UNIVERSITA' DEGLI STUDI DI SALERNO
FACOLTA' DI SCIENZE MM.FF.NN.
CORSO DI LAUREA MAGISTRALE IN INFORMATICA
CORSO DI LAUREA TRIENNALE IN INFORMATICA

GESTIONE DEI PROGETTI SOFTWARE
INGEGNERIA DEL SOFTWARE
A.A. 2013/2014

18/12/2013

Object Design Document

ODD di VViSeR UniSa

Team 8
VER 1.0

Coordinatore del progetto

Prof.ssa Ferrucci Filomena
Prof. De Lucia Andrea

Partecipanti

Nome	Matricola
Davide Scarano	0512100693
Antonio De Piano	0512101245
Eugenio Gigante	0510201455
Simone Romano	0512101343
Maria Vittoria Coda	0512101147
Giuseppe Sabato	0512101137
Michele Roviello	0512101217
Salvatore Angiuoli	0512101383

Scritto da:	Antonio De Piano, Giuseppe Sabato

Revision History

Data	Versione	Descrizione	Autore
18/12/2013	1.0	Compattazione documento	Giuseppe Sabato

Sommario

Indice delle tabelle

Indice delle figure

1 INTRODUZIONE

1.1 Tradeoff dell'object design

Comprensibilità vs Costi

La comprensibilità del codice è un aspetto molto importante soprattutto per la fase di testing. Ogni classe e metodo deve essere facilmente interpretabile anche da chi non ha collaborato al progetto. Nel codice si useranno i commenti standard e Javadoc per aumentare la comprensione del codice sorgente. Ovviamente questa caratteristica aggiungerà dei costi allo sviluppo del nostro progetto.

Sicurezza vs Efficienza

Per gestire la sicurezza, abbiamo previsto un'autenticazione nel momento in cui l'utente accede al sistema, in modo tale che possa visualizzare solo i propri dati. Quando un utente inserisce le credenziali (email e password), il sistema effettua una ricerca nella tabella utente del database, al fine di verificare innanzitutto se l'utente esiste, e in tal caso la correttezza della password inserita. In caso contrario(utente insesitente o password errata), l'accesso viene negato. Questa politica di gestione, non appesantice molto il sistema e rappresenta un buon compromesso tra sicurezza ed efficienza.

Prestazioni vs Costi

Non avendo a disposizione alcun budget, utiliziamo materiale open source per la realizzazione del software VViSeR con lo scopo di rendere il sistema performante ed efficiente.

Tempo di risposta vs Spazio di memoria

Per la memorizzazione, abbiamo scelto il DB relazionale, perchè i dati da memorizzare richiedono accessi ad un livello di dettaglio più raffinato e inoltre viene fornito l'accesso veloce

e concorrente ad essi. Lo svantaggio è che i DB richiedono circa il triplo dello spazio occupato dai dati reali.

Memoria vs efficienza

Abbiamo preferito un sistema efficiente a discapito della memoria utilizzata. Quindi abbiamo aumentato la ridondanza dei dati nel database, al fine di renderli subito disponibili.

1.2 Definizioni, acronimi e abbreviazioni

Tabella 1 Tabella acronimi

Acronimo	Descrizione
RAD	Requirement Analysis Document
DBMS	Data Base Management System
SSD	System Design Document
SQL	Structured Query Language
HW	Hardware
SW	Software
GUI	Graphical User Interface
ODD	Object Design Document

1.3 Riferimenti

- Bernd Bruegge e Allen H. Dutoit - Object-Oriented Software Engineering (using UML, Patterns and JavaTM) – Prentice Hall.
- Sommerville, Software Engineering – Addison Wesley .
- Per la redazione dell'ODD si è utilizzato lo standard del libro - Object-Oriented Software Engineering (using UML, Patterns and JavaTM).
- VViSeR_RAD - Requirements and analysis Document_2.3.pdf

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 è .java;
- Per i file Bytecode è .class.

2.2 Organizzazione dei file

Un file consiste di sezioni che dovrebbero essere separate da linee bianche e un commento opzionale che identifica ogni sezione. File più lunghi di 2000 linee sono ingombranti e devono essere evitati.

2.2.1 File sorgenti

Ogni file sorgente Java contiene una singola classe pubblica o un'interfaccia. Quando ci sono classi e interfacce private associate con la classe pubblica, è possibile inserirle nello stesso file sorgente della classe pubblica. La classe pubblica deve essere la prima classe o interfaccia nel file. I file sorgenti Java hanno la seguente struttura:

Commenti di inizio: Tutti i file sorgenti devono iniziare con un commento in stile C, che elenca il nome della classe, descrizione, autore e informazioni sulla versione, informazioni di copyright:

```
/**
 * Nome della classe
 *
 * Descrizione
 *
 * Author
 * Informazioni di versione
 *
 * 2012 - Copyright by SESA Lab - DMI - University of Salerno
 */
```

Istruzioni di package e import: La prima linea non commento di molti file sorgenti Java è l'istruzione package, che può essere seguita da istruzioni import. Ad esempio:

```
package sdi.classeUtilita;
import java.sql.Time;
```

Dichiarazioni di classe e interfaccia: L'ordine in cui le dichiarazioni di una classe o interfaccia devono apparire è il seguente:

- Commento di documentazione della classe/interfaccia (/**...*/);
- 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 (static): Prima le variabili di classe public, poi quelle protected e infine quelle private;
- Variabili di istanza: Prima quelle public, poi quelle protected e infine quelle 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 è specificato come costruire l'indentazione (se usare spazi o tabulazioni). Le tabulazioni devono essere settate ogni otto spazi (non quattro).

2.3.1 LUNGHEZZA DELLE LINEE

Evitare linee più lunghe di ottanta caratteri, perché esse non vengono ben gestite da molti terminali e strumenti software. Per la documentazione si utilizza una più corta lunghezza di linea, generalmente non più 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 prima di 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;

- Se le regole precedenti rendono il codice più confuso o il codice è troppo spostato verso il margine destro, utilizzare solo otto spazi di indentazione.

```
nomeMetodo(espressioneLunga1, espressioneLunga2, espressioneLunga3,
               espressioneLunga4, espressioneLunga5);

var=nomeMetodo(espressioneLunga1,
        nomeMetodo2(espressioneLunga2,
            espressioneLunga3));
```

I seguenti sono due esempi di interruzione di espressioni aritmetiche. Il primo è da preferire, dal momento che l'interruzione avviene al di fuori dell'espressione tra parentesi, che è ad un livello più alto.

```
nomeLungo1= nomeLungo2 * (nomeLungo3 + nomeLungo4 - nomeLungo5)
        * 4 * nomeLungo6; //PREFERIRE

nomeLungo1= nomeLungo2 * (nomeLungo3 + nomeLungo4
        - nomeLungo5) * 4 * nomeLungo6; //EVITARE
```

Quelli che seguono sono due esempi di indentazione di dichiarazioni di metodi. Il primo è il caso convenzionale, mentre il secondo sposta la seconda e la terza linea più lontano dal margine destro rispetto all'uso convenzionale, utilizzando un'indentazione con solo otto spazi.

```
//INDENTAZIONE CONVENZIONALE
nomeMetodo(int unArg, Object unAltroArg, String ancoraUnArg,
        Object eAncoraUnArg) {

    ...
}

//INDENTAZIONE 8 SPAZI PER EVITARE INDENTAZIONI TROPPO PROFONDE
private synchronized nomeMetodoLungo(int unArg,
        Object unAltroArg, String ancoraUnArg,
        Object eAncoraUnArg) {

    ...
}
```

Lo spezzettamento delle linee per l'istruzione if dovrebbe generalmente usare la regola degli otto spazi, poiché quella convenzionale a quattro spazi rende più difficoltosa la visibilità del corpo. Ad esempio:

```
//NON USARE QUESTA INDENTAZIONE
if((condizione1&&condizione2)
    ||(condizione3&&condizione4)
    ||!(condizione5&&condizione6)){
   faiQualcosa();
}

//USARE QUESTA INDENTAZIONE
if((condizione1&&condizione2)
    ||(condizione3&&condizione4)
    ||!(condizione5&&condizione6)){
   faiQualcosa();
}
```

Quelli che seguono sono tre modi accettabili di formattare le espressioni ternarie:

```
alpha = (unaLungaEspressioneBooleana) ? beta : gamma;

alpha = (unaLungaEspressioneBooleana) ? beta
                : gamma;

alpha = (unaLungaEspressioneBooleana)
        ? beta
        : gamma;
```

2.4 Commenti

I programmi Java possono avere due tipi di commenti: commenti d'implementazione e commenti di documentazione. I commenti d'implementazione sono quelli classici del C++, che sono delimitati da / *...*/ e //. I commenti di documentazione (noti anche come doc comments) sono esclusivi del Java, e sono delimitati da /**...*/. I doc comments possono essere estratti in file HTML utilizzando lo strumento javadoc.

I commenti di implementazione sono dei mezzi per commentare il codice o per commentare una particolare implementazione. I doc comments vengono utilizzati per descrivere la specifica del codice da una prospettiva non implementativa, per essere letti da sviluppatori che non devono necessariamente avere il codice in mano.

I commenti dovrebbero essere usati 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 per leggere e comprendere il programma. Ad esempio, informazioni su come il package corrispondente è costruito o in quale directory risiede non dovrebbero essere incluse in un commento.

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 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, considerare il caso 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 COMMENTO DI 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. I commenti di blocco possono essere usati all'inizio di ogni file e prima di ogni metodo. Possono inoltre essere usati in altri punti, come all'interno dei metodi. 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.

```
/*
 * Qui c'è un commento di blocco
 */
```

Commenti a linea singola: Sono brevi commenti che possono apparire su una singola linea di codice ed indentati al livello del codice che seguono. Se un commento non può essere scritto su una linea singola, deve seguire il formato di commento di blocco. Un commento a linea singola deve essere preceduto da una linea bianca. Quello che segue è un esempio di commento a linea singola nel codice Java:

```
if(condizione) {
    /* Gestisce la condizione */
    ...
}
```

Commenti di fine linea: Il delimitatore di commento // può commentare una linea completa o una parte di essa. Non dovrebbe essere usato su più linee consecutive per commenti testuali; comunque, può essere usato su più linee consecutive per commentare sezioni del codice. Seguono tre esempi dei tre stili:

```
if(i>0) {
    //Fa qualcosa
    ...
} else {
    i--; //Spiega il perché qui
}

//if(x==0){
//
// Fa qualcos'altro
// ...
```

2.4.2 COMMENTI DI DOCUMENTAZIONE

I “doc comments” descrivono classi Java, interfacce, costruttori, metodi e campi. Ogni doc comment è compreso all’interno dei delimitatori di commento /**...*/, con un commento per ogni classe, interfaccia o membro. Questo commento deve apparire solo prima della dichiarazione:

```
/**
 * La classe Esempio fornisce...
 */
public class Esempio {
    ...
```

Un doc comment si compone di una descrizione seguita da un blocco di tag. I tag da utilizzare sono @author, @exception, @param, @return, @see.

```
/**
 * Verifica l'equivalenza tra due oggetti.
 * Ritorna un boolean che indica se l'oggetto in cui mi trovo
 * è equivalente all'oggetto specificato come parametro.
 *
 * @author          Marco Rossi
 * @param   obj     l'oggetto che viene confrontato
 * @return  true    se i due oggetti sono equivalenti
 *          false   altrimenti
 *
 * @see             java.util
 */
public boolean equals(Object obj) {
    return (this == obj);
}
```

Notiamo che classi ed interfacce al alto livello non sono indentate, mentre lo sono i loro membri. La prima linea del commento di documentazione (/**) per classi e interfaccia non è indentata; le linee di commento successive hanno ognuna uno spazio di indentazione (per allineare verticalmente gli asterischi). I membri, inclusi i costruttori, hanno quattro spazi per la

prima linea del doc comment e cinque spazi per quelli successivi. Se si ha necessità di dare informazioni circa la classe, l'interfaccia, la variabile o il metodo, che non sono appropriate per la documentazione, usare un commento di implementazione di blocco o a singola linea immediatamente dopo la dichiarazione. Per esempio, i dettagli sull'implementazione di una classe devono andare in un commento di blocco seguente l'istruzione class, non nel doc comment della classe. i doc comments non devono essere posizionati dentro il blocco di definizione di un metodo o un costruttore, perché Java associa i commenti di documentazione con la prima dichiarazione dopo il commento.

2.5 Dichiarazioni

2.5.1 NUMERO PER LINEA

Una dichiarazione per linea è raccomandata dal momento che incoraggia i commenti. In altre parole

```
int livello;        // livello di indentazione
int dimensione;     // dimensione della tabella
```

è preferito rispetto a

```
int livello, dimensione;
```

Non inserire tipi differenti sulla stessa linea:

```
int livello, varArray[]; //NO!
```

Un'altra alternativa accettabile è usare le tabulazioni, cioè:

```
int     livello;          // livello di indentazione
int     dimensione;       // dimensione della tabella
float   posizioneCorrente; // posizione della tabella attualmente selezionata
```

2.5.2 INIZIALIZZAZIONE

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

2.5.3 POSIZIONE

Mettere le dichiarazioni all'inizio dei blocchi. Un blocco è un codice racchiuso entro parentesi graffe aperta e chiusa. Non aspettare di dichiarare le variabile al loro primo uso: può confondere il programmatore inesperto e impedire la portabilità del codice dentro lo scope. L'unica eccezione a questa regola sono gli indici dei cicli for che in Java possono essere dichiarati nell'istruzione stessa. Evitare dichiarazioni locali che nascondono dichiarazioni a più alto livello. Ad esempio, non dichiarare una variabile con lo stesso nome in un blocco interno.

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;
- La parentesi graffa chiusa "}" inizia una linea indentantosi per mapparsi con la corrispondente istruzione di apertura, eccetto il caso in cui c'è un'istruzione vuota, allora la "}" dovrebbe essere immediatamente dopo la "{".

2.6 Istruzioni

2.6.1 ISTRUZIONI SEMPLICI

Ogni linea deve contenere al massimo un'istruzione.

2.6.2 ISTRUZIONI COMPOSTE

Le istruzioni racchiuse dovrebbero essere indentate ad un ulteriore livello rispetto all'istruzione composta. La parentesi graffa aperta dovrebbe stare alla fine della linea che inizia l'istruzione composta, mentre la parentesi graffa chiusa dovrebbe iniziare una linea es essere indentata verticalmente con l'inizio dell'istruzione composta.

Le parentesi graffe vanno usate per tutte le istruzioni, anche quelle singole, quando sono parte di una struttura di controllo come nelle istruzioni if-else o for.

2.6.3 ISTRUZIONI RETURN

Un'istruzione return con un valore non dovrebbe usare parentesi, a meno che queste rendano in qualche modo il valore ritornato più ovvio.

2.6.4 ISTRUZIONI IF, IF-ELSE, IF-ELSE-IF ELSE

La classe di istruzioni if-else deve avere la seguente forma

```
if(condizione) {
    ...
}
if(condizione) {
    ...
} else {
    ...
}
if(condizione) {
    ...
} else if(condizione) {
    ...
} else {
    ...
}
```

Le istruzioni if usano sempre le parentesi graffe.

2.6.5 ISTRUZIONI FOR

Un'istruzione for dovrebbe avere la seguente forma

```
for(inizializzazione; condizione; aggiornamento) {
        ...
}
```

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

2.6.6 ISTRUZIONI WHILE

Un'istruzione while dovrebbe avere la seguente forma

```
while(condizione) {
    ...
}
```

2.6.7 ISTRUZIONI DO-WHILE

Un'istruzione do-while dovrebbe avere la seguente forma

```
do {
    ...
} while(condizione);
```

2.6.8 Istruzioni switch

Un'istruzione switch dovrebbe avere la seguente forma

```
switch(condizione) {
case ABC:
    ...
    /* Prosegue oltre */
case DEF:
    ...
    break;
default:
    ...
    break;
}
```

Ogni volta che un caso non include l'istruzione break, e quindi prosegue al caso successivo, aggiungere un commento nel punto in cui normalmente dovrebbe esserci l'istruzione break. Ogni istruzione break dovrebbe includere un caso di default. Nel caso di default, l'istruzione break è ridondante, ma previene un errore nel caso in cui venga aggiunto un altro case.

2.6.9 ISTRUZIONI TRY-CATCH

Un'istruzione try-catch dovrebbe avere la seguente forma

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

Un'istruzione try-catch può inoltre essere seguita da finally, che viene 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 e interfaccia;

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:

- Una parola chiave seguita da una parentesi dovrebbe essere separata da uno spazio;
- Uno spazio bianco non dovrebbe essere usato fra il nome di un metodo e le sue parentesi d'apertura;
- Uno spazio bianco dovrebbe essere interposto dopo le virgole nelle liste di argomenti;
- Tutti gli operatori binari eccetto . (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 di classe dovrebbero essere sostantivi, con lettere minuscole e, sia la prima lettera del nome della classe sia la prima lettera di ogni parola interna, deve essere maiuscola (camel case). Cercare di rendere i nomi delle classi semplici, descrittivi e che rispettino il dominio

applicativo. Usare parole intere evitando acronimi e abbreviazioni (a meno che l'abbreviazione sia più usata della forma lunga, come URL o HTML).

Non dovrebbero essere usati underscore per legare nomi. Per le Servlet è necessario far iniziare il nome della classe con il prefisso Servlet.

2.8.2 INTERFACCE

I nomi di interfaccia iniziano con la lettera I e seguono le stesse regole dei nomi di classi.

2.8.3 METODI

I nomi dei metodi devono essere verbi con iniziale minuscola e a gobba di cammello. Cercare di rendere i nomi dei metodi semplici, descrittivi e che rispettino il dominio applicativo. I nomi dei metodi non devono iniziare con caratteri di underscore o di dollaro. Usare parole intere evitando acronimi e abbreviazioni (a meno che l'abbreviazione sia più usata della forma lunga, come URL o HTML). Non dovrebbero essere usati underscore per legare nomi.

2.8.4 VARIABILI

Tutte le variabili e le istanze di classe devono essere scritte con iniziale minuscola e a gobba di cammello. I nomi delle variabili non devono iniziare con caratteri di underscore o dollaro. La scelta di un nome deve essere mnemonica e deve rispettare il dominio applicativo.

Le variabili che identificano una collezione di oggetti devono chiamarsi con lo stesso nome dell'oggetto contenuto nella lista. I nomi di variabili di un solo carattere dovrebbero essere evitati.

2.8.5 COSTANTI

I nomi delle variabili dichiarate come costanti di classe devono essere scritte in lettere tutte maiuscole con le parole separate da underscore. 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

Non rendere pubblica una variabile di istanza o di classe senza una buona ragione. Le variabili di istanza devono essere scritte o 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. Usare invece 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. Questo è compito del compilatore!

2.9.4 PARENTESI

E' generalmente una buona idea usare le parentesi liberamente in espressioni che coinvolgono operatori misti per evitare problemi di precedenza degli operatori.

2.9.5 VALORI RITORNATI

Provare a rendere la struttura del programma aderente alle proprie intenzioni.

3 PACKAGING

VViser è suddiviso in **pacchetti**:

- it.unisa.vviser.entity
- it.unisa.vviser.storage
- it.unisa.vviser.servlet
- it.unisa.vviser.exception
- it.unisa.vviser.test

Il pacchetto **it.unisa.vviser.entity** contiene le classi bean, che permettono la gestione dei dati, tramite operazione che si occupano di creare e manipolare le entità che fanno parte del sistema, entità che rappresentano un concetto ben distinto e con un proprio ruolo.

Il pacchetto **it.unisa.vviser.storage** contiente tutte le classi che gestiscono ed immagazzinano i dati persistenti, e per questo presenta tutte quelle operazioni che permettono la comunicazione con il Database, come operazioni per la connessione, l'accesso ai dati e l'interrogazione (query) di questi. Il nome di queste classi inizia con il prefisso DB.

Il pacchetto **it.unisa.vviser.servlet** contiene tutte le servlet, (una per ogni funzionalità del sistema), i cui nomi iniziano con il prefisso Servlet.

Il pacchetto **it.unisa.vviser.exception** contiene le classi di eccezione, che alterano il flusso di controllo in caso di errori durante l'esecuzione del codice, i cui nomi finiscono con il suffisso Exception.

Ed **infine il pacchetto it.unisa.vviser.test** contiente tutte le classi di test, i cui nomi iniziano con il prefisso Test.

Nelle immagini che seguono, si rappresentano i diagrammi relativi ai package e il diagramma delle classi relativo al package it.unisa.vviser.storage.

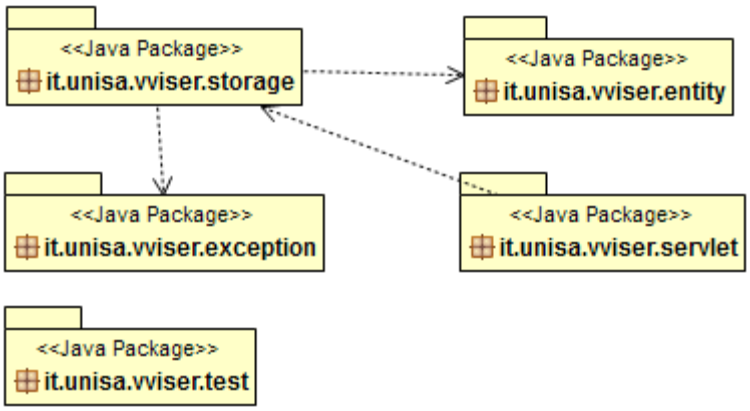


Figura 1 Diagramma dei package

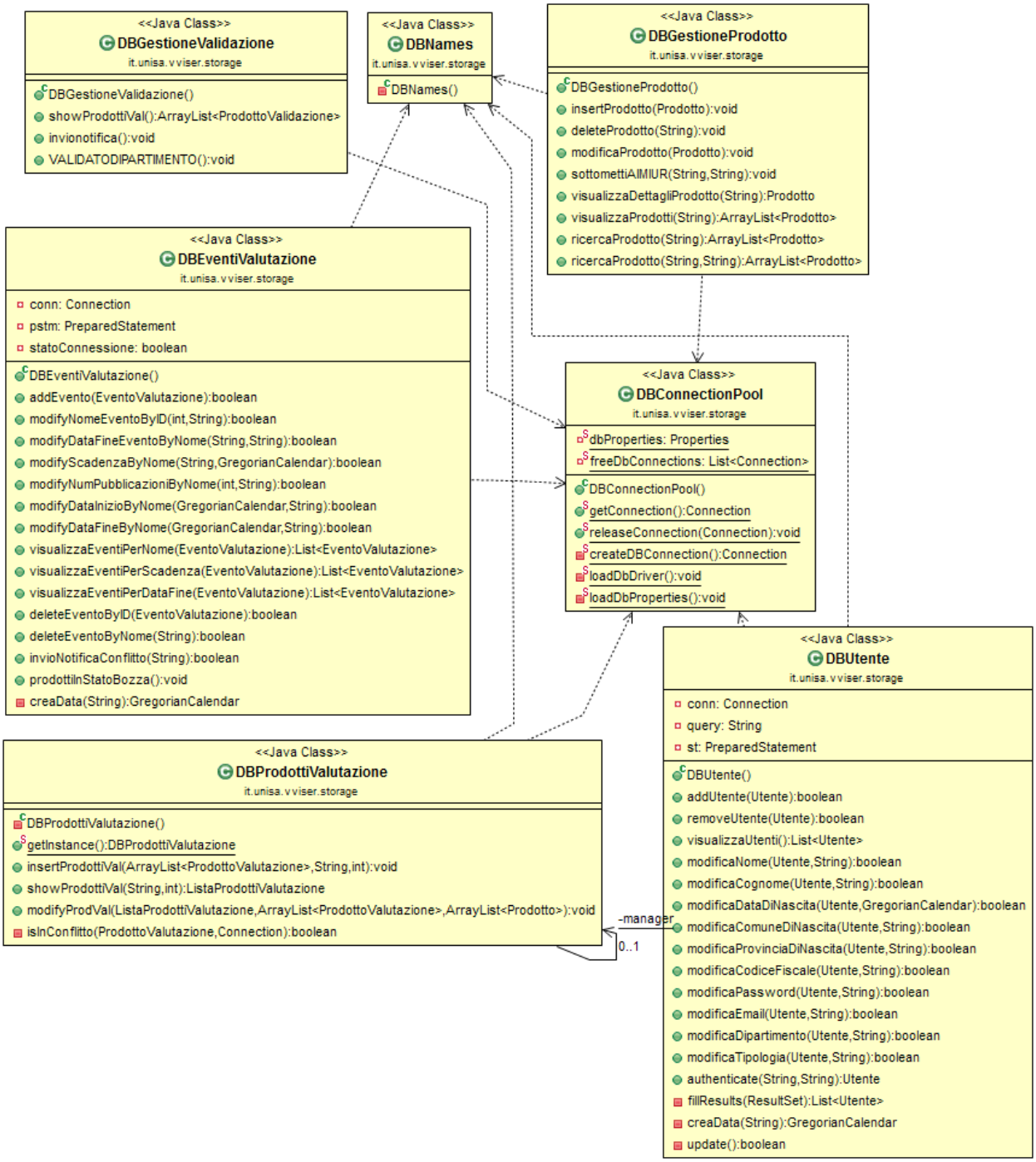


Figura 2 Diagramma delle classi appartenenti al package storage

4 DESIGN PATTERN

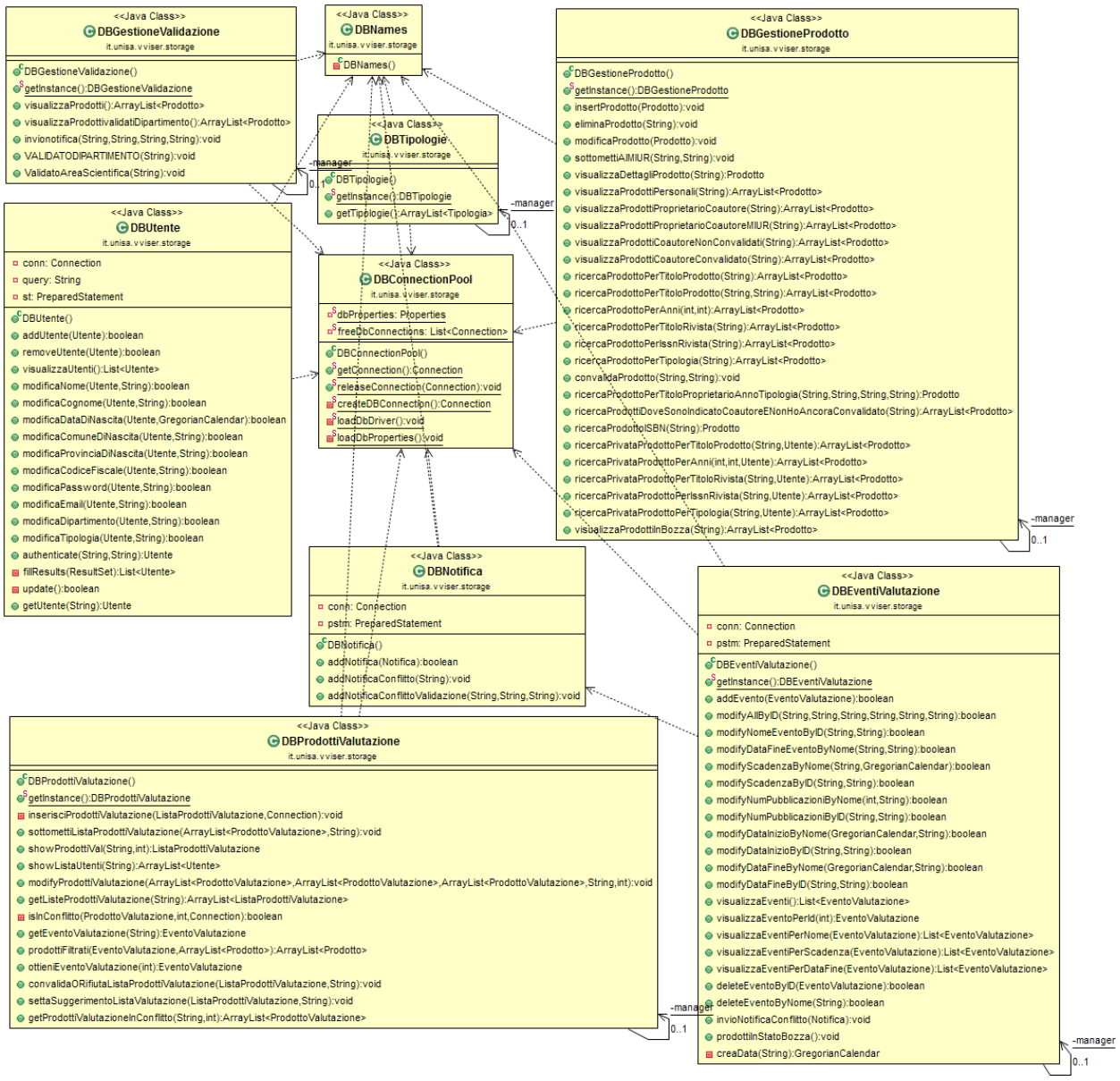
Nel nostro sistema abbiamo deciso di utilizzare principalmente due design pattern: singleton e facade.

Singleton Design Pattern: VViser fa uso del Singleton Pattern nello Storage Layer per impedire la creazione di più di una istanza di oggetto.

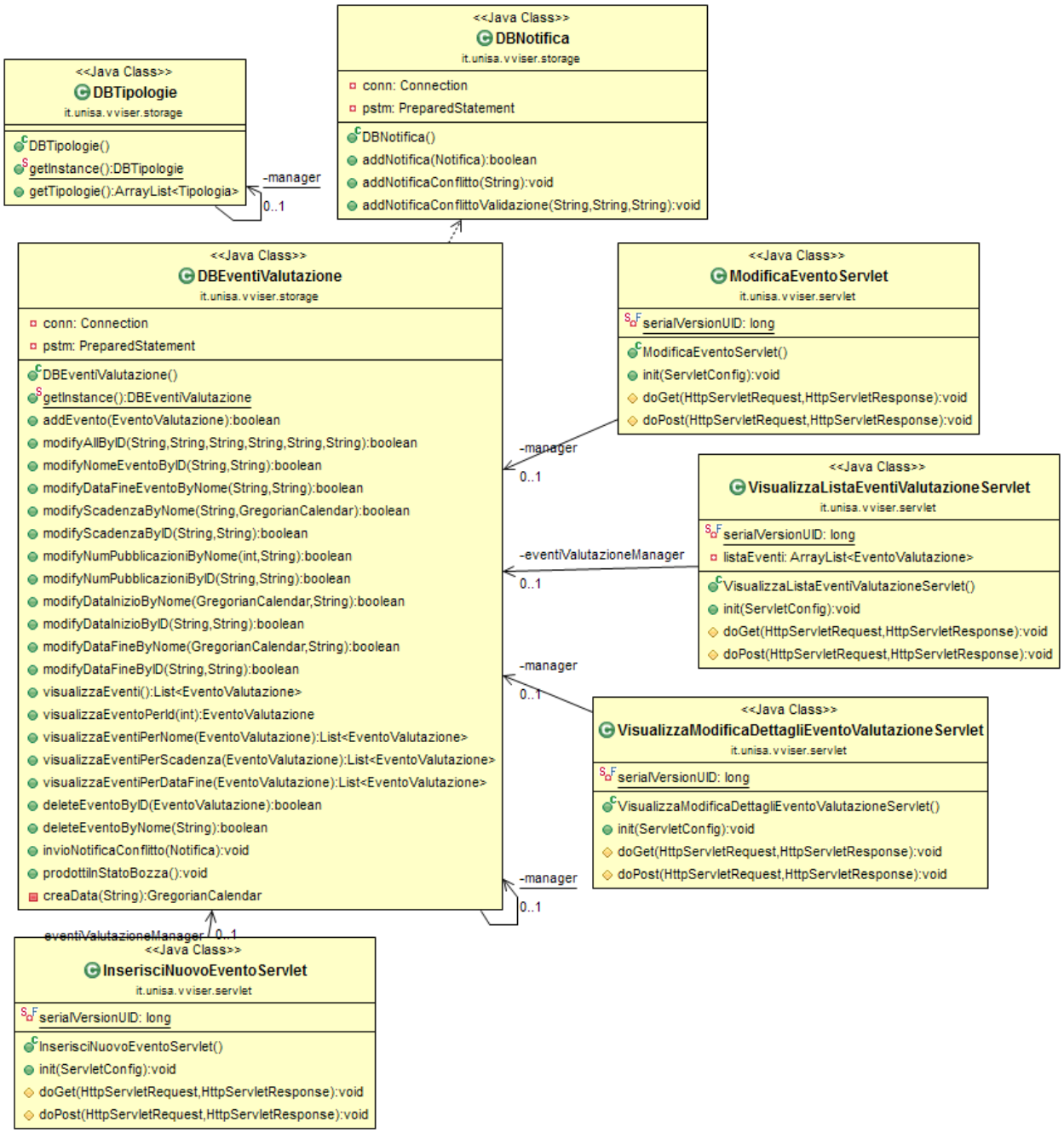
Façade Design Pattern: VViser fa uso del Façade Pattern per permettere di accedere alle funzionalità di tutte le gestioni tramite un'unica interfaccia di più semplice utilizzo.

5 CLASS DIAGRAM

5.1 Storage

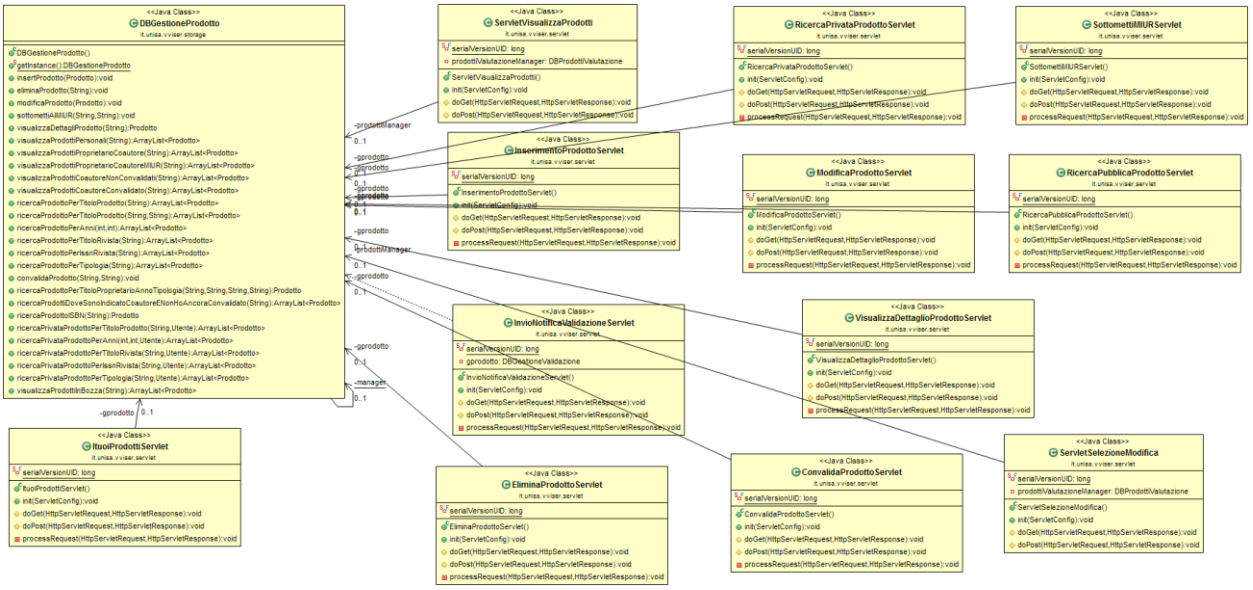


5.2 Gestione Sistema

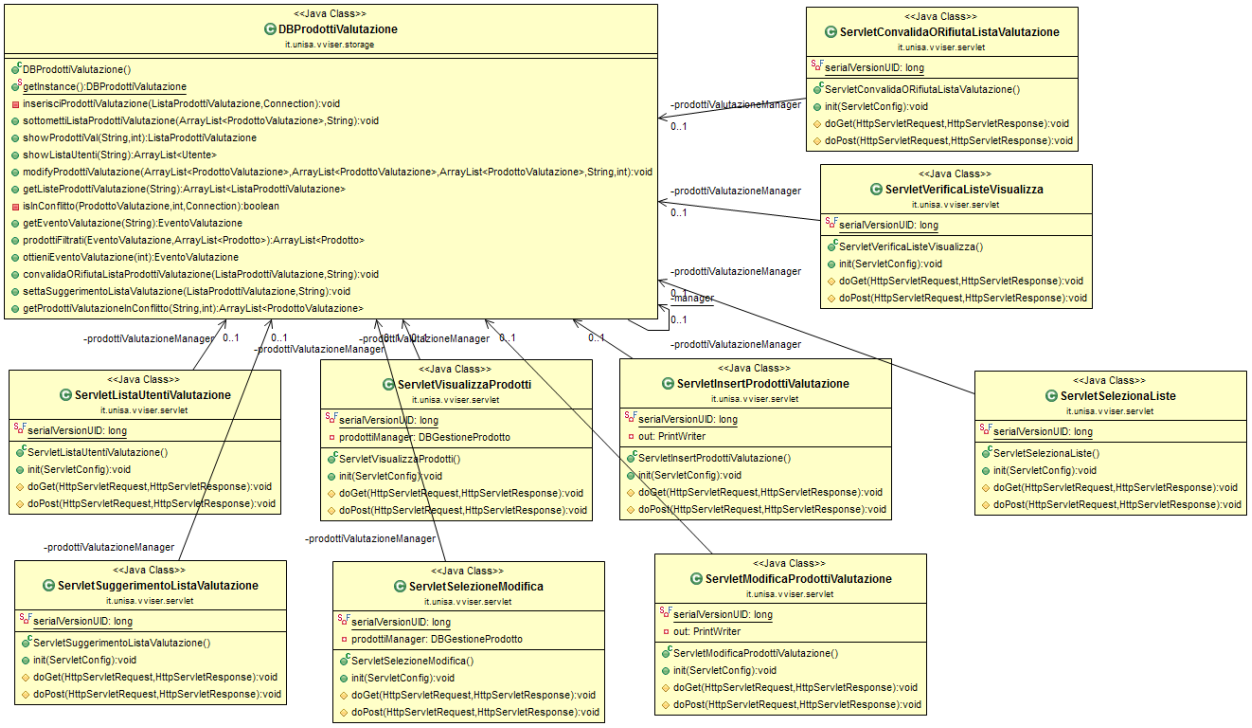


5.3 Gestione Utenti

5.4 Gestione Prodotti



5.5 Gestione Valutazione



5.6 Gestione Validazione