Università degli studi di Salerno

*Corso di Laurea in Informatica*

**Object Design Document**

**“fasTrain”**



**Autori:**

**Matricola**

0512104934

0512104814

0512104826

**Nome**

Acierno Erminio

Avallone Daniele

Paolillo Domenico

**1. Introduzione**

Lo scopo di questo documento è la realizzazione di un modello capace di integrare tutte le diverse funzionalità individuate durante lo sviluppo dei documenti precedenti.

Si presentano delle linee guida sull’interfaccia grafica, le classi, le operazioni, i tipi, gli argomenti e i sottosistemi definiti nel System Design Document.

**1.1 Object Design Trade-offs**

* **Comprensibilità vs. Tempo**

Il codice deve essere di chiara lettura e di facile comprensione, per poter facilitare la lettura, le modifiche ed il testing. Il codice quindi sarà provvisto di commenti, che chiariranno le varie funzionalità implementate.

* **Interfaccia vs. Usabilità**

L’interfaccia grafica del sistema verrà sviluppata in modo da renderne semplice e chiaro l’utilizzo.

Saranno quindi presenti menu, form, pulsanti e finestre di feedback in grado rendere l’utente finale in grado di navigare facilmente e consapevolmente all’interno del sistema.

* **Sicurezza vs Efficienza**

Per quanto possa essere estremamente importante la sicurezza, ci limiteremo ad un sistema username-password.

**1.2 Linee Guida per la Documentazione delle Interfacce**

**Naming Convention**

È buona norma usare nomi:

* Descrittivi
* Pronunciabili
* Di uso comune
* Di lunghezza medio-corta
* Non Abbreviati
* Utilizzando solo caratteri i consentiti: lettere maiuscole e minuscole dalla A alla Z e numeri dallo 0 al 9.

**Variabili**

I nomi delle variabili devono rispettare degli standard:

* incominciare con una lettera maiuscola;
* essere dichiarate all’inizio in blocco;
* occupare solamente una riga;
* essere facili da leggere;
* essere accompagnate da commenti qualora non fosse chiaro il loro scopo.

**Metodi**

I nomi dei metodi devono rispettare degli standard:

* incominciare con una lettera maiuscola;
* il nome che identifica il metodo è un verbo che suggerisce l’operazione che svolge il metodo;
* i metodi delle variabili “bean” dovranno essere del tipo get/set + NomeVariabile().

**Classi e pagine**

I nomi delle classi devono rispettare degli standard:

* devono incominciare con una lettera maiuscola;
* i loro nomi devono fornire informazioni circa il loro scopo.

La dichiarazione dei una classe deve avvenire nel seguente modo:

* Dichiarazione classe pubblica;
* Dichiarazione costanti;
* Dichiarazioni di classe;
* Dichiarazioni di variabili d’istanza;
* Costruttore;
* Commento e dichiarazione metodi.

**1.3 Definizioni, acronimi e abbreviazioni**

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

**1.4 Riferimenti**

* B.Bruegge, A. H. Dutoit, Object Oriented Software Engineering - Using UML, Pattern and Java, Prentice Hail, 3rd edition, 2009.
* Documento RAD.

**2. Design pattern**

* **Singleton**

Il pattern Singleton viene utilizzato quando si vuole garantire di avere un unico punto di accesso. Ad esempio, esso viene utilizzato quando si desidera avere un solo Window Manager oppure una sola Coda di Stampa oppure un unico accesso al database.

Un oggetto Singleton viene inizializzato nel momento in cui la classe viene invocata attraverso la definizione di un oggetto statico. La visibilità del suo costruttore viene modificata da public a private, così che non sia possibile istanziare la classe dall’esterno e fare in modo che soltanto la classe può istanziare sé stessa.

* **MVC**

In fase di system design, si è stabilito che il sistema proposto presenta l’architettura Model - View - Controller (MVC). Il pattern MVC viene utilizzato in un contesto dove l’applicazione deve fornire un’interfaccia grafica costituita da più schermate che mostrano vari dati all’utente, i quali devono risultare aggiornati in qualunque momento.

La soluzione fornita da questo pattern risulta essere particolarmente adatta al sistema proposto, poiché prevede che l’applicazione debba separare i componenti software che implementano le funzionalità di business dai componenti che implementano la logica di presentazione e di controllo, i quali utilizzano tali funzionalità.

I componenti previsti dal pattern MVC sono descritti di seguito nel dettaglio:

* Il **Model** definisce le regole di business per l'interazione con i dati, esponendo alla View ed al Control rispettivamente le funzionalità per l'accesso e l'aggiornamento dei dati.
* Il **Control** realizza la corrispondenza tra l’input dell'utente e i processi eseguiti dal Model, oltra a selezionare le schermate della View richieste ed implementare la logica di controllo dell'applicazione.
* La **View** si occupa della logica di presentazione dei dati

**3. Packages**

Il package fasTrain contiene sottopackage che a loro volta inglobano classi atte alla gestione delle richieste utente. Le classi contenute nel package svolgono il ruolo di gestore logico del sistema.

Il sistema presenta una suddivisione basata su tre livelli (three-tier):

● **Interface Layer**: livello di interfaccia visualizzato dall’utente finale offre la possibilità di interagire con il sistema ricevendo input e restituendo output

● **Application Logic Layer**: elabora i dati da inviare al client e quelli ricevuti dal database tramite lo storage layer. Si occupa delle gestioni di: utenti, corse, biglietti.

● **Storage Layer**: memorizza tutti i dati di compilazione inseriti nel sistema attraverso un DBMS e risponde alle richieste dell’application logic layer restituendo i dati richiesti.

**3.1 Package View**

|  |  |
| --- | --- |
| CLASSE | DESCRIZIONE |
| BigliettiAcquistati.jsp | Pagina che mostra la lista dei biglietti acquistati |
| CompletaAcquisto.jsp | Pagina della fase di pagamento acquisto |
| ModificaCorsa.jsp | Pagina che permette all’admin di modificare le informazioni di una corsa |
| RisultatiRicerca.jsp | Pagina che mostra i risultati della ricerca effettuata |
| Homepage.jsp | Pagina iniziale |
| InfoBiglietto.jsp | Pagina che mostra le informazioni di una corsa |
| InserisciCorsa.jsp | Pagina che permette all’admin di inserire una nuova corsa |
| Login.jsp | Pagina che permette all’utente di loggare |
| ModificaBiglietto.jsp | Pagina che permette all’utente di modificare alcune informazioni sul biglietto acquistato |
| ModificaDatiUtente.jsp | Pagina che permette all’utente di modificare i dati inseriti in fase di registrazione |
| Offerte.jsp | Pagina che mostra tutte le corse in offerta |
| Registrazione.jsp | Pagina che permette all’utente di registrarsi al sistema |
| LaNostraStoria.jsp | Pagina che permette all’utente di visualizza la storia del sito |
| InfoPrezzi.jsp | Pagina che permette all’utente di visualizzare i prezzi offerti dal sito |
| Contattaci.jsp | Pagina che mostra all’utente come contattare i gestori del sito in caso di bisogno. |

**3.2 Package Model**

|  |  |
| --- | --- |
| CLASSE | DESCRIZIONE |
| DriverManagerConnectionPool.java | Classe che si occupa di gestire un pool di connessioni al database. |
| PrenotazioneDAO.java | Classe contenente i metodi che permettono di effettuare inserimento, ricerca e cancellazione di una prenotazione. |
| PasseggeroDAO.java | Classe contenente i metodi che permettono di effettuare inserimento, ricerca e cancellazione di un passeggero. |
| CorsaDAO.java | Classe contenente i metodi che permettono di effettuare inserimento, ricerca e cancellazione di una corsa. |
| CorsaBean.java | Classe contenente tutti i metodi relativi all'ogetto corsa. |
| PrenotazioneBean.java | Classe contenente tutti i metodi relativi all'ogetto Prenotazione. |
| PasseggeroBean.java | Classe contenente tutti i metodi relativi all'ogetto Passeggero. |
| UtenteBean.java | Classe contenente tutti i metodi relativi all'ogetto Utente. |
| UtenteDAO.java | Classe contenente i metodi che permettono di effettuare inserimento e login di un Utente. |

**3.3 Package Control**

|  |  |
| --- | --- |
| CLASSE | DESCRIZIONE |
| LoginServlet.java | Questa servlet si occupa di ricevere i dati di login, elaborarli e decidere se consentire o meno l’accesso all’area personale. |
| LogoutServlet.java | Questa servlet si occupa di invalidare la sessione per consentire il log out. |
| AggiungiCorsaServlet.java | Questa servlet si occupa di aggiungere corse all’interno del DB. |
| EliminaCorsaServlet.java | Questa servlet si occupa di eliminare una corsa all’interno del DB. |
| ModificaCorsaServlet.java | Questa servlet si occupa di modificare dettagli relativi ad una corsa nel DB. |
| AggiungiPrenotazioneServlet.java | Questa servlet si occupa di aggiungere una prenotazione e la lista dei suoi passeggeri all’interno del DB. |
| ModificaPasseggeroServlet.java | Questa servlet si occupa di modifica i dettagli relativi ad un o più passeggeri nel DB |
| ModificaUtenteServlet.java | Questa servlet si occupa di modificare dettagli relativi ad un utente registrato nel DB. |
| RicercaCorsaServlet.java | Questa servlet si occupa di ricercare una corsa all’interno del DB. |
| RegistrazioneServlet.java | Questa servlet si occupa di ricevere i dati relativi alla registrazione e di inviarli al DB. |
| VisualizzaOfferteServlet.java | Questa servlet si occupa di ricercare una corsa in offerta all’interno del DB. |
| VisualizzaBigliettiAcquistatiServlet.java | Questa servlet si occupa di ricercare le prenotazioni effettuate dall’utente. |
| InformazioniCorsaServlet.java | Questa servlet si occupa di restituire le informazioni del singolo biglietto |
| CompletaAcquistoServlet.java | Questa servlet consente l’inizio della gestione della prenotazione. |

**4. Class Interfaces**

|  |  |
| --- | --- |
| Nome | UtenteDAO |
| doSave | * Context   utenteDAO: doSave(utenteRegistrato)   * Pre   utente!=null &&  utente.Email!=null &&  utente.Nome!=null &&  utente.Cognome!=null &&  utente.DataDiNascita!=null &&  utente.NumeroTelefono!=null &&  utente.Citta!=null &&  utente.Provincia!=null &&  utente.Indirizzo!=null &&  utente.Password!=null  utente->forAll(u | u.email != Utente.Email);   * Post   Utente->include(utente) |
| doRetreiveByKey | * Context   UtenteDAO: doRetreiveByKey(email)   * Pre   email!=null &&  utente->exixsts(u | u.email==email)   * Post   utente->select(u | u.email==email); |
| doUpdate | * Context   UtenteDAO:doUpdate(Utente utente)   * Pre   email!=null&&utente-> exists(u|u.email==utente.email)   * Post   utente->update(Utente) |
| doCheckForEmail | * Context   UtenteDAO: doCheckForEmail(email)   * Pre   email!=null &&  select(email | u.email=email)   * Post   If Result(select) == true -> return true;  else return false |

|  |  |
| --- | --- |
| Nome | CorsaDAO |
| doSave | * Context   CorsaDAO: doSave(corsa)   * Pre   corsa!=null &&  corsa.codiceCorsa!=null &&  corsa.codiceTreno!=null &&  corsa.stazionePartenza!=null &&  corsa.stazioneArrivo!=null &&  corsa.oraPartenza!=null &&  corsa.oraArrivo!=null &&  corsa.numeroPostiEconomy!=null &&  corsa. numeroPostiPremium!=null &&  corsa. numeroPostiBusiness!=null &&  corsa.prezzoClassePremium!=null &&  corsa.prezzoClasseEconomy!=null &&  corsa.prezzoClasseBusiness!=null &&  corsa.sconto!=null&&  corsa.data!=null  corsa->forAll(c|c.codiceCorsa!= Corsa.codiceCorsa);   * Post   Corsa->include(corsa) |
| doRetreiveByKey | * Context   CorsaDAO: doRetreiveByKey(codiceCorsa)   * Pre   codiceCorsa!=null &&  corsa->exixsts(c| c.codidceCorsa==codiceCorsa)   * Post   corsa->select(c | c.codidceCorsa==codiceCorsa); |
| doUpdate | * Context   CorsaDAO:doUpdate(Corsa corsa)   * Pre   corsa!=null&&  corsa-> exists(c|c.codiceCorsa==codiceCorsa)   * Post   corsa->update(Corsa) |
| doRetreiveByDate | * Context   CorsaDAO: doRetreiveByDate(data,stazioneP,stazioneA)   * Pre   data!=null &&  corsa->exixsts(c| c.data==data, stazioneP==c.stazioneP, stazioneA==c.stazioneA)   * Post * corsa->select(c | c.data==data && stazioneP==c.stazioneP && stazioneA==c.stazioneA); |
| doCheckForCodice | * Context   CorsaDAO:doCheckForCodice(codiceCorsa)   * Pre   codiceCorsa!=null &&  select(c|c.codiceCorsa=codiceCorsa)   * Post   If Result(select) == true -> return true;  else return false |

|  |  |
| --- | --- |
| Nome | PasseggeroDAO |
| doUpdate | * Context   PasseggeroDAO:doUpdate(Paseggero passeggero)   * Pre   codiceDocumento!=null&&passeggero-> exists(p|p.codiceDocumento==passeggero.codiceDocumento)   * Post   passeggero->update(Passeggero) |
| doDelete | * Context: doDelete(codicePrenotazione) * Pre:   passeggero -> exist(p|p.codicePrenotazion==codicePrenotazione);   * Post:   !passegggero-exist(p|p.codicePrenotazione==codicePrenotazione); |
| doRetreiveByForeingKey | Context  PasseggeroDAO: doRetreiveByForeingKey(codicePrenotazione)   * Pre   codicePrenotazione!=null &&  passeggero->exixsts(p| p.codicePrenotazione == codicePrenotazione)   * Post   corsa->select(p | p.codicePrenotazione == codicePrenotazione); |
| doGetCodice | * Context   PrenotazioneDAO: doGetCodice(codiceCorsa)   * Pre   codiceCorsa!=null &&  prenotazione->exixsts(p| p.codidceCorsa==codiceCorsa)   * Post   prenotazione->select(p | p.codiceCorsa==codiceCorsa); |
| getMaxCode | * Context   PrenotazioneDAO: doGetCodice(codicePrenotazione)   * Pre   codicePrenotazione!=null &&  prenotazione->exixsts(p| p.codicePrenotazione==codicePrenotazione)   * Post   prenotazione->select(MAX(p | p.codicePrenotazione==codicePrenotazione)); |

|  |  |
| --- | --- |
| Nome | PrenotazioneDAO |
| doUpdate | * Context   PrenotazioneDAO:doUpdate(Prenotazione prenotazione)   * Pre   codicePrenotazione!=null&&prenotazione-> exists(p|p.codicePrenotazione==prenotazione.codicePrenotazione)   * Post   prenotazione->update(Prenotazione) |
| doDelete | * Context: doDelete(codicePrenotazione) * Pre:   prenotazione -> exist(p|p.codicePrenotazion==codicePrenotazione);   * Post:   !prenotazione-exist(p|p.codicePrenotazione==codicePrenotazione); |
| doRetreiveByForeingKey | Context  PasseggeroDAO: doRetreiveByForeingKey(codicePrenotazione)   * Pre   codicePrenotazione!=null &&  passeggero->exixsts(p| p.codicePrenotazione == codicePrenotazione)   * Post   corsa->select(p | p.codicePrenotazione == codicePrenotazione); |