Università degli Studi di Salerno

Corso di Ingegneria Del Software

Eat & Reorder

Object Design

Document

**Docente:**

Prof. Andrea De Lucia

**Studenti:**

Abate Francesco

Dello Buono Marco

De Martino Vincenzo

Gagliardi Rosario

Sommario

[1. Introduzione 3](#_Toc25831082)

[1.1. Trade-offs 5](#_Toc25831093)

1. Introduzione

Dopo aver stilato il documento di Requirements Analysis e il documento di System Design, occorre concentrarsi sugli aspetti implementativi. L’Object Design Document (ODD) ha come obiettivo quello di produrre un modello che sia in grado di unire tutte le informazioni accumulate nelle fasi precedenti. In tale documento verranno definite le interfacce delle classi, le operazioni supportate, i tipi dei dati, i parametri delle procedure, i signatures dei sottosistemi definiti nel documento di System Design, i trade-offs e le linee guida, per evitare compromessi di progettazione.

1.1. Trade-offs

**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 materiali reperibili in maniera gratuita, come un server open source e un ambiente di sviluppo,partendo da zero minimizzando così i costi e rendendo il sistema più che soddisfacente.

**Perfomance VS Usabilità**

L’interfaccia, grazie all’utilizzo delle form e di un cammino semplice e intuitivo, permette un uso facilitato del sistema per tutti i tipi di utenti all’interno della nostra piattaforma. E’ stata preferita l’usabilità per rendere l’esperienza d’utilizzo il più agevole possibile rendendo l’interfaccia chiara e pulita. Il sistema deve garantire tempi di risposta brevi per ogni funzionalità.

**Interfaccia VS Tempo di risposta**

Il tempo di risposta tra server e interfaccia sono rapidi per soddisfare le esigenze di tutti i tipi di utenti collegati al sistema, negli input degli utenti verrà fatto un controllo più meticoloso dei dati andando a specificare i parametri errati di caso di insuccesso.

## **1.2. Linee guida per la Documentazione delle Interfacce**

Per l’implementazione del sistema saranno utilizzate le seguenti convenzioni.

**Naming Convention:**

È buona norma utilizzare nomi:

* Descrittivi
* Pronunciabili
* Di uso comune
* Di lunghezza medio-corta
* Non abbreviati
* Evitando la notazione ungherese
* Utilizzando solo caratteri consentiti (a-z, A-Z, 0-9)

**Variabili:**

* I nomi delle variabili devono iniziare con la lettera minuscola, e le parole successive con la lettera maiuscola. La dichiarazione delle variabili deve essere effettuata ad inizio blocco; in ogni riga vi deve essere una sola dichiarazione di variabile e va effettuato l’allineamento per migliorare la leggibilità.
* In determinati casi, è possibile utilizzare il carattere underscore “\_”, ad esempio quando si fa uso di variabili costanti oppure quando si fa uso di proprietà statiche.

**Metodi:**

* I nomi dei metodi devono iniziare con la lettera minuscola, e le parole successive con la lettera maiuscola. Di solito il nome del metodo è costituito da un verbo che identifica un’azione, seguito dal nome di un oggetto. I nomi dei metodi per l’accesso e la modifica delle variabili dovranno essere del tipo getNomeVariabile() e setNomeVariabile(). Se viene dichiarata una variabile all’interno di un metodo quest’ultima deve essere dichiarata appena prima del suo utilizzo e deve servire per un solo scopo, per facilitare la leggibilità. Esempio: getId(), setId()
* Ai metodi va aggiunta una descrizione, la quale deve essere posizionata prima della dichiarazione del metodo, e deve descriverne lo scopo. La descrizione del metodo deve includere anche informazioni riguardanti gli argomenti, il valore di ritorno, le eccezioni. I metodi devono essere raggruppati in base alla loro funzionalità.

**Classi e pagine:**

* I nomi delle classi e delle pagine devono iniziare con la lettera maiuscola, e anche le parole successive all’interno del nome devono iniziare con la lettera maiuscola. I nomi delle classi e delle pagine devono essere evocativi, in modo da fornire informazioni sullo scopo di quest’ultime. Ogni file sorgente .java contiene una singola classe e dev’essere strutturato in un determinato modo:
* Una breve introduzione alla classe che ne specifichi lo scopo

/\*\*

\* sommario dello scopo della classe.

\*

\*

\*

\*/

* L’istruzione import che permette di importare all’interno della classe gli altri oggetti che la classe utilizza.
* La dichiarazione di una classe è caratterizzata da:

1. Dichiarazione della classe pubblica
2. Dichiarazioni di costanti
3. Dichiarazioni di variabili di classe
4. Dichiarazioni di variabili d’istanza
5. Costruttore
6. Commento e dichiarazione metodi e variabili

## **1.3. Definizioni, acronimi e abbreviazioni**

**Acronimi:**

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

**Abbreviazioni:**

* DB: DataBase

## **1.4. Riferimenti**

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

## **1.5. Design Pattern**

Data Access Object

È un pattern architetturale che consente di separare i servizi della logica di business

dalle operazioni per la gestione dei dati persistenti. La funzionalità di questo pattern è

nascondere dall’applicazione tutte le complessità coinvolte nell’esecuzione dell’operazioni che

interagiscono con la sorgente dei dati in modo da permettere ad entrambi i livelli di evolversi

separatamente senza sapere nulla dell’altro.

Il DAO implementa il meccanismo di accesso richiesto per lavorare con la sorgente

dei dati. I moduli della logica di business utilizzano l’interfaccia esposta dal DAO, tale

interfaccia definisce le operazioni standard da eseguire. La classe DAO concreta che

implementerà l’interfaccia è responsabile dell’archivazione e restituzione dei dati. Per

archiviare i dati recuperati dalla classe DAO si utilizzano gli oggetti Bean che contengono i

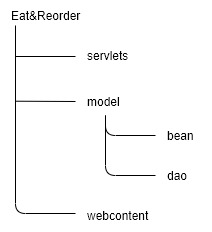
metodi get/set per memorizzare i dati recuperati.

Il design pattern sarà usato per la gestione della persistenza dei dati. L’oggetto del

controller comunicherà con l’interfaccia DAO che verrà implementata dalle classi dao specifiche.

## **2. Packages**

La decomposizione dei sottosistemi in packages è la seguente:



Il package servlets contiene al suo interno tutte le servlet che rappresentano i servizi offerti dai sottosistemi Gestione Utente e Gestione Ordini individuati nel SDD;

Il package model contiene al suo interno tutti i bean e tutti i dao, i quali rappresentano i servizi offerti dai sottosistemi. Ovviamente, i bean e i dao sono raccolti in appositi package contenuti nel package model;

Il package webcontent contiene tutte le componenti che raffigurino la presentazione del sistema, quindi contiene html e jsp.

La suddivisione dei package in servlets, model e webcontent è motivata dall’utilizzo dell’architettura MVC, la quale permette di separare la logica di business (servlets) dalla presentazione (webcontent) e dalla gestione dei dati (model). Il package model è a sua volta suddiviso in bean e dao. Le servlet nel package servlets si occuperanno dell’elaborazione dei dati.

## 2.1. Servlets package

Nel package servlets sono presenti tutte le servlet che implementano la logica di business e che tramite le interfacce dao, contenute nell’apposito package, comunicheranno con le jsp per la gestione dei contenuti nei bean, presenti nell’apposito package.

Di seguito vediamo tutte le classi contenute in tale package, suddivise in base al sottosistema.

*Sottosistema “Gestione Utente”:*

* RegistrazioneCliente.java
* RegistrazioneAzienda.java
* RegistrazioneFattorino.java
* Login.java
* ModificaProfiloCliente.java
* ModificaProfiloAzienda.java
* ModificaProfiloFattorino.java
* CreaSegnalazione.java
* PannelloBan.java
* VisualizzaProfilo.java
* Logout.java
* RicercaAziende.java
* ModificaProdotto.java
* RimuoviProdotto.java
* InserimentoProdotto.java
* VisualizzaListino.java

*Sottosistema “Gestione Ordini”:*

* AggiungiAlCarrello.java
* ModificaQuantitaProdotto.java
* RimuoviProdottoCarrello.java
* VisualizzaCarrello.java
* Ordinazione.java
* VisualizzaOrdiniPreparazione.java
* VisualizzaDettagliOrdine.java
* VisualizzaConsegne.java
* ConfermaRitiro.java
* ConfermaConsegna.java

[Altre verranno aggiunte proseguendo]

## 2.2. Bean package

Nel package bean sono presenti tutti gli oggetti entity che definiscono gli oggetti di dominio e i quali saranno utilizzati come bean dalle servlet presenti nel package servlets.

## **3. Interfacce delle classi**

|  |  |
| --- | --- |
| Nome Classe | Carrello |
| Descrizione | Classe che gestisce il carrello |
| Pre-condizioni | **Context**::**Carrello** public aggiornaQtaCarrello(prod : Prodotto, qta : Int)  **pre**: prod !=null and qta > 0 and prodotti->include(p | p.getCodice() = prod.getCodice())  **Context**::**Carrello** public rimuoviProdotto(prod : Prodotto)  **pre**: prod !=null and prodotti->include(p | p.getCodice() = prod.getCodice())  **Context**::**Carrello** public svuota()  **pre**: prodotti->size()>0  **Context**::**Carrello** public checkAziendaCarrello(prod: Prodotto)  pre: prod != null  **Context**::**Carrello** public checkInCarrello(prod: Prodotto)  **pre**: prod != null  **Context**::**Carrello** public aggiungiProdotto(prod: Prodotto)  **pre**: prod.getAzienda() = self.getCurrentAzienda() |
| Post-condizioni | **Context**::**Carrello** public aggiornaQtaCarrello(prod : Prodotto, qta : Int)  **post**: self.prodotti->include(p| p.getCodice() = prod.getCodice() and p.getQta() = qta)  **Context**::**Carrello** public rimuoviProdotto(prod : Prodotto)  **post**: prodotti-> !include(p | p.getCodice() = prod.getCodice())  **Context**::**Carrello** public svuota()  **post**: prodotti->size() = 0 and self.CurrentAzienda() = null  **Context**::**Carrello** public checkAziendaCarrello(prod: Prodotto)  **post:**  **Context**::**Carrello** public checkInCarrello(prod: Prodotto)  **post:**  **Context**::**Carrello** public aggiungiProdotto(prod: Prodotto)  **post**: prodotti->include(p | p.getCodice() = prod.getCodice()) |
| Invarianti | **Context::Carrello** **inv** self.prodotti->forAll(p | p.getAzienda() = self.getCurrentAzienda()) |

|  |  |
| --- | --- |
| Nome Classe | AccountAzienda |
| Descrizione | Classe che rappresenta l’account di una Azienda |
| Pre-condizioni | Context::AccountAzienda public dammiProdotto(id :Long)  **pre**: id != null and prodotti->include(p | p.getCodice() = id)  Context::AccountAzienda public aggiungiProdotto(prod : Prodotto)  **pre**: prod != null and prodotti->!include(p | p.getCodice() = prod.getCodice())  Context::AccountAzienda public rimuoviProdotto(prod : Prodotto)  **pre**: prod != null and prodotti->include(p | p.getCodice() = prod.getCodice())  Context::AccountAzienda public aggiornaProdotto(prod : Prodotto)  **pre**: prod != null and prodotti->include(p | p.getCodice() = prod.getCodice()) |
| Post-condizioni | Context::AccountAzienda public dammiProdotto(id :Long)  **post**:  Context::AccountAzienda public aggiungiProdotto(prod : Prodotto)  **post**: prodotti->include(p | p.getCodice() = prod.getCodice())  Context::AccountAzienda public rimuoviProdotto(prod : Prodotto)  **post**: prodotti->!include(p | p.getCodice() = prod.getCodice())  Context::AccountAzienda public aggiornaProdotto(prod : Prodotto)  **post**: prodotti->include(p | p.getCodice() = prod.getCodice() and p.getNome() = prod.getNome() and p.getImmagine() = prod.getImmagine() and p.getDescrizione() = prod.getDescrizione() and p.getPrezzo() = prod.getPrezzo()) |
| Invarianti |  |

|  |  |
| --- | --- |
| Nome Classe | Ordine |
| Descrizione | Classe che rappresenta un ordine |
| Pre-condizioni | **Context**::Ordine public cambiaStato(stato :String)  **pre** : !self.stato.equals(“Consegnato”) |
| Post-condizioni | Context::Ordine public cambiaStato(stato :String)  **post** : if stato.equals(“Ritirato”) and @pre self.stato.equals(“In preparazione”) then self.stato = stato  if stato.equals(“Consegnato”) and @pre self.stato.equals(“Ritirato”) then self.stato = stato |
| Invarianti |  |

|  |  |
| --- | --- |
| Nome Classe | GestoreOrdineDAOImpl |
| Descrizione | Classe che implementa l’interfaccia GestoreOrdineDao per la gestione dei dati persistenti riguardanti l’ordine. |
| Pre-condizioni | **Context::GestoreOrdineDAOImpl** public controlloEsistenzaMail(mail : String)  **pre** : mail != null  **Context::GestoreOrdineDAOImpl** public creaOrdine(order : Ordine, user : AccountUtente, cart : Carrello)  **pre** : order != null and user!= null and cart != null and cart.prodotti->size()>0  **Context::GestoreOrdineDAOImpl** public dammiOrdiniPreparazione(azienda : AccountAzienda)  **pre** : azienda != null  **Context::GestoreOrdineDAOImpl** public dammiOrdine(codice : Long)  **pre** : codice != null  **Context::GestoreOrdineDAOImpl** public dammiConsegne(fattorino : AccountFattorino)  **pre** : fattorino != null  **Context::GestoreOrdineDAOImpl** public ordineSetRitirato(orderCode : Long)  **pre** : order != null  **Context::GestoreOrdineDAOImpl** public ordineSetConsegnato(orderCode : Long)  **pre** : order != null |
| Post-condizioni | - |
| Invarianti | - |
| Nome Classe | **GestoreMailImpl** |
| Descrizione | Classe che implementa l’interfaccia GestoreMail per la gestione e l’invio di mail da parte del sistema |
| Pre-condizioni | **Context::GestoreMailImpl** public inviaMailModetatore(idOrder :Long, description : String)  pre: idOrder != null and description != null  **Context::GestoreMailImpl** public inviaMailBan(azienda :AccountAzienda, description : String)  pre: azienda != null and description != null  **Context::GestoreMailImpl** public inviaMailOrdine(order :Ordine)  pre: azienda != null and description != null |
| Post-condizioni | - |
| Invarianti | - |

## **4. Class Diagram**

