

|  |
| --- |
| Object Design Document  Versione 1.0  **ANNO ACCADEMICO 2020/2021** |
|  |
| RISTOMANAGER  Ambrosio Salvatore  Costante Marco  Benitozzi Simone  Nappo Carla Alessia |



PARTECIPANTI

|  |  |
| --- | --- |
| **NOME** | **MATRICOLA** |
| Salvatore Ambrosio | 0512106166 |
| Costante Marco | 0512105772 |
| Benitozzi Simone | 0512105742 |
| Nappo Carla Alessia | 0512105956 |

REVISION HISTORY

|  |  |  |  |
| --- | --- | --- | --- |
| **DATA** | **VERSIONE** | **DESCRIZIONE** | **AUTORE** |
| 19/01/2021 | 1.0 | Prima stesura del template | Benitozzi Simone |
| 19/01/2021 | 1.1 | Definizione di Introduzione e Design Pattern | Benitozzi Simone |
| 25/01/2021 | 1.2 | Correzioni e aggiunta dettagli | Costante Marco |

# INDICE

[INDICE 4](#_Toc61968727)

[1.Introduzione 5](#_Toc61968728)

[1.1 Object Design Trade-offs 5](#_Toc61968729)

[1.2 Componenti off-the-shelf 6](#_Toc61968730)

[1.3 Linee Guida 7](#_Toc61968731)

[1.3.1 Package 7](#_Toc61968732)

[1.3.2 Naming Convention 7](#_Toc61968733)

[1.3.3 Variabili 8](#_Toc61968734)

[1.3.4 Costanti 8](#_Toc61968735)

[1.3.5 Metodi 8](#_Toc61968736)

[1.3.6 Classi e Pagine 8](#_Toc61968737)

[1.4 Acronimi 9](#_Toc61968738)

[1.5 Riferimenti 10](#_Toc61968746)

[2. Design Patterns 11](#_Toc61968747)

[2.1 Model-View-Controller 11](#_Toc61968748)

[2.2 Data Access Object 11](#_Toc61968749)

[3. Packages 12](#_Toc61968751)

[4. Class Interfaces 13](#_Toc61968752)

[5. Class Diagram Completo 14](#_Toc61968753)

# 1.Introduzione

## **1.1 Object Design Trade-offs**

* Usabilità vs Funzionalità: Il sistema dovrà essere prima di tutto intuitivo e di facile comprensione da parte dell’utente, questo anche al costo di rinunciare alla implementazione di alcune funzionalità che potrebbero complicarne l’utilizzo.
* Tempi di Sviluppo vs Sicurezza: Il meccanismo di sicurezza implementato, a causa dei tempi e le risorse limitate, prevederà un semplice sistema di login e differenziazione degli utenti in ruoli, con filtri per il controllo degli accessi ad operazioni dedicate.
* Comprensibilità vs Costi: Il codice sviluppato dovrà risultare quanto più possibile comprensibile e chiaro, questo a costo di un maggiore tempo dedicato alla scrittura, allo scopo di avere come risultato un codice riutilizzabile e integrabile in futuro con modifiche e aggiornamenti.

## **1.2 Componenti off-the-shelf**

Per l’implementazione del sistema verrà fatto uso di componenti software ausiliari che ne facilitino lo sviluppo

* //Bootstrap, Bulma, Jquery, AJAX, JSON ???
* JUnit: framework utilizzato per il test di unità delle componenti implementate.
* Mockito: framework di test open source per Java per il test di componenti lato server.
* Selenium: suite di tool utilizzati per automatizzare i test di sistema eseguendoli sul web browser.

## **1.3 Linee Guida**

Per la produzione di un codice comprensibile ed uniforme, che favorisca il riutilizzo e futuri aggiornamenti, andranno seguite nella sua stesura le linee guida di seguito elencate:

## **1.3.1 Package**

Il codice sorgente deve essere opportunamente suddiviso in package, per modellare i sottosistemi che compongono il sistema, che minimizzino l’accoppiamento tra le classi ed evidenzino la coesione tra classi con comportamenti simili.

A tale proposito, i package principali nei quali le classi verranno suddivise, comprensivi di eventuali sub-package per un maggiore partizionamento, sono i seguenti:

* Control: contenente le classi Servlet, quindi i comportamenti lato server del sistema.
* Model: contenente le classi che definiscono i Bean del sistema, i quali possono rappresentare sia entità persistenti nel DB, che oggetti temporanei utili per l’esecuzione delle funzionalità del programma a runtime.
* DAO: contenente le classi che implementano il pattern architetturale DAO, per la gestione della persistenza, le quali presentano dalle più semplici operazioni CRUD, a più complesse query di interrogazione al DB.
* Test: contenente le classi utilizzate per il testing del sistema.

## **1.3.2 Naming Convention**

Per la documentazione delle interfacce sarà necessario seguire una naming convention precisa, nel rispetto delle classiche notazioni dell’Object Oriented, che porti ad adottare nomi:

* Descrittivi
* Intuitivi e comprensibili
* Di lunghezza medio-corta
* Comprensivi possibilmente di carattere alfabetici, evitando quelli numeri ove possibile

## **1.3.3 Variabili**

Le variabili dovranno essere identificate attraverso la classica “camelCase” notation, evitando underscore laddove il nome sia costituito da più parole.

La dichiarazione delle stesse dovrà essere chiara e possibilmente separata dall’inizializzazione per garantire un migliore leggibilità del codice.

## **1.3.4 Costanti**

Le costanti dovranno invece essere identificate attraverso una “UPPER\_CASE” notation, utilizzando underscore laddove il nome sia costituito da più parole.

La dichiarazione delle stesse dovrà essere chiara e coincidente con l’inizializzazione, pertanto non dovranno esserne dichiarate più di una nello stesso rigo, seppur dello sesso tipo, questo per garantire un migliore leggibilità del codice.

## **1.3.5 Metodi**

I metodi dovranno essere identificati attraverso la classica “camelCase” notation.

Il nome dei metodi sarà tipicamente costituito da un verbo che ne specifica l’azione e l’oggetto su cui essa viene applicata.

Tutti i metodi saranno commentati secondo la documentazione JavaDoc.

## **1.3.6 Classi e Pagine**

Le classi e le pagine dovranno essere identificate attraverso la classica “CamelCase” notation, con l’iniziale maiuscola, per distinguerle da attributi e metodi.

Il loro nome dovrà fornire una descrizione intuitiva di quello che è lo scopo ed essere coerente con le operazioni implementate al loro interno.

## **1.4 Acronimi**

# RAD: Requirements Analysis Document

# SDD: System Design Document

# ODD: Object Design Document

# OCL: Object Constraints Language

# DB: Database

# DAO: Data Object Access

# MVC: Model View Controller

## **1.5 Riferimenti**

* Bern Bruegge, Allen H. Dutoit, Object-Oriented Software Engineering - Using UML, Patterns, and JAVA, 3rd edition.
* RAD\_RistoManager
* SDD\_RistoManager

# 2. Design Patterns

## **2.1 Model-View-Controller**

A seguito delle osservazioni fatte nel SDD, abbiamo ritenuto opportuno utilizzare, nello sviluppo del sistema, il Design Pattern Model-View-Controller, il quale prevede che i sottosistemi appartengano a 3 distinte categorie, ciascuna delle quali con un compito diverso.

In particolare:

Il model si occupa della di gestire i dati e quindi sarà responsabile dell’interazione con il database.

La view si occupa di curare l’interazione con l’utente e quindi avrà il compito di gestire la formattazione dei dati che verranno visualizzati.

Il controller dopo aver ricevuto i comandi forniti dall’utente si occuperà di elaborare i dati, passarli al model se necessario e inviare la risposta al view appropriato.

## **2.2 Data Access Object**

Il DAO è un design pattern per la gestione della persistenza, che prevede l’implementazione di classi che presentano dalle più semplici operazioni CRUD, a più complesse query di interrogazione al DB, che recuperano quindi entità persistenti, tramutandole in oggetti Java utilizzabili per la logica di business.

Il vantaggio relativo all'uso del DAO è dunque il mantenimento di una rigida separazione tra le componenti di un'applicazione, e si presta quindi in maniera ottimale al supporto di un'applicazione basata sul paradigma MVC.

# 3. Packages

# 4. Class Interfaces

# 5. Class Diagram Completo