Università degli Studi di Torino
Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea in Informatica percorso STISI
ANNO ACCADEMICO 2008/2009

Tesi di laurea triennale

ModelMapper

Un framework Java per la persistenza basato sulla filosofia Convention Over Configuration

Candidato: **Querella Luca e Usbergo Alex**
Relatore: **Bono Viviana**
Co-Relatori: **Pasteris Paolo**

Indice

Capitolo 1

Introduzione

1.1 Obiettivi

L'obiettivo di questa tesi è lo sviluppo di un framework ORM Java.

Per framework ORM si intende uno strumento che permetta l'integrazione tra due tipi di dati incompatibili tra di loro: gli oggetti della programmazione object-oriented e le entità del modello relazionale.

Nello specifico il nostro obiettivo è fornire al programmatore un servizio di persistenza degli oggetti astraendo le caratteristiche e il comportamento del database dove vengono memorizzati i dati.

Questo servizio vuole da un lato essere accessibile allo sviluppatore che non ha specifiche conoscenze nel campo dei database e ha l'esigenza di salvare il contenuto informativo di alcuni oggetti, dall'altro deve permettere al programmatore più esperto, una drastica diminuzione delle righe di codice che dovrebbero essere destinate alle routine per il recupero e la manipolazione dei dati presenti sul database, di conseguenza riducendo il numero di errori, il tempo debugging e di mantenimento del progetto.

Oltre alla semplicità e alla velocità di sviluppo, alla sicurezza e alla mantenibilità del progetto, un ulteriore vantaggio che questo strumento introduce è la portabilità: è possibile interfacciarsi a DBMS diversi senza la necessità di modificare il codice sorgente cosa che altrimenti sarebbe impossibile.

1.2 ModelMapper

Chiarite le finalità di un generico ORM passiamo ora alla trattazione di *ModelMapper*, è questo infatti il nome scelto per il progetto. Il motivo di questa scelta va ricercata nell'etimologia del verbo inglese *"to map"* e nel significato della parola *"Model"* in ingegneria del software.

"To map" da un punto di vista matematico significa infatti comportarsi come una funzione, creare un'immagine di un dominio. Tornando alle nostre finalità vogliamo creare un'immagine di un nostro Oggetto (o più propriamente, come vedremo Modello) su uno strumento di persistenza: rappresentare le informazioni contenute in questo in una logica relazionale.

In un'architettura multi-tier ("a strati") i modelli compongono il modulo di gestione del contenuto informativo, non vogliamo infatti rendere persistenti tutti gli oggetti, molti di questi sono dedicati alla logica applicativa del software e non alla rappresentazione dell'informazione.

Non da meno è l'influenza che ha avuto nello sviluppo il pattern architetturale *DataMapper* descritto da Martin Fowler che analizzeremo in seguito.

Le linee guida applicate nello sviluppo sono state fondamentalmente 3:

- Convention over Configuration: abbiamo utilizzato questo paradigma per il design di software (in seguito approfondito) per ridurre lo sforzo che il programmatore deve compiere per imparare ad usare il framework e in generale per ridurre il tempo di sviluppo.

- DRY "don't repeat your self": filosofia secondo la quale non vi deve essere ridondanza di informazione (approfondito meglio in seguito). Abbiamo applicato questa filosofia quanto più possibile nel rispetto dei vincoli imposti dal linguaggio.

- Omogeneità tra la semantica Object Oriented e la sintassi di definizione dei modelli richiesta al programmatore. Vengono riportati i concetti di ereditarietà, aggregazione e di composizione tipici del paradigma OO.

1.2.1 Convention Over Configuration

Convention Over Configuration (o **Coding by Convention**) è un paradigma di design che prevede configurazione minima (o addirittura assente) per il programmatore che utilizza un framework che lo rispetti, obbligandolo a configurare solo

gli aspetti che si differenziano dalle implementazioni standard o che non rispettano particolari convenzioni di denominazione o simili.

Questo tipo di approccio semplifica notevolmente la programmazione soprattutto agli stadi iniziali dello studio di un nuovo framework, senza necessariamente perdere flessibilità o possibilità di discostarsi dai modelli standard.

Framework mal ingegnerizzati molto spesso necessitano di molteplici file di configurazione, ognuno con diversi parametri. Questi parametri forniscono informazioni specifiche per ogni progetto, cha vanno da `URI` a mapping tra classi e tabelle del database. Un grande numero di file di configurazione è spesso un indicatore di complessità non necessaria nel design dell'applicazione.

Per esempio, nelle prime versioni dell'ORM Java **Hibernate** i mapping tra classi e tabelle erano descritti in file XML. Molte di queste informazioni potevano essere ottenute da un mapping convenzionale, ovvero asserendo che i nomi delle tabelle coincidevano con quelli delle classi, e quelli delle colonne con quelli dei campi.

Molti frameworks moderni usano un approccio *C*oC, come per esempio Spring, Ruby on Rails, Kohanna PHP, Grails, CakePHP e Maven.

Va sottolineato che questa filosofia non è nuova. Infatti nelle radici della libreria Java si possono notare massicci utilizzi di `Coding by Convention`; per esempio citando le secifiche `JavaBean`:

"As a general rule we don't want to invent an enormous java.beans.everything class that people have to inherit from. Instead we'd like the JavaBeans runtimes to provide default behaviour for normal objects, but to allow objects to override a given piece of default behaviour by inheriting from some specific java.beans.something interface."

1.2.2 DRY - Don't Repeat Yoursel

Don't Repeat Yourself (DRY, anche conosciuto come "Single Point of Truth") è un principio secondo il quale l'informazione non debba essere ripetuta e ridondante e non si debba esprimere lo stesso concetto più di una volta, specie se in forma diversa.

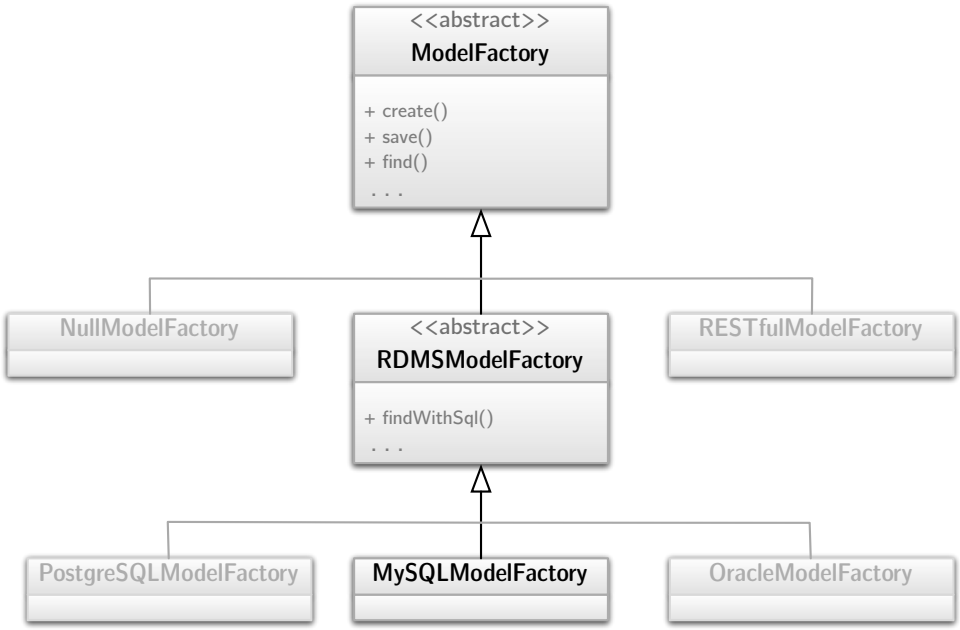
Di particolare importanza, nell'informatica, diventa un Design pattern della programmazione secondo il quale bisogna evitare il più possibile la duplicazione del codice, poiché questa complica la mantenibilità e la leggibilità del codice stesso.

Un codice DRY riduce al minimo le informazioni ridondanti e le duplicazioni, risulta molto più pulito, mantenibile e leggibile e ricorre, dove possibile, all'utilizzo di funzioni per accorpare in un unico punto funzionalità usate più volte.

1.3 Non solo RDBMS

Sebbene ORM sia acronimo di *Object-Relational Mapping* e quindi si riferisca esplicitamente al modello relazione abbiamo voluto creare uno strumento che si adatti anche a differenti strumenti di persistenza. E' possibile estendere facilmente il framework sviluppando un adapter per generici datasource. Esempi di datasource possono essere DB Object-Relational, RESTful, CouchDB, file XML o YAML, o qualsivoglia strumento che permetta di salvare informazioni con una specifica codifica.

Nello specifico abbiamo deciso di concentrarci nello sviluppo di un solo adapter, quello per il database MySQL recentemente acquistato da Sun Microsystem. L'implementazione di metodi specifici relativi a questo database sono presenti nella classe `MySQLModelFactory`, che eredita la classe astratta RDBMSModelFactory , figlia a sua volta di ModelFactory come rappresentato nel diagramma sottostante.



Il metodo `findWithSql()` infatti è definito nella classe `RDBMSModelFactory` perché è proprio dei modelli relazionali.

Un esempio di adapter rappresentato nello schema è il NullModelFactory: paradossalmente questo non fornisce nessuna persistenza, salva le informazioni in strutture dati allocate nella memoria temporanea, questo strumento permette allo svilup-

patore che non ha ancora deciso l'adapter di provare il proprio progetto e rimandare questa decisione.

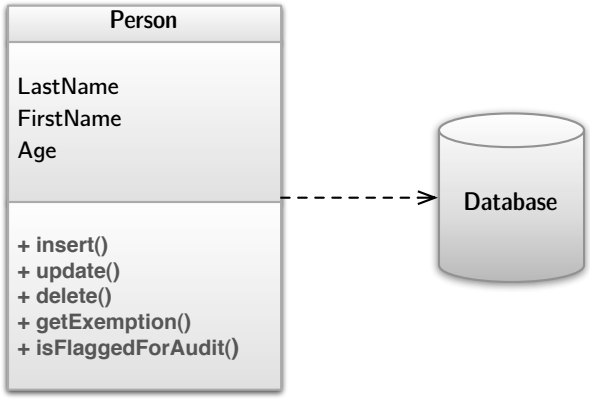
Come è facile notare tutti i nomi degli adapter terminano con il suffisso "factory", questo perché è l'adapter che si occupa di istanziare i modelli, sia nel caso si stiano cercando istanze già presente del datasource sia nel caso se ne stiano creando di nuove. I design patterns utilizzati verranno presentati in seguito.

1.4 Pattern architetturali

Lo sviluppo del progetto è stato fortemente influenzato sia da framework di persistenza per altri linguaggi di programmazione (primo fra tutti *R*ails), sia da diversi design pattern architettturali. In particolare due pattern descritti da *M*artin Fowler: **Active Record** (da cui prende il nome il componente del model-tier di Rails), **DataMapper**.

1.4.1 Active Record Pattern

Un oggetto che avvolge un record presente in una tabella o in una vista del database, incapsula l'accesso al database, e aggiunge logica di dominio sui dati.



Nella scelta di una strategia di persistenza per gli oggetti, la prima decisione riguarda dove collocare la responsabilità di gestire il *mapping*. Ci sono due principali alternative: (1) la classe stessa rappresentante l'entità può essere responsabile del mapping, oppure (2) una classe separata può effettuare il mapping al database.

La prima scelta corrisponde ad Active Record: i metodi di persistenza dell'oggetto vengono inseriti direttamente nell'*entity-object*, e quest'ultimo incapsula una singola riga di una tabella del database.

Segue quindi che l'essenza di Active Record è un *Domain Model* nel quale le classi coincidono con la struttura del database sottostante; ogni oggetto è responsabile delle operazioni `CRUD` (`create`, `read`, `update` e `delete`) e della domain logic che opera sui dati. La struttura di questi ultimi deve coincidere esattamente con quella di un record del database: un campo nella classe per ogni colonna nella tabella.

Una classe Active Record solitamente ha metodi che svolgono le seguenti operazioni:

- Costruire un'istanza `ActiveRecord` da un result set SQL.
- Costruire un'instanza predisposta per un successivo inserimento nel database.
- Fornire *getter* e *setter* per ogni campo.
- Implementare porzioni di business logic.

La classe fornisce inoltre metodi *finder* statici per incapsulare query SQL comunemente usate e ritornare una collezione di `ActiveRecord`.

È da sottolineare che i metodi *get* e *set* possono compiere operazioni intelligenti come convertire tipi SQL in tipi nativi del linguaggio, oppure ritornare l'istanza di un entità in relazione (facendo un *lookup* nella tabella).

Per una maggiore comprensione del pattern, viene presentato un esempio di implementazione senza utilizzo di framework per un semplice modello.

Esempio: A Simple Person

Si deve innanzitutto definire la classe `Person`:

```
class Person {
        private int id;
        private String firstName;
        private String lastName;
        private int numberOfDependents;
        ...
}
```

Lo schema del database è costruito con la stessa struttura.

```
CREATE TABLE People( id int PRIMARY KEY, firstname varchar,
lastname varchar, number_of_dependents int )
```

Per caricare un oggetto, la classe `Person` deve fornire metodi statici per la ricerca, quindi:

```
private static String findStatement = "SELECT␣*␣FROM␣People␣WHERE␣id␣=␣?";

public static Person find(int id) {
        PreparedStatement stm;
        ResultSet rs;
        try {
                ...(obtain the statement from the JDBC connection)
                rs.findStatement.executeQuery();
                rs.next();
                return load(rs);
        } catch (SQLException e) {...}
}

public static load(ResultSet rs) throws SQLException {
        int id = rs.getInt(1);
        String firstName = rs.getString(2);
        ...
        result = new Person(id, firstName, lastName, numDependents);
        return result;
}
```

Mentre per eseguire il salvataggio dei dati si implementa il metodo d'istanza *update*

```
private final static String updateStatement =
"UPDATE␣People␣set␣lastname␣=␣?,␣firstname␣=␣?" +
"number_of_dependents␣=␣?␣WHERE␣id␣=␣?";

public void update() {
        PreparedStatement stm;
        try {
                ...(obtain the statement from the JDBC connection)
                stm.setString(1, lastName);
                stm.setString(2, firstName);
                ...
```

```
                stm.execute();
        } catch (SQLException e) {...}
}
```

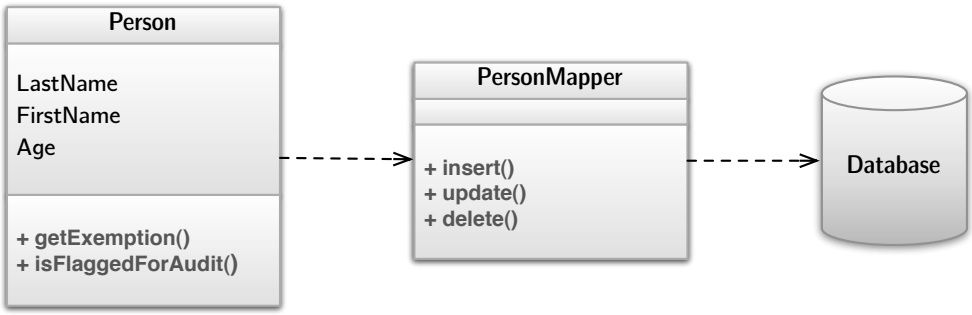
Nella stessa maniera si implementano le altre operazioni `CRUD`, come l'inserimento e la cancellazione. Come si può vedere, anche in un esempio così semplice la mole di lavoro per implementare il pattern Active Record, e quindi per gestire la persistenza è notevole.

Implementazioni di Active Record

L'implementazione di Active Record più popolare è realizzata in Ruby, ed è presente nel framework Rails. Questa usa pesantemente la metaprogrammazione per creare runtime campi e metodi di accesso e ricerca che corrispondano a tabelle e attributi dababase.

Come verrà esposto in seguito, la chiave per la realizzazione di ModelMapper è una strategia simile, realizzata grazie all' *instance-proxying* e la *reflection* offerta da Java.

1.4.2 Data Mapper Pattern



Un pattern architetturale come Active Record rende l'utilizzo degli entity-object estremamente facilitato, ma molto spesso la struttura delle classi differisce da quella delle tabelle del database.

Infatti non è sempre immediato trovare un *fit* tra le classi che si vogliono descrivere (per i requisiti della business logic) e lo schema del database. In questi casi viene in aiuto il pattern Data Mapper: questo non fa altro che aggiungere una indirezione (quindi un nuovo livello) al model-tier, costituito da classi *mapper* del datasource.

Questo comporta quindi una certa indipendenza tra datasource e domain logic; si può scrivere l'intera logica del modello senza pensare a come e dove i dati saranno resi persistenti.

Questo pattern può comunque risultare più complesso in molti use-case, ovvero tutti quelli che si possono esprimere naturalmente con Active Record.

Segue nuovamente un caso di implementazione del pattern.

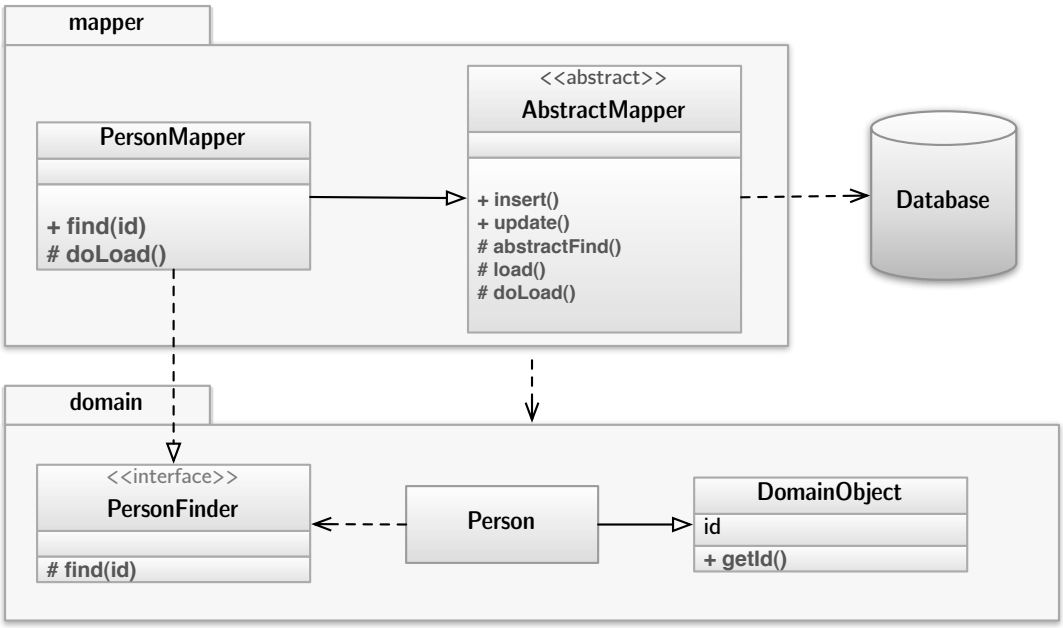
Esempio: A Simple Person Mapper

La struttura dell'esempio si rifà al modello `Person` definito prima, con schema del database isomorfo. Viene prima introdotta una strategia presente in questo esempio e largamente utilizzata all'interno dei framework di persistenza per gestire la consistenza tra istanze multiple di oggetti in memoria, ossia l'*Indentity Map*.

L' Identity Map è un design pattern per l'accesso al database usato per incrementare le prestazioni e garantire consistenza, fornendo una cache in memoria per prevenire il reperimento di uno stesso oggetto dal database.

Se i dati richiesti sono stati precedentemente caricati dal database, la Identity Map ritorna l'istanza dell'oggetto già creato, altrimenti esegue il fetch dei dati dal database e inserisce l'oggetto creato nell'Identity Map.

Il nostro esempio è descritto dal seguente diagramma delle classi:



La classe `PersonMapper` implementa implementa sia il `Finder` che la `Identity Map`, mentre `AbstractMapper` implementa comportamenti comuni a tutte le classi mapper.

Ipotizzando di voler ricercare le persone per cognome, il codice necessario sarà:

```
class PersonMapper...

private satic String findLastNameStatement =
"SELECT␣*␣FROM␣People␣WHERE␣lastname␣LIKE␣?";

public List findByLastName(String name) {
        PerparedStatement stm;
        ResultSet rs;

        try {
                ...(obtain the statement from the JDBC connection)
                stm.setString(1, name);
                rs = stm.executeQuery();
                return loadAll(rs);
        } catch (SQLException e) {...}
}
```

```
class AbstractMapper...

protected List loadAll(ResultSet rs) throws SQLException {
        List result = new ArrayList();
        while (rs.next()) result.add(load(rs));
        return result;
}
```

In questo modo però alcune righe del `ResultSet` potrebbero corrispondere ad istanze già caricate; per evitare duplicati è necessario controllare nella `Identity Map` la presenza di ogni `id` caricato.

```
class AbstractMapper...

protected Map identityMap = new HashMap();
protected DomainObject load(ResultSet rs) {
        DomainObject result = identityMap.get(rs.getInt("Id"));
        if (result != null) return result;
        else return doLoad(rs);
}
protected abstract DomainObject doLoad(ResultSet rs) throws SQLException;

class PersonMapper...

protected DomainObject doLoad(ResultSet rs) throws SQLException {
        int id = rs.getInt(1);
        String firstName = rs.getString(2);
        ...
        result = new Person(id, firstName, lastName, numDependents);
        return result;
}
```

Capitolo 2

Scelte progettuali

In una prima fase di progettazione del framework, eravamo orientati allo sviluppo di un preprocessore, il cui compito fosse modificare alcune classi Java, in cui erano stati inseriti dallo sviluppatore elementi sintattici destinati alla definizione dei modelli stessi, inserendo codice Java in grado di effettuare il mapping con il datasource. In una seconda fase, questa idea è stata abbandonata: i modelli vengono definiti dallo sviluppatore mediante interfacce Java con opportune *annotation*. Le interfacce devono rispettare alcune convenzioni; insieme al framework viene fornito uno strumento "Validator" che si occupa di controllare che queste convenzioni vengano rispettate, senza però modificarne in alcun modo il codice.

Di seguito prenderemo prima in esame i preprocessori e le annotations, infine ci dedicheremo alla spiegazione dei motivi che ci hanno portato a scegliere quest'ultime.

2.1 Preprocessori

Con preprocessore si intende un programma in grado di effettuare sostituzioni testuali sul codice sorgente. Spesso il preprocessing precede la compilazione, l'output del preprocessore viene infatti passato al compilatore in grado di produrre codice macchina. La differenza sostanziale tra il preprocessore e il compilatore è che il primo effettua soltanto modifiche testuali il secondo invece si occupa della traduzione del codice. Tipi comuni di sostituzione testuale sono l'espansione di macro, l'inclusione di file e la selezione condizionale.

Vediamo un esempio per chiarire meglio il concetto, di seguito riportiamo un frammento di codice C

```
#include <stdio.h>
```

```
#define HW "hello world"

int main() {
        #ifdef HW
        printf(HW);
        #endif
}
```

in questo codice sono presenti alcune direttive precedute dal carattere #. Queste direttive vengono analizzate e sostitute dal preprocessore prima della compilazione. E possibile vedere il risultato del preprocessing (l'input del compilatore) mediante il comando `gcc -E file.c`. Eccone riportato un frammento di seguito:

```
[..]
int printf(const char * , ...) ;
int putc(int, FILE *);
int putchar(int);
[..]

int main() {
 printf("hello world");
}
```

come è possibile notare è stata effettuata l'inclusione del file `stdio.h` (non interamente riportato), la sostituzione della costante `HW` e la scelta condizionale.

Procedendo in maniera analoga, il nostro preprocessore avrebbe sostituito opportuni elementi sintattici (non appartenenti alla sintassi Java) con frammenti di codice in grado di gestire l'interazione con lo strumento di persistenza.

2.2 Java Annotations

Le annotazioni in Java sono etichette che si inseriscono nel codice sorgente per marcare determinati elementi e permettere la loro elaborazione attraverso appositi strumenti. Queste etichette possono essere processate a livello di codice sorgente, oppure il compilatore li può includere nei file delle classi.

Le annotazioni possono avere molti utilizzi *(come per esempio la generazione automatica di codice per il test, il logging e la semantica delle transazioni o di file ausiliari, come i descrittori di distribuzione)* e non modificano mai il modo con il quale vengono compilati i programmi.

2.2.1 Metadati nei programmi

I metadati sono dati che descrivono dati. In questo contesto sono dati che descrivono il codice. Un esempi tipico di metadati sono i commenti *Javadoc*.

Questi commenti descrivono il codice ma non ne modificano il significato. A partire da `JDK 5.0`, si possono utilizzare le annotazioni per inserire dati arbitrari nel proprio codice sorgente. Il motivo per compiere questa operazione è legato al fatto che i metadati sono utili solo in combinazione con gli strumenti.

Di seguito è riportato un esempio di una semplice annotazione:

```
public class Foo {
        ...
        @Test public void doBar()
}
```

L'annotazione `@Test` riguarderà il metodo `doBar`. In Java, un annotazione viene utilizzata come un *modifier* (una parola chiave come `public` o `static`) e si colloca prima della voce annotata, senza un punto e virgola. Il nome di ogni annotazione è preceduto dal simbolo `@`.

L'annotazione `@Test` non esegue alcuna operazione per conto proprio e per essere utile ha bisogno di essere elaborata. Per esempio, per verificare una classe, uno strumento di test potrebbe chiamare tutti i metodi che hanno un etichetta `@Test`. Un altro strumento potrebbe rimuovere tutti i metodi di test da un file di classi, in modo che non vengano distribuiti con il programma alla fine dei test.

Le annotazioni possono essere definite in modo che abbiano *elementi*, per esempio: `@Test(id=1F6)` .

Oltre ai metodi, è possibile annotare le classi, campi e variabili locali, in quanto un'annotazione può essere ovunque si possa inserire un modificatore.

Ogni annotazione deve essere definita da un'*interfaccia di annotazione*. I metodi dell'interfaccia corrispondono agli elementi dell'annotazione.

```
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)

public @interface Test { String id() default = "[n/a]"; }
```

La dichiarazione `@interface` crea un interfaccia Java vera e propria. Gli strumenti che elaborano le annotazioni ricevono oggetti che implementano l'interfaccia

di annotazione. Uno strumento può chiamare il metodo `id` per ricavare l'elemento id di una particolare annotazione `@Test`.

Le annotazioni `Target` e `Retention` sono *meta annotazioni*, che annotano `Test`, contrassegnandola come un annotazione che può essere applicata solo a metodi e che si ricava quando il file delle classi viene caricato nella macchina virtuale.

2.2.2 Sintassi delle annotazioni

Un annotazione è definita come un interfaccia di annotazione:

```
modifiers @interface AnnotationName {
        element declaration,
        element declaration,
        ...
}
```

La dichiarazione di ogni elemento assume la forma:

```
type elementName(); | type elementName() default value;
```

Per esempio l'annotazione indicata di seguito ha due elementi:

```
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)

public @interface BugReport {
        String assignedTo() default "[none]";
        int severity() = 0;
}
```

E può essere utilizzata in questa maniera (l'ordine degli elementi non ha importanza): `@BugReport(assignedTo=Foo, severity=9)`

Se non viene specificato il valore di un elemento, si utilizza il valore predefinito della dichiarazione.

Un annotazione priva di elementi prende il nome di *annotazione di marcatura*. Se un elemento ha il nome speciale `value` e non viene specificato nessun altro elemento, si può tralasciare il nome dell'elemento e il simbolo `=` .

Tutte le interfacce di annotazione estendono implicitamente l'interfaccia `Annotation`. Questa è un interfaccia regolare, non un'interfaccia di annotazione.

Non è possibile estendere le interfacce di annotazione, in altre parole, tutte le interfacce di annotazione estendono direttamente `Annotation`.

La meta annotazione `@Retention` specifica per quanto si deve mantenere un'annotazione; si specifica almeno uno dei seguenti valori. (Il valore predefinito è `RetentionPolicy.CLASS`).

Policy di contenimento	Descrizione
`SOURCE`	Incluse nel class file.
`CLASS`	Incluse nel class file, ma la macchina virtuale non le deve caricare.
`RUNTIME`	Incluse nel class file e vengono caricate dalla macchina virtuale.

2.2.3 Meta-annotations

Le *meta-annotations* sono interfacce di annotazione con `@Target` definito come `ANNOTATION_TYPE`, e specificano proprietà che si applicano alle annotazioni stesse.

La meta annotazione `@Documented` offre un'indicazione a strumenti di documentazione, per esempio *Javadoc*. Le annotazioni documentate devono essere considerate in modo analogo ad altri modificatori, per esempio `protected` o `static`, per ciò che riguarda i propositi della documentazione stessa.

La meta annotazione `@Inherited` si applica solo ad annotazioni che riguardano le classi. Quando una classe ha un'annotazione ereditata, automaticamente tutte le sue sottoclassi hanno la stessa annotazione. Ciò facilita la creazione di annotazioni che funzionano in modo analogo alle interfacce di marcatura, per esempio `Serializable`.

In effetti, un'annotazione `@Serializable` risulta essere più appropriata rispetto ad un interfaccia di marcatura `Serializable` senza metodi. Una classe è serializzabile perché esiste un supporto a runtime per la lettura/scrittura dei suoi campi, non a causa di un qualsiasi principio teorico di progetto object oriented, e un'annotazione descrive questo fatto molto meglio di quanto possa fare l'ereditarietà delle interfacce.

```
@Inherited @Serializable { }
@Serializable class Person { ... }
class Student extends Employee { ... } //anche questa classe è @Serializable
```

2.3 Preprocessore vs Annotations

Innanzitutto è necessaria una precisazione: i due strumenti non sono equivalenti o intercambiabili. Le annotation da sole non permettono di effettuare alcuna sostituzione testuale, permettono esclusivamente l'introduzione di etichette. Sono necessari

ulteriori strumenti (come ad esempio proprio dei preprocessori) per dare uno scopo a queste etichette.

Nello specifico gli strumenti che abbiamo deciso di utilizzare sono le API Java per la *reflection* e il meccanismo del *proxying* che approfondiremo entrambi in seguito, ma quello che è utile sapere ora, è che a differenza dei preprocessori, questi agiscono runtime ovvero quando il codice è in esecuzione e non coinvolgono in alcun modo il processo di compilazione.

Veniamo ora al motivo della nostra scelta: abbiamo voluto prediligere le annotations per i seguenti motivi.

- Semplicità: allo sviluppatore viene richiesto di definire le interfacce dei propri modelli seguendo alcune convenzioni e quando necessario anteponendo alla dichiarazione di taluni metodi delle annotation, riteniamo che seguire delle regole per strutturare un'interfaccia Java, sia più semplice che imparare una nuova sintassi differente da quella di Java, con cui lo sviluppatore ha certamente già confidenza.

- Comodità: non è necessario installare alcun tool, l'unico requisito tecnico per l'uso del framework è quello di aver inserito nel proprio `ClassPath` l'archivio jar contenenti le classi di **ModelMapper**.

- Compatibilità: una nuova sintassi vuol dire un nuovo linguaggio, probabilmente incompatibile con altri strumenti che lo sviluppatore utilizza nel suo lavoro, come ad esempio IDE, editor di testi avanzati, o altri preprocessori o generatori di codice.

Inoltre le annotation sono uno strumento già disponibile in Java e si prestavano al nostro scopo, utilizzare queste anziché definire una nuova sintassi e sviluppare un preprocessore in grado di analizzarla ed elaborarla ci ha permesso una maggiore rapidità di sviluppo. Inoltre le API Java per la reflection (con cui abbiamo effettivamente analizzato il codice scritto dallo sviluppatore) sono ben documentate e facili da usare a differenza di altri strumenti.

Utilizzare qualcosa che esiste già, anziché ricrearlo da zero è quasi sempre la scelta migliore, va notato però che non sempre è possibile: nel nostro caso la espressività delle annotation era sufficiente in altri casi questa è limitante e quindi è necessario introdurre nuovi costrutti.

Capitolo 3

Strumenti utilizzati

3.1 Design Patterns

Nello sviluppo di **ModelMapper** si è ricorsi all'utilizzo di diversi *design patterns*. Utilizzare i pattern significa produrre progetti standard, riusabili e ben documentati. *Standard* perché non cercano soluzioni complesse a problemi noti, ma usano lo stato dell'arte del mercato. *Riusabili* perché il pattern di per sé è già un concetto riusabile: sposta l'idea di riuso dal semplice codice alla sua progettazione. *Ben documentati* perché un pattern è un pattern, e quindi è lecito ritenere che gli esperti del settore sappiano di cosa si sta parlando.

3.1.1 Tipi di Design Patterns

I Design Pattern variano per livello di astrazione, granularità ed entità coinvolte (classi/oggetti). Si possono distinguere tre tipi di Design Pattern: *creazionali*, *strutturali* e *comportamentali* . Si riporta l'analisi delle varie categorie.

- **Pattern creazionali**: supportano la creazione di oggetti in un sistema senza dover identificare una specifica classe; aiutano quindi a costruire un sistema indipendente da come i suoi oggetti sono creati, composti e rappresentati. Un pattern creazionale usa l'ereditarietà per variare la classe che viene istanziata. Propongono quindi soluzioni per creare oggetti. I Design Pattern creazionali astraggono il processo di istanziazione. Incapsulano la conoscenza di quali classi concrete vengono usate e nascondono il come vengono istanziate e composte. Rendono flessibili il cosa viene creato, chi, come e quando lo crea.

- **Pattern strutturali**: controllano le relazione tra grosse porzioni di applicazione e migliorano la riusabilità e le funzionalità del sistema. Propongono quindi soluzioni per la composizione strutturale di classi e oggetti; descrivono come classi e oggetti possono essere composti per formare strutture più grandi.

- **Pattern comportamentali**: riguardano gli algoritmi e l'assegnazione delle responsabilità tra gli oggetti. Descrivono gli oggetti e le classi, la loro comunicazione e collaborazione, ottimizzando il modo con cui lo stato del sistema viene trasmesso e modificato. Semplificano e aumentano la manutenibilità dell'applicazione. Propongono quindi delle soluzioni per gestire le suddivisioni delle responsabilità delle classi e degli oggetti. Si occupano quindi di algoritmi e di assegnamento di responsabilità tra oggetti.

3.1.2 Patterns utilizzati

Segue ora la presentazione dei core patterns utilizzati nello sviluppo di **ModelMapper**.

3.2 Proxy/Surrogate

Un oggetto svolge il proprio lavoro a supporto dell'interfaccia pubblica che lo rende visibile. Può accadere, tuttavia, che un oggetto non riesca a soddisfare in pieno questa responsabilità base. In questi casi, un oggetto proxy può prendere su di sè la responsabilità di soddisfare le richieste di un client. Lo scopo di un proxy è di fornire un surrogato o un segnaposto ad un altro oggetto per controllare l'accesso a tale oggetto. Una ragione per effettuare un controllo sull'accesso a un oggetto può essere quella di rinviare il costo della sua creazione e inizializzazione fino a quando l'oggetto non è effettivamente necessario. Si consideri per esempio un editor di documenti che consenta di inserire oggetti grafici all'interno del documento, alcuni dei quali, per esempio immagini di grandi dimensioni, possano introdurre costi significativi per la loro creazione. L'apertura di un documento dovrebbe essere veloce, pertanto dovremmo evitare di creare oggetti "costosi" tutti insieme al momento dell'apertura del documento. Tale restrizioni dovrebbero suggerire di creare oggetti costosi *on demand*, il che significherebbe in questo caso, ogni volta che un'immagine diviene visibile.

Questa ottimizzazione non deve però avere impatto sul codice per la visualizzazione.

La soluzione consiste nell'utilizzare un altro oggetto, un *proxy* dell'immagine, che ha lo stesso comportamento dell'immagine vera e propria e si preoccupa di istanziarla quando richiesto.

Il proxy crea l'immagine soltanto quando l'editor di documenti chiede al proxy di disegnarla sullo schermo. Tutte le richieste successive vengono trasferite dal proxy direttamente all'immagine, pertanto il proxy deve mantenere un riferimento all'immagine dopo la sua creazione.

3.2.1 Applicabilità

Il pattern Proxy è applicabile ogni volta che si ha la necessità di avere un riferimento ad un oggetto che sia più versatile o raffinato di un semplice puntatore.

Esistono svariate situazioni in cui il pattern Proxy è applicabile:

1. **Un proxy remoto** fornisce un rappresentante locale per un oggetto in un diverso spazio d'indirizzamento.

2. **Un proxy virtuale** gestisce la creazione su richiesta di oggetti "costosi".

3. **Un proxy di protezione** controlla l'accesso a un oggetto. Questo tipo di proxy si rivela utile quando possono essere definiti diritti di accesso diverso per gli oggetti.

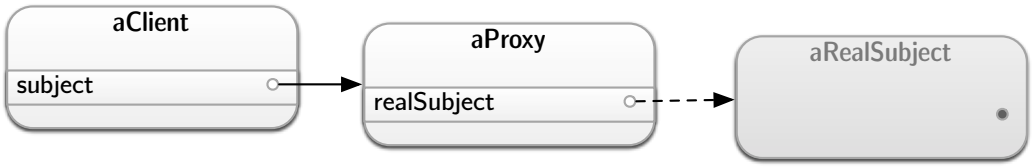
4. **Un riferimento intelligente** sostituisce un puntatore puro a un oggetto, consentendo l'esecuzione di attività aggiuntive quando si accede all'oggetto referenziato. Ecco alcuni utilizzi tipici:

   - Il conteggio dei riferimenti all'oggetto reale, in modo da gestire automaticamente la sua distruzione quando non ci sono più riferimenti attivi – implementazione di un meccanismo di garbage collection basato su *release/retain*.

- Il caricamento di un oggetto persistente in memoria quando viene referenziato per la prima volta.
- La verifica che l'oggetto rappresentato sia riservato (*locked*) quando si richiede di accedervi, in modo che nessun altro oggetto possa modificarlo.

3.2.2 Conseguenze

Il pattern proxy introduce un livello d'indirezione nell'accesso a un oggeto.



L'indirezione aggiuntiva ha finalità diverse, secondo il tipo di proxy considerato:

1. un proxy remoto può nascondere agli utilizzatori il fatto che un oggetto risiede in uno spazio di indirizzamento diverso;

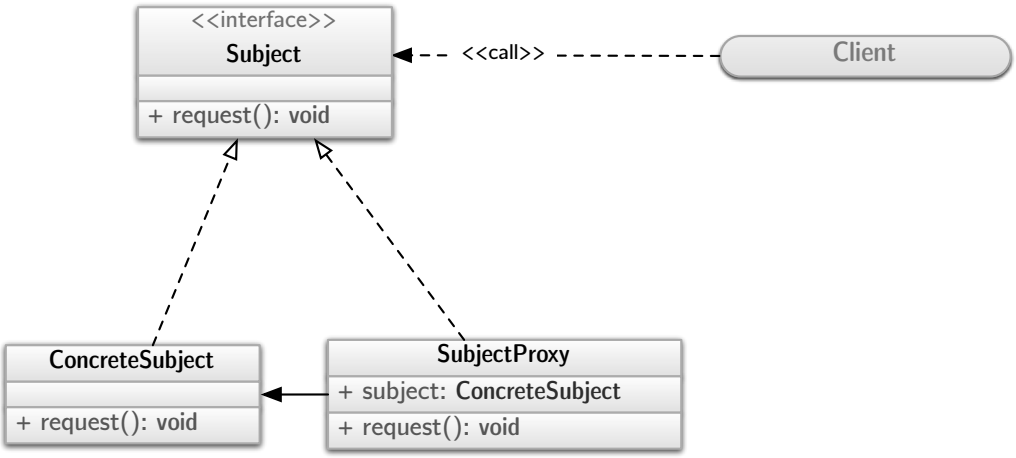
2. un proxy virtuale può svolgere ottimizzazioni quali la creazione di un oggetto su richiesta;

3. sia i proxy di protezione che i riferimenti intelligenti consentono di svolgere operazioni di gestione aggiuntive quando è richiesto l'accesso ad un oggetto.

Esiste un'altra ottimizzazione che può essere effettuata con il pattern proxy e che può essere nascosta al client. Tale ottimizzazione è correlata alla creazione su richiesta e viene chiamata *copy-on-write*. La clonazione/copia di un oggetto complicato e di grandi dimensioni può rivelarsi un'operazione costosa e, se la copia non è modificata, non c'è motivo di sostenere tale costo.Utilizzando un proxy per posticipare il processo di clonazione possiamo assicurare che il costo della copia dell'oggetto è sostenuto soltanto quando l'oggetto viene modificato,cioè quando la copia è realmente necessaria. Per utilizzare questa tecnica occorre mantenere nel proxy un contatore di riferimenti dell'oggetto rappresentato. Copiare il proxy produce semplicemente un incremento nel contatore dei riferimenti; soltanto quando il client richiede un'operazione di modifica dell'oggetto rappresentato, il proxy effettua la copia vera e propria e decrementa il contatore dei riferimenti all'oggetto.

3.2.3 Implementazione

L'uso del proxy da parte di codice client deve essere trasparente, ossia il codice chiamante non deve assolutamene accorgersi del fatto che sta usando un proxy piuttosto che direttamente l'oggetto servitore.



Il pattern proxy riccorre proprio all'utilizzo delle *interfacce* per realizzare la trasparenza di invocazione dei metodi desiderati. Come si può vedere nel diagramma UML, il codice client vede solo un'interfaccia, la stessa implementata dall'oggetto servitore (chiamato `ConcretSubject`) e dal proxy (`subjectProxy`).

In Java è presente la classe Proxy (`java.lang.reflect.Proxy`) e classi di utilità che permettono di creare oggetti proxy dalle proprie classi che implementano una data interfaccia.

Di seguito viene presentato del codice di esempio che usa il supporto ai proxy fornito da Java. Viene anzi tutto definita una semplice interfaccia con un solo metodo

```
public interface TargetInterface {
        public void print();
}
```

e un'implementazione di questa interfaccia

```
public class TargetImplementation implements TargetInterface {
        public void print() {
                System.out.println("hello world");
        }
}
```

La classe `ProxyTarget` creerà il proxy della classe `TargetImplementation` (o di una qualsiasi implementazione di `TargetInterface`) e a cui verrano delegate le chiamate dei metodi dell'interfaccia `TargetInterface`:

```
import java.lang.reflect.*;
public class ProxyTarget implements InvocationHandler {
        public static Object createProxy(Object obj) {
                /* creates the proxy class according the instance of the target class */
                return Proxy.newProxyInstance(obj.getClass().getClassLoader(),
                obj.getClass().getInterfaces(), new ProxyTarget(obj));
        }
        private Object target;
        private ProxyTarget(Object target) { this.target = target; }
        //dispatching
        public Object invoke(Object proxy, Method method, Object[] args) {
                Object result = null;
                try { /* do what you want */ result = method.invoke(target,args);
                } catch(Exception e){ out.println(e.getMessage()); }
                return result;
        }
}
```

3.3 Factory Method

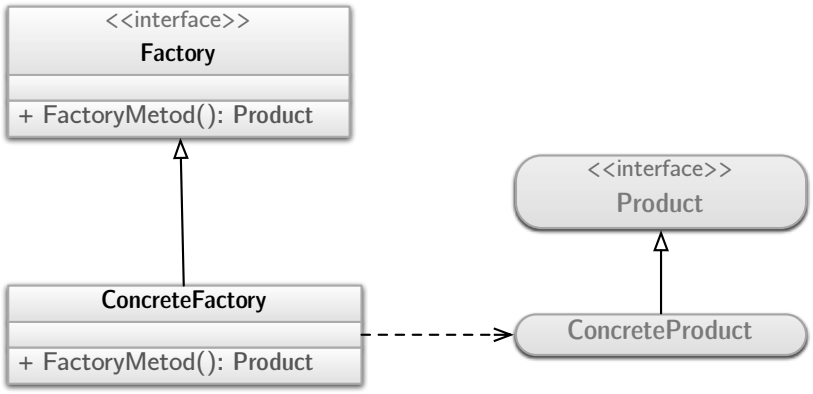
Il Factory Method, come altri *creational pattern*, fornisce un metodo per istanziare un oggetto senza sapere a priori la sua esatta classe. Questo pattern raggiunge il suo scopo fornendo un'interfaccia per creare un oggetto, ma lascia che le sottoclassi decidano quale oggetto istanziare. Esso può servire per costruire oggetti per uno specifico proposito senza che il richiedente conosca la classe specifica che viene istanziata. I framework utilizzano classi astratta per definire e mantenere le relazioni tra oggetti e spesso sono anche responsabili della creazione di questi oggetti.

3.3.1 Applicabilità

Il pattern Factory Method può essere applicato quando:

- una classe non è in grado di sapere in anticipo le classi degli oggetti che deve creare;

- una classe vuole che le sue sottoclassi scelgano gli oggetti da creare;
- le classi delegano la responsabilità a una o più classi di supporto (*helper classes*), e si vuole localizzare in un punto ben preciso la conoscenza di quale o quali classi di supporto vengano delegate.



3.3.2 Conseguenze

L'utilizzo di metodi factory elimina la necessità di riferirsi a classi dipendenti dall' applicazione all'interno del codice. Il codice si basa unicamente sull'interfaccia `Product` e pertanto può operare con qualunque classe `ConcreteProduct` definita dall'utente. Un potenziale svantaggio dell'utilizzo dei metodi di factory è che gli utilizzatori potrebbero essere costretti a definire delle sottoclassi di `Factory` solo per creare un particolare oggetto `ConcreteProduct`. Quando le sottoclassi devono essere utilizzate comunque, questo non è un problema, in casi contrario l'utilizzatore deve tener conto di un altro possibile elemento variabile nel codice dell'applicazione. L'utilizzo del pattern Factory Method ha due ulteriori conseguenze.

- *Fornisce un punto d'aggancio per le sottoclassi.* Utilizzare un metodo factory per la creazioni di oggetti in una classe fornisce sempre una flessibilità maggiore rispetto alla creazione diretta dell'oggetto. Il pattern Factory Method fornisce alle sottoclassi un punto d'aggancio per la produzione di una versione specializzata di un oggetto.
- *Connette gerarchie di classi parallele.* Gerarchie di classi parallele si ottengono quando una classe delega alcune delle sue responsabilità ad una classe separata.

3.3.3 Implementazione

Consideriamo le seguenti questioni legate al'applicazione del pattern Factory Method.

1. *Due varianti principali.* Le due varianti principali del pattern Factory Method sono: **(1)** il caso in cui la class `Factory` è una classe astratta e non fornisce un'implementazione standard per il metodo `factory` dichiarato, e **(2)** il caso in cui la classe `Factory` è una classe concreta e fornisce un'implementazione di base per tale metodo. E' altresì possibile avere una classe astratta che fornisce un'implementazione di base, ma è meno comune dei due casi considerati. Il primo caso richiede che le sottoclassi definiscano un'implementazione per il metodo `factory`, poiché non esiste una ragionevole implementazione di base. Questo approccio fornisce una soluzione al problema dell'istanziazione di classi non identificabile a priori. Nel secondo caso, la classe `Factory` concreta utilizza il metodo factory principalmente per questioni di flessibilità. La regola seguita è *sposta la creazione di oggetti in un operazione separata, in modo che le sottoclassi possano ridefinire il modo con cui gli oggetti sono creati.* Questa regola assicura che i progettisti della sottoclasse possano cambiare, se necessario, la classe degli oggetti istanziati dalla loro classe progenitrice.

2. *Metodi factory parametrici.* Un'altra variante del pattern consente al metodo `factory` di creare tipologie di prodotto multiple. In questo caso il metodo factory riceve un parametro che identifica il tipo di oggetto che deve essere creato. Tutti gli oggetti creati dal metodo factory devono condividere l'interfaccia `Product`.

```
public interface ProductA extends Product { ... }
public class Factory {
        public static <T extends Product> factory(Class<T> type) { ... }
}
...
ProductA p = Factory.factory(ProductA.class);
```

3.4 Lazy Initialization

La Lazy Initialization (inizializzazione pigra) constiste nella tattica di istanziare un oggetto, inizializzare una variabile, effettuare un calcolo od eseguire un processo solo nel momento in cui tale operazione è richiesta. Tipicamente, questo si ottiene

memorizzando in un *flag* l'avvenimento di un determinato processo. Ogni volta che avviene un certo evento, si esamina il flag. Se questo è abbassato, si continua, altrimenti si inizializza una certa variabile o si istanzia un certo oggetto.

3.4.1 Applicabilità

Dal punto di vista dei design pattern, la lazy initialization si usa spesso con un *factory method*. Questo combina tre idee:

- usare un factory method per instanziare una classe;
- memorizzare l'istanza di una mappa, in modo tale da poter riprendere la stessa istanza la volta successiva che si richiede la stessa con certi parametri;
- usare la lazy initialization per istanziare un oggetto la prima volta che è richiesto.

3.4.2 Implementazione

Un piccolo esempio: la classe `Fruit` non fa nulla qui, questo è solo un esempio per mostrare l'architettura. La variabile `types` è una mappa usata per memorizzare le istanze di `Fruit` per tipo.

```
public class Fruit {
        private static Map types = new HashMap();
        private String type;

        private Fruit(String type){ this.type = type; types.put(type, this); }

        public static Fruit getFruit(String type) {
                Fruit f;
                if (types.containsKey(type)) f = (Fruit) types.get(type);
                else f = new Fruit(type);

                return f;
        }
}
```

3.5 Java Reflection API

Le API reflection, definite nel package `java.lang.reflection` sono un'infrastruttura di classi messe a disposizione dello sviluppatore che ha l'esigenza di ispezionare a tempo di esecuzione alcune proprietà di altre classi. E' possibile esaminare le proprietà di una classe (come i campi, i metodi, costruttori e le interfacce implementate) senza sapere a priori neppure il nome o la struttura, ma ottenendo queste informazioni durante l'esecuzione. Si tratta di uno strumento indispensabile per chi desidera scrivere tools ispezionatori di classi, debuggers, caricatori di plugin o più in generale per la creazione di applicazioni a componenti. L'uso della reflection è sconsigliata agli sviluppatori meno esperti e va evitata in tutte quelle situazioni in cui è possibile farne a meno, è infatti uno strumento da un lato molto potente ma anche molto pericoloso: può incidere negativamente sulla performance e sulla sicurezza (permette infatti di compiere operazioni, com ad esempio la lettura di campi privati, che non dovrebbero essere generalmente permesse).

Oggetto della reflection sono le classi, quindi partendo dal principio vediamo come è possibile ottenere un riferimento ad oggetto `Class` (come è noto infatti nella programmazione object-oriented tutto è un oggetto, anche le classi stesse), esistono tre metodi per farlo.

Possiamo trovare una classe a partire da una stringa:

```
Class c = Class.forName(className);
```

dove className specifica il percorso della classe cercata, come ad esempio `"javax.swing.Timer"`. Si noti che come abbiamo accennato in precedenza la stringa potrebbe non essere nota a tempo di compilazione (potrebbe ad esempio essere letta da uno standard di input). Alternativamente possiamo trovare la classe a partire da un'istanza:

```
Class c = o.getClass();
```

anche questa volta, la classe non sempre è nota a tempo di compilazione, come mostra il frammento di codice sottostante

```
Object o;
if (b)
        o = new A();
else
        o = new B();
```

dove b è un boolean (che potrebbe essere differente ad ogni esecuzione). Infine se invece la classe è nota già in fase di compilazione possiamo semplicemente recuperare la classe aggiungendo il suffisso `.class`, come mostrato di seguito.

```
Class c = javax.swing.Timer.class
```

Una volta recuperata la classe di nostro interesse le API ci mettono a disposizione diversi metodi per ispezionarli, ne vediamo alcuni:

```
boolean        isInterface()
boolean        isPrimitive()
```

che ci indicano se la classe è un'interfaccia o meno (il primo), o se è un tipo primitivo (come int, void double, etc..) o meno (il secondo)

```
Method         getDeclaredMethod(String name, Class[] parameterTypes)
Method[]       getDeclaredMethods()
Method         getMethod(String name, Class[] parameterTypes)
Method[]       getMethods()
```

che ci permettono di cercare (il primo e il terzo), oppure listare (il secondo e il quarto) i metodi dichiarati in una classe (il primi due) o quelli disponibili, e quindi eventualmente anche ereditati (i secondi due). In maniera analoga possiamo cercare i campi, con i seguenti metodi:

```
Field          getDeclaredField(String name)
Field[]        getDeclaredFields()
Field          getField(String name)
Field[]        getFields()
```

o i costruttori. Una volta ottenuti i riferimenti agli oggetti `Field`, `Method` o `Constructor` possiamo ispezionare anche questi. Prendiamo per esempio la classe `Method` ecco di seguito alcuni metodi che fornisce:

```
Class          getDeclaringClass()
Class[]        getExceptionTypes()
String         getName()
Class[]        getParameterTypes()
Class          getReturnType()
```

che ci forniscono in ordine, la classe di appartenenza, le eccezioni sollevabili, il nome, i parametri e il tipo di ritorno. Fino a qua ci siamo limitati ad analizzare i metodi, vediamo che è possibile anche invocare dinamicamente uno di questi attraverso:

```
Object  invoke(Object obj, Object[] args)
```

Molte altre operazioni sono disponibili anche per Field e Constructor. Ci siamo limitati alla trattazione di una piccola parte delle potenzialità del package `java.lang.reflection`, per maggiori dettagli invitiamo il lettore a consultare il tutorial fornito da Sun.

Capitolo 4

Manuale

Forniamo questo capitolo una breve panoramica su come utilizzare il framework.

4.1 "Getting started"

Come prima cosa è necessario assicurarsi di aver scaricato l'archivio jar contenente le classi del framework e aver inserito questo nel classpath del sistema oppure di lanciare gli eseguibili java e javac anteponendo il parametro `-cp /.../.../ModelMapper.jar`.

Segue un esempio di una classe `Test.java` che utilizza il framework.

```
import modelmapper.*;
import modelmapper.provider.mysql.*;

public class Test {

        public static void main(String[] args) {

                ModelFactory factory = new MySQLModelFactory(
                        "jdbc:mysql://localhost/mm",
                        "username", "password");
                Person p = factory.create(Person.class);
                p.setFirstName("luca");
                p.save();
        }
}
```

Come prima cosa notiamo le due direttive import, il primo serve per importare le generiche classi di ModelMapper, il secondo per le classi che gestiscono l'interazione che il database MySQL ed è infatti in questo package che si trova la definizione di `MySQLModelFactory`.

Nella prima riga del main viene istanziato un oggetto di tipo `ModelFactory`, gli oggetti di questo tipo sono il cuore del framework e permettono di creare nuovi istanze di modelli o di recuperare dati già presenti nello strumento di persistenza.

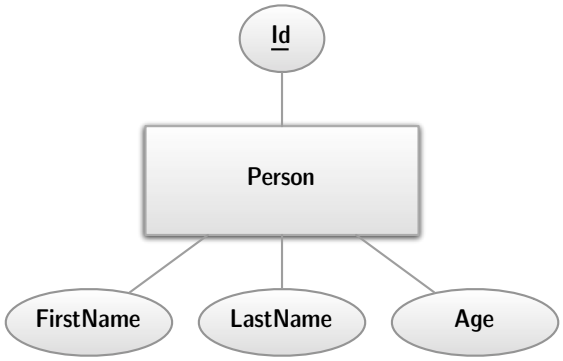
Per ogni strumento di persistenza supportato dal framework si presuppone che esista un factory associato, anche se a questo stadio dello sviluppo è presente soltanto il supporto al DBMS MySQL.

Il costruttore di `MySQLModelFactory` prende in input un uri jdbc, e le credenziali (username e password) per accedere al db.

Nel restante codice viene creata un istanza del modello di tipo *Person*, viene impostato un campo e viene salvato sul database. Prima di approfondire nel dettaglio queste righe e i metodi offerti dal `ModelFactory` ci concentreremo sulla definizione di un modello.

4.2 Definizione di un modello

Mostreremo come definire i modelli procedendo per esempi. Iniziamo da un semplice caso di studio. Vogliamo definire il modello `Person` che dispone dei soli tre attributi `FirstName`, `LastName` e `Age`, e di un attributo identificatore `Id`. Vediamo prima una rappresentazione secondo il modello ER, in seguito il codice necessario per la definizione in ModelMapper




```
import modelmapper.*;
```

```
import modelmapper.annotation.*;

public interface Person extends Model {

        public String getFirstName();
        public void setFirstName(String name);

        public String getLastName();
        public void setLastName(String name);

        public int getAge();
        public void setAge(int age);

        public boolean isStudent();
        public void setStudent(boolean student);
}
```

Ogni modello è definito tramite un'interfaccia che estende `Model`, o un altro modello già definito (come vedremo negli esempi relativi all'ereditarietà dei modelli), inoltre è necessario utilizzare le direttive import così come mostrato nell'esempio.

Per ogni campo è necessario definire un metodo get e set seguito dal nome. I metodi get hanno come tipo di ritorno il tipo del campo e nessun parametro, al contrario i metodi set hanno come tipo di ritorno `void` e un parametro del tipo del campo. Fanno eccezioni i campi booleani il cui metodo `get` è sostituito dal metodo con il prefisso `is`

Notiamo che non è stato necessario definire `getId` e `setId`, `Id` è infatti l'attributo identificante di default (intero e autoincrementante) nel caso non ne venga definito uno diversamente. Se ad esempio avessimo voluto identificare il modello tramite la stringa codice fiscale avremmo dovuto aggiungere all'interfaccia la seguente dichiarazione:

```
        @Id
        public String getCF();
        public void setCF(String cf);
```

Ci può essere un solo attributo identificante che sostituisce quello di default, il tipo di questo campo deve essere esclusivamente `int` o `String` e il nome chiaramente deve differire da `Id`. Come intuibile questo attributo viene mappato negli RDBMS come `PrimaryKey`.

Il simbolo `@Id` è un esempio di annotation utilizzata per etichettare la proprietà di un campo, queste devono precedere il metodo `get` o `is` del campo a cui sono riferite, segue un elenco delle annotation che è possibile utilizzare:

Annotazione	Descrizione
`@Id`	Attributo identificante
`@NotNull`	Attributo non nullo
`@Unique`	Attributo unico
`@AutoIncrement`	Attributo auto incrementato

Va precisato che `@AutoIncrement` può essere usato sia in combinazione con `@Id` e `@Unique` che da solo, inoltre ovviamente `@Id` e `@Unique` sono tra loro incompatibili.

Tutti i metodi definiti fin ora sono relativi ai campi del modello, è possibile definire altri metodi di business logic, anteponendo l'annotation `@BusinessLogic` e fornendone un'implementazione. Vediamo un esempio estendendo il nostro modello `Person`:

```
import modelmapper.*;
import modelmapper.annotation.*;

public interface Person extends Model {
        @Id
        public String getCF();
        public void setCF(String cf);

        public String getFirstName();
        public void setFirstName(String name);

        public String getLastName();
        public void setLastName(String name);

        public int getAge();
        public void setAge(int age);

        public boolean isStudent();
        public void setStudent(boolean student);

        @BusinessLogic
        public String fullName();
}
```

I metodi di business logic non possono avere il prefisso `set`, `get`, `is`, `add` o `remove` e non possono chiamarsi `save`, `validate`, `find` o `migrate`.

Per i metodi `@BusinessLogic` di `Person` è necessario fornire un'implementazione, quindi definiamo la classe `PersonImpl`

```
import modelmapper.*;

public class PersonImpl extends ModelProxy {

        public String fullName {
                return getFirstName() + "␣" + getLastName();
        }
}
```

Notiamo che `PersonImpl` estende `ModelProxy` e non implementa `Person` (come d'altronde è ovvio, infatti non viene fornita nessuna implementazione dei metodi `get` e `set`). In queste classi forniamo esclusivamente l'implementazione dei metodi `@BusinessLogic`; la classe si deve trovare nello stesso package dell'interfaccia che definisce il modello e per convenzione ha il suffisso `Impl`; è possibile utilizzare un altro nome specificandolo attraverso l'annotation di classe `@Implementation` come mostrato di seguito:

```
import modelmapper.*;
import modelmapper.annotation.*;

@Implementation(MyPersonImplementation.class)
public interface Person extends Model {
        [..]
}
```

quindi:

```
import modelmapper.*;

public class MyPersonImplementation extends ModelProxy {
        [..]
}
```

Ora che abbiamo finito la definizione del modello, possiamo riprendere l'esempio fornito nel paragrafo **6.1** per provarlo. Anticipiamo che i metodi accessibili per l'oggetto `Person` saranno quelli definiti nell'interfaccia più `save()`, `delete()` e `validate()`. Li analizzeremo in seguito nel dettaglio.

4.3 Connection

Abbiamo visto, fino ad ora, come definire un singolo Modello, vediamo ora come mettere in relazione diversi modelli tra di loro, questo è possibile tramite le *connection*.

Partiamo da un esempio, estendendo `Person` e aggiungendogli il collegamento con uno o più `Pet`.

```
public interface Person extends Model {
        [..]

        @Connection(name = "PP", type = ConnectionType.Aggregation)
        public Pet[] getPets();
        public void setPets(Pet[] pets);
}

public interface Pet extends Model {
        [..]

        public String getName();
        public void setName(String name);

        @Connection(name = "PP", type = ConnectionType.BelongsTo)
        public Person getOwner();
        public void setOwner(Person owner);
}
```

Ogni `@Connection` ha un tipo e un nome, i metodi `set` e `get` hanno come parametro e tipo di ritorno un oggetto Model o un array di Model a seconda della cardinalità della relazione.

Le connection mettono in relazione sempre esattamente due modelli, a meno che siano simmetriche (tralasciamo questo caso per il momento, vedremo un esempio in seguito) pertanto per ognuna di queste annotation deve esserci la reciproca, con lo stesso nome, e tipo appropriato, similmente a come abbiamo visto nell'esempio precedente.

Esistono 4 tipi di `@Connection`:

- **Aggregation**: Un modello possiede uno o più oggetti, "una persona possiede uno o più animali" nell'esempio mostrato. Vi è quindi una cardinalità **1..N** il tipo di ritorno sarà un array di modelli (ad esempio `Pet[]`) Se vogliamo ridurre

la cardinalità a **1..1** cambiamo il tipo di ritorno in un singolo modello (ad esempio `Pet`). Se il modello che possiede l'altro viene eliminato, questo non cessa di esistere, nel nostro esempio "se una persona muore l'animale continua a vivere, cambiando padrone"

- **Composition**: è una variante della aggregation, valgono le stesse regole ad eccezione del fatto che se l'oggetto che possiede gli altri viene eliminato questi vengono automaticamente cancellati. Un esempio potrebbe essere la relazione tra una cartella e i file contenuti in essa.
- **BelongsTo**: l'oggetto è posseduto da altri tramite un'aggregation o una composition. Se non viene specificato diversamente una `@Connection` è di questo tipo.
- **ManyToMany**: permette la relazione [**N..M**], "un insegnante ha tanti allievi, un allievo ha tanti insegnanti". Il tipo di ritorno del metodo `get` sarà sempre un array di modelli. Come nel caso dell'aggregation se un oggetto viene cancellato quelli a esso connessi non vengono eliminati.

Per le connection aggregation e composition del tipo [**1..N**] e per le ManyToMany, oltre ai metodi `get` e `set` è possibile definire anche i metodi `add` e `delete`. Segue un esempio.

```
public interface Student extends Model {
    [..]

    public String getLogin();
    public void setLogin(String name);

    @Connection(name = "SM", type = ConnectionType.Composition)
    public Mark getMarks();
    public void setMarks(Mark[] marks);
    public void addMarks(Mark... marks);
    public void deleteMarks(Mark... marks);
}
```

4.4 Ereditarietà dei modelli

A differenza di altri framework ModelMapper permette l'ereditarietà tra modelli.

Vogliamo ad esempio che il modello `Student` estenda `Person` infatti possiamo dire che "uno studente è una persona", la dichiarazione è molto naturale: si usa la parola chiave extends propria delle interfaccie Java.

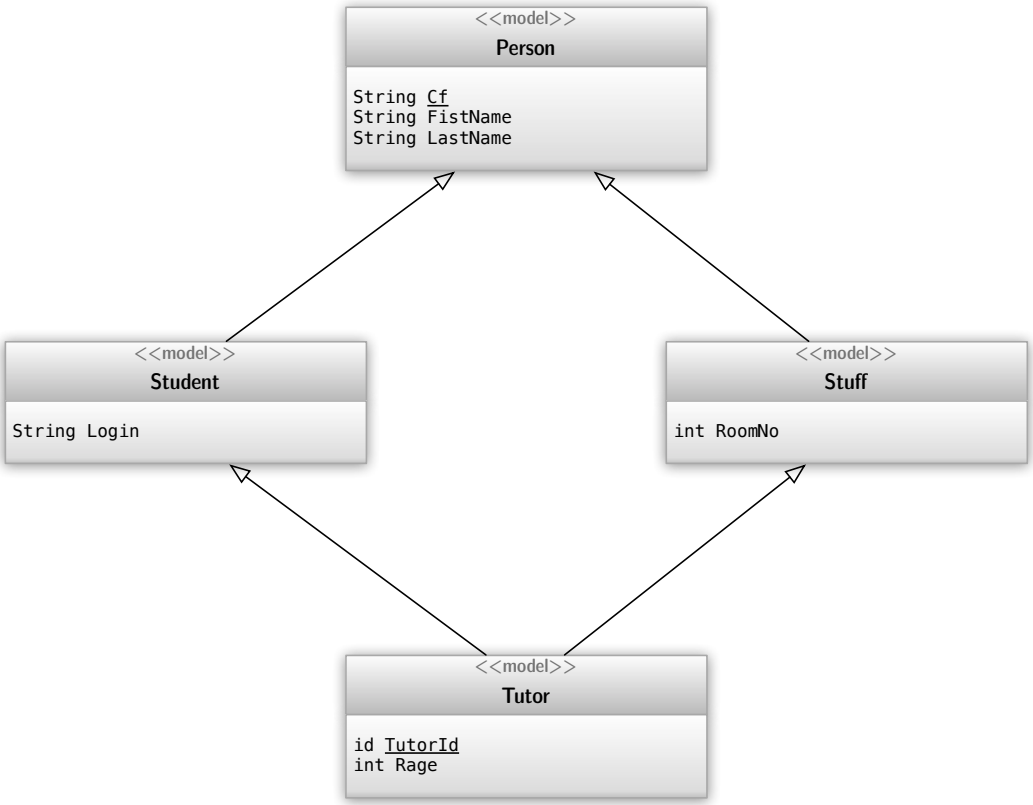
```
public interface Person extends Model {
    @Id
    public String getCF();
    public void setCF(String cf);

    public String getFirstName();
    public void setFirstName(String name);

    public String getLastName();
    public void setLastName(String name);
}

public interface Student extends Person {
    public String getLogin();
    public void setLogin(String name);
}
```

`Student` disporrà di tutti i suoi campi più quelli di `Person`.

L'interfaccia figlia non può ridefinire una componente del padre, perché risulterebbe ridondante nella definizione del modello.

Come è noto, in Java le interfacce supportano l'ereditarietà multipla, così abbiamo voluto rendere disponibile questa caratteristica anche in `ModelMapper`.

Vediamo un esempio partendo da un diagramma UML.



Analizziamo quindi la definizione dei modelli:

```
public interface Person extends Model {
    @Id
    public String getCF();
    public void setCF(String cf);

    public String getFirstName();
    public void setFirstName(String name);

    public String getLastName();
    public void setLastName(String name);
}

public interface Student extends Person {
    public String getLogin();
```

```
    public void setLogin(String name);
}

public interface Staff extends Person {
    public int getRoomNo();
    public void setRoomNo(int roomNo);
}

public interface Tutor extends Staff,Student {
        @Id
        public int getTutorId();
        public void setTutorId(int id);

    public int getRate();
    public void setRate(int rate);
}
```

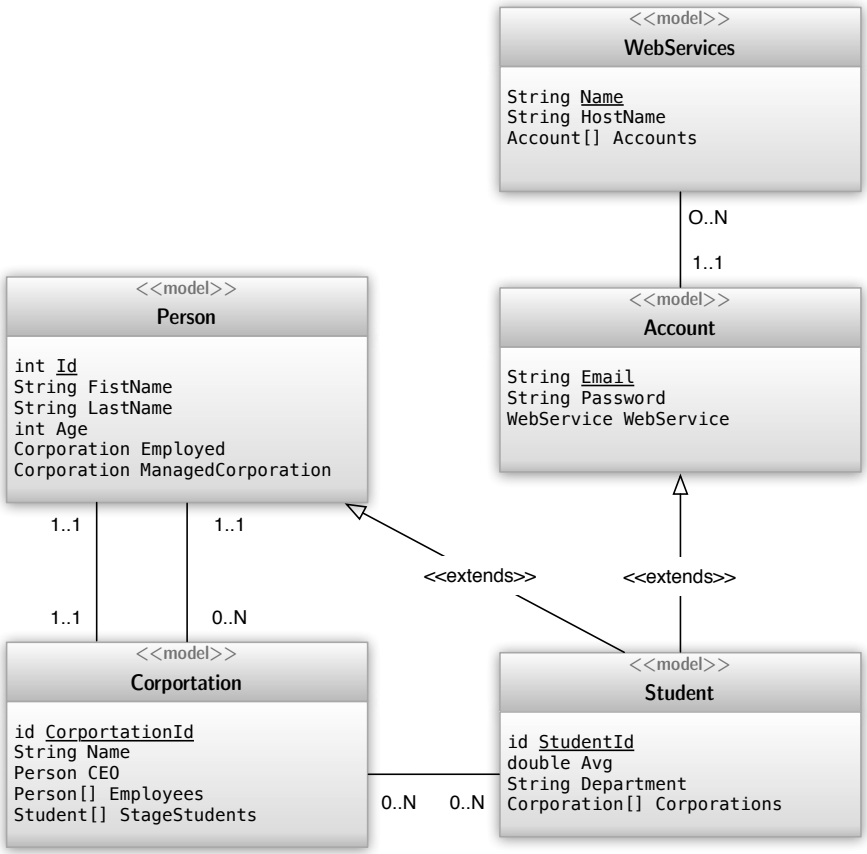
Il `Tutor` disporrà di tutti i suoi campi più quelli di `Staff`, `Student` e `Person`.

Quando si usa l'ereditarietà multipla in ModelMapper bisogna tenere presente che non è possibile estendere due classi che definiscono la stessa componente, quindi è compito del programmatore risolvere manualmente i conflitti.

4.5 Caso di studio

Per concludere questa panoramica sulla definizione dei modelli, esaminiamo un caso di studio: analizzeremo prima uno schema UML, poi la traduzione nella sintassi di definizione dei modelli di ModelMapper, infine la traduzione SQL generata dall'adapter MySQL.

4.5.1 Schema UML



Si noti che è un esempio strettamente didattico, dire che "uno studente è un account" anzichè "uno studente ha un account" è una forzatura semantica per mostrare le possibilità date dal framework

4.5.2 Definizione in ModelMapper

```
public interface Account extends Model {

        @Id
        public String getEmail();
        public void setEmail(String email);
```

```
        public String getPassword();
        public void setPassword(String password);

        @Connection(name = ”AS”)
        public WebService getWebService();
        public void setWebService(WebService service);


}

public interface Corporation extends Model {

        @Id
        @AutoIncrement
        public int getCorporateId();
        public void setCorporateId(int id);

        public String getName();
        public void setName(String name);

        @Connection(name = ”CEO”, type = ConnectionType.Aggregation)
        public Person getCEO();
        public void setCEO(Person founder);

        @Connection(name = ”Employed”, type = ConnectionType.Composition)
        public Person[] getEmployees();
        public void setEmployees(Person[] s);

        @Connection(name = ”Stage”, type = ConnectionType.ManyToMany)
        public Student[] getStageStudents();
        public void setStageStudents(Student[] s);

}

public interface Person extends Model {

        public String getFirstName();
```

```
        public void setFirstName(String name);

        public String getLastName();
        public void setLastName(String name);

        public int getAge();
        public void setAge(int age);

        @Connection(name = "Employed")
        public Corporation getEmployer();
        public void setEmployer(Corporation c);

        @Connection(name = "CEO")
        public Corporation getManagedCorporation();
        public void setManagedCorporation(Corporation c);
}

public interface WebService extends Model {

        @Id
        public String getName();
        public void setName(String name);

        public String getHostName();
        public void setHostName(String hostName);

        @Connection(name = "AS", type = ConnectionType.Composition)
        public Account[] getAccounts();
        public void setAccounts(Account[] a);

}
```

4.5.3 Traduzione SQL generata

```
CREATE TABLE Corporation(
    Name varchar(255) NULL,
    CorporateId int AUTO_INCREMENT NOT NULL,
    CreatedAt datetime NULL,
```

```
    UpdatedAt datetime NULL,
    PRIMARY KEY(CorporateId)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

CREATE TABLE Person(
    FirstName varchar(255) NULL,
    LastName varchar(255) NULL,
    Age int NULL,
    Employer int NULL,
    ManagedCorporation int NULL,
    Id int AUTO_INCREMENT NOT NULL,
    CreatedAt datetime NULL,
    UpdatedAt datetime NULL,
    PRIMARY KEY(Id)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

CREATE TABLE WebService(
    Name varchar(255) NOT NULL,
    HostName varchar(255) NULL,
    CreatedAt datetime NULL,
    UpdatedAt datetime NULL,
    PRIMARY KEY(Name)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

CREATE TABLE Account(
    WebService varchar(255) NULL,
    Password varchar(255) NULL,
    Email varchar(255) NOT NULL,
    CreatedAt datetime NULL,
    UpdatedAt datetime NULL,
    PRIMARY KEY(Email)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

CREATE TABLE Student(
    StudentId int AUTO_INCREMENT NOT NULL,
    Avg double NULL,
    Department varchar(255) NULL,
    CreatedAt datetime NULL,
```

```
    UpdatedAt datetime NULL,
    PersonId int NOT NULL,
    AccountEmail varchar(255) NOT NULL,
    PRIMARY KEY(StudentId)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

ALTER TABLE Person
    ADD FOREIGN KEY(Employer)
        REFERENCES Corporation(CorporateId) ON DELETE CASCADE,

    ADD FOREIGN KEY(ManagedCorporation)
        REFERENCES Corporation(CorporateId);

ALTER TABLE Account
    ADD FOREIGN KEY(WebService)
        REFERENCES WebService(Name) ON DELETE CASCADE;

ALTER TABLE Student
    ADD FOREIGN KEY(PersonId)
        REFERENCES Person(Id) ON DELETE CASCADE,

    ADD FOREIGN KEY(AccountEmail)
        REFERENCES Account(Email) ON DELETE CASCADE;

CREATE TABLE Stage (
    Id int AUTO_INCREMENT,
    StageStudents int,
    Corporations int,
    PRIMARY KEY(Id),
    FOREIGN KEY(StageStudents) REFERENCES Student(StudentId),
    FOREIGN KEY(Corporations) REFERENCES Corporation(CorporateId)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

4.6 Due strumenti forniti: Validator e Migration

Insieme al framework vengono forniti due strumenti. Il primo che chiamato “Validator”, permette di verificare la correttezza sintattica della definizione di un modello: se non si sono osservate le regole fornite da questo capitolo il Validator elencherà

gli errori. Si noti che è consigliabile testare i propri modelli, ma non strettamente necessario in quanto, quando viene caricato un modello viene comunque controllato (per performance viene controllato soltanto una volta ogni esecuzione anche se viene utilizzato molteplici volte) lanciando un eccezione run-time in caso di problemi.

E' possibile utilizzare questo strumento, per verificare la correttezza ad esempio dei modelli `Person` e `Account` mediante il seguente comando.

```
java modelmapper.Validator Person Account
```

L'elenco delle classi da controllare, può essere arbitrariamente lungo.

Il secondo strumento, "Migrator" permette di inizializzare il datasource per permettere il salvataggio di dati. Ovviamente questo è strettamente dipendente dal tipo di datasource, nel nostro caso ad esempio avendo implementato l'adapter per un RDBMS abbiamo sviluppato un generatore di schema SQL.

Di seguito un esempio.

```
java modelmapper.Migratior Person Account ––mysql jdbc:mysql://localhost/mm username pas
```

Questo strumento risolve automaticamente le dipendenze, se nell'esempio precedente `Person` dichiarasse una qualche connection o ereditasse un modello non passato come parametro, il tool si occuperà di inizializzare anche queste.

4.7 Utilizzo dei modelli

Vediamo in quest'ultima parte del capitolo come utilizzare i modelli creati.

4.7.1 Creazione di una nuova istanza

La creazione di una nuova istanza avviene tramite un oggetto di tipo `ModelFactory` dove è definito il metodo

```
<T extends Model> T create(Class<T> type)
```

che prende come parametro un oggetto di tipo classe (dove la classe è di tipo Model). Vediamo un semplice esempio

```
ModelFactory factory = [..]
Person p = factory.create(Person.class);
```

Si faccia riferimento alla Sezione 3.5 per ulteriori tecniche su come ottenere i riferimenti ad un oggetto classe.

4.7.2 Ricerca

Analogamente alla creazione, la ricerca avviene mediante il factory, ma in questo caso abbiamo diverse modalità. La prima avviene tramite l'invocazione dei metodi `find` o `advancedFind`.

```
<T extends Model> T[] find (Class<T> type,
                            String criteria,
                            Object... params)

<T extends Model> T[] advancedFind (Class<T> type,
                        String criteria,
                        String orderBy,
                        boolean desc,
                        int limit,
                        Object... params)
```

Il primo parametro è l'oggetto di tipo `class`, analogamente a quanto succede nelle creazione. I parametri `criteria` e `params` costituiscono il cuore della ricerca, si tratta di una sintassi SQL-like che permette di specificare i criteri di ricerca, analogamente a quanto faremmo nella clausola WHERE di una ricerca SQL (l'esempio servirà a chiarire meglio questo concetto).

```
ModelFactory factory = [..]
Person[] p = factory.find(Person.class,
                "(Age␣=␣?␣or␣Age␣>␣?)␣AND␣FirstName␣=␣?",10,18,"Luca");
```

La variabile `p` conterrà un array di persone che hanno esattamente 10 anni o ne hanno più di 18 il cui nome è Luca.

Per quanto riguarda il metodo `advancedFind` sono anche disponibili i parametri, `limit` per specificare il numero massimo di risultati che desideriamo ottenere, `orderBy` per specificare il campo secondo il quale vogliamo ordinare, e il `boolean` per specificare se vogliamo un ordine decrescente o meno.

```
ModelFactory factory = [..]
Person[] p = factory.advancedFind(Person.class, "FirstName␣=␣?␣AND␣LastName␣=␣?",
                    "FirstName",true,10,"foo","bar");
```

La seconda modalità avviene tramite l'invocazione del metodo `findWithSql` definito in `RDBMSModelFacotory` e quindi disponibile soltanto quando si usano database relazionali, vediamo solamente un esempio analogo al precedente con questa sintassi.

```
RDBMModelFactory factory = [..]
Person[] p = factory.findWithSql(Person.class, "SELECT␣*␣FROM␣'Person'␣" +
                    "WHERE␣LastName␣=␣foo␣AND␣FirstName␣=␣bar␣" +
                    "ORDER␣BY␣'FirstName'␣LIMIT␣10");
```

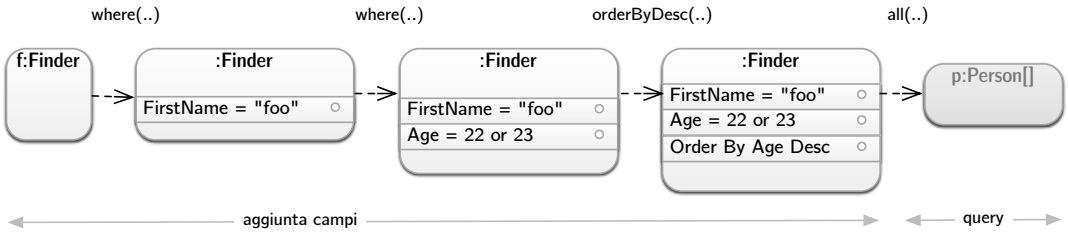
Infine l'ultima modalità è secondo noi la più interessante. Per utilizzarla bisogna in prima cosa creare un oggetto `Finder` attraverso il suo costruttore.

```
Finder(Class model, ModelFactory factory)
```

Il primo parametro è l'oggetto classe, mentre il secondo è il factory che stiamo utilizzando. Per spiegarne l'utilizzo partiamo da un esempio.

```
ModelFactory factory = [..]
Finder f= new Finder(Person.class,factory);
Person[] p = f.where("FirstName","foo").where("Age",22,23)
                                .orderByDesc("Age").all()
```

Tecnicamente finder è un oggetto immutable e quando viene invocato un metodo come `where` o `orderBy` viene creato e restituito un nuovo finder che contiene anche il nuovo parametro di ricerca. Sull'ultimo finder (quello che contiene tutti i parametri di ricerca) viene invocato il metodo `all` che esegue la ricerca e restituisce l'array di modelli.



Di seguito mostriamo l'elenco dei metodi di `Finder`.

```
T[]         all()
long        count()
T           first()
T[]         first(int limit)
Finder<T>   include(String... connectionsNames)
T           last()
T[]         last(int limit)
```

```
Finder<T>   orderByAsc(String field)
Finder<T>   orderByDesc(String field)
Finder<T>   where(String field, Object... args)
Finder<T>   whereGt(String field, Object... args)
Finder<T>   whereLt(String field, Object... args)
Finder<T>   whereNot(String field, Object... args)
```

4.7.3 Modifica, salvataggio e cancellazione

La modifica e la lettura di un modello è possibile esclusivamente tramite i metodi specificati nell'interfaccia di definizione del modello.

Attraverso il metodo `save` viene salvato l'oggetto nello strumento di persistenza, viene inserito nel caso sia nuovo e viene aggiornato nel caso sia già presente.

Prima della chiamata di `save` il framework esegue una chiamata a `validate`, questo può essere ridefinito nella classe di implementazione dei metodi di business logic, e nel caso venga restituito un `false` il framework non procede con il salvataggio e lancia un'eccezione `InvalidModelException`.

Per la cancellazione di una singola istanza è possibile chiamare il metodo `delete` sull'oggetto oppure invocare il metodo omonimo definito in `ModelFactory` passandogli come prima parametro l'oggetto classe, e come successivi parametri le istanze da cancellare.

Concludiamo così il capitolo dedicato all'utilizzo del framework, per maggiori dettagli rimandiamo alla documentazione del progetto.

Capitolo 5

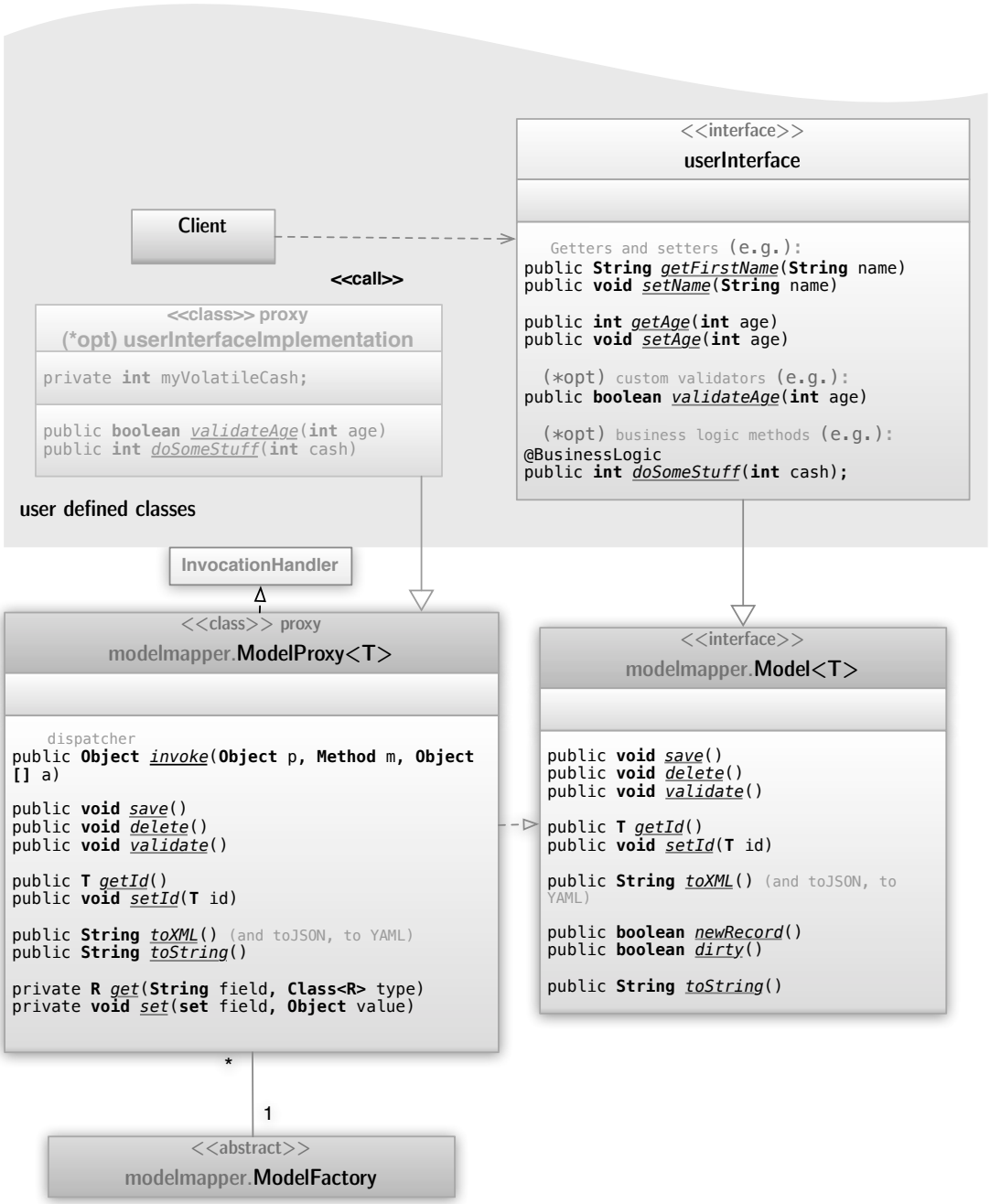
Implementazione

Dopo aver analizzato gli aspetti della progettazione ed aver fornito una panoramica sull'utilizzo di ModelMapper, forniamo ora una trattazione degli aspetti più interessanti dell'implementazione.

Si consideri innanzitutto che il framework da cui abbiamo preso maggiormente ispirazione è ActiveRecord, un ORM Ruby che si basa sull'omonimo pattern architetturale di cui si è precedentemente parlato. Un'implementazione in Ruby è decisamente più naturale, per proprietà intrinseche al linguaggio, come l'aggiunta di metodi a run-time e il duck-typing. Ne consegue che la realizzazione dei due framework risulta essere completamente diversa.

Nonostante l'uso del framework sia abbastanza naturale (basti pensare che un modello è descritto da una comune interfaccia Java), la sua implementazione sfrutta pattern piuttosto inusuali. La strategia chiave per la realizzazione di ModelMapper è l'*instance proxying*. L'instance proxying è una delle parti meno conosciute delle Java Reflection API che permette ad un oggetto che implementa l'interfaccia `InvocationHandler` di poter intercettare tutte le chiamate dei metodi ad un'interfaccia di un determinato tipo. In questo modo la classe `ModelFactory` può dinamicamente creare istanze che forniscono un'implementazione ad un'arbitraria sottointerfaccia di `Model`. L'istanza creata è un proxy che parsifica tutte le chiamate ai metodi presenti nell'interfaccia (definita dall'utente), e si occupa di gestire automaticamente le operazioni `CRUD` legate al modello.

Segue ora il diagramma UML (tralasciando il `ModelFactory`), per descrivere meglio il meccanismo spiegato sopra.


<<interface>>
userInterface
Client
<<call>>
Getters and setters (e.g.):
public String getFirstName(String name)
public void setName(String name)
public int getAge(int age)
public void setAge(int age)
(*opt) custom validators (e.g.):
public boolean validateAge(int age)
(*opt) business logic methods (e.g.):
@BusinessLogic
public int doSomeStuff(int cash);
<<class>> proxy
(*opt) userInterfaceImplementation
private int myVolatileCash;
public boolean validateAge(int age)
public int doSomeStuff(int cash)
user defined classes
InvocationHandler
<<class>> proxy
modelmapper.ModelProxy<T>
dispatcher
public Object invoke(Object p, Method m, Object [] a)
public void save()
public void delete()
public void validate()
public T getId()
public void setId(T id)
public String toXML() (and toJSON, to YAML)
public String toString()
private R get(String field, Class<R> type)
private void set(set field, Object value)
<<interface>>
modelmapper.Model<T>
public void save()
public void delete()
public void validate()
public T getId()
public void setId(T id)
public String toXML() (and toJSON, to YAML)
public boolean newRecord()
public boolean dirty()
public String toString()
*
1
<<abstract>>
modelmapper.ModelFactory

5.1 Descrizione delle classi

Verranno ora trattate nel dettaglio le classi *core* di ModelMapper.

5.1.1 ModelProxy

La classe proxy che implementa `InvocationHandler` è `ModelProxy`. Il metodo più importante di questa classe è `invoke`, che intercetta tutte le chiamate delle interfacce `Model`, parsifica i nomi dei metodi seguendo le regole imposte dalle convenzioni ed esegue la corrispondente operazione `CRUD` per mezzo delle opportune chiamate alla classe `ModelFactory`.

La seguente porzione di codice può chiarire meglio il meccanismo:

```
public Object invoke(Object proxy, Method method, Object[] args) {

  [..]
  if (isAGetter(method) && [..]) {
    String methodName = factory.getCache().fieldName(method);
    Class  retType = method.getReturnType();

    return get(methodName, retType);
  }

  if (isASetter(method) && [..]) {
    String methodName = factory.getCache().fieldName(method);
    set(mN, args[0]); return null;
  }
  [..]
}
```

`isAGetter`/`isASetter` controllano che il metodo sia un valido *get* o *set*, in seguito viene estrapolato il nome dell'attributo e invocato il metodo `get` passandogli il tipo di ritorno aspettato:

```
private <R extends Object> R get(String field, Class<R> returnType) {
  if (fields().containsKey(field)) return (R) f.get(field);
  return null;
}
```

I valori dei campi sono contenuti in una `HashMap` e il metodo privato `fields` si occupa di astrarre dai problemi legati all'ereditarietà mutipla (l'implementazione dell'ereditarietà tra i modelli verrà trattata in seguito).

I modelli legati da una `@Connection` e accessibili tramite i metodi getter definiti dall'utente, sono anch'essi contenuti in una struttura hash, ma con una gestione più complessa:

```
protected Map<String, List<Model>>
  fetched = new HashMap<String, List<Model>>();

private <R extends Model> R getConnection(Class<R> type, String field,
Connection connection) {

  if (!fetched.containsKey(field)) {

    R[] objs = (R[]) factory.fetch(type, this, c);
    List<Model> list = new ArrayList<Model>();

    if (objs != null)
      for (R o : objs) list.add(o);

    fetched.put(field, list);
  }

  List<Model> back = fetched.get(field);

  return
    back.toArray((R[])
    java.lang.reflect.Array.newInstance(type, back.size()));
}
```

Come è stato illustrato nel CAPITOLO MANUALE, è possibile estendere la classe `ModelProxy` per implementare i metodi annotati come `@BusinessLogic` e per definire i validatori dei campi.

5.1.2 ModelFactory

La classe `ModelFactory` è fondamentale nell'architettura di ModelMapper. Questa è responsabile dell'istanziazione dei modelli e non solo.

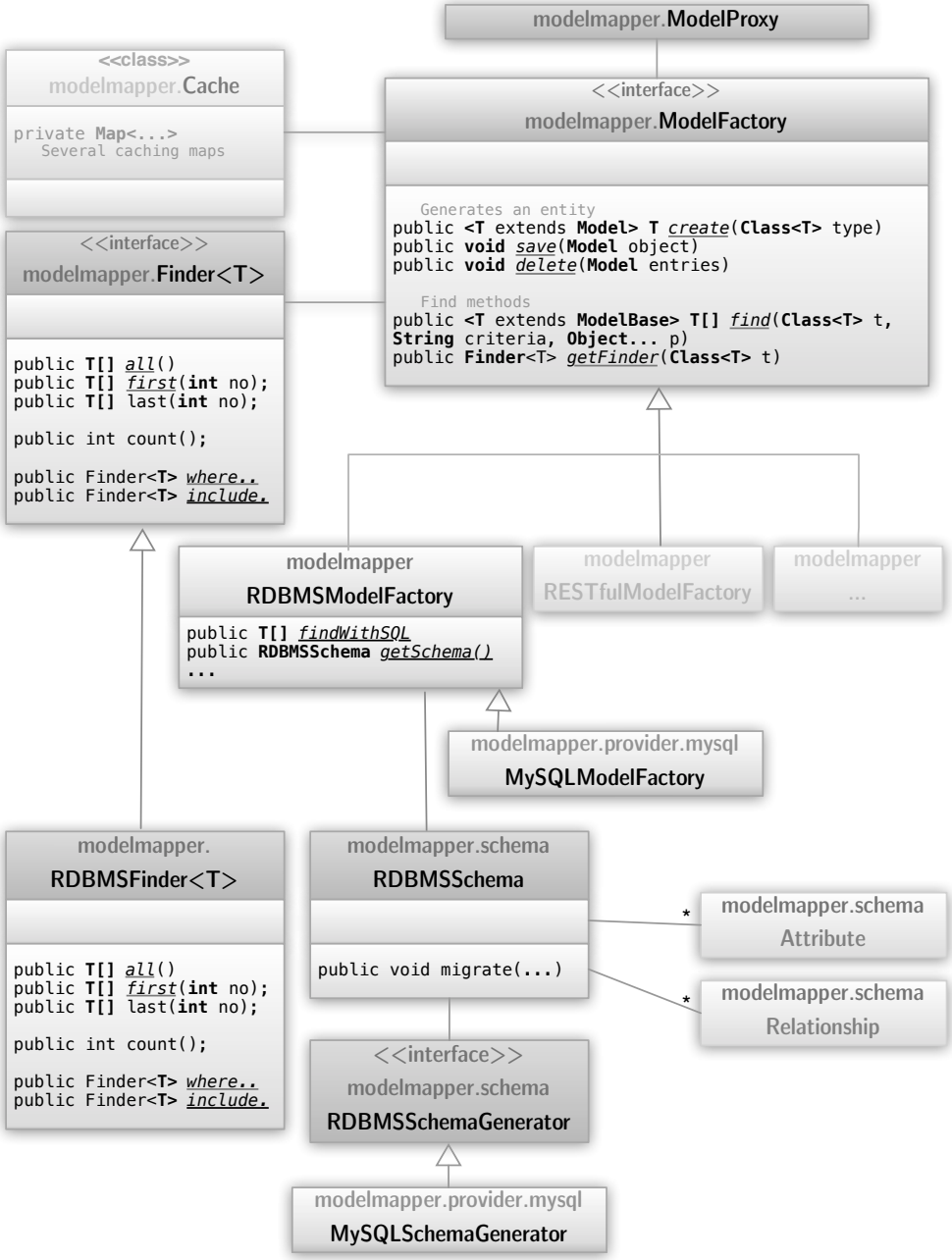
Nonostante lo sviluppo di questa classe ricalchi in modo significativo il design pattern *FactoryMethod* descritto nella sezione 3.3, la classe è impropriamente chiamata `factory`: essa infatti si occupa non solo della creazione delle istanze ma anche della ricerca, della cancellazione, del salvataggio e di altre funzioni minori come la *migration*, che verrà trattata più avanti.

Viene ora riportata la porzione di codice riguardante la creazione delle istanze proxy.

```
protected <T extends Model> T create(Class<T> type) {
  T back = null;
  try {
    back = (T) Proxy.newProxyInstance(type.getClassLoader(),
           new Class[] {type},
           getCache().getImplementation(type));
  } catch (Exception e) { ... }
  [..]
  return back;
}
```

Come si può vedere il metodo è parametrico rispetto al tipo della classe passata come argomento. Nel blocco `try-catch` viene istanziata la specifica sottoclasse di `ModelProxy`.

Il diagramma sottostante mostra le associazioni e le sottoclassi di `ModelFactory`.


modelmapper.ModelProxy
<<class>>
modelmapper.Cache
private Map<...>
Several caching maps
<<interface>>
modelmapper.ModelFactory
Generates an entity
public <T extends Model> T create(Class<T> type)
public void save(Model object)
public void delete(Model entries)
Find methods
public <T extends ModelBase> T[] find(Class<T> t, String criteria, Object... p)
public Finder<T> getFinder(Class<T> t)
<<interface>>
modelmapper.Finder<T>
public T[] all()
public T[] first(int no);
public T[] last(int no);
public int count();
public Finder<T> where..
public Finder<T> include.
modelmapper
RDBMSModelFactory
public T[] findWithSQL
public RDBMSSchema getSchema()
...
modelmapper
RESTfulModelFactory
modelmapper
...
modelmapper.provider.mysql
MySQLModelFactory
modelmapper.
RDBMSFinder<T>
public T[] all()
public T[] first(int no);
public T[] last(int no);
public int count();
public Finder<T> where..
public Finder<T> include.
modelmapper.schema
RDBMSSchema
public void migrate(...)
*
modelmapper.schema
Attribute
*
modelmapper.schema
Relationship
<<interface>>
modelmapper.schema
RDBMSSchemaGenerator
modelmapper.provider.mysql
MySQLSchemaGenerator

5.1.3 RDBMSModelFactory

`RDBMSModelFactory` è l'implementazione relazionale del factory. Questa classe si occupa di tutte le operazioni di traduzione tra la logica object-oriented dei modelli e quella relazionale delle tabelle.

É da sottolineare che non è questa classe a generare direttamente gli *statement* SQL, ma questo compito spetta alle varie implementazioni presenti nel package `modelmapper.provider`.

Tutte le informazioni sulla struttura delle tabelle, e le definizioni delle chiavi esterne necessarie per la navigazione tra gli oggetti, sono contenuti in un'istanza della classe `RDBMSSchema` presente nel factory, richiamabile grazie a `getSchema()`.

Invocando il metodo `migrate(Class... models)` su questo oggetto, vengono generati gli statement SQL per la creazione delle tabelle per il specifico database istanziato.

Di cruciale importanza nel framework è il fetch delle classi collegate da una `@Connection`, fondamentale per la navigazione tra gli oggetti. Come visto nella porzione di codice riportata sopra, per la classe `ModelProxy`, il metodo *protected* `getConnection` richiama il metodo `fetch` del factory, che si occupa di gestire il recupero dei record presenti sul database legati da una relazione con il modello.

```
protected Model[] fetch(Class toFetch, Model invokedBy,
modelmapper.annotation.Connection conn) {

Relationship r = getSchema().getNamedRelationship(conn.name());

String idA = getCache().getModelId(r.getClassA());
String idB = getCache().getModelId(r.getClassB());

//belongs to
if (conn.type().equals(ConnectionType.BelongsTo)) {

  String condition =
  r.getTableB() + "." + r.getFieldB() + "␣=␣" + r.getTableA() + "." + idA +
  "␣and␣" + r.getTableB() + "." + idB + "␣=␣?␣";

  return find(toFetch, condition,
              invokedBy.fields().get(idB));

//agreggation - composition
```

```
} else if (!conn.type().equals(ConnectionType.ManyToMany)) {

  String condition = [..]

  return find(toFetch, condition,
              invokedBy.fields().get(idA));

//many to many
} else {

  String condition = [..]

  return find(toFetch, condition,
              invokedBy.fields().get(idParam));
}

}
```

5.2 Implementazione dell'ereditarietà

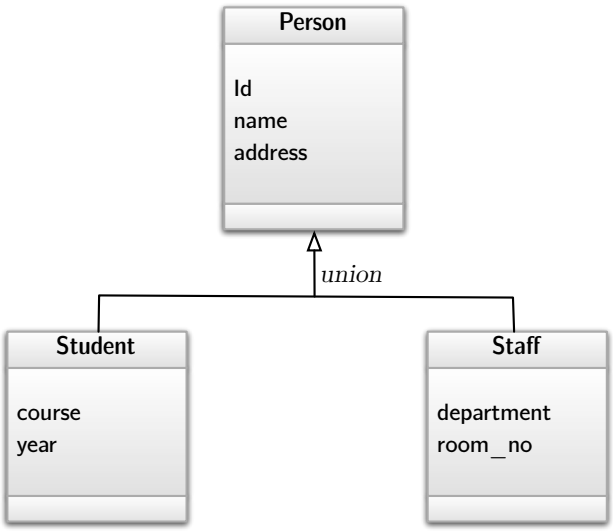
I modelli in ModelMapper sono definiti scrivendo delle comuni interfacce Java, le quali vantano ereditarietà multipla. Ne segue che ModelMapper supporta l'ereditarietà multipla tra i modelli, permettendo il polimorfismo a livello di linguaggio e di database.

L'implementazione dell'ereditarietà multipla è stata lungamente oggetto di discussione; i problemi sono legati sia alla definizione delle tabelle e dei vincoli nel modello relazionale, sia alla simulazione del meccanismo di ereditarietà multipla nel framework per effettuare il *dispatch* alla giusta sovraclasse.

Nel modello relazionale, l'implementazione dell'ereditarietà è definita usando vincoli di chiave e vincoli di integrità referenziale per simulare le relazioni tra sovraclassi e sottoclassi. Seguono tutte le principali strategie per implementare l'ereditarietà su basi di dati relazionali e object relational; gli esempi proposti sono composti da porzioni di `SQL` per la definizione dei dati in versione Oracle. Questi possono essere compatibili con altri database con pochi adattamenti.

5.2.1 Union Inheritance

Il seguente diagramma mostra un esempio di una relazione di *union inheritance*.



Ogni oggetto della sovraclasse è un oggetto di almeno una delle sottoclassi e questo non preclude il fatto che un membro di una sottoclasse possa anche essere membro di altre sottoclassi; nell'esempio sopra esposto una persona che è membro dello staff può anche essere uno studente.

Per simulare questo tipo di ereditarietà, `Student` e `Staff` presenti nell'esempio dovranno avere la chiave primaria della sovraclasse, `Person`, nelle loro tabelle relazionali; inoltre la chiave primaria della sovraclasse diventerà chiave primaria ed esterna della sottoclasse.

Il vincolo di integrità referenziale assicura che un elemento della sottoclasse sia sempre un elemento della sovraclasse.

Nelle basi di dati *Object Relational* solitamente questo tipo di ereditarietà viene espressa tramite la keyword `UNDER`.

Un esempio di implementazione relazionale della *union inheritance* è mostrata di seguito:

```
CREATE TABLE Person (
  id    VARCHAR2(10) NOT NULL,
  name    VARCHAR2(20),
  address   VARCHAR2(35),
  PRIMARY KEY(id));
```

```
CREATE TABLE Student (
  id    VARCHAR2(10) NOT NULL,
  course    VARCHAR2(10),
  year    VARCHAR2(4),
  PRIMARY KEY(id),
  FOREIGN KEY(id) REFERENCES Person ON DELETE CASCADE);

CREATE TABLE Staff (
  id    VARCHAR2(10) NOT NULL,
  department  VARCHAR2(10),
  room_no   VARCHAR2(4),
  PRIMARY KEY(id),
  FOREIGN KEY(id) REFERENCES Person ON DELETE CASCADE);
```

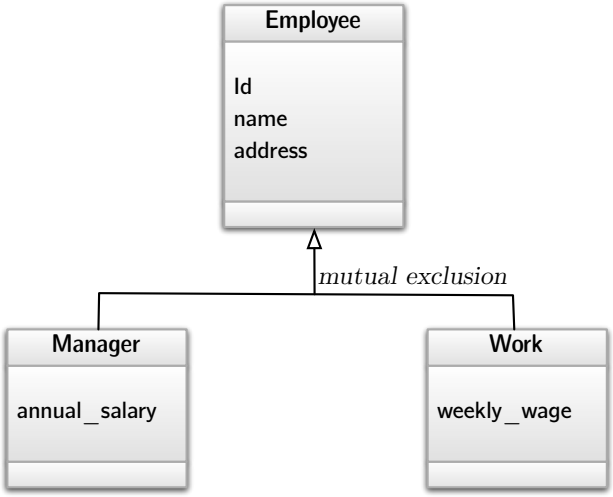
Utilizzando feature object-relational la descrizione delle tabelle risulta invece:

```
CREATE OR REPLACE TYPE Person_T AS OBJECT (
  id    VARCHAR2(10),
  name    VARCHAR2(20),
  address   VARCHAR2(35)) NOT FINAL

CREATE TABLE Person OF Person_T (
  id    NOT NULL,
  PRIMARY KEY(id));

CREATE OR REPLACE TYPE Student_T UNDER Person_T (
  course    VARCHAR2(10),
  year    VARCHAR2(4))

CREATE TABLE Student OF Student_T (
  id    NOT NULL,
  PRIMARY KEY(id));

CREATE OR REPLACE TYPE Staff_T UNDER Person_T (
  department  VARCHAR2(10),
  room_no   VARCHAR(4))

CREATE TABLE Staff OF Staff_T (
```

```
  id    NOT NULL,
  PRIMARY KEY(id));
```

5.2.2 Mutual-Exclusion Inheritance

Una relazione di ereditarietà mutuamente esclusiva definisce un gruppo di sottoclassi disgiunte. Un esempio può essere quello presentato nel seguente schema:



Questo esempio descrive l'impossibilità di un manager di essere un lavoratore, e vice versa. È da sottolineare che questa relazione non è totale, ma parziale; ci può essere cioè un impiegato che non è rappresentato da nessuna delle due sottoclassi.

Il miglior modo per gestire l'ereditarietà con mutua esclusione senza perdere la semantica della relazione è quello di aggiungere alla sovraclasse un attributo che rispecchi il tipo della sottoclasse.

Nella tabella `Employee` verrà quindi aggiunto un attributo `emp_type` che potrà assumere il valore `manager`, `worker` o `null`.

Vengono presentati di seguito i dettagli implementativi.

```
CREATE TABLE Employee (
  id    VARCHAR2(10) NOT NULL,
  name    VARCHAR2(20),
  address   VARCHAR2(35),
  emp_type  VARCHAR2(8)
  CHECK(emp_type IN ('Manager', 'Worker', NULL))
```

```
  PRIMARY KEY(id));

CREATE TABLE Manager (
  id    VARCHAR2(10) NOT NULL,
  annual_salary NUMBER,
  PRIMARY KEY(id),
  FOREIGN KEY(id) REFERENCES Employee(id) ON DELETE CASCADE);

CREATE TABLE Worker (
  id    VARCHAR2(10) NOT NULL,
  weekly_wage NUMBER,
  PRIMARY KEY(id),
  FOREIGN KEY(id) REFENCES Employee(id) ONE DELETE CASCADE);
```

La descrizione object-relational risulta essere:

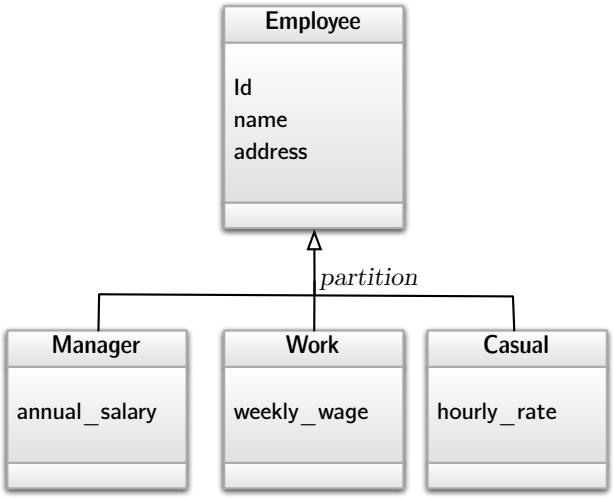
```
CREATE OR REPLACE TYPE Employee_T AS OBJECT (
  id    VARCHAR2(10),
  name    VARCHAR2(20),
  address   VARCHAR2(35),
  emp_type  VARCHAR(8)) NOT FINAL

CREATE TABLE Employee OF Employee_T (
  id    NOT NULL,
  emp_type  CHECK(emp_type in ('Manager', 'Worker', NULL))
  PRIMARY KEY(id));

CREATE OR REPLACE TYPE Manager_T UNDER Employee_T (
  annual_salary NUMBER)

CREATE OR REPLACE TYPE Worker_T UNDER Employee_T (
  weekly_wage NUMBER)
```

In quest'ultima implementazione, viene creata solo la tabella per la sovraclasse. Non abbiamo bisogno delle tabelle per le sottoclassi in quanto un oggetto può essere membro di una sola sottoclasse.

5.2.3 Partition Inheritance

La *partition inheritance* è definita come un gruppo di sottoclassi che partizionano la sovraclasse. Una partizione richiede che gli insiemi che ne prendono parte siano disgiunti, e che la loro unione sia uguale all'insieme partizionato. È quindi una relazione totale ed esclusiva, ovvero una combinazione delle due *inheritance* trattate precedentemente.

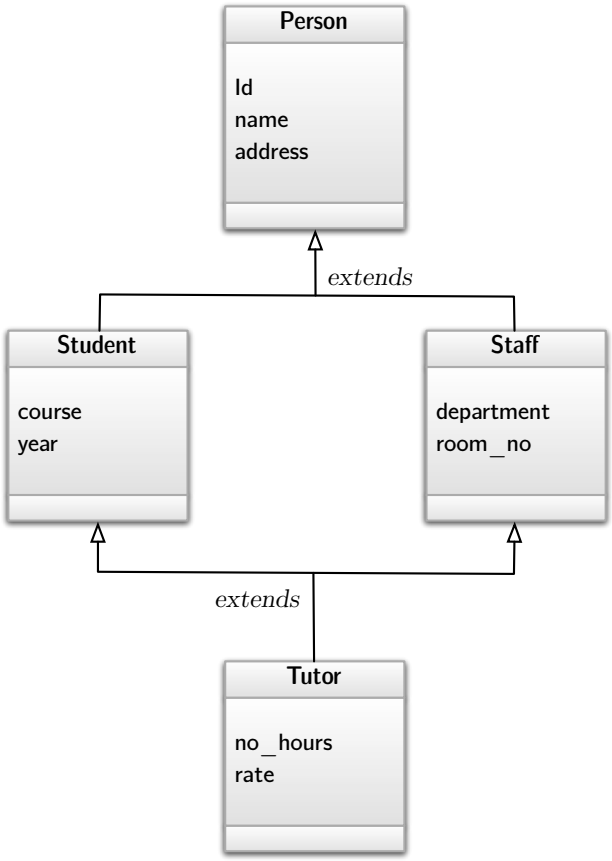
Riprendiamo l'esempio di prima, e aggiungiamo una sottoclasse `Casual`; si assume che ogni membro della classe `Employee` possa appartenere ad un solo membro della partizione.



Come nella mutua esclusione, viene aggiunto un attributo `emp_type` alla tabella della sovraclasse; a questo viene però aggiunto il vincolo `NOT NULL`. Questo assicura che ogni oggetto appartenga ad una ed una sola sottoclasse.

5.2.4 Multiple Inheritance

L'ultima relazione analizzata è l'ereditarietà multipla.



Come si può vedere dal esempio riassunto dal diagrama qui sopra, si può dire che un `Tutor` è sia uno studente che un membro dello staff.

Il miglior modo per gestire questa relazione è l'utilizzare una tabella per ogni classe ed tenere un chiave esterna per ogni sovraclasse.

Nel database object-relational analizzato la keyword `UNDER` è riservata all'eredità singola, quindi si è obbligati a scegliere una classe padre e simulare l'ereditarietà multipla tramite associazioni.

Segue la descrizione relazionale dell'esempio sopra illustrato.

```
CREATE TABLE Person (
  id    VARCHAR2(10) NOT NULL,
```

```
  name    VARCHAR2(20),
  address   VARCHAR2(35),
  PRIMARY KEY(id));

CREATE TABLE Student (
  id    VARCHAR2(10) NOT NULL,
  course    VARCHAR2(10),
  year    VARCHAR2(4),
  PRIMARY KEY(id),
  FOREIGN KEY(id) REFERENCES Person(id) ON DELETE CASCADE);

CREATE TABLE Staff (
  id    VARCHAR2(10) NOT NULL,
  department  VARCHAR2(10),
  room_no   VARCHAR2(4)
  PRIMARY KEY(id),
  FOREIGN KEY(id) REFERENCES Person(id) ON DELETE CASCADE);

CREATE TABLE Tutor (
  id    VARCHAR2(10) NOT NULL,
  no_hours  NUMBER,
  rate    NUMBER,
  PRIMARY KEY(id),
  FOREIGN KEY(id) REFERENCES Person(id) ON DELETE CASCADE);
```

ModelMapper gestisce l'ereditarietà multipla per i Relational DBMS proprio in questa maniera. La classe `modelmapper.schema.RDBMSSchema` si occupa di generare lo schema partendo dalle interfacce definite dall'utente, e nel caso di ereditarietà multipla lo schema viene creato seguendo la strategia sopra descritta.

Questo permette di mantenere la semantica dell'ereditarietà multipla nella base di dati, e grazie ad un meccanismo di *dispatching* delle chiamate viene garantita l'esecuzione

Capitolo 6

Conclusione

6.1 Perchè un altro ORM?

TODO

L'intero mercato degli ORM Java è dominato dal framework di JBoss, Hibernate. Questo è talmente utilizzato da essere stato recentemente portato anche su piattaforma .NET (NHibernate). Quindi, se esiste già un framework così tanto utilizzato e maturo da soddisfare tutti gli use-case più importanti, perché crearne un altro?

La risposta è da ricercarsi nella complessità di questo ORM. Hibernate è estremamente complesso, e sicuramente ciò è dovuto alla sua versatilità, ma molto spesso gli use-case sono piuttosto lineari e non richiedono tutte le possibilità di configurazione offerte dal framework. ModelMapper è stato progettato per essere il più possibile di semplice utilizzo, con la minima configurazione possibile. Infatti il framework non ha alcun file di configurazione, ma il mapping è dedotto direttamente dalle convenzioni di scrittura dei metodi, e da poche semplici annotazioni.

Attualmente l'industria è sempre più ai framework RAD (*Rapid Application Development*) basati sulla filosofia Convention over Configuration, e ModelMapper cerca di seguire questo principio il più possibile.

6.2 Vantaggi e svantaggi rispetto JPA/Hibernate/JMaestrale

TODO