|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Object Design Document  Progetto  STORYTELLING   |  |  | | --- | --- | | Riferimento |  | | Versione | 0.3 | | Data | 27/12/2021 | | Destinatario | Prof. Gravino | | Presentato da | Alessandro Marigliano, Antonio Scotellaro, Emmanuele Virginio Coppola, Muriel Rossi. | | Approvato da |  | |

# RevisionHistory

|  |  |  |  |
| --- | --- | --- | --- |
| **Data** | **Versione** | **Descrizione** | **Autori** |
| 19/12/2021 | 0.1 | Prima stesura | Emmanuele Virginio Coppola |
| 26/12/2021 | 0.2 | Inserimento Packages, Interfacce delle classi, design pattern | Muriel Rossi, Alessandro Marigliano, Antonio Scotellaro, Emmanuele Virginio Coppola |
| 27/12/2021 | 0.3 | Inserimento design pattern, revisione | Muriel Rossi |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Team Members

|  |  |  |
| --- | --- | --- |
| **Nome** | **Acronimo** | **Informazioni di contatto** |
| Emmanuele Virginio Coppola | EVC | e.coppola37@studenti.unisa.it |
| Alessandro Marigliano | AM |  |
| Antonio Scotellaro | AS |  |
| Muriel Rossi | MR | m.rossi67@studenti.unisa.it |

Sommario

[Revision History 2](#__RefHeading___Toc699_1509244861)

[Team Members 3](#__RefHeading___Toc701_1509244861)

[1 Introduzione 4](#__RefHeading___Toc703_1509244861)

[1.1 Object design goals 4](#__RefHeading___Toc705_1509244861)

[1.2 Object Trade-Off 4](#__RefHeading___Toc707_1509244861)

[1.3 Linee guida per la documentazione dell’interfaccia 5](#__RefHeading___Toc709_1509244861)

[1.4 Definizioni, acronimi e abbrevizioni 5](#__RefHeading___Toc711_1509244861)

[1.5 Riferimenti 6](#__RefHeading___Toc713_1509244861)

[2 Packages 6](#__RefHeading___Toc715_1509244861)

[3 Interfacce delle classi 6](#__RefHeading___Toc717_1509244861)

[4 Design Patterns 7](#__RefHeading___Toc719_1509244861)

[5 Glossario 8](#__RefHeading___Toc721_1509244861)

# 1 Introduzione

Storytelling si propone di creare una piattaforma libera di pubblicità e …

In questa prima sezione del documento verranno descritti i trade-offs e le linee guida per l’implementazione , riguardante la nomenclatura, la documentazione e le convenzioni sui formati.

## 1.1 Object design goals

**Modularità:**

Il sistema Storytelling deve avere la possibilità di poter cambiare e aggiungere funzionalità senza dover modificare molto codice.

**Robustezza:**

Il Sistema deve risultare robusto, reagendo in maniera adeguata alle situazioni impreviste grazie alla gestione delle eccezioni evitando di bloccare le componenti non coinvolte del sistema.

**Riusabilità:**

Il Sistema deve basarsi sulla riusabilità del codice attraverso l’utilizzo delle interfacce e dei design pattern

## 1.2 Object Trade-Off

**Modularità vs Tempi di esecuzione:**

Il sistema sarà costruito in maniera tale da preferire ove possibile la modularità rispetto al tempo di Esecuzione.

**Robustezza vs Tempo di esecuzione:**

Il sistema è costruito per preferire effettuare i controlli necessari per rendere il sistema più robusto anche se ci sono tempi di esecuzione più lughi.

## 1.3 Linee guida per la documentazione dell’interfaccia

Le linee guida sono una serie di regolamenti usati dagli sviluppatori per la progettazione delle interfacce per facilitarne la comprensione anche per altri team di sviluppatori. Le linee guida sono definite in base alla convenzione Java **Sun Java Coding Conventions** [Sun, 2009].

Link a documentazione ufficiale sulle convenzioni

Di seguito una lista di link alle convenzioni usate per definire le linee guida:

**• Java Sun:** <https://checkstyle.sourceforge.io/sun_style.html>  
**• HTML:**<https://www.w3schools.com/html/html5_syntax.asp>

## 1.4 Definizioni, acronimi e abbrevizioni

Vengono riportati di seguito alcune definizioni presenti nel documento:

* **Package:** raggruppamento di classi, interfacce o file correlati;
* **Design pattern:** template di soluzioni a problemi ricorrenti impiegati per ottenere riuso e flessibilità;
* **Interfaccia:** insieme di signature delle operazioni offerte dalla classe;
* **View:** nel pattern MVC rappresenta ciò che viene visualizzato a schermo da un utente e che gli  
  permette di interagire con le funzionalità offerte dalla piattaforma;
* **lowerCamelCase:** è la pratica di scrivere frasi in modo tale che ogni parola o abbreviazione nel  
  mezzo della frase inizi con una lettera maiuscola, senza spazi o punteggiatura intermedi;
* **UpperCamelCase:** è la pratica di scrivere frasi in modo tale che ogni parola o abbreviazione inizi  
  con una lettera maiuscola, senza spazi o punteggiatura intermedi;
* **Javadoc:** sistema di documentazione offerto da Java, che viene generato sottoforma di interfaccia  
  in modo da rendere la documentazione accessibile e facilmente leggibile
* **IDE(Integated develoment enviroment):** software che sfrutta l’implementazione di tecnologie diverse per velocizzare implementazione e testing del codice.
* **POJO(Plain Old Java Object):** Oggetti java semplici che non sono legati a restrizioni speciali aumentando quindi leggibilità e riutilizzo del codice.
* **CACHE:** Archivio di dati temporanei consistenti con il database

## 1.5 Riferimenti

Di seguito una lista di riferimenti ad altri documenti utili durante la lettura:  
• Statement Of Work;  
• Business Case;  
• Requirements Analysis Document;  
• System Design Document;  
• Object Design Document;  
• Test Plan;  
• Matrice di tracciabilità;  
• Manuale di installazione;  
• Manuale utente;

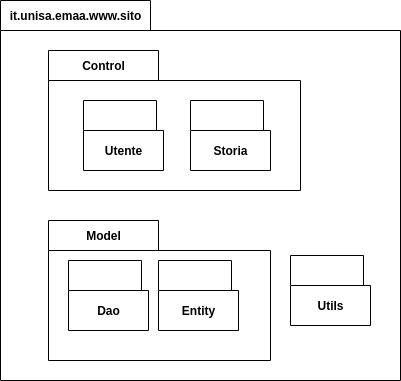
Per definizioni di design pattern applicati in ambienti java:

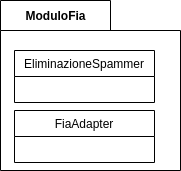
<https://www.oracle.com/java/technologies>

# 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 e di quella dei progetti generati dall’IDE **Intellij IDEA .**





# 

# 3 Interfacce delle classi

Package Utils

|  |  |
| --- | --- |
| **Nome Classe** | Validazione |
| Descrizione | Classe che si occupa delle operazioni di registrazione Utente e controllo della correttezza dei dati presentati |
| Metodi | +utenteIsPresent(Utente utente):Boolean presente  +emailRegex(String email):Boolean valid  +passwordHasher(String password):String hash |
| Invarianti di Classe | / |

|  |  |
| --- | --- |
| **Nome Metodo** | +utenteIsPresent(Utente utente):Boolean presente |
| Descrizione | Metodo che verifica la presenza di un’utente nel sistema |
| Pre-condizione | **/** |
| Post-Condizione | / |
| **Nome Metodo** | +emailRegex(String email):Boolean valid |
| Descrizione | Metodo che verifica se la stringa passata è un’email |
| Pre-condizione | **/** |
| Post-Condizione | / |
| **Nome Metodo** | +passwordHasher(String password):String hash |
| Descrizione | Metodo che trasforma una password in chiaro in un’hash |
| Pre-condizione | **/** |
| Post-Condizione | / |

Package Utente

|  |  |
| --- | --- |
| **Nome Classe** | RegistrazioneUtente |
| Descrizione | Classe che si occupa delle operazioni di registrazione Utente e controllo della correttezza dei dati presentati |
| Metodi | -controlloDati(String email,String password,String passwordTest,String nome,Boolean eula): Boolean corretto  -effettuaRegistrazione(Utente utente):void |
| Invarianti di Classe | / |

|  |  |
| --- | --- |
| **Nome Metodo** | controlloDati(String email,String password,String passwordTest,String nome,Boolean eula): Boolean corretto |
| Descrizione | Metodo che verifica la presenza di tutti i campi richiesti e che l'email passi la relativa regular expression come anche la password |
| Pre-condizione | **/** |
| Post-Condizione | / |
| **Nome Metodo** | effettuaRegistrazione(Utente utente): Boolean corretto |
| Descrizione | Metodo che effettua la registrazione dell’utente |
| Pre-condizione | **context:** effettuaRegistrazione(Utente utente): Boolean corretto  **pre:** RegistrazioneUtente::controlloDati(String email,String password,String passwordTest,String nome,Boolean eula)  &&  not Validazione::utenteIsPresent(Utente utente) |
| Post-Condizione | **context:** effettuaRegistrazione(Utente utente): Boolean corretto  **post:** Validazione::utenteIsPresent(Utente utente) |

|  |  |
| --- | --- |
| **Nome Classe** | Login |
| Descrizione | Classe che si occupa del login dell’utente |
| Metodi | -controllaUtente(String email,String password):Utente utente  -recuperaUtente(String email, String password:Boolean corretto |
| Invarianti di Classe | / |

|  |  |
| --- | --- |
| **Nome Metodo** | -controllaUtente(String email,String password):Boolean corretto |
| Descrizione | Metodo che verifica che la presenza e la corrispondenza dei dati con quelli presente nel datatbase |
| Pre-condizione | / |
| Post-Condizione | / |
| **Nome Metodo** | -recuperaUtente(String email):Utente utente |
| Descrizione | Metodo che preleva i dati dal database relativi all’utente effettuando il login e cancellando la password nella sessione |
| Pre-condizione | **context:** Login:: controllaUtente(email,password)  **pre:** UserDao::retrieveByEmail(email)==null |
| Post-Condizione | **context:** Login:: controllaUtente(email,password)  **post:** isLogged(utente)==true |

|  |  |
| --- | --- |
| **Nome Classe** | EliminazioneUtente |
| Descrizione | Classe che si occupa della rimozione dell'account |
| Metodi | -eliminaUtente(String email,String password):void  -//da controllare il controlla Utente |
| Invarianti di Classe | / |

|  |  |
| --- | --- |
| **Nome Metodo** | -eliminaUtente(String email,String password):void |
| Descrizione | Metodo che elimina i dati dell'utente dal database |
| Pre-condizione | **context:** Login:: controllaUtente(email,password)  **pre:** notUserDao::retrieveByEmail(email)==null |
| Post-Condizione | **context:** Login:: controllaUtente(email,password)  **post:** UserDao::retrieveByEmail(email)==null  &&  notisLogged(utente)==true |

Package Utente.Asincrono

|  |  |
| --- | --- |
| **Nome Classe** | ValidaEmail |
| Descrizione | Classe che si occupa del controllo della presenza di un'email all'interno del database chiamando |
| Metodi | -eliminaUtente(String email,String password):void  -//da controllare il controlla Utente |
| Invarianti di Classe | / |

Package Storia

|  |  |
| --- | --- |
| **Nome Classe** | VisualizzaBacheca |
| Descrizione | Classe che si occupa della visualizzazione della bacheca |
| Metodi | -recuperaListaStorie():List<Storia> Bacheca |
| Invarianti di Classe | / |

|  |  |
| --- | --- |
| **Nome Metodo** | -recuperaListaStorie():List<Storia> Bacheca |
| Descrizione | Metodo che recupera le storie dalla cache |
| Pre-condizione | / |
| Post-Condizione | / |

|  |  |
| --- | --- |
| **Nome Classe** | PubblicaStoria |
| Descrizione | Classe che si occupa pubblicazione di una storia |
| Metodi | -pubblicazioneStoria(Stringa storia,Utente utente):int idStoria  -aggiunta |
| Invarianti di Classe | / |

# 4 Design Patterns

Nella presente sezione si andranno a descrivere e dettagliare i design patterns utilizzati nello sviluppo dell’applicativo Storytelling.

Per ogni pattern si darà:  
• Una brevissima introduzione teorica.  
• Il problema che doveva risolvere all’interno di Storytelling.  
• Una brevissima spiegazione di come si è risolto il problema in Storytelling.  
• Un grafico della struttura delle classi che implementano il pattern.

**Adapter**

L’Adapter è un design pattern strutturale , ovvero una soluzione già provata nell’industria informatica che semplifica le relazioni tra le entità. In questo caso l’Adapter è un pattern che facilita la comunicazione tra il sistema Storytelling e il modulo fia.

Infatti il problema riguarda la comunicazione con il suddetto modulo fia che communica con il sistema con messaggi HTTP, rendendo quindi necessaria la creazione di una classe che permettesse di convertire i messaggi inviati dal modulo fia in oggetti java utilizzabili dal sistema e viceversa.

**DAO**

Un DAO (Data Access Object) è un pattern che offre un’interfaccia astratta nascondendo i dettagli implementativi del database usato dagli utilizzatori.

Il problema è la quantità di dati da gestire e la modularità del sistema e quindi permettere di poter cambiare facilmente il database implementato nel sistema non cambiando l’interfaccia mostrato ai sottosistemi utilizzatori del database rendendo quindi la gestione dei dati semplice e anche facilmente modificabile.

I DAO si occuperanno di salvare e leggere i messaggi all’interno del database rendendo maneggevoli le entità Utenti,Commenti, Reazioni, Storie implementandoli con dei POJO che possono essere generati automaticamente durante l’implementazione grazie all’implementazione standard dell’IDE **intellij Idea.**

**Bridge**

Il Bridge è un particolare design pattern strutturale che consente di dividere set di classi accoppiate, in due gerarchie separate (ovvero una di astrazione ed una di implementazione), in modo da poterle sviluppare indipendentemente, implementando un'astrazione della classe e del comportamento della classe che la implementa concretamente. E' stato utilizzato all'interno del progetto E.M.A.A. per introdurre maggiore modularità fra le classi, che risultavano fortemente accoppiate e permettendo così di utilizzare classi ancora non completamente sviluppate ma necessarie ad altri concetti.

# 5 Glossario