Università degli Studi di Salerno
Corso di Ingegneria del Software

E-store Libri&Musica

Object Design Document

ODD
Versione 0.5

Data ultima modifica: 16/02/2017

Progetto: E-store Libri&Musica | Versione: 0.6

Documento: ODD | Data: 16/02/2017

Partecipanti:

Nome	Matricola
Fimiani Luca	0512103180

Scritto da: Fimiani Luca

Revision History

Data	Versione	Descrizione	Autore
15/11/2016	0.1	Inizio stesura documento	Fimiani Luca
16/11/2016	0.2	Completamento paragrafo 1	Fimiani Luca
17/11/2016	0.3	Stesura paragrafo 2	Fimiani Luca
18/11/2016	0.4	Aggiunta dei package e dei sottosistemi	Fimiani Luca
19/11/2016	0.5	Revisione elaborato	Fimiani Luca
16/02/2017	0.6	Revisione	Fimiani Luca

INDICE DEGLI ARGOMENTI

1.Introduzione…… 3
1.1 Compromessi dell'Object Design…….. 3
1.2 Riferimenti……. 3

2. Linee guida per l'implementazione…… 4

2.1 Nomi di file……..4
2.2 Organizzazione dei file……4
2.2.1 File sorgenti…… 4
2.3 Indentazione……5
2.3.1 Lunghezza delle linee……5
2.3.2 Spostamento di linee…… 5
2.4 Commenti…… 6
2.4.1 Formattazione del commento di implementazione…… 6
2.4.2 Formattazione dei commenti di documentazione……6
2.5 Dichiarazioni……7
2.5.1 Numero per linea……7
2.5.2 Inizializzazione…… 7
2.5.3 Posizione…… 7
2.5.4 Dichiarazione di classe e interfaccia……8
2.6 Istruzioni……8
2.6.1 Istruzioni semplici……8
2.6.2 Istruzioni composte……8
2.6.3 Istruzioni return……8
2.6.4 Istruzioni if, if-else, if-else-if-else ……8
2.6.5 Istruzioni for ……9
2.6.6 Istruzioni while …… 9
2.6.7 Istruzioni do-while …… 9
2.6.8 Istruzioni switch ……9
2.6.9 Istruzioni try-catch ……10
2.7 Spazi Bianchi …… 10
2.7.1 Linee bianche …….. 10
2.7.2 Spazi bianchi …… 10
2.8 Convenzioni di nomi …….. 11
2.8.1 Classi …… 11
2.8.2 Interfacce …… 11
2.8.3 Metodi …… 11
2.8.4 Variabili …… 11
2.8.5 Costanti …… 11
2.9 Consuetudini di programmazione …….. 12

2.9.1 Fornire accesso a variabili di istanza o di classe 12
2.9.2 Riferire variabili e metodi di classe 12
2.9.3 Assegnamento di variabili 12
2.9.4 Parentesi 12
2.9.5 Valori restituiti 12
3 Packages 13
3.1 Package beans 13
3.2 Package dataModel 14
3.3 Package servletAdmin 14
3.4 Package servletUser 15
3.5 Package dbConnection 15
3.6 Sottosistema GestioneUtente 16
3.7 Sottosistema GestioneAdmin 17
4 Design Patterns 18
5 Components off-the-shelf 18
6 Class Interfaces 18
6.1 Tabella OrdineBean 18
6.2 Tabella ProdottiBean 19
6.3 Tabella UtenteBean 20
7 Glossario 21

1.Introduzione

Tale documento definisce l'object design del sistema. Lo scopo è di chiudere il divario fra gli oggetti individuati in fase di analisi e la piattaforma hardware/software scelta durante il system design. Si intende definire precisamente e senza ambiguità di comprensione la specifica delle classi, degli attributi, delle interfacce , delle operazioni e dei vincoli su questi. In maniera simile si intende individuare i compromessi delle scelte più importanti, individuare i design pattern adatti alle particolari necessità e le componenti off-the-shelf da riutilizzare.

1.1 Compromessi dell'Object Design

Memoria vs Efficienza

La scelta effettuata fa in modo di privilegiare l'utilizzo della memoria, in quanto si prevede di dover gestire una grande quantità di dati dopo qualche anno di utilizzo del sistema. Il database nonostante comporti alcuni svantaggi nell'utilizzo della memoria, conserva molte garanzie per quanto riguarda la gestione consistente dei dati e degli accessi in concorrenza.

Sicurezza vs Efficienza

I requisiti di sicurezza prefissati devono essere rispettati. Ad esempio ad ogni richiesta di un client il sistema deve essere certo dell'autenticazione dell'utente ed anche validare i dati ricevuti. A tal proposito si farà uso di sessioni e di controlli della tipologia di utente ad ogni operazione richiesta. Tale controllo comporterà sicuramente l'aumento del carico di dati che viaggiano sulla rete fra client e server, andando ad aumentare i tempi di risposta e rendendo meno gestibili le situazioni di picchi di carico. Questa scelta, però, deve essere obbligatoriamente rispettata in modo da rendere coerente la selezione dei requisiti specificati in fasi di analisi.

Comprensibilità della codifica vs Costi

Per favorire le fasi di test e debug bisogna dare importanza anche alla leggibilità del codice. Il comportamento di oggetti e metodi deve risultare comprensibile anche agli sviluppatori che non le hanno realizzate. L'uso dei commenti e della documentazione javadoc non è obbligatoria, visto lo scopo del progetto, ma sarebbe opportuno. Commenti e documentazione javadoc favorirebbero lo sviluppo del codice , in particolar modo porterebbero a maggior comprensibilità e conseguentemente alla riduzione del tempo necessario a completare la scrittura del codice.

Prestazioni vs Costi

Non avendo budget a disposizione per l'acquisto di hardware e software si utilizzeranno solamente componenti open source.

Costi vs implementazione

Data l'assenza di budget ogni componente verrà realizzata partendo da zero, o meglio, partendo da un sistema che è stato da noi già implementato ma che non è stato ultimato.

Costi vs Mantenimento

L'utilizzo di componenti open source non nega un buon mantenimento. Tali componenti, infatti, permettono un buon mantenimento del codice attraverso i propri strumenti,

1.2 Riferimenti

- E-storeLibri&Musica_RAD
- E-storeLibri&Musica_SDD
- B. Bruegge, A.H. Dutoit, Object Oriented Software Engineering

2.Linee guida per l'implementazione

2.1 Nomi di file

Il software Java utilizza i seguenti suffissi per i file :

- Per i sorgenti Java il suffisso è ".java"
- Per le jsp il suffisso è ".jsp"
- Per i fogli di css il suffisso è ".css"
- Per i file javascript il suffisso è ".js"
- Per i file Bytecode il suffisso è ".class"

2.2 Organizzazione dei file

Un file consiste di sezioni di codice che dovrebbero essere opportunamente separati da spazi vuoti e da commenti che offrono maggior dettaglio del codice scritto. Quest'ultimi sono opzionali .

2.2.1 File sorgenti

Ogni singolo file sorgente Java contiene del codice che raffigura il comportamento di tale classe.

I file sorgenti dovrebbero avere la seguente struttura :

- Commenti di inizio (Opzionali) : tutti i file sorgenti dovrebbero iniziare con un commento che riporta il nome della classe, la descrizione, l'autore, le informazioni sulla versione e le informazioni di copyright.
- Istruzioni di package e import : la prima linea non commentata di molti file sorgenti Java è l'istruzione *package* che descrive il package in cui la classe è collocata.
  In seguito a tale istruzione possono esserci vari *import* che raffigurano le classi importate dall'esterno del package corrente.
- Dichiarazioni di classe e interfaccia : tali istruzioni indicano se la classe è un'interfaccia o una semplice classe. L'ordine da rispettare è il seguente :

- Istruzione “class” o “interface”
- Commento di implementazione della classe/interfaccia, se necessario.
  Questo commento deve contenere informazioni generali sulla classe o interfaccia che non sono appropriate per il commento di documentazione;
- Variabili di classe nel seguente ordine : static, public, protected, private.
  Variabili di istanza nel seguente ordine : public , protected, private.
- Costruttori
- Metodi: questi metodi devono essere raggruppati in base alla loro funzionalità piuttosto che in base a regole di visibilità o accessibilità. Ad esempio, un metodo di classe privato può stare tra due metodi pubblici. L’obiettivo è rendere più semplice la lettura e la comprensione del codice.

2.3 Indentazione

Come unità di indentazione devono essere usati quattro spazi, ma non viene precisamente evidenziato come costruire l’indentazione (se usare spazi o tabulazioni). Le tabulazioni devono essere settate ogni quattro spazi.

2.3.1 Lunghezze delle linee

E’ consigliato ma non obbligatorio evitare linee più lunghe di cento caratteri per favorire un’ottima gestione da tutti i terminali e strumenti software. Per i commenti di documentazione si usa una linea più corta, non più lunga di settanta caratteri.

2.3.2 Spostamento di linee

Quando un’espressione supera la lunghezza della linea occorre spezzarla secondo i seguenti principi generali :

- Interrompere la linea dopo una virgola
- Interrompere la linea dopo un operatore
- Preferire interruzioni di alto livello rispetto ad interruzioni di basso livello
- Allineare la nuova linea con l’inizio dell’espressione nella linea precedente

2.4 Commenti

I programmi Java possono avere due tipi di commenti : commenti di implementazione e commenti di documentazione (entrambi sono opzionali nel nostro progetto, ma ne è consigliato l'utilizzo). I commenti d'implementazione sono quelli classici , delimitati da /*..*/. I commenti di documentazione (noti come javadoc documents) sono esclusivi di Java e sono delimitati da /**…..*/. I commenti in stile javadoc possono essere estratti in file HTML.

I commenti d'implementazione sono dei mezzi per commentare il codice o per commentare una particolare implementazione di esso.

I commenti di documentazione vengono utilizzati per descrivere la specifica del codice da una prospettiva non implementativa, in modo da facilitare la lettura e la comprensione di ciò che si è implementato agli sviluppatori che lo andranno a leggere. Tali commenti fanno in modo che gli sviluppatori possono comprendere il codice pur non avendolo avanti agli occhi.

I commenti dovrebbero essere utilizzati per dare una panoramica del codice e per fornire informazioni aggiuntive che non sono prontamente disponibili nel codice stesso. I commenti devono contenere solo informazioni rilevanti in modo tale da poter leggere e comprendere il programma. Ad esempio non devono essere descritte informazioni su come è composto il package corrispondente o in quale directory risiede .

La discussione sulle decisioni non banali o non ovvie è adatta, ma bisogna evitare di duplicare le informazioni che sono presenti in maniera chiara nel codice. E' molto facile che commenti ridondanti diventino obsoleti. In generale si dovrebbe evitare di inserire commenti suscettibili che rischiano di diventare obsoleti con l'evoluzione del software.
La frequenza dei commenti talvolta riflette una povera qualità del codice. Quando ci si sente obbligati ad aggiungere un commento bisogna considerare l'opzione di riscrivere il codice per renderlo più chiaro.
I commenti non dovrebbero essere inclusi in grandi riquadri tracciati con asterischi o altri caratteri, né dovrebbero includere caratteri speciali come backspace.

2.4.1 Formattazione del commento d'implementazione

I programmi possono avere tre tipi di commenti di implementazione :

- *Commenti di blocco* : Sono usati per fornire descrizioni di file, metodi, strutture dati e algoritmi. Tali commenti possono essere usati all'inizio di ogni file e prima di ogni metodo. Possono essere utilizzati anche all'interno dei metodi, se ve n'è necessità.
  I commenti di blocco dentro una funzione o un metodo dovrebbero essere indentati allo stesso livello del codice che descrivono. Un commento di blocco dovrebbe essere preceduto da una linea bianca di separazione dal codice.
- *Commenti a linea singola* : Sono brevi commenti che possono apparire su una singola riga di codice e devono essere indentati al livello del codice che seguono. Se un commento non può essere scritto su una linea singola deve seguire il formato di un commento di blocco. Un commento a linea singola deve essere preceduto da una linea bianca.

- *Commenti di fine linea* : Il delimitatore di commento " // " può commentare una linea completa o una parte di essa. Non dovrebbe essere mai usato su più linee consecutive per commenti testuali, anche se è consentito.

2.4.2 Formattazione dei commenti di documentazione

I commenti di documentazione descrivono classi Java, Interfacce, costruttori, metodi e campi. Ogni documento è compreso all'interno dei delimitatori /**…*/ , con un commento per ogni classe, interfaccia o membro. Questo commento deve apparire solo prima della dichiarazione.

Un commento javadoc si compone di una descrizione seguita da un tag :

- @author
- @param
- @exception

2.5 Dichiarazioni

2.5.1 Numero per linea

La dichiarazione per linea sarebbe opportuna per incoraggiare i commenti, ovvero :

```
int somma; //risultato dell'operazione somma
int differenza; //risultato dell'operazione sottrazione
```

è preferito rispetto a :

```
int somma, differenza;
```

2.5.2 Inizializzazione

E' opportuno provare ad inizializzare le variabili locali nel punto in cui sono dichiarate. L'unica ragione per non inizializzare una variabile dove è stata dichiarata è che il suo valore dipende da un calcolo che prima occorre eseguire.

2.5.3 Posizione

E' opportuno collocare le dichiarazioni all'inizio dei blocchi (codice racchiuso tra parentesi graffe) evitando di dichiarare le variabili dopo il primo uso. Ciò permette di non confondere il programmatore inesperto e non impedire la portabilità del codice dentro lo scope. L'unica eccezione sono gli indici dei cicli for che vengono dichiarati nell'istruzione stessa.
Evitare dichiarazioni locali che mascherano quelle globali, ovvero bisogna evitare che i nomi di variabili locali coincidano con quelli di varabili globali.

2.5.4 Dichiarazione di classe e interfaccia

Quando si codificano classi e interfacce Java, si dovrebbero rispettare le seguenti regole di formattazione:

- Non mettere spazi tra il nome del metodo e la parentesi "(" che apre la lista dei parametri
- La parentesi graffa aperta "{" si trova alla fine della stessa linea dell'istruzione di dichiarazione o nella riga successiva sotto l'identificatore della classe/interfaccia
- La parentesi graffa chiusa "}" inizia una linea indentandosi per mapparsi con la corrispondente istruzione di apertura, eccetto il caso in cui c'è un'istruzione vuota. In tal caso la "}" dovrebbe essere immediatamente dopo la "{"

2.6 Istruzioni

2.6.1 Istruzioni semplici

E' consigliato , ma non obbligatorio, che ogni riga debba contenere al massimo un'istruzione.

```
i++;          //CORRETTO
j++;          //CORRETTO
i++; j++;     //NO!!!
```

2.6.2 Istruzioni composte

Le parentesi graffe aperte dovrebbero stare o alla fine della linea che inizia l'istruzione composta o indentate nella linea successiva poste allineate verticalmente con l'inizio della rispettiva istruzione.
Le parentesi graffe vanno usate per tutte le istruzioni anche per quelle singole.

2.6.3 Istruzioni return

Un'istruzione return con un valore di restituzione non dovrebbe usare parentesi, a meno che queste rendano in qualche modo il valore ritornato più ovvio (specificando il caso di restituzione del valore) .

2.6.4 Istruzioni if, if-else, if-else-if-else

Forma che devono assumere tali istruzioni :

```
if(i == 0) {
    ~~~~
}
if(i > 0) {
    ~~~~
} else {
    ~~~~
}
if(somma > 0) {
    ~~~~
} else if(somma <= 0) {
    ~~~~
}
```

Le istruzioni if usano sempre le parentesi graffe .

2.6.5 Istruzioni for

Un'istruzione for dovrebbe avere la seguente forma :

```
for (int i=0; i<n; i++)
{
    //corpo
}
```

Quando si usa l'operatore virgola nella clausola di inizializzazione o aggiornamento di un'istruzione for bisogna evitare la complessità di utilizzare più di tre variabili. Se necessario usare istruzioni separate prima del ciclo for o alla fine di esso.

2.6.6 Istruzioni while

Un'istruzione while dovrebbe avere la seguente forma :

```
while (n > 0)
{
    //corpo
}
```

2.6.7 Istruzioni do-while

Un'istruzione do-while dovrebbe avere la seguente forma :

```
do {
    //corpo
} while (n > 0);
```

2.6.8 Istruzioni switch

Un'istruzione switch dovrebbe avere la seguente forma :

```
switch(choice)
{
    case 1 :
        //...
        break;
    case 2 :
        //...
        break;
    case 3 :
        //...
        break;
}
```

2.6.9 Istruzioni try-catch

Un'istruzione try-catch dovrebbe avere la seguente forma :

```
try {
    //...
} catch(Exception e)
{
    //...
}
```

Un'istruzione try-catch può comprendere anche la clausola finally, la quale verrà eseguita indipendentemente dal fatto che il blocco try sia stato o meno completato con successo.

```
try {
    //...
} catch(Exception e)
{
    //...
}
finally {
    //...
}
```

2.7 Spazi bianchi

2.7.1 Linee bianche

Due linee bianche dovrebbero essere sempre usate nelle seguenti circostanze :

- Fra sezioni di un file sorgente
- Fra definizioni di classe, interfaccia e costruttori

Una linea bianca dovrebbe essere sempre usata nelle seguenti circostanze :

- Fra metodi
- Fra le variabili locali in un metodo e la sua prima istruzione
- Prima di un commento di blocco o a singola linea
- Fra sezioni logiche all'interno di un metodo

2.7.2 Spazi bianchi

Spazi bianchi dovrebbero essere usati nelle seguenti circostanze :

- Uno spazio bianco non dovrebbe essere usato fra il nome di un metodo e le sue parentesi di apertura
- Tutti gli operatori binari eccetto l'operatore punto (" . ") dovrebbero essere separati dai loro operandi tramite spazi. Gli spazi bianchi non dovrebbero mai separare gli operatori unari come l'operatore meno (" -- "), l'incremento (" ++ ") e il decremento (" –").

2.8 Convenzioni di nomi

2.8.1 Classi

I nomi delle classi dovrebbero essere sostantivi che raffigurano l'oggetto che verrà istanziato nel momento in cui si dichiari una variabile di quella rispettiva classe. E' consigliato utilizzare la convenzione Java secondo cui il nome di una classe dovrebbe essere scritto con la sola lettera iniziale maiuscola.
Inoltre non dovrebbero essere usati underscore per legare i nomi. Per le Servlet è necessario inserire nel nome della classe il nome "Servlet" . La stessa operazione va effettuata con i "Bean".

Le classi sono disposte in package a seconda delle loro funzionalità. Servlet , Bean e Model sono separati opportunamente.

La scelta di un nome deve essere mnemonica e deve rispettare il dominio applicativo.

2.8.2 Interfacce

E' opportuno che si seguano le stesse regole utilizzate per le classi.

2.8.3 Metodi

I nomi dei metodi devono essere verbi con l'iniziale minuscola. Se un metodo è formato da più parole concatenate è opportuno che ogni parola, tranne quella iniziale, abbia la prima lettera maiuscola.
La scelta di un nome deve essere mnemonica e deve rispettare il dominio applicativo.

2.8.4 Variabili

Tutte le variabili e le istanze di classe devono essere scritte con iniziale minuscola seguendo la stessa regola della concatenazione di più parole utilizzata per i metodi (a gobba di cammello) .
I nomi di variabili di un solo carattere dovrebbero essere evitati.
La scelta di un nome deve essere mnemonica e deve rispettare il dominio applicativo.

Per i codici che rappresentano le chiavi all'interno del DB è consigliato scrivere il nome con la lettere iniziale maiuscola.

Per i codici isbn e isrc è opportuno che per facilitare l'identificazione essi siano scritti interamente in maiuscolo, ovvero "ISBN" e "ISRC".

2.8.5 Costanti

I nomi delle variabili dichiarate come costanti di classe devono essere scritte in lettere maiuscole con le parole separate da underscore. E' consigliabile che i nomi delle costanti siano in inglese. La scelta di un nome deve essere mnemonica e deve rispettare il dominio applicativo.

2.9 Consuetudini di programmazione

2.9.1 Fornire accesso a variabili di istanza o di classe

E' opportuno non rendere pubblica una variabile di istanza o di classe (globale) senza una buona ragione. E' opportuno che le variabili di istanza siano scritte e lette attraverso delle chiamate a metodi.

2.9.2 Riferire variabili e metodi di classe

Evitare di usare un oggetto per accedere a variabili o metodi di classe static. E' consigliato usare il nome della classe.

2.9.3 Assegnamento di variabili

Evitare di assegnare a più variabili lo stesso valore in una sola istruzione.
Non usare l'operatore di assegnamento in un punto in cui può essere facilmente confuso con l'operatore di uguaglianza.
Non usare assegnamenti innestati nel tentativo di migliorare le prestazioni a tempo di esecuzione.

2.9.4 Parentesi

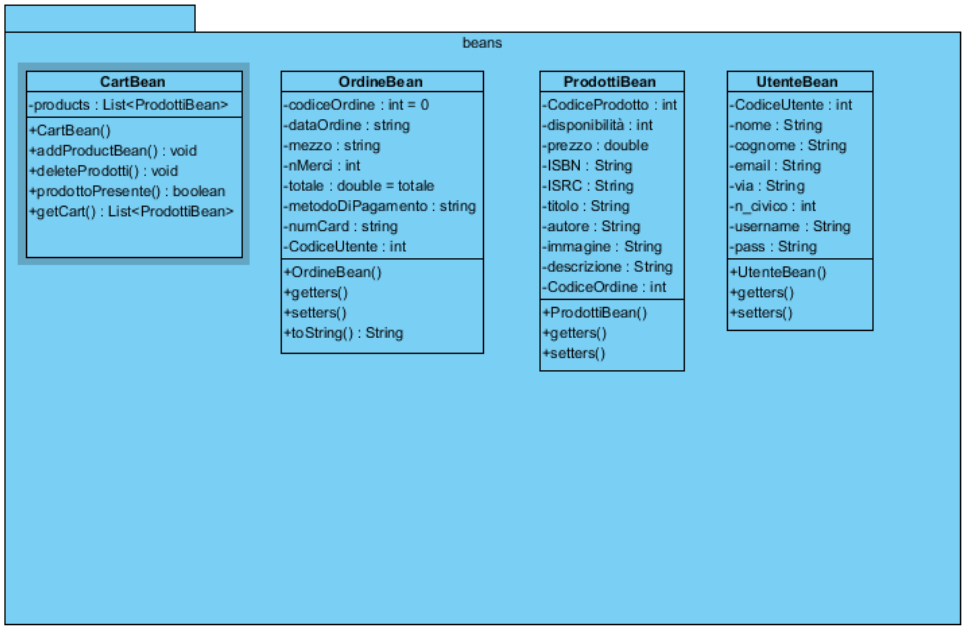
E' generalmente utile utilizzare le parentesi in espressioni che coinvolgono operatori misti in modo da evitare problemi di precedenza degli operatori.

2.9.5 Valori restituiti

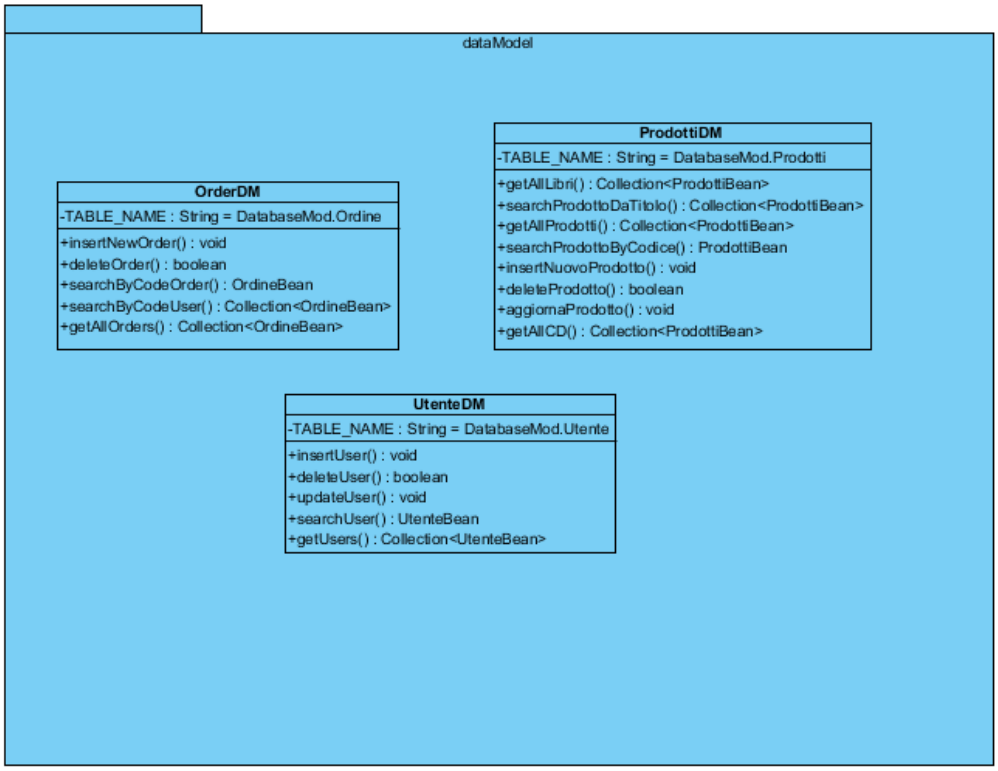
E' opportuno adattare la struttura del programma alle intenzioni del programmatore.

3. Packages

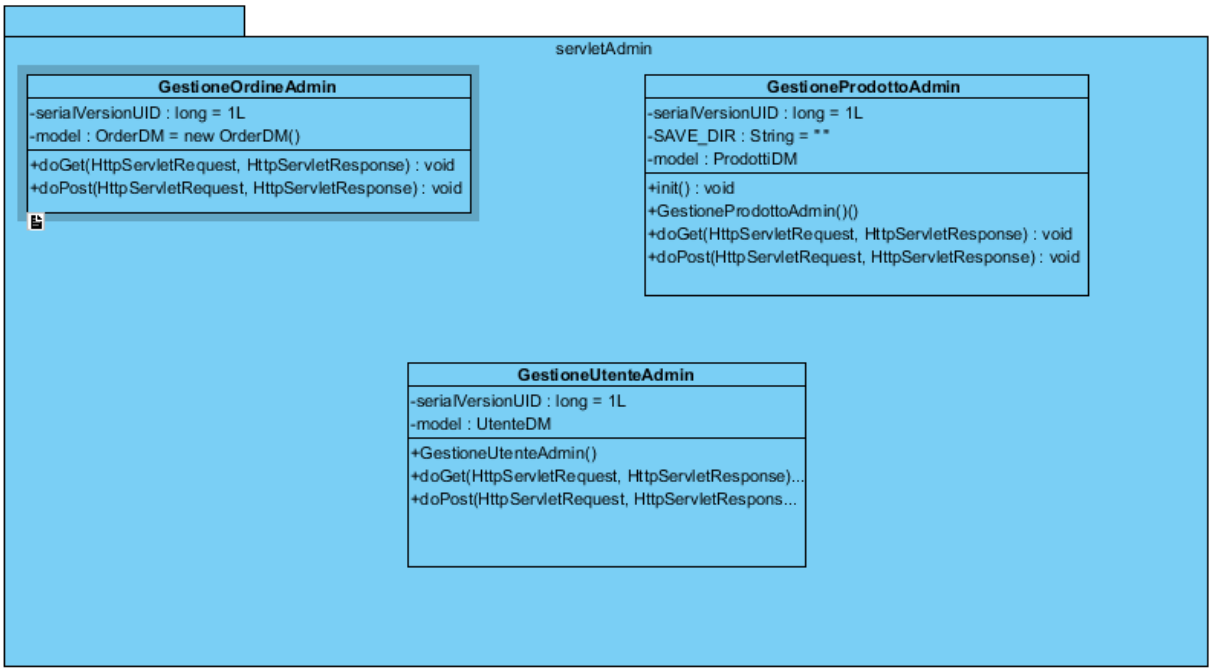
3.1 Package beans



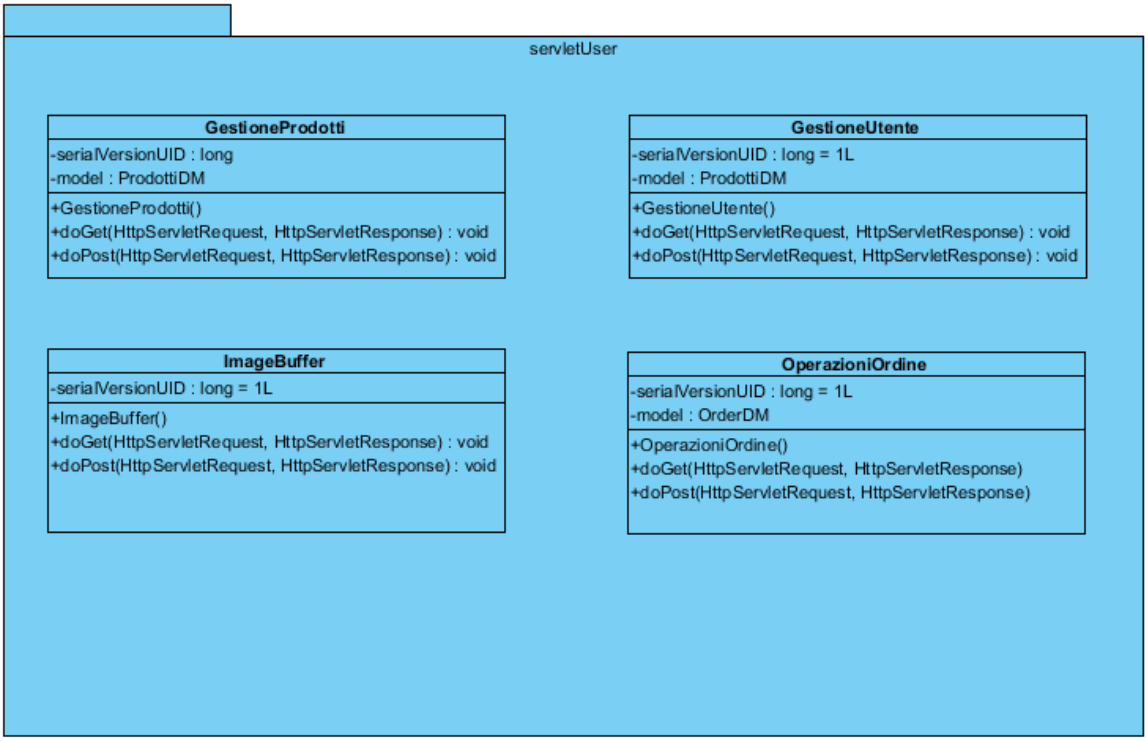
3.2 Package dataModel



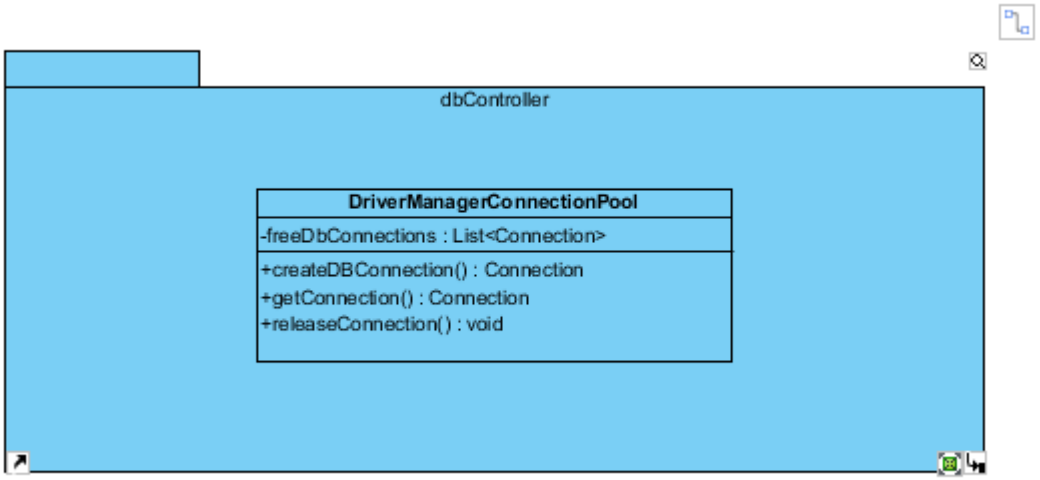
3.3 Package servletAdmin



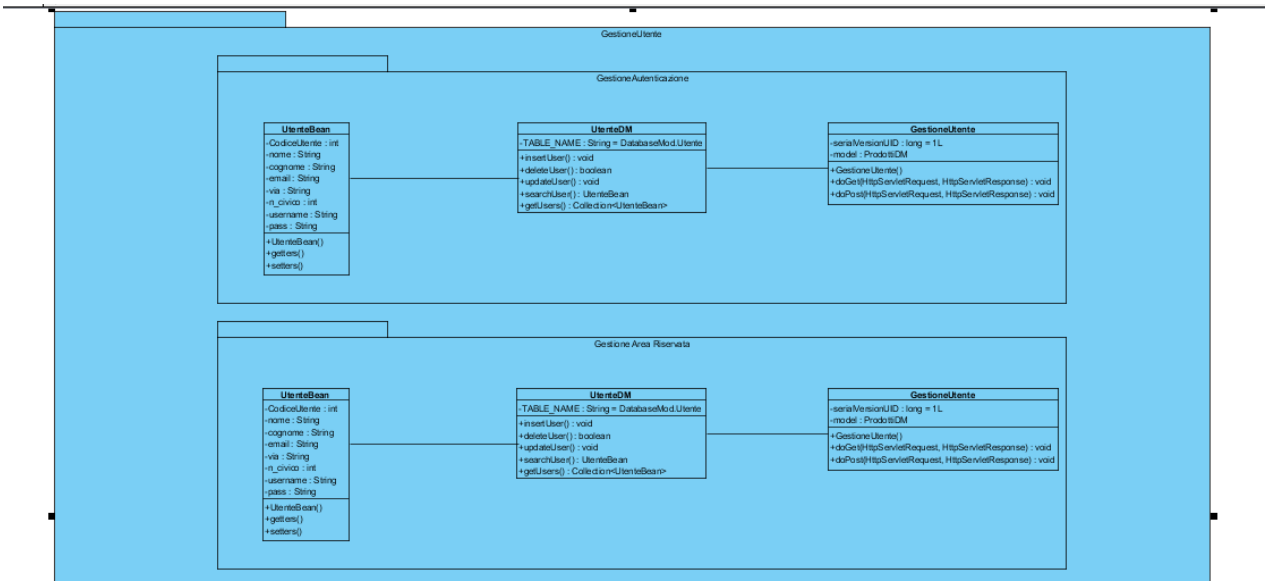
3.4 Package servletUser

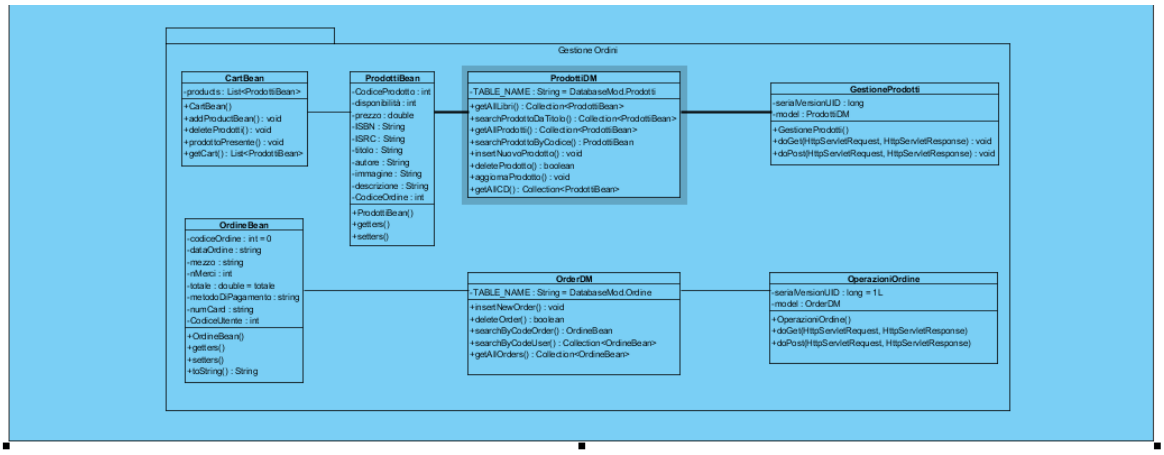


3.5 Package dbConnection

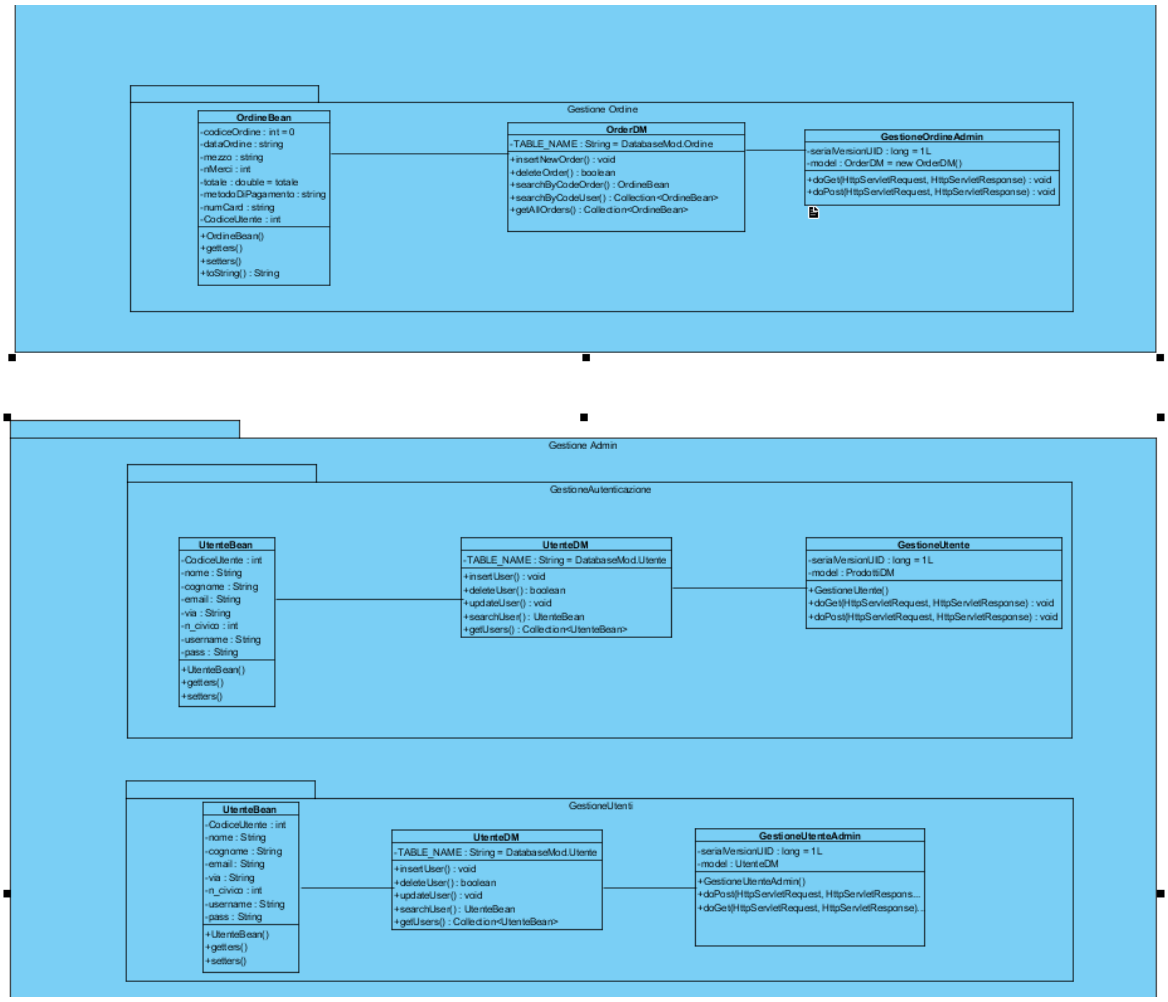


3.6 Sottosistema GestioneUtente





3.7 Sottosistema Gestione Admin



4. Design Patterns

Facade Pattern

Abbiamo preferito evitare di utilizzare il facade pattern. Il nostro sistema, infatti, è suddiviso in più sottosistemi che vengono invocati ed utilizzati quando necessario. Non è necessario , a mio avviso, implementare un'unica interfaccia.

Bridge Pattern

Fa in modo che vengano oscurati i dettagli implementativi relativi alla parte di Storage, disaccoppiando l'implementazione di basso livello dall'interfaccia.

Singleton Pattern

Tale pattern viene utilizzato per istanziare un'oggetto manager in caso di necessità.

In un primo momento si pensava fosse utile utilizzare i design patterns sopra citati, ma, successivamente ci si è resi conto che per il sistema progettato tali patterns potevano essere utilizzati in maniera opzionale.

5. Components off-the-shelf

Non è previsto l'uso di components off-the-shelf.

6. Class Interfaces

6.1 Tabella OrdineBean

Nome Classe	**OrdineBean**
Descrizione	Questa classe indica un ordine sul nostro sito

Pre-condizioni	`this.codiceOrdine = 0;`
	`this.dataOrdine = "";`
	`this.mezzo = "";`
	`this.nMerci = 0;`
	`this.totale = 0;`
	`this.metodoDiPagamento = "";`
	`this.numCard = "";`
	`this.CodiceUtente = 0;`

Post-condizioni	• **context** OrdineBean :: getCodiceOrdine(), **post** codiceOrdine *autogenerated* • context OrdineBean :: getDataOrdine(), **post** dataOrdine != null • **context** OrdineBean :: getMezzo(), **post** mezzo.lenght() > 0 && mezzo.lenght <= 45 • **context** OrdineBean :: getTotale(), **post** totale.numberMaxDigits() == 10,2

	• **context** OrdineBean :: getMetodoDiPagamento(), **post** metodoDiPagamento.lenght() > 0 && metodoDiPagamento.lenght() <= 20 • **context** OrdineBean :: getNumCard(), **post** numCard.lenght() > 0 && numCard.lenght() <= 23

Invarianti	**context** OrdineBean **invariant** :
	codiceOrdine *autogenerated;*
	dataOrdine != null; mezzo.lenght() >0 && mezzo.lenght() <= 45;
	totale.numberMaxDigits() == 10,2;
	metodoDiPagamento.lenght() > 0 && metodoDiPagamento.lenght() <= 20;
	numCard.lenght() > 0 && numCard.lenght() <= 23; ,

6.2 Tabella ProdottiBean

Nome Classe	**ProdottiBean**
Descrizione	Questa classe indica un Prodotto sul nostro sito

Pre-condizioni	`this.CodiceProdotto=000000000;`
	`this.Disponibilità=0;`
	`this.prezzo=0;`
	`this.ISBN="";`
	`this.ISRC="";`
	`this.Descrizione="";`
	`this.titolo="";`
	`this.autore="";`
	`this.immagine="";`
	`this.codiceOrdine = 0;`

Post-condizioni	• **context** ProdottiBean :: getCodiceProdotto(), **post** codiceProdotto != null • **context** ProdottiBean :: getPrezzo(), **post** prezzo.numberMaxDigits() == 8,2 • **context** ProdottiBean :: getISBN(), **post** ISBN.lenght() ==14 • **context** ProdottiBean :: getISRC(), **post** ISRC.lenght() == 12 • **context** ProdottiBean :: getTitotlo(), **post** titolo.lenght() > 0 && titolo.lenght() <= 30 • **context** ProdottiBean :: getAutore(), **post** autore.lenght() > 0 && autore.lenght() <= 25 • **context** ProdottiBean :: getImmagine(), **post** immagine.lenght() > 0 && immagine.lenght() <= 500 • **context** ProdottiBean :: getDescrizione(), **post** descrizione.lenght() > 0 && descrizione.lenght() <= 200

Invarianti	**context** ProdottiBean **invariant** :
	codiceProdotto != null;
	prezzo.numberMaxDigits() == 8,2;
	ISBN.lenght() ==14;
	ISRC.lenght() == 12;
	titolo.lenght() > 0 && titolo.lenght() <= 30;
	autore.lenght() > 0 && autore.lenght() <= 25;
	immagine.lenght() > 0 && immagine.lenght() <= 500;
	descrizione.lenght() > 0 && descrizione.lenght() <= 200;

6.3 Tabella UtenteBean

Nome Classe	**UtenteBean**
Descrizione	Questa classe indica un utente sul nostro sito
Pre-condizioni	
Post-condizioni	• **context** UtenteBean :: getCodiceUtente(), **post** codiceUtente *autogenerated && codiceUtente != null* • **context** UtenteBean :: getNome(), **post** nome.lenght() > 0 && nome.lenght() <= 25 • **context** UtenteBean :: getCognome(), **post** cognome.lenght() > 0 && cognome.lenght() <= 25 • **context** UtenteBean :: getEmail(), **post** email.lenght() > 0 && email.lenght() <= 60 • **context** UtenteBeano :: getVia(), **post** via.lenght() > 0 && via.lenght() <= 30 • **context** UtenteBean :: getN_civico(), **post** n_civico.numberMaxDigits() == 3 • **context** UtenteBean :: getCitta(), **post** citta.lenght() > 0 && citta.lenght() <= 30 • **context** UtenteBean :: getUsername(), **post** username.lenght() > 0 && username.lenght() <= 30 && username != null • **context** UtenteBean :: getPass(), **post** pass.lenght() > 0 && pass.lenght() <= 30 && username !=null •

Invarianti	**context** UtenteBean **invariant** :
	codiceUtente *autogenerated && codiceUtente != null;*
	nome.lenght() > 0 && nome.lenght() <= 25;
	cognome.lenght() > 0 && cognome.lenght() <= 25*;*
	email.lenght() > 0 && email.lenght() <= 60;
	via.lenght() > 0 && via.lenght() <= 30*;*
	n_civico.numberMaxDigits() == 3;
	citta.lenght() > 0 && citta.lenght() <= 30;
	username.lenght() > 0 && username.lenght() <= 30 && username != null;
	pass.lenght() > 0 && pass.lenght() <= 30 && username !=null;

7. Glossario

Acronimo	Descrizione
RAD	Requirements Analysis Document
DBMS	Database Management System
SDD	System Design Document
HW	Hardware
SW	Software
JDBC	Java DataBase Connectivity
DB	DataBase
SQL	Structured Query Language