PublicationReportDAO**:** getReportPublication (String autore, int codicePubblicazione);  **post:** return ReportPublication  **context** PublicationReportDAO**:** removeAllReportByPub (int codicePubblicazione);  **post:** removeAllReportByPub =True |
| Invarianti |  |

1. **Design Pattern con Class Diagram**
   1. **Data Access Object Pattern (DAO)**

**Nome e classificazione:** Data Access Object Pattern (DAO) ed è un design pattern architetturale.

**Scopo:** La sua funzione consiste nel nascondere all'applicazione tutte le complessità relative all'esecuzione delle operazioni CRUD nel meccanismo di archiviazione sottostante. Ciò consente a entrambi i livelli di evolversi separatamente senza sapere nulla l'uno dell'altro.

**Applicabilità:** L’oggetto DAO gestisce la connessione con il data source per estrarre e/o immagazzinare i dati. Gli oggetti DAO implementano i meccanismi di accesso richiesti per lavorare con il data source.

**Class diagram:**

**Partecipanti:** I partecipanti al pattern sono:

* BusinessObject: rappresenta il client del dato. È l’oggetto che richiede l’accesso al data source per ottenere o immagazzinare informazioni.
* DataAccessObject: è l’oggetto principale di questo pattern.

Il DataAccessObject astrae l’implementazione dell’accesso ai dati per il BusinessObject per rendere l’accesso trasparente e indipendente dalla tecnologia utilizzata.

* DataSource: è il data source con la sua implementazione attuale, ovvero il nostro database MySQL.
* TransferObject: rappresenta il Transfer Object usato per trasportare i dati. Il DataAccessObject può usare il Transfer Object per ritornare i dati al client. Il DataAccessObject può anche ricevere dati dal client tramite un Transfer Object per aggiornare il data source.

**Conseguenze:** Il DAO nasconde del tutto ai suoi client i dettagli implementativi di accesso al data source. Poiché l’interfaccia esposta dal DAO non cambia quando il sottostante data source cambia implementazione, questo pattern permette al Data Access Object di adattarsi a differenti schemi implementativi senza che questo abbia alcun effetto sui client o sui componenti del business layer.

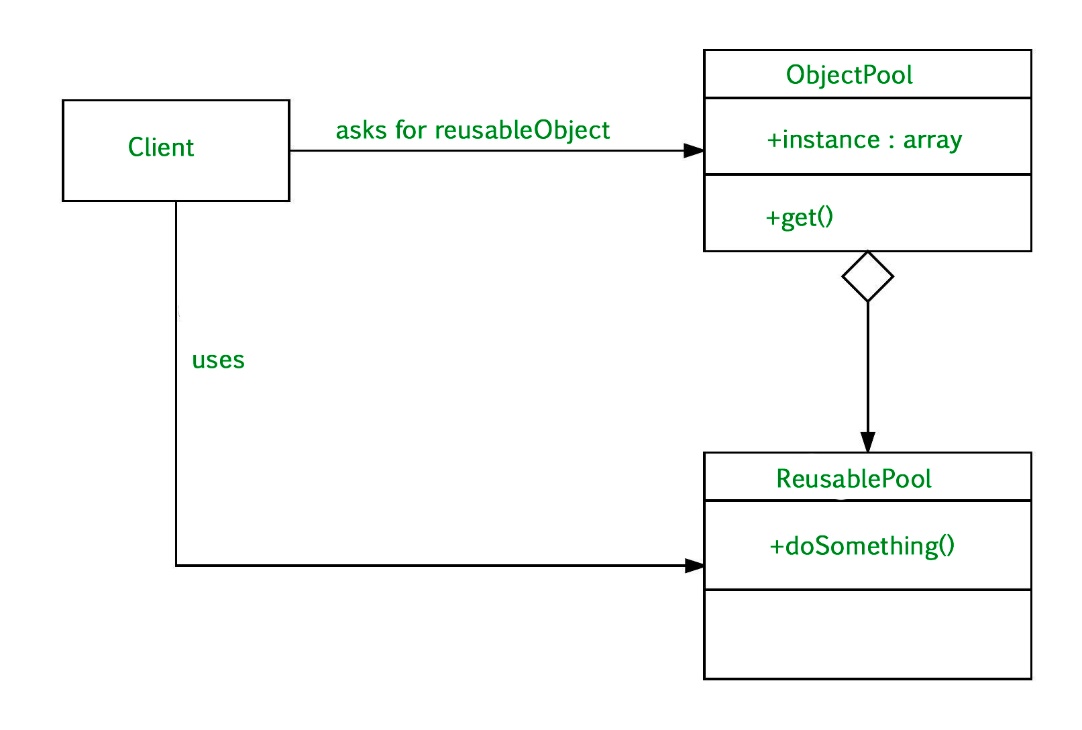
**Utilizzo:** all’interno del nostro prodotto software, verrà utilizzato per la gestione delle connessioni e dei dati del database MySQL.

* 1. **Object Pool Design Pattern**

**Nome e classificazione:** Object Pool Design Pattern ed è un design pattern creazionale.

**Scopo:** Questo pattern utilizza una serie di oggetti inizializzati mantenuti in un pool, piuttosto che allocarli e distruggerli su richiesta. Un client del pool farà richiesta di un oggetto e eseguirà operazioni sull’oggetto restituito. quando il client avrà finito l’oggetto sarà restituito all’object pool, invece di essere distrutto.

**Applicabilità:** Quando è necessario lavorare con un gran numero di oggetti le cui istanze sono particolarmente costose da creare e ogni oggetto è necessario solo per un breve periodo di tempo, le prestazioni di un'intera applicazione possono essere influenzate negativamente. Un modello di progettazione del pool di oggetti può essere considerato desiderabile in casi come questi.

**Class Diagram:**

**Partecipanti:** I partecipanti al pattern sono:

* Client: è la classe che utilizza un oggetto del tipo mantenuto nell’ ObjectPool.
* ReuseablePool: è la classe la cui istanziazione è lenta o dispendiosa, o la cui disponibilità è limitata, motivo per il quale le sue istanze sono mantenute nell’ObjectPool.
* ObjectPool: è la classe che mantiene una lista di oggetti disponibili e una collezione di oggetti che sono già stati richiesti al Pool.

**Conseguenze:** il vantaggio dell’ObjectPool è che offre un significativo boost delle performance quando il rate di inizializzazione di un’istanza di una classe è alto.

**Utilizzo:** all’interno della nostra piattaforma verrà utilizzato per mantenere una lista di connessioni disponibili al database. I DataTransferObject, che implementano i relativi DataAccessObject, faranno richiesta di una connessione al Pool ogni qualvolta vorranno accedere al Database.

1. **Glossario**

**MySQL:** MySQL è un database relazionale (RDBMS) multithread open source, sviluppato nel 1996 da una società di consulenza svedese, la TcX, che aveva bisogno di un database veloce e che richiedeva poche risorse, pur dovendo gestire notevoli quantità di dati. La diffusione di MySQL è dovuta principalmente alla sua natura open source e gratuita, oltre alle sue doti di velocità e flessibilità.

**DBMS:** Un DBMS è sostanzialmente uno strato software che si frappone fra l’utente ed i dati veri e propri. Grazie a questo strato intermedio l’utente e le applicazioni non accedono ai dati così come sono memorizzati effettivamente, cioè alla loro rappresentazione fisica, ma ne vedono solamente una rappresentazione logica. Ciò permette un elevato grado di indipendenza fra le applicazioni e la memorizzazione fisica dei dati.

**RDBMS:** Il termine relational database management system (RDBMS) (sistema per la gestione di basi di dati relazionali) indica un database management system basato sul modello relazionale. Un database relazionale è costituito da tabelle, ognuna delle quali è composta da righe identificate da un codice univoco denominato chiave. Le tabelle che compongono il database non sono del tutto indipendenti tra loro ma relazionate da legami logici.

**Java:** Java è un linguaggio di programmazione ad alto livello, orientato agli oggetti e a tipizzazione statica, che si appoggia sull'omonima piattaforma software di esecuzione, specificamente progettato per essere il più possibile indipendente dalla piattaforma hardware di esecuzione.

**HTML:** HTML (HyperText Markup Language) è un linguaggio di markup. Oggi è utilizzato principalmente per il disaccoppiamento della struttura logica di una pagina web e la sua rappresentazione, gestita tramite gli stili CSS per adattarsi alle nuove esigenze di comunicazione e pubblicazione all'interno di Internet.

**Javascript:** In informatica JavaScript è un linguaggio di scripting orientato agli oggetti e agli eventi, comunemente utilizzato nella programmazione Web lato client per la creazione, in siti web e applicazioni web, di effetti dinamici interattivi tramite funzioni di script invocate da eventi innescati a loro volta in vari modi dall'utente sulla pagina web in uso.

**CSS:** Il CSS, in informatica, è un linguaggio usato per definire la formattazione di documenti HTML, XHTML e XML, ad esempio i siti web e relative pagine web.

**CamelCase**: la notazione a cammello, o in inglese CamelCase, è la pratica nata durante gli anni 70 di

scrivere parole composte o frasi unendo tutte le parole tra loro, ma lasciando le loro iniziali maiuscole. Si può distinguere in un lowerCamelCase, in cui la prima lettera della prima parola viene lasciata minuscola, o in UpperCamelCase, in cui la prima lettera della prima parola è maiuscola.

**Bootstrap:** Bootstrap è una raccolta di strumenti liberi per la creazione di siti e applicazioni per il Web. Essa contiene modelli di progettazione basati su HTML e CSS, sia per la tipografia, che per le varie componenti dell'interfaccia, come moduli, pulsanti e navigazione, così come alcune estensioni opzionali di JavaScript.