Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Triennale in Ingegneria Informatica

Studio di Hibernate attraverso i costrutti del modello relazionale

Laureando

Marco Carraro

Relatore

Prof. Giorgio Maria Di Nunzio

Anno Accademico 2011/2012

Alla mia famiglia e a Silvia ringraziandovi per il vostro sostegno, la vostra pazienza e la vostra comprensione

Marco Carraro

Indice

1 Introduzione

Ogni programmatore prima o poi nella sua vita incontra il problema di come rendere persistenti gli oggetti che usa nelle sue applicazioni, ovvero come far sopravvivere tali oggetti anche dopo la fine di un'applicazione. Un primo metodo che risolve questo problema è quello di utilizzare dei file, manovra che però porta via moltissimo tempo nella scrittura di classi e metodi ad hoc per ogni applicazione e che non fornisce un controllo degli accessi concorrenti. Tale metodo è quindi poco usato anche se rimane una buona soluzione per necessità di pochi dati persistenti utilizzati da un'unica persona.
Un metodo alternativo consiste l'utilizzo di sistemi di gestione di basi di dati che consentono di accelerare il lavoro dei programmatori fornendo la possibilità di gestire l'accesso multiplo ai dati, controllando la ridondanza nei dati, introducendo meccanismi di *backup and restore* e molti altri vantaggi.
Nel linguaggio Java per accedere ai database bisogna ricorrere alle connessioni JDBC[1] (Java DataBase Connectivity) che fungono da collegamento tra applicazione e base di dati. Uno studio fatto recentemente tra i programmatori Java sottolinea come il setting dei database attraverso connessioni JDBC occupi mediamente un terzo del tempo totale per lo sviluppo di un'applicazione (tra scrittura del codice e mantenimento delle connessioni).
Per ovviare a questo problema sono nati gli ORM (Object Relational Mapping)[2] che sono delle tecniche di programmazione che forniscono interfacce per trasformare dati da modello relazionale a modello ad oggetti e viceversa. Tali applicazioni consentono al programmatore un elevato riutilizzo del codice e permettono di snellire enormemente la procedura per rendere gli oggetti persistenti.
Hibernate[3] è un'implementazione open source degli ORM. L'obiettivo primario dei suoi sviluppatori è quello di semplificare il processo di persistenza degli oggetti, rendendo trasparente l'accesso ai DBMS[4](DataBase Management System) e nascondendo il particolare tipo di implementazione utilizzato.
Hibernate riesce a soddisfare tale obiettivo introducendo i seguenti vantaggi:

- Produttività: Il codice relativo alla persistenza degli oggetti può essere la parte più tediosa di un'applicazione. Hibernate elimina gran parte del codice necessario per rendere persistenti gli oggetti. Questo per ogni tipo di strategia: analizzando un problema e creando il database relativo o partendo da un database esistente.

- Mantenimento: Scrivere meno righe di codice rende un'applicazione più comprensibile poichè enfatizza la logica relativa al problema in esame, ma questo non è l'unico vantaggio di Hibernate che attraverso i suoi tool permette anche una migliore gestione delle modifiche allo schema relazionale. Se c'è bisogno infatti di modifiche da una

[1] `http://it.wikipedia.org/wiki/JDBC`
[2] `http://it.wikipedia.org/wiki/Object-relational_mapping`
[3] `http://it.wikipedia.org/wiki/Hibernate`
[4] `http://it.wikipedia.org/wiki/Database_management_system`

parte o dall'altra le operazioni da effettuare sono ben definite e limitate in modo da migliorare il mantenimento dell'applicazione.

- Controllo migliore delle performance: Hibernate introduce anche delle tecniche che permettono di controllare le performance di un'applicazione ottimizzando al massimo le interazioni con il/i database utilizzati nell'applicazione creata.

- Indipendenza da architetture/prodotti: Hibernate riesce ad astrarre l'applicazione rispetto al DataBase Management System utilizzato e al dialetto SQL utilizzato. Questo permette di spostare l'applicazione su altre macchine che utilizzano diversi DBMS mantenendone il funzionamento.

1.1 Obiettivi

Hibernate è un progetto nato diversi anni fa che è molto diffuso tra gli sviluppatori. Per questo c'è un'ampia letteratura che riguarda il mondo degli ORM e che si pone l'obiettivo di semplificare la vita ai programmatori che si avvicinano a questo mondo.
Poichè Hibernate si pone a metà tra il mondo dei database e la programmazione ad oggetti per comprenderne tutte le sfaccettature è necessario capire tutti e due i mondi. Uno dei maggiori problemi riguardante la letteratura dell'argomento è che le spiegazioni e la documentazione presente è quasi interamente dedicata ad un esperto programmatore Java che deve rendere persistenti i suoi oggetti.
Non c'è, o comunque è imprecisa, della letteratura che analizzi Hibernate dal punto di vista del modello relazionale e di come implementare efficacemente in Hibernate situazioni descritte molto bene da uno schema relazionale.
L'obiettivo di questa tesi è quello quindi di fornire della documentazione su come far funzionare Hibernate a partire da problemi modellizzati con uno o più schemi relazionali. La trattazione che verrà seguita sarà quindi la seguente:

1. Analizzare problemi reali (verranno ripresi esempi a partire dal sito di aste online presente nel libro in bibliografia alla voce [1]);

2. Ricavare schemi Entity-Relationship secondo lo standard del libro in bibliografia alla voce [2];

3. Mostrare possibili traduzioni in schemi relazionali (a volte più di una possibile traduzione);

4. Fornirne l'implementazione con Hibernate.

Tutti gli esempi mostrati sono stati tutti testati con gli strumenti che verranno indicati nel paragrafo seguente. Per non appesantire la trattazione dei vari esempi non verrà riportato l'intero codice ma solo dei frammenti significativi. Per vedere l'intero codice funzionante è

possibile collegarsi a:
`http://www.dei.unipd.it/~carraro1`

1.2 Strumenti Utilizzati

1.2.1 Eclipse

Eclipse[5] è un IDE(Integrated Development Environment) ovvero un ambiente di sviluppo multi-linguaggio e multi-piattaforma open-source. E' stata utilizzata la versione Indigo per la realizzazione degli esempi Java con Hibernate.

1.2.2 PostgreSQL

PostgreSQL[6] è un completo database relazionale open-source. E' stato utilizzato negli esempi sebbene come già detto nei precedenti paragrafi la scelta di un DBMS non sia vincolante per il funzionamento dell'applicazione. Ho utilizzato questo DBMS per coerenza con i corsi che ho frequentato in questo corso di laurea.

1.2.3 OmniGraffle

OmniGraffle[7] è un software di disegno che gira su ambiente Mac. Permette di disegnare flow-chart, Diagrammi Entity-Relationship e numerosi altri tipi di grafici e diagrammi. E' stato utilizzato per creare le varie figure presenti nella tesi.

1.2.4 PhpPgAdmin

PhpPgAdmin[8] è un'applicazione PHP libera che consente di amministrare in modo semplificato database di PostgreSQL tramite un qualsiasi browser. E' stato utilizzato per verificare i risultati delle varie classi di Test dei vari esempi e per la produzione delle immagini dei risultati.

[5] `http://it.wikipedia.org/wiki/Eclipse_(informatica)`
[6] `http://it.wikipedia.org/wiki/PostgreSQL`
[7] `http://www.omnigroup.com/products/omnigraffle/`
[8] `http://it.wikipedia.org/wiki/PhpPgAdmin`

2 Modello Concettuale e Modello Relazionale

In questa tesi si farà più volte riferimento al modello concettuale ed al modello relazionale. In questo capitolo verranno dati i fondamenti teorici riguardanti i due modelli per poter meglio comprendere di cosa si sta parlando.

2.1 Modello Concettuale

La modellazione concettuale è una fase molto importante nella progettazione di una buona base di dati. Infatti solitamente la progettazione si articola nei seguenti passaggi:

1. Raccolta ed analisi dei requisiti attraverso interviste al committente;
2. Progettazione concettuale;
3. Progettazione logica;
4. Progettazione fisica.

Dalla lista precedente si può capire come la progettazione concettuale sia un passo fondamentale per arrivare ad una buona base di dati. In questa tesi per i vari esempi si partirà direttamente da uno schema concettuale già fatto, si tradurrà in schema relazionale (progettazione logica) e si implementeranno con Hibernate i vari schemi trovati.
Il modello concettuale utilizzato sarà il modello Entity Relationship[9], che è un modello ad alto livello di astrazione formalizzato da Peter Chen[10]. Poichè non esiste uno standard per tale diagramma verrà utilizzato quello adottato nel corso di Basi Di Dati frequentato nel terzo anno della laurea triennale di Ingegneria Informatica a Padova. Tale standard è quello usato nel libro in bibliografia alla voce [2].
Il modello Entity-Relationship si basa su tre definizioni ovvero:

- **Entità**: Un'entità rappresenta una classe di oggetti che ha caratteristiche comuni ai fini dell'applicazione di interesse. In ogni schema entity relationship l'entità viene indicata con un rettangolo con all'interno il suo nome univoco;
- **Associazione**: Un'associazione rappresenta un legame esistente tra due o più entità. Il numero di entità collegate dall'associazione è chiamato grado dell'associazione. In questa tesi sono presenti solo associazioni binarie e ternarie in quanto le altre sono spesso traducibili come combinazioni di queste. L'associazione viene indicata con un rombo con all'interno il suo nome;

[9] `http://it.wikipedia.org/wiki/Modello_E-R`
[10] `http://en.wikipedia.org/wiki/Peter_Chen`

- **Attributo**: Le entità e le associazioni sono descritte da una serie di attributi che esprimono delle caratteristiche e dell'informazione riguardante un'entità o un'associazione. Gli attributi vengono indicati con un pallino collegato all'entità o all'associazione a cui appartengono. Il pallino può essere vuoto se indica un attributo con valori che possono ripetersi tra un'istanza ed un'altra dell'associazione o dell'entità oppure pieni.

In figura 1 è presente un'esempio di schema Entity-Relationship dove X e W possono essere o 0 o 1 anche diversi tra loro mentre Y e Z possono assumere 1 o N dove N assume il significato di molti.

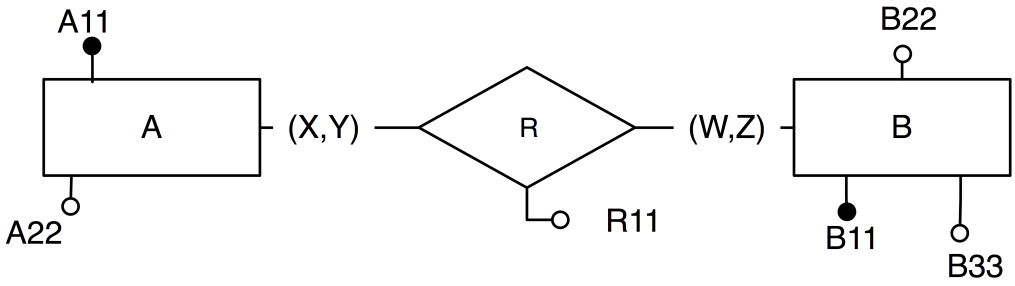

Figura 1: Esempio di schema E-R

2.2 Modello Relazionale

Il modello relazionale fu introdotto da Codd[11] della IBM nel 1970. Ebbe molto successo in quanto molto semplice grazie al fatto che usava i concetti matematici di insieme e della logica del primo ordine.
Il modello relazionale presenta la base di dati come una collezione di relazioni basandosi quindi sulla definizione di *relazione matematica*, formata da stato della relazione e schema di relazione.
Prima di introdurre tali concetti è opportuno introdurre i seguenti:

- Dominio: Un dominio D è un insieme di valori atomici. Per atomici si intende che ogni valore nel dominio è indivisibile, per lo meno per quanto riguarda il modello relazionale.

- Attributo: Un attributo A_i fornisce il nome del ruolo interpretato da un certo dominio D.

[11] `http://en.wikipedia.org/wiki/Edgar_F._Codd`

Di seguito i concetti di schema di relazione e stato della relazione:

- Uno **schema di relazione** R, indicato con $R(A_1, A_2, ..., A_n)$, è costituito da un nome di relazione R e da un elenco di attributi $A_1, A_2, ..., A_n$. Ogni attributo A_i come già detto è il nome del ruolo interpretato da un certo dominio D nello schema di relazione R. Tale D è detto dominio di A_i.
  Uno schema di relazione è usato per descrivere una relazione; R è detto nome della relazione, mentre il grado della relazione è n ovvero il numero di attributi che ne fanno parte.

- Uno **stato di relazione** r dello schema di relazione $R(A_1, A_2, ..., A_n)$, indicato anche con r(R), è un insieme di n-tuple $r = \{t_1, t_2, ..., t_n\}$. Ogni n-tupla t è un elenco ordinato di n valori $t =< v_1, v_2, ..., v_n >$ dove ogni valore $v_i, 1 \leq i \leq n$ è un elemento del dominio di A_i oppure è uno speciale valore null.

Il concetto di relazione è rappresentabile efficacemente mediante una tabella. Quest'ultima però rappresenta un concetto differente rispetto a quello di relazione. In una relazione infatti non è presente il concetto di ordine tra le tuple al suo interno cosa che mediante rappresentazione tabellare si introduce. Infatti in generale una relazione è un insieme di $\mathbb{R}^n$ mentre una tabella è una rappresentazione piatta di tale insieme.
Nelle relazioni in una base di dati è possibile anche introdurre dei vincoli, alcuni dei quali già introdotti tramite la definizione di una relazione. Di seguito sono elencati questi vincoli:

- Vincoli di dominio: Questo tipo di vincolo impone che all'interno di ciascuna tupla il valore di ogni attributo A deve essere un valore atomico del suo dominio D o al massimo un valore speciale null.

- Vincoli di chiave: Una relazione come già detto è formata da un insieme di tuple. Poichè si parla di insieme non sono ammessi valori duplicati al suo interno, quindi le tuple devono essere differenti tra loro. Tali differenze sono visibili osservando un sottoinsieme di attributi SK di uno schema di relazione R che hanno la proprietà che prese due tuple distinte $t_i, t_j \Rightarrow t_i[SK] \neq t_j[SK]$. Tali sottoinsiemi che soddisfano questa proprietà sono detti superchiavi di R.

- Vincoli di Integrità Referenziale: Questo tipo di vincolo introduce delle imposizioni sul valore di un insieme di attributi di una relazione R1 rispetto ai valori di un altro insieme di attributi di un'altra relazione R2. Sono usati per collegare tra di loro le relazioni. Sono questi i tipi di vincoli più interessanti, in quanto sono i più difficili da capire per Hibernate.

3 Implementazione di Schemi Relazionali con Hibernate

In questa sezione si fornirà della documentazione per mappare correttamente con Hibernate situazioni possibili di schemi relazionali possibili nella modellazione di un database. Per ogni esempio si cercheranno di dare tutte le possibili traduzioni in schema relazionale possibili.

3.1 Configurazione di Hibernate

In questo paragrafo verrà mostrata la configurazione di Hibernate utilizzata per far funzionare correttamente tutti gli esempi illustrati in seguito. Come già descritto nell'Introduzione, uno dei vantaggi introdotti da Hibernate è il largo riutilizzo del codice da un'applicazione all'altra, per questo affinchè vi siano meno ripetizioni possibili nei paragrafi seguenti verranno omesse delle classi e dei file di configurazione identici o molto simili tra i vari esempi. Tali configurazioni e classi verranno mostrati quindi una sola volta.

3.1.1 Package e directory

Tutti gli esempi sono stati testati con i package e le directory presenti nella seguente figura:

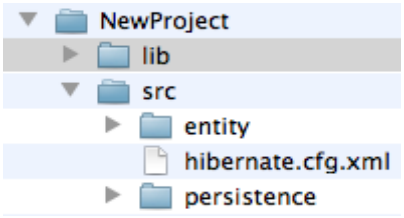

Figura 2: Package And Directory

La cartella src contiene tutto il codice relativo al progetto (classi e file .xml), in particolare contiene due ulteriori package ovvero entity e persistence più un file di configurazione hibernate.cfg.xml che verrà spiegato in dettaglio nei prossimi paragrafi. La cartella lib contiene tutte le librerie necessarie al corretto funzionamento di Hibernate, tali librerie devono venire correttamente importate da Eclipse alla creazione del progetto nuovo (non basta metterle dentro la cartella lib). Il package entity è quello che contiene tutte le classi e i file xml relativi ad ogni esempio, mentre il package persistence contiene al suo interno la classe HibernateUtil.java. Tutto il codice da qui in avanti si rifarà a tale organizzazione.

3.1.2 hibernate.cfg.xml

Il file hibernate.cfg.xml è il file da cui Hibernate preleva le configurazioni per poter funzionare correttamente. Lo scopo principale di tale file è quello di costruire una SessionFactory

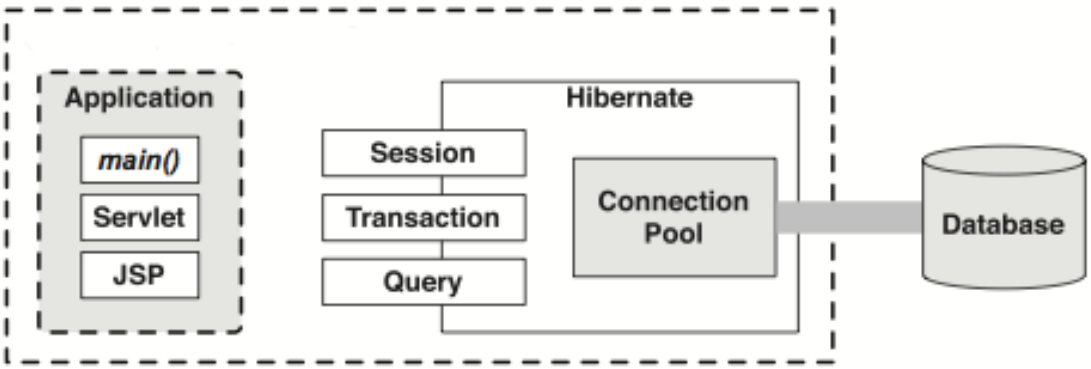

Figura 3: Hibernate e Connection Pool

a partire dalle configurazioni date, che servirà per poter usare le sessioni ovvero gli oggetti che serviranno a colloquiare con il database. Nel listato 1 si mostra quello utilizzato per gli esempi in questa tesi. Il DTD usato (righe da 2 a 4) serve per dare possibilità al parser XML di validare il file.
Le righe da 7 a 21 danno ad Hibernate le informazioni necessarie perchè conosca il DBMS utilizzato nel sistema corrente, queste sono le uniche righe da modificare in caso di cambio di DBMS (anche in presenza di applicazioni complicate si modificano solo poche righe di codice e tutto può girare anche su sistemi con DBMS diversi).
Le righe da 23 a 31 servono a settare il Database Connection Pool che è un insieme di connessioni mantenute in cache per minimizzare il costo dell'apertura e della chiusura di queste e velocizzare le performance delle interazioni con un database. Per questa tesi è stato usato c3p0[12] che è un Database Connection Pool open-source facilmente reperibile. In figura 3 è schematizzato l'ambiente Java con Hibernate e il Connection Pool.

Le righe relative al connection pool sono opzionali ma ci sono tre buoni motivi per utilizzarle:

- Acquisire una nuova connessione è costoso. Inoltre alcuni DBMS fanno partire un nuovo processo server per ogni connessione;
- Mantenere connessioni idle è costoso per i DBMS e il pool ne ottimizza l'uso;
- Il pool fornisce un meccanismo di caching che permette di immagazzinare query finchè la connessione non sia pronta a riceverle.

[12] `http://sourceforge.net/projects/c3p0/`

Listing 1: File Hibernate.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">
            org.postgresql.Driver
        </property>
        <property name="hibernate.connection.password">
            hibernate
        </property>
        <property name="hibernate.connection.url">
            jdbc:postgresql:hibernate-test
        </property>
        <property name="hibernate.connection.username">
            hibernate
        </property>
        <property name="hibernate.dialect">
            org.hibernate.dialect.PostgreSQLDialect
        </property>

        <property name="hibernate.c3p0.min_size">5</property>
        <property name="hibernate.c3p0.max_size">20</property>
        <property name="hibernate.c3p0.timeout">300</property>
        <property name="hibernate.c3p0.max_statements">
            50
        </property>

        <property name="hbm2ddl.auto">create</property>

        <mapping resource="entity/Item.hbm.xml"/>
        <mapping resource="entity/Shipment.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

Modificando quindi le righe di codice relative al pool è possibile agire direttamente sulle performance di un'applicazione che usa Hibernate. Ecco la spiegazione relativa ai campi del Database Connection Pool:

- La riga 23 fornisce la quantità minima di connessioni tenute pronte dal connection pool, mentre la riga 24 da il numero massimo di connessioni nel pool.

- La riga 25 fornisce il tempo massimo dopo il quale una connessione idle viene rimossa dal pool

- Le righe 26-28 danno il numero massimo di prepared statements che possono essere in cache. Questo campo è molto importante per le performance di Hibernate.

Infine per poter usare c3p0 è necessario aggiungere le librerie necessarie al suo funzionamento (verranno spiegate nella sezione 3.1.4).
La riga 30 è molto importante a livello di debug, infatti la proprietà hbm.dll.auto dice ad Hibernate cosa fare sul database sul quale si lavora. In questo caso la proprietà è settata su create, in questo modo Hibernate provvede a rimuovere lo schema presente appena l'applicazione si avvia e lo rimpiazza con quello nuovo generato dall'applicazione. In poche parole con questa impostazione si perderebbero tutti i dati immagazzinati sul database ad ogni riavvio dell'applicazione. Ovviamente ciò non è quello che si vuole normalmente ma in fase di creazione del database questa proprietà è utile. Una volta che si è verificata la bontà dell'applicazione creata la riga va commentata o tolta.
Infine la righe da 32 a 33 forniscono un'indicazione di dove si trovano i file di mappatura necessari per far capire ad Hibernate la struttura delle classi e come dialogare con il database. Tali file di mappatura sono differenti da esempio ad esempio e vanno sempre specificati sempre tutti in queste righe.

3.1.3 Package Persistence

Il package persistence contiene al suo interno un'unica classe ovvero HibernateUtil.java. Tale classe indicata nel listato 2 serve per creare correttamente la SessionFactory a partire dal file di configurazione hibernate.cfg.xml. Poichè tali operazioni sono comuni a tutti gli esempi si è scelto di creare tale classe per poi importarla nelle classi di Test degli esempi per rendere meglio visibile il codice.

3.1.4 Librerie

Come già specificato, la cartella lib contiene tutte le librerie utilizzate negli esempi di questa tesi. In tale cartella sono presenti tutte le librerie necessarie per il funzionamento di Hibernate che sono:

Listing 2: HibernateUtil.java

```
1  package persistence;
2  import org.hibernate.*;
3  import org.hibernate.cfg.*;
4  public class HibernateUtil {
5     private static SessionFactory sessionFactory;
6     static{
7        try{
8           sessionFactory=new Configuration().configure().
9                                    buildSessionFactory();
10       }catch(Throwable ex){
11          throw new ExceptionInInitializerError(ex);
12       }
13    }
14    public static SessionFactory getSessionFactory(){
15       return sessionFactory;
16    }
17    public static void shutdown(){
18       getSessionFactory().close();
19    }
20 }
```

- Librerie associate al core di Hibernate[13]
- Libreria associata al DBMS[14]
- Libreria di c3p0[15]

3.2 Classi e Files di Mappatura

Per poter correttamente costruire un ponte tra il mondo database e il mondo Java, Hibernate ha bisogno di conoscere le informazioni relative all'applicazione come gli attributi che verranno utilizzati nelle classi e che relazioni si intende ottenere. Tali informazioni vengono prelevate da dei file di mappatura scritti in XML che si accompagnano alle varie classi. In tali file sono presenti informazioni sugli attributi e soprattutto sui vincoli tra una classe e l'altra dalle quali Hibernate potrà tradurre correttamente i vincoli nel mondo relazionale.

[13] Le librerie sono scaricabili da
`http://sourceforge.net/projects/Hibernate/files/latest/download?source=files`
al percorso dist/lib/required

[14] Per PostgreSQL tale file è postgresql-9.1-901.jdbc4.jar scaricabile da
`http://jdbc.postgresql.org/download/postgresql-9.1-902.jdbc4.jar`

[15] Il file si chiama c3p0.jar ed è scaricabile da
`http://sourceforge.net/projects/c3p0/files/latest/download` dentro la cartella lib

Listing 3: Formato file di mappatura XML

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="package.ClassName" table="TableName">
        <id name="IdName" column="ID_COLUMN_NAME">
            <generator class="CLASS">
                ...
            </generator>
        </id>
        <property name="via" column="VIA"/>
        ...
    </class>
</hibernate-mapping>
```

Un file di mappatura XML ha la struttura mostrata nel listato 3. Come si può vedere dunque ogni attributo che non abbia un vincolo di integrità referenziale viene mappato tramite un'unica riga mentre attributi che riferiscono a campi di relazioni diverse saranno mappate attraverso diversi tag come si vedrà nei prossimi esempi.

3.3 Associazioni Binarie Uno a Uno

Per associazione binaria uno ad uno si intende un'associazione che coinvolge due entità con cardinalità (0,1) o (1,1) da entrambi i lati dell'associazione. Ciò implica che un'istanza di un'entità può essere collegata tramite l'associazione al massimo ad un'istanza dell'altra entità. A seconda che la cardinalità minima sia 0 o 1 la traduzione possibile in schema relazionale può variare. Se la cardinalità è (1,1) ambo i lati siamo in una situazione di questo tipo:

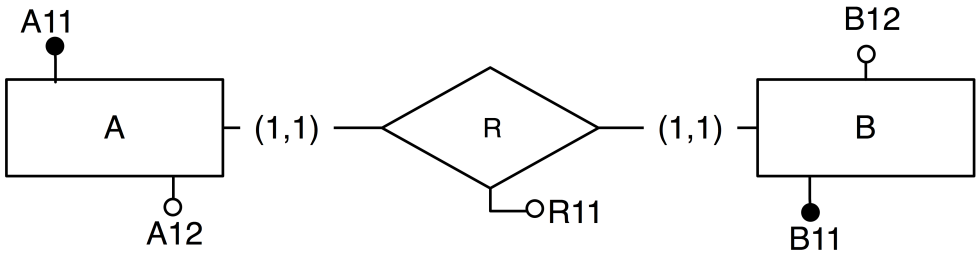

Figura 4: Modello Entity-Relationship Ass Uno ad Uno con cardinalità (1,1)

Solitamente è opportuno tradurre tale situazione nel seguente schema relazionale:

A(A11,A12,R11,B12)

Ciò è dovuto al fatto che A11 e B11 assumono gli stessi valori e normalmente l'entità A e l'entità B non rappresentano concetti molto differenti tra loro. Tale soluzione verrà ripresa nei prossimi esempi di associazioni uno ad uno con particolarità quali entità deboli o partecipazioni parziali di alcune entità all'associazione.

3.3.1 Associazione Uno ad Uno con Entità Debole

Consideriamo una base di dati che raccoglie gli utenti di un sito web. Ognuno di questi utenti deve avere un solo indirizzo e gli indirizzi di due utenti qualsiasi nella base di dati devono essere diversi, non può quindi capitare che due utenti condividano lo stesso indirizzo. Lo schema concettuale che descrive il problema è il seguente:

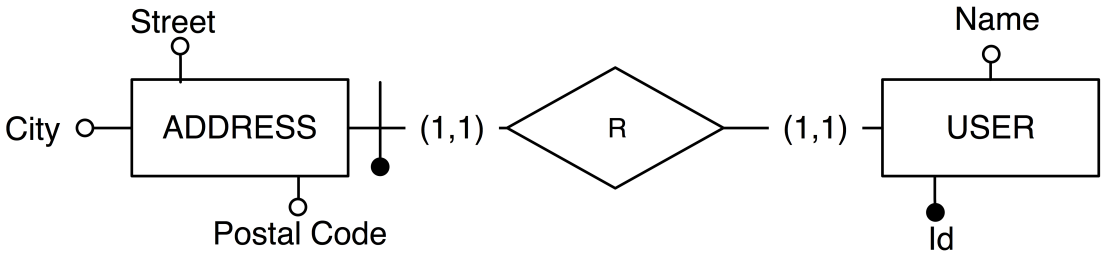

Figura 5: Modello Entity-Relationship Address e User

Ci sono due possibili schemi relazionali ottenibili mediante traduzione ovvero:

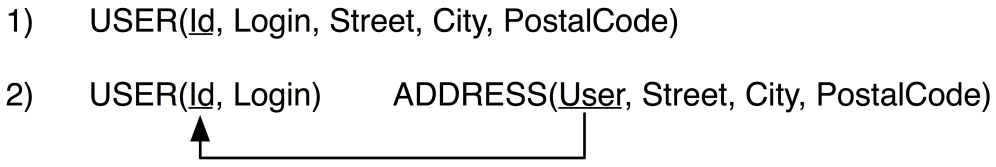

Figura 6: Schemi Relazionali Possibili

Normalmente il primo caso è preferibile al secondo in quanto si hanno meno relazioni, ma per completezza verranno illustrate entrambe le mappature anche perchè la scelta va fatta valutando volume dei dati e query più frequenti.
Per rappresentare correttamente i concetti del problema in esame si creeranno due classi Java una relativa a User ed una per Address che chiameremo User.java ed Address.java. Per questo esempio verranno mostrate tali classi rispettivamente nei listati 4 e 5, mentre

Listing 4: User.java

```
1  package entity;
2
3  public class User {
4          private Long id;
5          private String login;
6          private Address address;
7
8          public User(){}
9
10         public User(String user){
11                 login=user;
12         }
13         public User(String user, Address address){
14                 login=user;
15                 this.address=address;
16         }
17         //getter and setter
18 }
```

per i prossimi non saranno mostrate. A seconda che la traduzione scelta sia la prima o la seconda bisogna accompagnare le classi Java a delle determinate mappature in XML, di seguito si analizzano tutti e due i casi:

- **Accorpamento in un'unica entità**: E' necessario utilizzare un unico file XML di mappatura che si occuperà di creare l'unica relazione presente nello schema relazionale. Gli attributi accorpati ovvero Street, PostalCode e City vengono riconosciuti attraverso l'utilizzo del tag <component> che richiama una classe esterna rispetto a quella che si sta mappando. Nel listato 6 è presente il file di mappatura per User. Tale file va salvato assieme alle classi Java nel package entity ed è sufficiente per la corretta creazione/interazione con il modello relazionale di fig 6 caso 1)

- **Due Relazioni Distinte**: Se la traduzione scelta è quella di tradurre lo schema in due relazioni distinte allora è necessario modificare il file XML User.hbm.xml ed aggiungere anche un file di mappatura Address.hbm.xml per la relazione Address. Un'altra modifica da fare è rendere Address una Entity type. I concetti di Entity Type e Value Type sono molto importanti in Hibernate, il primo indica una classe con un identificatore univoco mentre il secondo rappresenta una classe senza identificatore e che viene identificata esternamente. Tale concetto ricorda molto quello di entità debole ed entità forte.
  Basta quindi aggiungere un attributo privato di tipo long chiamato id alla classe Address insieme al suo metodo *getter* e al suo metodo *setter* per renderla un Entity

Type. Per rendere bidirezionale l'associazione inoltre bisogna aggiungere anche un attributo di tipo User ad Address chiamato address. I listati 7 e 8 mostrano i file di mappatura. Come si può notare gli identificatori della tabella relativa ad Address sono chiavi esterne verso gli identificatori della tabella relativa ad User. Ciò si ottiene attraverso il tag <generator> con attributo class=foreign che richiede di specificare a che parametro rifarsi per la generazione degli identificatori. In questo caso tutto funziona perchè è presente il tag <one-to-one> alla fine del file Address.hbm.xml che collega user (l'oggetto di tipo User nella classe Address) alla classe User. L'attributo constrained=true è necessario perchè venga creata correttamente la chiave esterna alla tabella relativa ad User.

3.3.2 Associazione Uno Ad Uno con Entità Debole e Partecipazione Parziale Dell'Entità Forte

Riprendendo l'esempio della sezione precedente supponiamo che non sia obbligatorio che ogni utente indichi il proprio indirizzo (che rimane comunque al massimo uno). Questa piccola differenza si riflette nella cardinalità minima di User che diventa 0 come mostrato nello schema Entity-Relationship di figura 7. Anche in questo caso verranno utilizzate le

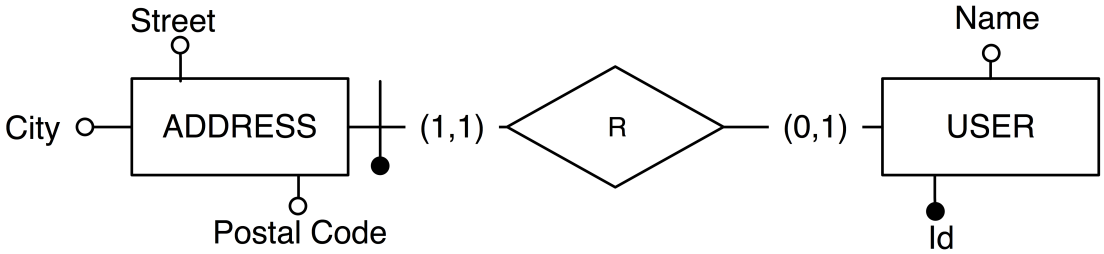

Figura 7: Schema Entity-Relationship User-Address con partecipazione facoltativa di User

stesse classi già mostrate nei listati relativi ad User ed Address. La traduzioni in schema relazionale possibili sono analoghe a quelle indicate precedentemente (fig. 6), in questa situazione però il caso 1) andrebbe scelto solo se si fosse sicuri che la maggior parte degli utenti indichi il proprio indirizzo altrimenti si rischierebbe di sprecare molto spazio per valori nulli.
In questo paragrafo dunque si analizzerà il secondo caso che prevede due relazioni separate. Per tradurre correttamente con Hibernate lo schema E-R in due relazioni è necessario aggiungere al file User.hbm.xml anche un altro file di mappatura Address.hbm.xml. Tale file presenta sempre le solite caratteristiche per la corretta mappatura ma ha una dichiarazione dell'identificativo particolare come mostrato dal frammento nel listato 9. Come si può notare dal file quando viene creato un oggetto di tipo Address e viene reso persistente questo prende come identificatore lo stesso dell'istanza di User a cui è collegato (tale informazione è presa dall'attributo user di tipo User presente in Address). Se non è collegato a nessun

Listing 5: Address.java

```java
package entity;

public class Address {
        private String street;
        private int num;
        private int cap;

        public Address(){}
        public Address(String via,int num,int cap){
                this.street=via;
                this.num=num;
                this.cap=cap;
        }
        //setter and getter
}
```

Listing 6: User.hbm.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
   <class name="entity.User" table="USERS">
      <id name="id" column="USER_ID">
        <generator class="native"/>
      </id>
      <property name="login"/>
      <component name="address" class="entity.Address">
        <property name="via"/>
        <property name="numero"/>
        <property name="cap"/>
      </component>
   </class>
</hibernate-mapping>
```

Listing 7: User.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
        <class name="entity.User" table="USERS">
       <id name="id" column="USER_ID">
          <generator class="identity"/>
       </id>
       <property name="login"/>
       <one-to-one name="address" class="entity.Address"
                cascade="save-update"/>
        </class>
</hibernate-mapping>
```

Listing 8: Address.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
        <class name="entity.Address" table="ADDRESSES">
                <id name="id" column="ADDRESS_ID">
           <generator class="foreign">
              <param name="property">user</param>
           </generator>
                </id>
                <property name="via" column="VIA"/>
                <property name="numero" column="NUMERO"/>
                <property name="cap" column="CAP"/>
                <one-to-one name="user" class="entity.User"
            constrained="true"/>
        </class>
</hibernate-mapping>
```

Listing 9: Address2.hbm.xml

```
...
<id name="id" column="ADDRESS_ID">
        <generator class="foreign">
                <param name="property">user</param>
        </generator>
</id>
...
```

oggetto User non appena si tenta di rendere persistente l'oggetto Address viene lanciata un'eccezione da Hibernate.

3.3.3 Associazione Uno Ad Uno con Partecipazione Facoltativa Di Un'Entità

Consideriamo sempre lo stesso esempio che coinvolge User e Address. In questo caso però abbiamo dei vincoli che ci impongono che uno User può avere oppure no un indirizzo (e se lo ha ne ha uno e uno solo) e ogni Address è associato ad uno e un solo User. La situazione può essere schematizzata dal seguente schema E-R:

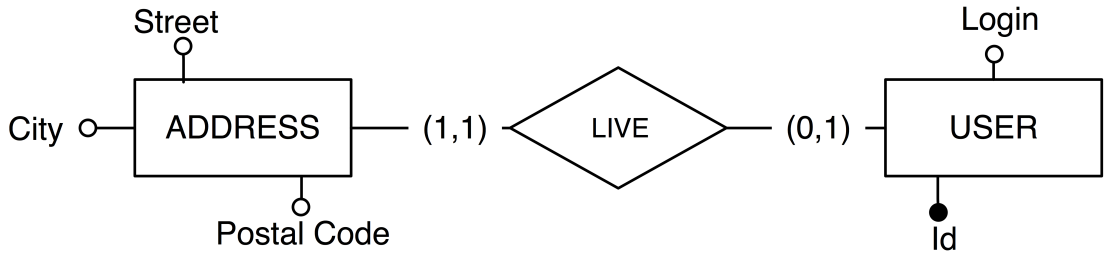

Figura 8: Modello Entity-Relationship sezione 3.3.3

Le classi Java necessarie per la mappatura sono sempre User.java e Address.java con l'eccezione che ora Address necessita di un'attributo di tipo long chiamato id che rappresenterà l'identificatore di Address. Anche per questo caso sono possibili traduzioni analoghe a quelle fatte in precedenza, e se si sceglieranno tali traduzioni le modifiche da effettuare ai casi precedenti sono poche. Analizziamo invece la seguente traduzione:

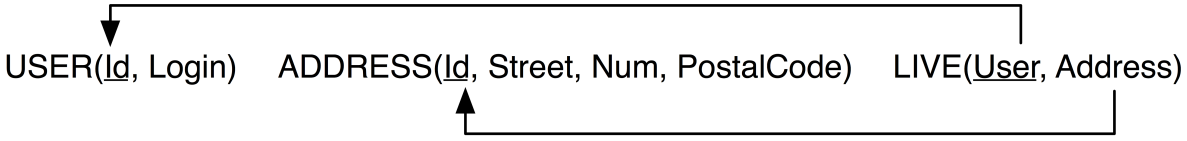

Figura 9: Schema Relazionale a tre relazioni

Listing 10: Frammento User.hbm.xml

```
...
<join table="LIVE" optional="true">
    <key column="USER_ID"/>
    <many-to-one name="address" column="ADDRESS_ID"
            not-null="true" unique="true" cascade="save-update"/>
</join>
...
```

In questo caso dobbiamo generare una relazione centrale che manterrà i collegamenti tra una relazione e l'altra attraverso dei vincoli di integrità referenziali. Tale scelta è ottenibile mediante i soliti due file di mappatura, uno per User e l'altro per Address, denominati User.hbm.xml e Address.hbm.xml. Quest'ultimo non verrà mostrato in quanto non rappresenta nessuna particolarità, l'unica differenza rispetto ai casi precedenti è che la classe Address è diventata un'Entity Type vera e propria con il proprio identificatore di tipo long. Pertanto nel campo id andrà specificato un <generator class=identity/> come per User.hbm.xml.
E' invece interessante notare come si ottenga la relazione centrale LIVE incorporando nel file User.hbm.xml il codice nel listato 10. Si può notare quindi come attraverso il tag <join table> sia possibile definire dei vincoli di integrità referenziali tra entità su una relazione differente. Tale scelta è comoda soprattutto in caso di opzionalità da ambo i lati nell' associazione per evitare scomodi valori nulli nelle due relazioni.

3.4 Associazioni Binarie Uno a Molti

Per associazione binaria uno a molti si intende un'associazione che coinvolge due entità con cardinalità (1,1) o (0,1) da un lato dell'associazione e cardinalità (1,N) o (0,N) dall'altro. Anche in questo caso a seconda che la cardinalità minima sia zero da una o tutte due le parti può far pendere la traduzione in modello relazionale verso una soluzione invece di un'altra. Nei prossimi paragrafi verranno descritte le soluzioni più comuni.

3.4.1 Associazione Uno a Molti Con Liste, Bags, Sets

Quando ci si trova davanti ad una situazione di questo tipo in un programma Java normalmente si hanno due classi una delle quali al suo interno ha come attributo una collezione di oggetti dell'altra classe (questa classe rappresenta l'entità coinvolta nell'associazione con cardinalità (0,N) o (1,N)). C'è quindi bisogno di un oggetto collezione che sia riconosciuto correttamente da Hibernate, così che si creino i corretti vincoli di integrità referenziale tra le relazioni coinvolte.
Hibernate supporta diversi tipi di collezioni tutti presenti o implementabili a partire da

classi contenute nel package *java.util*, in questa tesi verranno analizzate le tre collezioni più utilizzate ovvero Set, Bag e List. Le tre collezioni differiscono per le politiche di gestione degli elementi al loro interno in particolare:

1. **Set**: Un Set è un insieme di elementi non ordinati in cui i duplicati non sono ammessi. E' la scelta più comune con Hibernate e quella di cui è presente la maggior documentazione in rete. Un Set normalmente soddisfa tutte le necessità di una applicazione che debba interagire con database. In Java un Set si implementa utilizzando la classe java.util.HashSet. In questa tesi eccetto questa sezione in cui verranno esemplificati tutti e tre i tipi di collezione si userà un Set ogni qual volta ci sarà bisogno di una collezione (ovvero quando avremo una entità coinvolta in una associazione con cardinalità massima N).

2. **Bag**: Un Bag è una collezione di elementi non ordinati in cui sono ammessi i duplicati. In Java non esiste una classe che implementi un Bag nei package standard ma è possibile servirsi di java.util.Collection e java.util.ArrayList per realizzarne uno.

3. **List**: Un List è una struttura dati di tipo FIFO (First In First Out) e in Java si realizza tramite la classe java.util.ArrayList

Per vedere le differenze tra le mappature utilizzando i tre tipi di collezione analizziamo la seguente situazione:
Consideriamo una base di dati relativa ad un sito di aste online. Tale sito vende degli oggetti (ITEM) all'utente che fa l'offerta (BID) migliore. Per un ITEM possono esserci più offerte ma ogni offerta è relativa ad un unico oggetto. La base di dati deve tenere in memoria tutte le offerte e tutti gli item trattati nel sito.
Lo schema E-R che descrive la situazione è visualizzato in figura 10. Si potrebbe anche

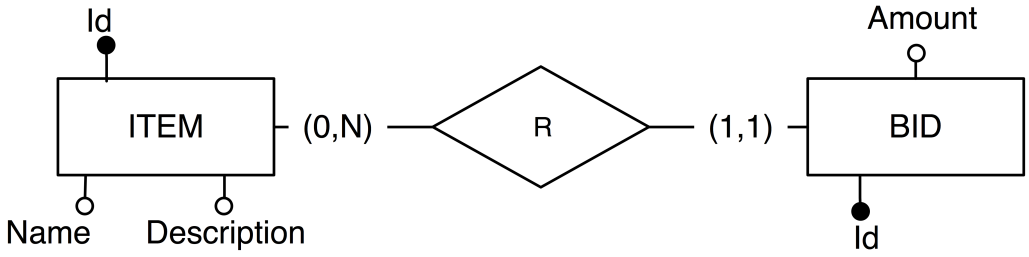

Figura 10: Schema E-R tra ITEM e BID

pensare all'entità ITEM come entità debole ma questo cambierebbe poco dal punto di vista della mappatura con Hibernate (cambierebbe il tag <generator> come si vede nel listato 8).
Le classi Java che sono necessarie per tale esempio sono dunque Bid.java e Item.java, nei listati 11 e 12 vengono mostrati i frammenti significativi di tali classi. Da notare che

Listing 11: Bid.java

```
package entity;
public class Bid {
    private long id;
    private double amount;
    private Item item;
    ...
    //constructors, setter and getter
}
```

Listing 12: Item.java

```
package entity;
public class Item {
    private long id;
    private String name;
    private String description;
    private Set<Bid> bids=new HashSet<Bid>(); //se si usa Set
    /*private List<Bid> bids=new ArrayList<Bid>();//se si usa Bag
        private Collection<Bid> bids=
                        new ArrayList<Bid>();//se si usa List*/
    ...
    //constructors setter and getter
}
```

Listing 13: Frammento di Bid.hbm.xml

```
...
<property name="amount" column="EURO"/>
<many-to-one name="item" column="ITEM_ID" class="Item"
                         not-null="true"/>
...
```

nel precedente listato relativo ad Item delle tre righe che definiscono l'attributo bids due devono essere commentate o rimosse per far funzionare la classe. La traduzione in schema relazionale che verrà presa in considerazione per l'analisi dei tre tipi di collezione è la seguente:

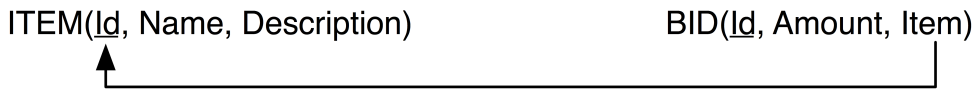

Figura 11: Schema Relazionale Item e Bid

Per tutte e tre le collezioni utilizzate tale schema relazionale è ottenibile mediante due file XML di mappatura ovvero Bid.hbm.xml e Item.hbm.xml, le differenze nei tre casi si vedono solo su Item.hbm.xml quindi nel listato 13 verrà indicato il frammento importante (quello riguardante il collegamento tra Bid e Item) di Bid.hbm.xml. Analizziamo ora le tre possibili collezioni:

1. **Set**: Se usiamo un Set nella classe Item.java bisogna lasciare delle tre righe relative all'attributo bids solo quello di tipo HashSet (che va importato all'inizio della classe Java). In questo caso il frammento significativo del file XML di Item è il seguente:

Listing 14: Frammento di Item.hbm.xml usando un Set

```
...
<set name="bids" inverse="true" cascade="save-update">
    <key column="ITEM_ID"/>
    <one-to-many class="Bid"/>
</set>
...
```

2. **Bag**: Se usiamo un Bag nella classe Item.java bisogna lasciare l'attributo bids di tipo ArrayList. Ecco il frammento XML per Item:

Listing 15: Frammento di Item.hbm.xml usando un Bag

```
...
<bag name="bids" inverse="true" cascade="save-update">
    <key column="ITEM_ID"/>
    <one-to-many class="Bid"/>
</bag>
...
```

3. **List**: Usando una collezione di tipo List bisogna lasciare bids di tipo ArrayList cancellando gli altri nella classe Item.java. Di seguito il frammento di codice per questo caso:

Listing 16: Frammento di Item.hbm.xml usando un List

```
...
<list name="bids" inverse="true" cascade="save-update">
    <key column="ITEM_ID"/>
    <list-index column="BID_POSITION"/>
    <one-to-many class="Bid"/>
</list>
...
```

3.4.2 Associazione Uno a Molti Con Partecipazioni Opzionali

Riprendiamo l'esempio del paragrafo precedente sul sito di aste online considerando la parte relativa all'informazione sul compratore di un certo oggetto. La base di dati raccoglie tutti gli utenti che interagiscono col sito e tutti gli oggetti che vi sono venduti. Chiaramente un utente può comprare più oggetti ed ogni oggetto può essere comprato da uno ed un solo utente. Questa situazione è correttamente schematizzata dallo schema Entity-Relationship di figura 12.

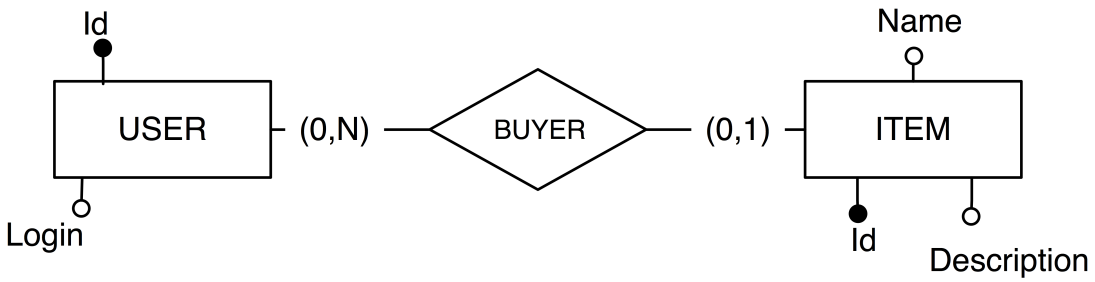

Figura 12: Schema E-R Item-User

Lo schema relazionale che è conveniente scegliere per evitare valori nulli nella base di dati è quello in figura 13.

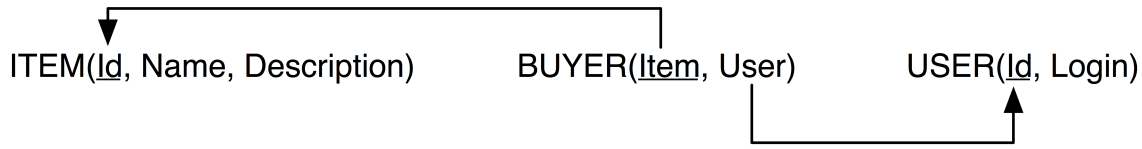

Figura 13: Schema Relazionale Item-User

Listing 17: Frammento di Item.hbm.xml

```
...
<join table="BUYERS" inverse="true">
    <key column="ITEM_ID" unique="true" not-null="true"/>
    <many-to-one name="buyer" class="User" column="USER_ID"/>
</join>
...
```

Per realizzare correttamente la situazione c'è bisogno di due classi: Item.java (analoga all'esempio precedente) e User.java(analoga al listato 4). Le uniche differenze delle due classi rispetto agli esempi precedenti sono che per Item non c'è più l'attributo bids (che rimarrebbe se analizzassimo un problema completo) mentre c'è un attributo in più chiamato buyer di tipo User il cui significato è evidente, per User invece abbiamo un attributo items che è un Set di Item. Per la corretta mappatura sono necessari due file XML ovvero Item.hbm.xml e User.hbm.xml (listati 17 e 18). Come si può notare dalla parte dal listato relativo ad Item.hbm.xml viene definita una tabella congiunta tra le due entità denominata BUYER, si definisce inoltre la chiave per tale tabella e il vincolo di integrità referenziale. Si tratta di un vincolo di tipo many-to-one ovvero l'altro lato della one-to-many. Nel modello relazionale non esiste il concetto di relazione many-to-one ma questo subentra quando si utilizzano Java ed Hibernate, in quanto Hibernate deve capire se un oggetto è riferito (o può esserlo) da più oggetti di un'altra classe.
Il listato relativo a User.hbm.xml invece, specificato che il Set deve essere mappato in una relazione esterna ovvero BUYER, deve definire un vincolo di integrità referenziale di tipo one-to-many utilizzando però il tag <many-to-many> ed impostando un vincolo unique=true. Tale operazione è assolutamente identica a creare una one-to-many ma l'implementazione di Hibernate non consente di mettere tale tag all'interno di un Set quindi per creare un vincolo uno a molti si forza il concetto di molti a molti.

3.4.3 Associazione Uno a Molti Con Attributo sull'Associazione

Analizziamo l'esempio precedente considerando però di voler tenere traccia anche del prezzo a cui è stato comprato un determinato prodotto da parte di un utente. La modifica allo

Listing 18: Frammento di User.hbm.xml

```
...
<set name="items" table="BUYERS" cascade="save-update">
    <key column="USER_ID"  not-null="true"/>
    <many-to-many class="Item" column="ITEM_ID" unique="true" />
</set>
...
```

schema E-R precedente è minima come si può vedere in figura:

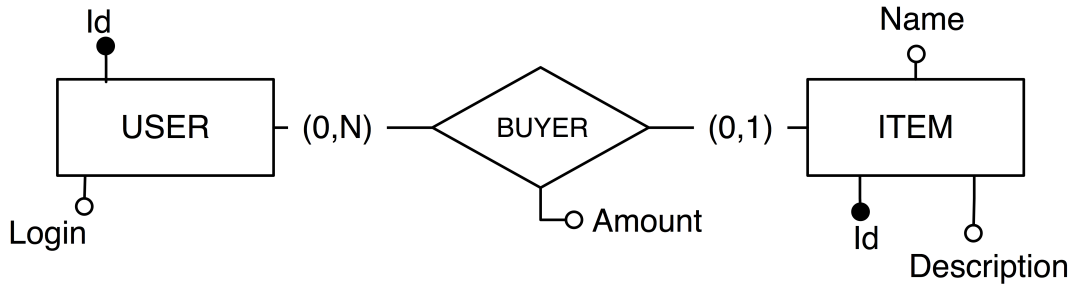

Figura 14: Schema E-R User - Item con prezzo di vendita

In questo caso lo schema relazionale tradotto è il seguente:

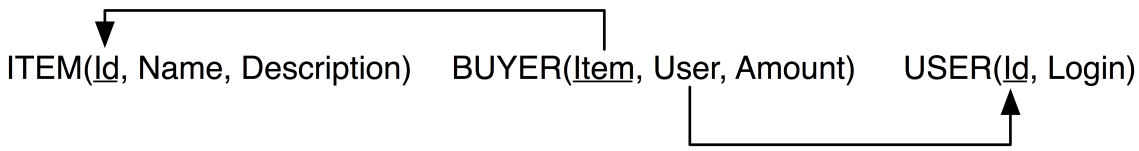

Figura 15: Schema Relazionale tradotto da fig. 3.4.3

Questa piccola modifica concettuale si rivela un'enorme modifica dal punto di vista Java/-Hibernate. Non è più possibile infatti esprimere la relazione intermedia Buyer per mezzo dei tag <join>. D'altronde ora la relazione Buyer ha un attributo valido per ogni sua istanza (Amount) quindi come è possibile mettere tale attributo nella classe User.java o Item.java se questi oggetti potrebbero anche non partecipare all'associazione (e quindi non aver bisogno di un attributo Amount)??
Tale problema si risolve aggiungendo due file: una classe Buyer.java (che evidentemente conterrà l'attributo Amount) e un file di mappatura Buyer.hbm.xml i cui frammenti sono mostrati nei listati 19 e 20. La classe relativa a Buyer presenta una nested class al suo interno che rappresenta l'identificatore di Buyer formato dagli id di Item e User coinvolti nella associazione. E' buona norma inoltre definire i metodi equals() ed hashCode() per

Listing 19: Frammento di Buyer.java

```
...
public class Buyer {
    public static class Id{
        private long itemId;
        private long userId;

        //constructors
        public boolean equals(Object o){
            if (o != null && o instanceof Id){
                Id that=(Id)o;
                return this.itemId.equals(that.itemId)
                                &&this.userId.equals(that.userId);
            }
            return false;
        }
        public int hashCode(){
            return itemId.hashCode()+userId.hashCode();
        }
    }
    private Id id=new Id();
    private User user;
    private Item item;
    private double amount;
    //constructors, setter and getter
}
```

Listing 20: Frammento di Buyer.hbm.xml

```
...
<composite-id name="id" class="Buyer$Id">
    <key-property name="itemId" access="field" column="ITEM_ID"/>
    <key-property name="userId" access="field" column="BUYER_ID"/>
</composite-id>
<property name="amount" column="AMOUNT"/>
<many-to-one name="user" column="BUYER_ID" not-null="true"
                                    insert="false" update="false"/>
<many-to-one name="item" column="ITEM_ID" not-null="true"
                                    insert="false" update="false"/>
...
```

Listing 21: Frammento di Item.hbm.xml

```
...
<join table="BUYER" inverse="true">
    <key column="ITEM_ID" unique="true"/>
    <many-to-one name="buyer" class="User" column="BUYER_ID"/>
</join>
...
```

Listing 22: Frammento di User.hbm.xml

```
...
<set name="items" table="BUYER">
    <key column="BUYER_ID"/>
    <one-to-many class="Buyer"/>
</set>
...
```

poter facilitare il lavoro di Hibernate con la gestione delle query.
Il file Buyer.hbm.xml mostra come definire chiave composte e come riferirsi a campi contenuti in nested class. Inoltre definisce i due vincoli di integrità sulle chiavi con due <one-to-many> cosa che non è concettualmente corretta come verrà discusso dopo.
I frammenti dei file di mappatura Item.hbm.xml e User.hbm.xml che completano l'esempio (le classi Item.java e User.java sono sempre le stesse) sono mostrati nei listati 21 e 22. La cosa importante da notare è l'imposizione su Item.hbm.xml attraverso il tag <join> del vincolo unique sulla chiave relativa ad Item. In questo modo si compensa lo squilibrio creato dal considerare fino a questo momento tutti vincoli come dei many-to-one. In poche parole questo esempio illustra l'unica incongruenza riscontrata tra modello relazionale e Hibernate. Infatti il modello relazionale creato in questo caso differisce da quello con cui siamo partiti per l'esempio in quanto verrà creato un modello relazionale analogo con però la relazione BUYER definita così:

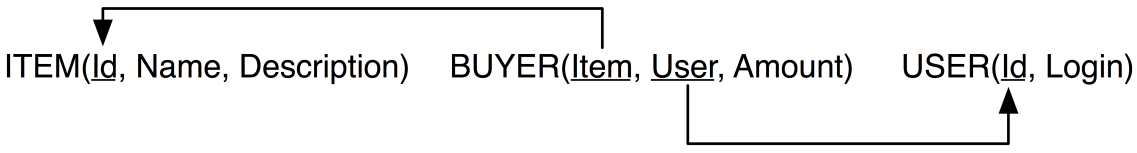

Figura 16: Schema relazionale realizzato con Hibernate

La situazione viene risolta imponendo un vincolo unique sull'attributo Item che fa parte della chiave realizzando quindi un modello relazionale che differisce semanticamente da quello di partenza ma che operativamente si comporta allo stesso identico modo.

3.5 Associazioni Binarie Molti a Molti

Per associazione molti a molti si intende una associazione che coinvolge due entità con cardinalità (0,N) o (1,N) da entrambi i lati. Per questi tipi di associazione dunque si avranno da tutti e due i lati una collezione di oggetti, come già detto prima verranno usati i Set per questi esempi ma il codice è riutilizzabile anche per List e Bag modificando solo i tag come mostrato nella sez: 3.4.1.

3.5.1 Associazioni Molti a Molti Semplici

Consideriamo una base di dati relativa ad un sito web di aste online. Tale base di dati deve raccogliere al suo interno tutti gli oggetti venduti nel sito che sono organizzati in categorie. Più precisamente una categoria può contenere uno o più oggetti e un oggetto appartiene ad una o più categorie. Ecco lo schema Entity Relationship che descrive la situazione:

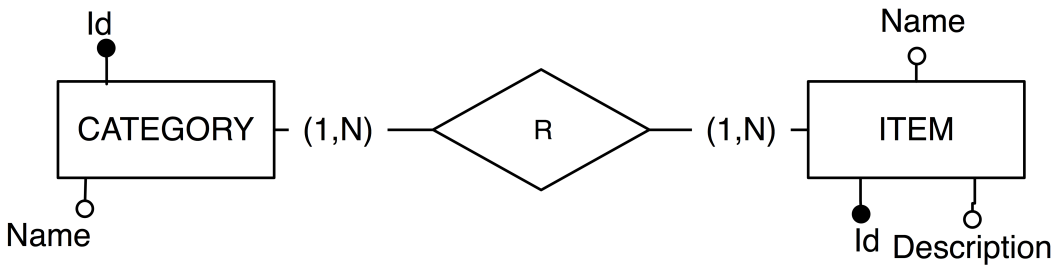

Figura 17: Schema E-R Category - Item

Lo schema relazionale tradotto è il seguente:

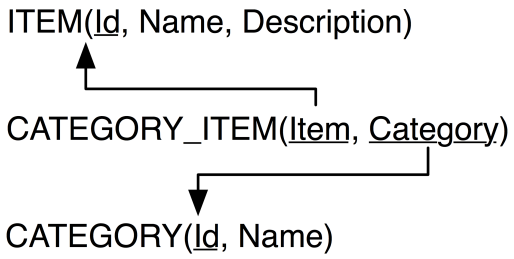

Figura 18: Schema Relazionale Category - Item

La relazione intermedia Category-Item in questo caso è ottenibile solamente tramite i file di mappatura delle due classi Category e Item. Per realizzare correttamente l'esempio con Hibernate c'è dunque bisogno di due classi Java (Category.java e Item.java) e due file di mappatura (Category.hbm.xml e Item.hbm.xml) i cui frammenti sono visibili nei listati 23, 24, 25 e 26 rispettivamente. La relazione intermedia Category_Item viene creata mappando su una tabella esterna i due set relativi alle due classi. Dichiarando su ogni file di

Listing 23: Frammento Category.java

```
...
public class Category {
    private long id;
    private String name;
    private Set<Item> items=new HashSet<Item>();
    //constructors, setter and getter
}
```

Listing 24: Frammento Item.java

```
...
public class Item {
    private String name;
    private long id;
    private String description;
    private Set<Category> categories=new HashSet<Category>();
    //constructors, setter and getter
}
```

Listing 25: Frammento Category.hbm.xml

```
...
<set name="items" table="CATEGORY_ITEM" cascade="save-update">
    <key column="CATEGORY_ID"/>
    <many-to-many class="Item" column="ITEM_ID"/>
</set>
...
```

Listing 26: Frammento Item.hbm.xml

```
...
<set name="categories" table="CATEGORY_ITEM"
                inverse="true" cascade="save-update">
    <key column="ITEM_ID"/>
    <many-to-many class="Category" column="CATEGORY_ID"/>
</set>
...
```

mappatura la chiave dell'altra entità si crea alla fine una relazione con una chiave doppia composta dai due identificatori delle classi Item e Category.

3.5.2 Associazioni Molti a Molti con Attributi Comuni

Consideriamo lo stesso esempio di prima volendo però tenere conto della data di quando è stato aggiunto un oggetto in una categoria. Tale situazione viene illustrata nel seguente E-R con lo schema relazionale ottenuto:

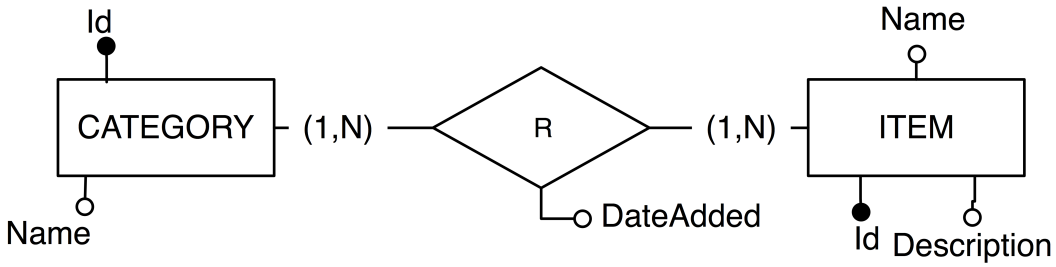

Figura 19: Schema ER Item - Category

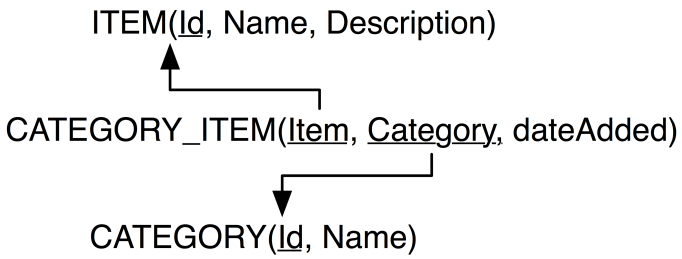

Figura 20: Schema Relazionale Item - Category

Questa piccola modifica come già visto per l'esempio trattato nella sezione 3.4.3 complica le cose dal punto di vista di Hibernate poichè c'è bisogno di un ulteriore classe Category_Item.java e di un ulteriore file di configurazione ma in questo caso lo schema relazionale proposto viene correttamente tradotto.
E' presente questa differenza rispetto a prima perchè il metodo utilizzato per mappare una associazione uno a molti con attributi comuni è nativo per le associazioni molti a molti con attributi comuni.
Di seguito i frammenti significativi dei file per realizzare correttamente la mappatura (listati 27, 28, 29 e 30) Come si può notare non ci sono differenze molto significative rispetto

Listing 27: Frammento classe CategorizedItem.java

```
...
public class CategorizedItem{
    public static class Id{
        public long categoryId;
        public long itemId;
        //constructors
        //metodi hashCode e equals
    }
    private Id id;
    private Date dateadded;
    private Item item;
    private Category category;
    //constructors, getter and setter
}
```

Listing 28: Frammento Category.hbm.xml

```
...
<set name="categorizedItems" inverse="true">
    <key column="CATEGORY_ID"/>
    <one-to-many class="CategorizedItem"/>
</set>
...
```

Listing 29: Frammento Item.hbm.xml

```
...
<set name="categorizedItems" inverse="true">
    <key column="ITEM_ID"/>
    <one-to-many class="CategorizedItem"/>
</set>
...
```

Listing 30: Frammento CategorizedItem.hbm.xml

```
...
<composite-id name="id" class="CategorizedItem$Id">
    <key-property name="categoryId" column="CATEGORY_ID"
                                                access="field"/>
    <key-property name="itemId" column="ITEM_ID" access="field"/>
</composite-id>
<property name="dateAdded" type="timestamp" not-null="true"/>
<many-to-one name="category" column="CATEGORY_ID" not-null="true"
                                    insert="false" update="false" />
<many-to-one name="item" column="ITEM_ID" not-null="true"
                                    insert="false" update="false"/>
...
```

all'associazione uno a molti con tabella intermedia appunto perchè il metodo usato per quell'esempio è nativo per questo.

3.6 Associazioni Ternarie

Per associazione ternaria si intende un'associazione che coinvolge tre entità con cardinalità (1,N) o (0,N) da tutti e tre i lati dell'associazione. Non verranno trattati i casi diversi in quanto è conveniente tradurre tali casi con associazioni binarie.
Consideriamo l'ultimo esempio che coinvolgeva oggetti e categorie e supponiamo di voler aggiungere l'informazione riguardante gli utenti che hanno aggiunto certi item nelle categorie afferenti.
Siamo nel caso di un'associazione ternaria schematizzata dallo schema Entity-Relationship di figura 21 mentre in figura 22 è mostrato lo schema relazionale che verrà implementato con Hibernate.

Questo caso è gestibile in modo simile al caso di molti a molti con tabella intermedia. Per poter quindi mappare correttamente la situazione sono necessari otto file totali ovvero:

- Quattro classi Java (Item.java, Category.java, User.java e CategorizedItem.java) che non verranno mostrate in quanto simili al caso precedente (ogni classe ha un Set di elementi chiamati categorizedItems di tipo Set<CategorizedItem> e CategorizedItem avrà una nested class in cui si costruisce l'identificatore con l'insieme dei tre id della altre tre classi).

- Quattro file XML di mappatura i cui frammenti significativi sono indicati nei listati 31, 32, 33 e 34

Listing 31: Frammento di User.hbm.xml

```
...
<set name="categorizedItems" inverse="true">
    <key column="ADDED_BY"/>
    <one-to-many class="CategorizedItem"/>
</set>
...
```

Listing 32: Frammento di Category.hbm.xml

```
...
<set name="categorizedItems" inverse="true">
    <key column="CATEGORY_ID"/>
    <one-to-many class="CategorizedItem"/>
</set>
...
```

Listing 33: Frammento di Item.hbm.xml

```
...
<set name="categorizedItems" inverse="true">
    <key column="ITEM_ID"/>
    <one-to-many class=ŰCategorizedItem"/>
</set>
...
```

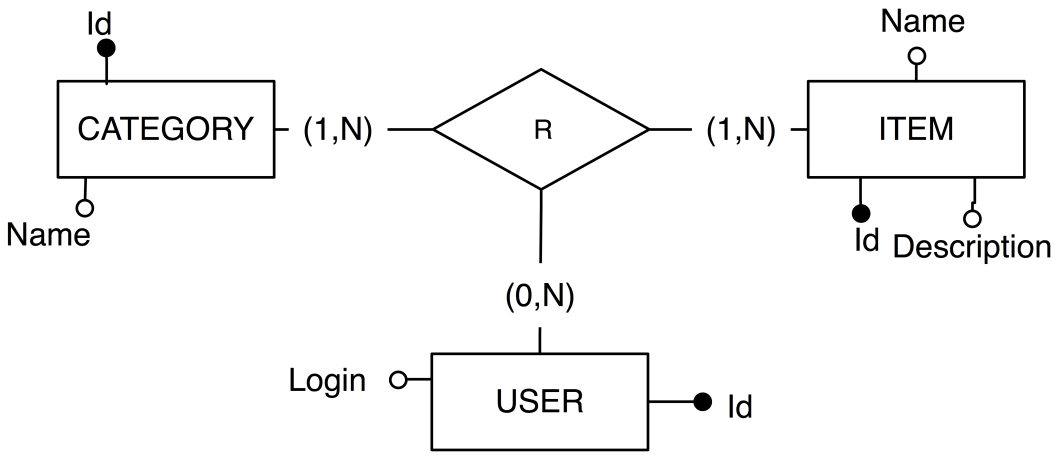

Figura 21: Schema Entity Relationship Item - Category - User

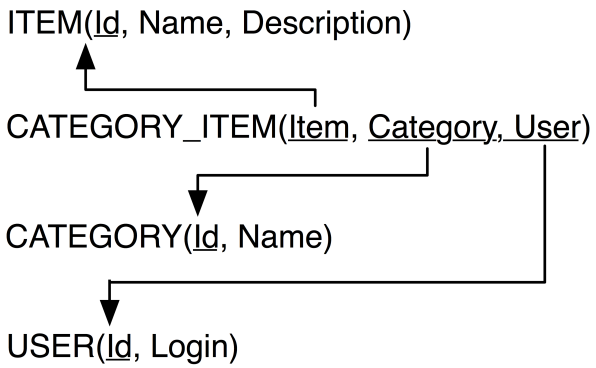

Figura 22: Schema Relazionale Item - Category

Da notare nei file di mappatura la presenza di update=false ed insert=false nel file di mappatura CategorizedItem.hbm.xml nella definizione dei vincoli di integrità referenziale. Tali comandi sono imposti da Hibernate in quanto la relazione Categorized_Item si intende sia una relazione le cui tuple vengono create alla creazione di un'associazione tra un utente e una coppia (categoria, oggetto) e quindi si presuppone che l'associazione sia non modificabile.
Infatti la riga dichiarativa del file CategorizedItem2.hbm.xml contiene mutable=false che sta a significare proprio quello appena descritto.
Nel caso di attributi comuni sulla associazione la soluzione varia pochissimo basta aggiungere tale attributo sulla classe CategorizedItem.java con il suo metodo *getter* ed il suo metodo *setter* e aggiungere la riga descrittiva dell'attributo sul file CategorizedItem.hbm.xml.

Listing 34: Frammento di CategorizedItem.hbm.xml

```
...
<class name="CategorizedItem" table="CATEGORY_ITEM"
        mutable="false">
    <composite-id name="id" class="CategorizedItem$Id">
        <key-property name="categoryId" column="CATEGORY_ID"/>
        <key-property name="itemId" column="ITEM_ID"/>
        <key-property name="userId" column="ADDED_BY"/>
    </composite-id>
    <many-to-one name="category" column="CATEGORY_ID"
                 not-null="true" insert="false" update="false"/>
    <many-to-one name="item" column="ITEM_ID"
                 not-null="true" insert="false" update="false"/>
    <many-to-one name="user" column="ADDED_BY"
                 not-null="true" insert="false" update="false"/>
</class>
...
```

4 Conclusioni

Dopo aver implementato e testato tutti gli esempi di questa tesi, Hibernate è risultato un ottimo tool per il colloquio tra applicazioni Java e Database che però presenta qualche svantaggio.
Uno dei suo obiettivi primari, che è risultato essere quello più evidente tra quelli proposti dagli sviluppatori, è la notevole riduzione delle righe di codice necessarie per colloquiare con uno o più database e soprattutto l'elevato tasso di riutilizzo del codice.
Quest'ultimo punto infatti risulta essere, a mio parere, determinante per chi vuole affacciarsi all'uso di Hibernate, ed è evidente soprattutto per chi deve scrivere molte applicazioni diverse ognuna delle quali agganciata ad un database. Il tempo che si risparmia riutilizzando tutto il codice ripetitivo tramite dei semplici *drag and drop* può essere utilizzato per affinare l'applicazione e per concentrarsi sui business problems ovvero sui problemi relativi all'applicazione e non sul colloquio di essa col database.
Un altro vantaggio che ho potuto confermare in questa tesi è come Hibernate mantenga estremamente leggibili le classi che devono essere rese persistenti e come mantenga tutto il progetto perfettamente scalabile e adattabile alle modifiche inevitabili che possono capitare nel ciclo di vita di un'applicazione.
Per quanto riguarda il miglior controllo delle performance, ciò si ha solamente se viene utilizzato c3p0 o un Database Connection Pool alternativo e ciò non è necessario affinchè Hibernate funzioni correttamente. Tale obiettivo quindi non è assimibilabile ad Hibernate, può comunque essere l'ago della bilancia per molti applicativi che colloquiano con database da milioni di tuple concorrentemente tra molti utenti.
Per quanto riguarda gli aspetti negativi di Hibernate si è notata una certa lentezza nell'avvio e nel mantenere il colloquio con i database. In generale confrontando la soluzione del colloquio attraverso connessioni JDBC all'interno del codice e con Hibernate, la prima soluzione risulta essere più performante dal punto di vista della velocità d'esecuzione. Ciò è anche spiegabile dal fatto che Hibernate è un insieme di classi ed interfacce che si frappone tra codice Java e Database che al suo interno usa connessioni JDBC, quindi quando si aggiunge uno strato aggiuntivo ad un qualcosa che funziona ad una certa velocità è logico aspettarsi che quello che si ottiene diminuisca la velocità di esecuzione.
Ultimo aspetto (parzialmente) negativo è la compatibilità del modello relazionale con Hibernate. Come accennato nell'introduzione l'obiettivo di questa tesi è quello di fornire una documentazione di come mappare con Hibernate le classi per implementare un certo schema relazionale. La maggior parte degli schemi relazionali sono facilmente mappabili come visto negli esempi precedenti. Se si vuole implementare però uno schema relazionale a tre relazioni ottenuto mediante traduzione dallo schema E-R di un'associazione binaria uno a molti e con attributi sull'associazione si riscontrano problemi. Come già descritto infatti non esiste un modo per poter creare una chiave unica nella relazione congiunta ma bisogna passare attraverso una chiave composta (tipica delle molti a molti) ed impostare un vincolo unique su una delle due. Questa controversia è una piccola differenza dal punto

di vista semantico che si traduce solo in un vincolo in più sul database mentre non ci sono differenze dal punto di vista operativo.
In conclusione Hibernate è un ottimo sistema per risparmiare tempo se si devono realizzare frequentemente applicativi che hanno bisogno di persistenza. Esso introduce comunque una latenza nel funzionamento dell'applicazione ma è un piccolo prezzo in confronto ai benefici derivanti dal tempo risparmiato, dalla chiarezza del codice e dall'adattabilità a sistemi differenti e alle modifiche.

5 Bibliografia

Riferimenti bibliografici

[1] Christian Bauer, Gavin King, Java Persistence With Hibernate ed. Manning, 2007;

[2] Atzeni, Ceri, Paraboschi, Torlone Basi di Dati Modelli e Linguaggi di Programmazione, ed. McGraw-Hill;

[3] Elmasri, Navathe Sistemi di Basi di Dati ed. Pearson, 2011;

[4] Cay Horstmann Concetti di informatica e fondamenti di Java ed. Apogeo;

[5] Wikipedia: `http://www.wikipedia.com`;

[6] Java: `http://www.java.com`;

[7] Hibernate: `http://www.Hibernate.org`;

[8] Eclipse: `http://www.eclipse.org`;

[9] OmniGraffle: `http://www.omnigraffle.com`;