GUIDA PER LO SVILUPPATORE

Regola kit

Autori:
Nicola Santi

8 maggio 2008

Indice

Prefazione

```
1 La societ aperta e aperta a piu valori, a piu visioni del mondo
  filosofiche e a piu fedi religiose, ad una molteplicita di proposte
  per la soluzione di problemi concreti e alla maggior quantita di
  critica. La societa aperta e aperta al maggior numero possibile di
  idee e ideali differenti, e magari contrastanti. Ma, pena la sua
  autodissoluzione, non di tutti: la societ aperta e chiusa solo agli
  intolleranti.
  (Karl R. Popper, La societa aperta e i suoi nemici, Vol. I, Platone
  totalitario, dalla IV di copertina.)
```

Non credo sia necessario abbracciare in toto la posizione filosofica nota come Relativismo per cogliere di quanto varino costumi e valori anche solo viaggiando nel spazio oggi: ogni società è unica, ogni costume trova giustificazione nel contesto in cui si è formato ed in questa diversità e di questa diversità si alimenta e prolifera il sapere umano. Se poi si azzardassero dei viaggi non solo nello spazio ma nelle profondità del tempo il nostro stupore si amplierebbe risalendo controcorrente il fiume dei secoli secoli. Il sistema di valori che assegna un giudizio positivo di un condottiero militare quale Alessandro Magno come avrebbe giudicato il Mahatma Gandhi? Quale sono le imprese per cui un padre (poté e possa) essere orgoglioso dei propri figli? Quale il modo etico di legiferare ed il concetto stesso di qualità di una casa od una vita intera come si è modificato attraversando anni di evoluzioni? Quale risposte possiate proporre a queste domande tutte dovranno, se non spiegare, scendere a compromessi con la variazione, la moda, il prendere atto che è tradizione che le cose cambino: è il panta rei di Eraclito, il tema eterno del divenire che procede spesso gradualmente mentre a volte in modo netto, traumatico. Ed è proprio lungo questa linea evanescente e discontinua che separa il nuovo corso (ancora impreciso e difficile da definire) dalla vecchia maniera (ancora presente ma senza più i favori del Tempo) destinata a scomparire; lungo questa linea che si dispongono schieramenti di persone in antitesi, a volte in conflitto aperto, cercando di lottare per quello in cui credono. Le une, come le altre senza alcuna garanzia di avere ragione e senza sfere di cristallo che li rinsaldino nelle loro convinzioni: solo passione, l'illuminisme ed il sentimento. Poca cosa, si dirà però è quello che è dato al genere umano ed è quanto basta in mano ad uomini e donne di valore. Un esempio di questo genere di confronto si è avuto nel XIV secolo all'interno del mondo letterario nella penisola italica tra lingua latina e lingua volgare. Un intellettuale fiorentino destinato ad una fama mondiale immensa chiamato Dante Alighieri partecipò allo scontro e scrisse un saggio a titolo de vulgari eloquentia. Il testo curiosamente è scritto in latino (perché destinato a dotti ed intellettuali dell'epoca) ma sostiene l'importanza e la dignità della lingua volgare proprio contro il latino (ed il greco). Ecco un passaggio tradotto in volgare del primo libro:

I-i-4 Di queste due lingue la più nobile è la volgare: intanto perché è stata adoperata per prima dal genere umano; poi perché il mondo intero ne

fruisce, benché sia differenziata in vocaboli e pronunce diverse; infine per il fatto che ci è naturale, mentre l'altra è, piuttosto, artificiale.

Una difesa delle lingue apprese in modo naturale, dalle nostre madri, quando ancora bambini muoviamo i primi passi nel mondo ed impariamo a chiamare le cose con il nome che gli uomini hanno attribuito loro. La lingua che utilizziamo tutti i giorni nelle miserie del quotidiano ma anche la lingua con cui ci dichiariamo ai nostri amati, con cui consoliamo i nostri figli e con cui cerchiamo di riempire il silenzio dell'infinito stellato. La lingua che l'Alighieri avrebbe usato poi per scrivere i suoi sonetti e la sua Commedia dimostrando così come il volgare non avesse nulla da invidiare all'accademico latino in quanto ad espressività, completezza: una lingua aulica, forgiata dal basso ed utilizzata da gente comune così come da poeti e letterati per il lavoro quotidiano e la più alta poesia.

Nello sviluppo in Java si è presentata una situazione per diversi tratti simile al panorama letterario italiano del XIV secolo. Fino al 1998 (ed oltre) si utilizzava Java seguendo l'imperante tradizione della progettazione ad oggetti; costrutti linguistici e tecniche come l'ereditarietà ed il polimorfismo venivano utilizzate per affrontare e risolvere in modo eleganti i problemi affrontati. Ogni intervento era realizzato nella cornice teorica di principi come l'Open/Closed, il principio di inversione delle dipendenze, il principio di sostituzione di Liskov ad altri ancora oggi del tutto validi ed accreditati. Con l'esplosione poi del movimento dei design pattern (come riferimento Gof del 1995) la scrittura raggiunse livelli di raffinatezza elevatissimi con architetture complesse ma intuitive e facili da mantenere ed espandere; inoltre la capacità di descrivere problemi e soluzioni in modo universalmente accettato ampliarono la banda di comunicazione tra i le persone coinvolte nello sviluppo in modo inaudito. Annotare una classe come implementazione, ad esempio, del pattern Facade consentiva a chiunque di comprendere il problema affrontato (dare una vista semplificata di un sottosistema complesso) e la soluzione messa in piedi (una classe ritagliata sulle esigenze dei client). Si andava diffondendo un modo di utilizzare Java come linguaggio di pattern e questo consentì di progettare soluzioni incredibili e di spostare il discorso dell'analisi fino ai confini stessi della progettazione ad oggetti, cogliendone i limiti e proponendo soluzioni complementari note poi come progettazione orientata agli aspetti. Proprio in questo contesto di ricerca febbrile e risultati poderosi che nel 1998 calano dall'alto delle torri di Sun (l'azienda che Java aveva inventato pochi anni prima) delle specifiche contenenti le applicazioni serie, quelle destinate alle aziende che si propongono sfide impegnative sia come scalabilità che come manutenzione. Queste specifiche presero il nome di J2EE (ora si chiamano JEE) ed arrivarono come le tavole di pietra della legge di Mosè: ovvero contenevano la verità ed il giusto modo. Tutto quello che si era fatto e si faceva doveva cessare, mutare, adattarsi perché J2EE era la sola via corretta per affrontare problemi complessi. A distanza di 1700 anni si ripresentava aulica ed imposto dall'autorità la scrittura

latina: ancora una volta artificiale, ancora una volta appannaggio di pochi intellettuali, ancora una volta scelta imposta. In verità, mentre nei corsi universitari fiorivano corsi dedicati a J2EE, nella pratica quotidiana il numero di applicazioni che lo utilizzava rappresentava un minoranza quasi esoterica. Infatti in pochi erano disposti ad accettare (o permettersi finanziariamente) i limiti che questo stack tecnologico imponeva; ad esempio non è mai stato possibile trovare un buon motivo per rinunciare all'ereditarietà. Così come è sempre parso innaturale la segregazione del modello all'interno dei confini degli EJB server senza poterli passare ai livelli di presentazione se non sotto forma di anemici delegati a nome DTO. Inoltre mentre i tempi di sviluppo salivano a causa delle lunghe attese per i riavvii degli application server si constatava con amarezza come J2EE rendesse difficile se non impossibile una metodologia di sviluppo nota come Test Driven Developement che aveva dato invece buona prova di sé nella creazione di applicazioni robuste. A causa di questi, ed altri motivi, apparve di nuovo la linea effimera di separazione tra fazioni apposte e lo schieramento della lingua volgare fece il suo ritorno. Anche questa volta si presentava come un linguaggio naturale, costruito dal basso per risolvere problemi concreti ma capace di accettare e vincere le sfide più complesse delle applicazioni aziendali. Anche questa volta intellettuali capaci si schierarono in difesa del volgare. Tra questi Rod Johson che fornì gli strumenti per affrontare in modo elegante, efficace, sistematico ed economicamente vantaggioso le sfide offerte dalle applicazioni aziendali progettando e rilasciando come open source il framework chiamato Spring. La scelta dell'open source è stata fondamentale per il successo di Spring in quanto ha consentito di integrare assieme tutte le soluzioni che la comunità aveva realizzato in quegli anni come risposta a J2EE tra cui ricordo solo Hibernate ed iBatis ma potrei parlare delle librerie commons, struts, tomcat e molti altri. In contrasto con J2EE Spring si fece chiamare contenitore leggero (o invertito) ed i mattoni di cui si componeva lo sviluppo classi Pojo, ovvero Plain Old Java Object per rimarcarne la semplicità e la tradizione legata alla progettazione ad oggetti. Il successo di mercato di Spring fu velocissimo ed imponente: applicazioni finanziarie, aeroportuali, governative, mediche, bancarie furono realizzate in tutto il mondo dimostrando l'efficacia del modello proposto. Il riscontro fu tale che la Sun stessa decise (tardivamente e parzialmente) di riadattare J2EE sottoponendolo ad una clamorosa inversione di orientamento architetturale per spingerlo ad abbracciare temi come Pojo Programming ed iniezione delle dipendenze. La portata del dietro front fu tale che anche il nome cambio in JEE e tutto lo stack tecnologico ufficiale iniziò il percorso che Spring aveva inaugurato molti anni prima e che forse un giorno poterà i due antagonisti a somigliarsi al punto dal diventare di fatto la stessa cosa. Ma questo solo quando JEE recupererà il tempo perso e solo se Spring dovesse arrestare la sua evoluzione.

Capitolo 1

Getting Started

Questo capitolo è una cura per gli impazienti; seguendo le istruzioni dei paragrafi seguenti sarete in grado di installare e predisporre una prima applicazioni web con Regola.

1.1 Installare Regola kit

Le applicazioni realizzate con Regola kit utilizzano Maven 2 per tutta la gestione del ciclo di build (compilazione, esecuzione dei test, creazione dei file war, ...). Quindi come prima cosa bisogna installare Maven 2 scaricandolo dal sito maven.apache.org e seguendo le istruzioni.

Finita l'installazione di Maven 2 verificate che tutto sia andato a buon fine aprendo una console di terminale e lanciando il seguente comando.

```
nicola@casper:~# mvn -version
Maven version: 2.0.8
Java version: 1.6.0_03
OS name: "linux" version: "2.6.22-14-generic" arch: "i386" Family:
"unix"
```

Se tutto è corretto Maven 2 risponde al comando restituendo la sua versione, quella di Java ed infine alcune informazioni circa il sistema operativo in uso.

Regola kit non richiede nessuna installazione particolare (anche se è possibile scaricare un pacchetto contente documentazione e comandi di utilità): quindi finita l'installazione di Maven 2 siete pronti già per utilizzare Regola kit.

■ Per maggiori informazioni su come installare Regola kit sulle vostre macchine di sviluppo si rimanda al capitolo 2 nella pagina 17. ■

1.2 Predisporre il database

Per questo tutorial ipotizziamo di avere a disposizione un database di tipo MySql già installato sulla macchina dove intendiamo creare il progetto. Assicuratevi che il database stia funzionando e digitate il comando seguente:

```
nicola@casper:~# mysql -u root -p
```

Vi troverete dentro la shell di amministrazione di MySql. Approfittatene per creare un nuovo database che sarà utilizzato dall'applicazione digitando il comando.

```
1 mysql> create database clienti;
```

(Attenzione al punto e virgola in fondo al comando). A questo punto non ci resta che creare anche l'utente utilizzato dalla nostra applicazione per accedere al database (nell'esempio porta il mio nome *nicola*).

```
1 mysql> grant all on clienti.* to 'nicola'@'localhost';
```

Bene, con il database abbiamo finito. Digitate questo ultimo comando per uscire dalla shell di amministrazione di MySql e passare al paragrafo seguente.

```
1 mysql> exit
```

■ Regola kit è in grado di utilizzare diversi DBMS (ad esempio Oracle, Microsoft Sql, PostgreSQL, Hypersonic, ...). Per sapere come configurare la vostra applicazione per utilizzare DBMS diversi da MySql si rimanda al capitolo 5 nella pagina 27. ■

1.3 Creare un progetto con Regola kit

Posizionatevi nella cartella dove volete creare il vostro nuovo progetto e digitate un comando simile al seguente con l'accortezza però di modificare il parametro gruopId (nell'esempio *com.acme*) con il nome del vostro package di default ed il parametro artifactId (nell'esempio *clienti*) con il nome del nuovo progetto.

```
1 nicola@casper:~# mvn archetype:create -DarchetypeGroupId=org.regola \
-DarchetypeArtifactId=regola-jsf-archetype \
3 -DarchetypeVersion=1.1-SNAPSHOT -DgroupId=com.acme \
-DartifactId=clienti
```

Attenzione: il comando qui sopra è riparito su diverse righe per chiarezza tipografica, deve invece essere digitato su una sola riga.

La prima volta che lanciate questo comando Maven 2 scarica tutte le librerie necessarie (la cosa potrebbe prendere un po' di tempo) e crea una sotto cartella col nome del progetto (nell'esempio la cartella clienti). Il progetto è questo punto è già stato creato, posizioniatevi all'interno della cartella *clienti* col comando:

```
nicola@casper:~# cd clienti
```

1.4 Collegarsi al database

Prima di lanciare la nostra nuova applicazione è necessario informarla circa le coordinate del database da utilizzare, per farlo bisogna apportare un modifica al file src/test/resources/jetty/env.xml con il vostro editor di testo preferito. Dovete inserire cambiare solo il nome del database (alla riga 6) e lo username (riga 8) da utilizzare per ottenere qualcosa di simile al frammento di xml seguente:

```
1 ...
<New id="jira-ds" class="org.mortbay.jetty.plus.naming.Resource">
3   <Arg>jdbc/Datasource</Arg>
    <Arg>
5     <New
      class="org.enhydra.jdbc.standard.StandardConnectionPoolDataSource">
        <Set name="Url">jdbc:mysql://localhost/clienti</Set>
7       <Set name="DriverName">com.mysql.jdbc.Driver</Set>
        <Set name="User">nicola</Set>
9     </New>
    </Arg>
11 </New>
...
```

■ Per avere maggiori informazioni sulle diverse configurazioni relative alle connessioni al database si rimanda al capitolo 4 ■

1.5 Avviare l'applicazione

Ora tutto è pronto per avviare l'applicazione, se avete lasciato la cartella principale del progetto tornateci e da lì lanciate il comando seguente (e lasciate aperta la console):

```
nicola@casper:~/projects/clienti# mvn jetty:run
```

Sullo schermo si susseguiranno diverse righe per informarvi che l'applicazione è stata inizializzata e quando, infine, apparirà la dicitura *Started Jetty Server* saprete che tutto è pronto.

Lasciando sempre aperta la console aprite un'istanza del vostro browser e collegatevi all'indirizzo localhost:8080/clienti per vedere la pagina di benvenuto della vostra applicazione.

Complimenti, avete appena fatto il primo passo nel mondo delle applicazioni Regola kit!

■ Per visualizzare l'applicazione state utilizzando un piccolo (ma molto completo) application server di nome Jetty. Per la messa in produzione però si consiglia di utilizzare dei container diversi (ad esempio Tomcat o JBoss). Per imparare a come creare i pacchetti per questi application server si rimanda al capitolo 6 nella pagina 29 ■

1.6 Struttura di un progetto Regola kit

La struttura della cartella di un progetto Regola kit si impronta alla struttra standard di un progetto web di Maven 2. Al primo livello troviamo:

pom.xml	il file di configurazione di Maven 2
src/	la cartella dei sorgenti
target/	contiene i file compilati ed i pacchetti per le consegne

La cartella target contiene quanto i pacchetti pronti per la consegna con la classi compilate ed i descrittori. Si tratta di una cartella il cui contenuto è ricreato ogni volta si lanci il comando:

```
1 nicola@casper:~/projects/clienti# mvn package
```

La cartella src contiene i sorgenti (html, js, css e java) dell'applicazione. Al suo interno potete trovare:

main/	contiene i sorgenti dell'appplicazione
test/	contiene i sorgenti dei test
site/	reportistica generata da Maven 2

La cartella main e la cartella test contengono entrambe i sorgenti java (nella sottocartella java) e le altre risorse (nella cartella risorse). Queste ultime sono i file di configurazione, i mappaggi orm e, in generale, tutto quello che non sono sorgenti Java ma devono finire comunque nel classpath. La differenza tra la cartella main e quella test e che il contenuto di quest'ultima non finisce mai nei pacchetti destinati alla produzione ma è usato esclusivamente per l'esecuzione dei test. Infine la cartella main contiene anche la sottocartella webapp dove si trova la webroot, ovvero le pagine web dell'applicazione ed il file web.xml.

■ Per una descrizione completa dei file e delle cartelle standard di un progetto Regola kit si rimanda al capitolo 3 nella pagina 19 ■

1.7 Persistenza su database

Regola kit abbraccia una metodologia di sviluppo incentrata sull'analisi del dominio del problema (Domain Driven Development) per cui il primo passo è quello di creare le classi di modello. Spesso queste classi sono persistite sul database per cui si potrebbe iniziare scrivendo la classe e poi creando la corrispondente tabella sul database. Oppure, al contrario come faremo tra poco, creando prima la tabella del database e facendoci poi creare in automatico la classe Java (nel prossimo paragrafo 1.9 nella pagina 14). Colleghiamoci nuovamente al database clienti.

```
1 nicola@casper:~/projects/clienti# mysql -u root -p clienti
```

Creiamo una piccola tabella con la sola chiave primaria (id) ed un campo di descrizione (label).

```
mysql> create table prodotti (id int(11) not null auto_increment,
label varchar(80) not null, primary key (id) );
mysql> describe prodotti;
+-------+-------------+------+-----+---------+----------------+
| Field | Type        | Null | Key | Default | Extra          |
+-------+-------------+------+-----+---------+----------------+
| id    | int(11)     | NO   | PRI | NULL    | auto_increment |
| label | varchar(80) | NO   |     |         |                |
+-------+-------------+------+-----+---------+----------------+
2 rows in set (0.02 sec)
```

Inseriamo qualche dato nella tabella, ad esempio alcune descrizioni di esempio per verificare poi il funzionamento dell'applicazione.

```
mysql> insert into prodotti values (null, 'book');
Query OK, 1 row affected (0.05 sec)

mysql> insert into prodotti values (null, 'bottle');
Query OK, 1 row affected (0.00 sec)

mysql> insert into prodotti values (null, 'paper');
Query OK, 1 row affected (0.00 sec)

mysql> select * from prodotti;
+----+--------+
| id | label  |
+----+--------+
|  1 | book   |
|  2 | bottle |
|  3 | paper  |
+----+--------+
3 rows in set (0.01 sec)
```

Adesso il database contiene una tabella con dei dati che possiamo usare per persistere le nostre classi di modello. Usciamo dal database e torniamo all'applicazione.

```
mysql> exit
```

1.8 (Ri)collegarsi al database

Al paragrafo 1.4 nella pagina 11 abbiamo configurato l'applicazione per utilizzare il nostro database in fase di esecuzione. Adesso dobbiamo fare in modo che anche in fase di sviluppo si possa accedere al database (ad esempio per usare i generatori di codice o lanciare la batteria di test). Il file da modificare è src/test/resources/designtime.properties e deve essere aggiornato in modo da contenere lo username, la password ed il nome del database. Il risultato finale deve risultare simile al seguente:

```
...
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class = com.mysql.jdbc.Driver
hibernate.connection.url = jdbc:mysql://localhost/clienti
hibernate.connection.username = nicola
```

```
hibernate.connection.password =
...
```

1.9 Classi di modello

Siete ora in grado di scrivere la classe di modello che sarà persistita sulla tabella prodotti... oppure potete farvela generare automaticamente e poi modificare convenientemente le classi prodotte. Userete gli Hibernate Tools che sono già configurati all'interno delle applicazioni prodotte con Regola kit e trovano nell'unico file src/test/resources/hibernate.reveng.xml la configurazione di tutto il processo di generazione inversa, a partire cioè dal database. Specificate il nome della tabella da cui partire (*prodotti* alla riga 2), il nome della classe (*Prodotto*, al singolare e con la prima lettera maiuscola nella riga 4), il package da utilizzare (*com.acme.model* sempre alla riga 2).

```
1 ...
  <table-filter match-name="prodotti"  package="com.acme.model"
  exclude="false"/>
3
   <table name="prodotti" class="Prodotto" >
5     <primary-key property="id" />
   </table>
7 ...
```

Ora avviate la generazione: posizionantevi nella cartella principale del vostro progetto ed utilizzate il plugin di Maven 2 Hibernate3 che consente di generare le classi java (il goal hbm2java), i file di mappaggio di hibernate (hbm2hbmxml) e la configurazione generale di hibernate (hbm2cfgxml).

```
1 nicola@casper:~/projects/clienti# mvn hibernate3:hbm2java
  hibernate3:hbm2hbmxml hibernate3:hbm2cfgxml
```

Il primo file generato si trova nella posizione src/main/java/com/acme/-model/Prodotto.java e contiene la classe Java:

```
1 public class Prodotto  implements java.io.Serializable {
3     private Integer id;
5     public Prodotto() {
      }
7
      public Integer getId() {
9         return this.id;
      }
11
      public void setId(Integer id) {
13        this.id = id;
      }
15 }
```

Poi è stato creato il file con i mappaggi di hibernate src/main/resource-s/com/acme/model/Prodotto.hbm.xml, molto semplice in questo caso:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD
 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Generated 14-apr-2008 12.23.38 by Hibernate Tools 3.2.0.CR1 -->
<hibernate-mapping>
    <class name="com.acme.model.Prodotto" table="prodotti"
    catalog="clienti">
        <id name="id" type="java.lang.Integer">
            <column name="id" />
            <generator class="identity" />
        </id>
    </class>
</hibernate-mapping>
```

Ed infine è stato inserito un riferimento a quest'ultimo file di mappaggio dentro la configuraizone principale di Hibernate src/main/resources/hibernate.cfg.xml (alla riga 13).

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property
        name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="hibernate.connection.password"></property>
        <property
        name="hibernate.connection.url">jdbc:mysql://localhost/clienti</property>
        <property
        name="hibernate.connection.username">nicola</property>
        <property
        name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

       <mapping resource="com/acme/model/Prodotti.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

Attenzione: il goal hibernate3:hbm2cfgxml cancella e riscrive ogni volta questo file ed inoltre vi aggiunge delle configurazioni che a runtime non sono usate (come username, password, url e driver class). Nell'impiego di tutti i giorni di Regola kit il nostro team di sviluppo non utilizza il goal hibernate3:hbm2cfgxml e si occupa di aggiungere manualmente i mappaggi delle risorse. Naturalmente le configurazioni non usate non costituiscono problema, per cui alla fine la scelta di impiegare o meno hibernate3:hbm2cfgxml è lasciata alla vostra discrezione.

■ Esistono altri goal disponibili, ad esempio la generazione degli script sql per creare le tabella a partire dalla configurazione delle classi. Si rimanda, per approfondimenti, al capitolo 5 nella pagina 27. ■

1.10 Dal modello alla presentazione

Adesso che avete creato la classe di modello è il momento di realizzare il codice per leggere e scrivere oggetti (della classe Prodotto) sul database,

le pagine web che elenchino questi oggetti così come la pagine di dettaglio per effettuare modifiche. Questo codice può essere scritto a mano oppure potete partire facendovi generare automaticamente della classi di default che utilizzerete come modello di partenza per le vostre modifiche.

Per avviare il generatore di Regola kit utilizzate il seguente comando:

```
1 nicola@casper:~/projects/clienti# mvn exec:java -Dexec.args="-c
com.acme.model.Prodotto -m"
```

Noterete tra i parametri passati al comando il nome della classe attorno a cui costruire i vari livelli e l'opzione *m* che specifica di utilizzare un'ampia catena di generatori, in particolare:

dao	produce il custom dao
modelPattern	produce la classe necessaria a Model Pattern
properties	aggiunge le chiavi per la localizzazione
list-handler	genera il controller dietro la pagina di lista
list	genera la pagina di lista
form-handler	genera il controller dietro la pagina di dettaglio
form	genera la pagina di dettaglio

■ I generatori possono anche essere avviati individualmente. Per scoprire come e conoscere anche altri generatori forniti con Regola kit si rimanda al capitolo 7.4 nella pagina 35. ■

Capitolo 2

Installazione

2.1 Maven 2

TODO: Qui si spiega che Regola kit si fonda su Maven 2 per tutta la gestione del ciclo di build.

2.2 Librerie

TODO: Dove si elencano i moduli di cui si compone

2.3 Dipendenze

TODO: Dove si elencano le dipendenze, ricordando però che sono gestite da Maven 2

2.4 Eclipse IDE

TODO: Come lavorare ad un progetto di Regola utilizzando Eclipse 3.3 ed i plugin necessari per il funzionamento

Capitolo 3

Struttura di un progetto

Questo capitolo funziona un po' come una cartina stradale e rimanda ad altri capitoli per l'approfondimento.

/	pom.xml	il file principale di Maven 2
src/main/resources/	runtime.properties	proprietà di runtime
	log4j.xml	la verbosità dei log
	hibernate.cfg.xml	la configurazione principale di Hibernate
	ApplicationResources.properties	i testi tradotti nelle varie lingue
	applicationContext-*.xml	le configurazioni di Spring
	<stesse cartelle dei packages>	i mappaggi di Hibernate
src/test/resources/	designtime.properties	proprietà a design time
	log4j.xml	la verbosità dei log
	hibernate.reveng.xml	Hibernate tools
	applicationContext-*-test.xml	le configurazioni di Spring
	jetty/env.xml	datasource per Jetty

3.1 Lo standard Maven 2

Fra i tanti vantaggi offerti di cui si può beneficiare utilizzando Maven 2 vi sono quelli derivanti dall'adesione ad uno standard (che i teorici dei giochi definiscono equilibrio di Nash); infatti disponendo i diversi file di un progetto in modo convenuto consente agli sviluppatori (e i sistemisiti) di orientarsi da subito su un progetto estraneo, a sistemi di supporto alla scrittura di trovare senza configurazione le risorse di cui abbisognano ed in generale consentono a soggetti diversi e lontani (nello spazio e nel tempo) di cooperare sullo stesso progetto senza conflitti.

La struttura esatta di progetto web è reperibile su (mav), però in estrema sintesi, la gerarchie delle cartelle prevede al primo livello la cartella dei sorgenti (src) e quella con gli artifatti prodotti (target), ad esempio le classi compilate ed i war file. La cartella src è primariamente suddivisa in due sottocartelle speculari, una contiene sorgenti di produzione (main) ed una quelli di test. Queste sottocartelle (di main e test) contengono i sorgenti java (cartella java) e le risorse (dentro resources), prevalentemente i file di configurazione. Da rilevare che la cartella main contiene anche la sottocartella webapp dove si trova la webroot, ovvero le pagine web dell'applicazione ed il file web.xml.

3.2 L'iniezione delle dipendenze

TODO: La distinzione tra i tre livelli di bean

3.3 La localizzazione

La localizzazione consiste nelle tecniche per tradurre i testi, l'aspetto e le form di input di un'applicazione nelle diverse lingue. Regola kit utilizza i Resource Bundle di Java per risolvere questo aspetto come indicato in (i18). Nella cartella src/main/resources sono presenti diversi file di proprietà il cui nome presenta la radice comune ApplicationResources, ad esempio:

- ApplicationResources en.properties
- ApplicationResources it.properties

Queste file replicano la stessa chiave traducendola nella varie lingue. Ad esempio dentro ApplicationResources en possiamo trovare la chiave errors.required a cui è associato un testo inglese:

```
1 errors.required=the field is required.
```

Nel file ApplicationResources it sarà possibile fornire una traduzione per la chiave errors.required semplicemente ridefinendola.

```
1 errors.required=campo necessario.
```

Affinché questo meccanismo di localizzazione funzioni Regola kit provvede automaticamente a selezionare il giusto file in base alle impostazioni del browser che utilizza l'applicazione. Inoltre è necessario rammentarsi di non includere mai direttamente dei testi all'interno delle pagine web ma di richiamare le chiavi definite dentro i file ApplicationResources. Ad esempio nelle pagine jsp bisogna inserire qualcosa di simile a:

```
1 <fmt:message key="errors.required"/>
```

Per le pagine jsf è invece possibile utilizzare il managed bean mgs.

```
1 #{msg[errors.required]}
```

3.4 Le connessioni al database

Le connessioni al database sono di due tipi a seconda dell'ambiente in cui sta girando l'applicazione. Per l'ambiente di run-time è necessario indicare il nome JNDI del datsource fornito dall'application server mentre a design-time bisogna proprio impostare tutte le caratteristiche di una connessione. In entrami i casi la configurazione riguarda l'impostazione di proprietà di configurazione dentro i file design-time.properties e run-time.properties.
Per maggiori dettagli ed esempi di configurazione per i diversi application server si rimanda a 4 nella pagina 23.

3.5 Verbosità dei log

TODO: log4j e come interagisce con JBoss

3.6 La sezione dei test

TODO: come configurare i test e quali file utilizzano.

3.7 Applicazione Servlet

TODO: Davvero in breve la struttura

Capitolo 4

Database

4.1 Configurazione di run-time

Le applicazioni JEE generalmente lasciano la gestione delle connessioni database al container. In questo modo è possibile per i sistemisti modificare, ad esempio, la url di connessione oppure il numero di connessioni in pool senza modificare l'applicazione né dovere effettuare una riconsegna (redeploy). Ogni container configura in modo diverso però ogni datasource è caratterizzato necessariamenta da un nome JDNI, ad esempio java:comp/env/jdbc/miodatabase. Questo nome è utilizzato dall'applicazione per recuperare la connessione e, in definitiva, connettersi al database. Nelle applicazioni Regola kit il nome JNDI del datasource è specificato nella proprietà di runtime jee.datasrouce (nel file src/main/resource/runtime.properties).

```
1 jee.datasource=java:comp/env/jdbc/Datasource
```

Con questo la configurazione di runtime della vostra applicazione è terminata, non ci sono altre variabili da impostare. Un problema tipico di configurazione riportato di diversi utenti è l'impossibilità di connettersi al datasource per avere utilizzato un indirizzo JNDI sbagliato. La causa del problema è che il nome con cui l'application server espone la connessione non è *esattamente* quello specificato nella configurazione: perché ad esempio viene preposta la stringa 'java:' o a volte 'java:comp/env/'.

Per facilitare la soluzione di questo tipo di problema concludiamo il paragrafo riportando qualche configurazione di datasource per gli application server più diffusi. Ad esempio JBoss richiede di ricopiare nella cartella di deploy del server utilizzato un file con il nome del tipo *-ds.xml (ad esempio miadatasource-ds.xml) con un contenuto simile al seguente:

```
1 <?xml version="1.0" encoding="UTF-8"?>

3 <datasources>
```

```
<local-tx-datasource>
    <jndi-name>jdbc/services</jndi-name>

    <connection-url>jdbc:oracle:thin:@133.222.0.1:1522:SID</connection-url>
    <user-name>username</user-name>
    <password>*****</password>

    <driver-class>oracle.jdbc.driver.OracleDriver</driver-class>
    <exception-sorter-class-name>
        org.jboss.resource.adapter.jdbc.vendor.OracleExceptionSorter
    </exception-sorter-class-name>
    <min-pool-size>5</min-pool-size>
    <max-pool-size>20</max-pool-size>

</local-tx-datasource>
</datasources>
```

In questo caso nella configurazione dell'applicazione dovete indicare il nome indicato però preceduto da 'java:' come indicato di seguito.

```
jee.datasource=java:jdbc/services
```

Per quanto riguarda Jetty il file di configurazione env.xml contiene l'indicazione del datasource.

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD
Configure//EN" "http://jetty.mortbay.org/configure.dtd">
<Configure class="org.mortbay.jetty.webapp.WebAppContext">

  <New id="jira-ds" class="org.mortbay.jetty.plus.naming.Resource">
     <Arg>jdbc/Datasource</Arg>
       <Arg>
         <New
         class="org.enhydra.jdbc.standard.StandardConnectionPoolDataSource">
           <Set name="Url">jdbc:mysql://localhost/clienti</Set>
           <Set name="DriverName">com.mysql.jdbc.Driver</Set>
           <Set name="User">nicola</Set>
         </New>
       </Arg>
  </New>

</Configure>
```

Jetty aggiunge al nome configurato la stringa 'java:comp/env/' per cui l'indirizzo JNDI da utilizzare è il seguente:

```
jee.datasource=java:comp/env/jdbc/Datasource
```

TODO: Aggiungere la configurazione per Tomcat

TODO: Mi chiedo come mai il dialetto sia specificato come proprietà di designtime ma non come proprietà di runtime.

4.2 Configurazione di design-time

Tra le caratteristiche invidiabili del framework Spring vi è la possibilità di testare l'applicazione fuori dal container con notevoli risparmio di tempo e relativo aumento di produttività. I servizi necessari ai test sono

offerti direttamente da Spring che si sostituisce al container, ad esempio, nella gestione delle transazioni e nei datasource. In quest'ultimo caso è necessario specificare tutte le proprietà di una connessione (come il driver, la url, la username e la password) come proprietà di designtime (nel file src/test/resources/designtime.properties).

```
...
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class = com.mysql.jdbc.Driver
hibernate.connection.url = jdbc:mysql://localhost/clienti
hibernate.connection.username = nicola
hibernate.connection.password =
...
```

Da notare che è necessario specificare anche la variabile hibernate.dialect con il tipo di *dialetto* di database utilizzato. I tipi disponibili sono molti ed elencati nella documentazione di Hibernate, alcuni esempi comuni sono MySQL5Dialect, OracleDialect, PostgreSQLDialect, HSQLDialect e SQLServerDialect.

Capitolo 5

Persistenza

5.1 Hibernate

TODO: Qui si dice che l'orm predefinito da Regola kit è Hibernate, perché è estremamente flessibile e su database preesistenti consente di gestire anche i casi più anomali. Perché non usiamo ancora JPA (immaturo).

5.2 Configurazione

TODO: I file hibernate.cfg.xml ed i mappaggi. Alcuni esempi di base per questi file.

5.3 Generatori automatici

TODO: Esempi di configurazione di hibernate.reveng.xml ed esempi (tutti) dei vari goals disponibili a riga di comando.

5.4 Altri ORM

TODO: Si parla del supporto sperimentale per gli altri orm.

Capitolo 6

Messa in produzione

6.1 Produrre i pacchetti war ed ear

TODO: Qui si spiega come produrre i pacchetti e cosa contengano al loro interno

6.2 Application Server

TODO: Perché non usare Jetty ed esempi di configurazione per Tomcat e JBoss. Si rimanda al capitolo dei database per la configurazione dei datasource.

6.3 Integrazione continua

TODO: Perché serve e come configurare i file standard dei progetti di Regola kit

Capitolo 7

Sviluppo

Questo capitolo apre la parte dedicata alle funzionalità offerte da Regola kit relativamente allo sviluppo. Lasciandosi alle spalle le configurazioni.

7.1 Domain Driven Development

TODO: Molto in breve si spiega come ci si incentri sul modello.

7.2 Livelli

TODO: Si presentano i vari livelli e si rimanda a capitoli successivi per i dettagli

7.3 Model Pattern

Regola kit nasce per agevolare la realizzazione di applicazioni web destinate a gestire moli piuttosto ampie di dati generalmente persistite su un database relazionale. Un problema ricorrente in questi contesti è quello che si potrebbe sinteticamente indicare come come la questione dei sottoinsiemi, ovvero fornire una risposta generale alle domande: come estrarre in modo conveniente sottoinsiemi di una certa entità dal database? E come rappresentare in modo sintetico questi sottoinsiemi all'interno di un form html oppure in un'applicazione desktop?
La soluzione proposta da Regola kit riteniamo che possa essere ritenuta sufficientemente assodata e generica da ambire al rango di design pattern (magari un minore); se così fosse pensiamo che la descrizione del problema unitamente alla soluzione individuata possa essere identificata col nome di Model Pattern (o in alternativa Selection).

Nei prossimi paragrafi cercheremo di formalizzare questo nuovo pattern descrivendone intenti e forze in gioco. Si avvisa fin d'ora che materremo il discorso a livello teorico presentando solo esempi astratti utili unicamente per comprendere la portata della soluzione proposta. Rimandiamo al paragrafo 8.4 nella pagina 37 per illustrare l'implementazione di riferimento di Model Pattern (quella contenuta in Regola kit) e spiegarne l'utilizzo all'interno delle vostre applicazioni.

7.3.1 Intento

Model Pattern vuole fornire un design appropriato per individuare un sottoinsieme di oggetti del modello di dominio e rappresentare questo sottoinsieme in modo conveniente all'interno del livello di presentazione.

7.3.2 Forze

In una categoria piuttosto ampia di applicazioni (che comprende la quasi totalità di applicazioni web) gli oggetti del modello salvano il loro stato in modo persistente su database, magari attraverso dei framework ORM (ad esempio Hibernate, JPA, ecc.). Non di rado il numero di istanze persistenti, per tipologia di classe, è veramente elevato cosa che complica in primis l'estrazione (1) (ovvero il filtraggio) delle istanze. Sempre dalla mole dei dati deriva la necessità di ordinare (2) e paginare (3) gli oggetti estratti: ovvero raggrupparli in sotto gruppi contenenti al più un certo numero di elementi. Un altra richiesta tipica delle applicazioni di cui stiamo parlando è quella di rappresentare gli oggetti del modello estratti in modo parziale, ovvero mostrare solo un sotto gruppo di proprietà, in modo da fornire, in un solo colpo d'occhio, tutti gli elementi necessari per prendere delle decisioni sui dati eliminando le proprietà superflue. Questo sezionamento delle classi del modello si chiama proiezione (4).
Del sottoinsieme degli elementi del modello individuato e estratto dal database (reidratato) è inoltre necessario fornire delle rappresentazioni sintetiche da utilizzare in diversi punti dell'applicazione. Si prenda per esempio la classe Studente ed il form html che consente di effetture le operazione di editing sulle sue proprietà quali nome, cognome ed altre. Si consideri poi che tra le proprietà della classe Studente vi sia anche la collezione di classi del tipo Esami che contenga gli esami sostenuti fino alla data corrente. Bene, il problema è spesso quello di mostrare nello stesso form html contenete le proprietà di Studente anche la collezione di Esami: in questo caso la rappresentazione corretta potrebbe variare in base alle finalità applicative. Ad esempio le segreterie di facoltà potrebbero essere interessati all'elenco completo di tutti gli esami (nessuna sintesi), i docenti potrebbero volere esclusivamente la media delle valutazione degli esami (applicare una metrica), l'ufficio delle imposte potrebbe essere interessato solo al fatto di aver sostenuto almeno un esame

(e quindi un intero che esprime il numero di esami superati). Tutte queste rappresentazioni (elenco, metrica ed intero) sono una descrizione del sottoinsieme di oggetti del modello Esami.
Inoltre si consideri il caso in cui la collezione di Esami fosse vuota o nulla, in questo caso diverse rappresentazioni tradurrebbero il valore null nel modo più appropriato rispetto al contesto (ad esempio con una stringa che indica l'assenza di esami, o con una segnalazione di errore o quant'altro). In ogni caso per ogni sottoinsieme di oggetti del modello emerge prepotente la necessità di provvedere ad una o più rappresentazione sintetica (5).
Quindi ricapitolando le forze che caratterizzano il problema:

1. estrazione di dati efficiente da una mole ampia o ampissima
2. ordinare le istanze di modello estratte
3. paginare le istanze di modello estratte
4. effettuare delle proiezioni sulle istanze di modello estratte
5. fornire delle rappresentazioni sintetiche della totalità di istanze estratte (cioè del sottoinsieme di tutta la popolazione di istanze estratto)

7.3.3 Esempio

Per amore di concretezza riportiamo un esempio che possa aiutare a comprendere meglio il problema descritto nel paragrafo precedente e la soluzione proposta da Model Pattern. Immaginiamo di avere una collezione di oggetti persista su database, tutti appartenenti alla medesima classe Prodotto, che presentano solo due proprietà: id, un intero che tiene la chiave primaria, e la descrizione del prodotto stesso.

```
public class Prodotto {

    private Integer id;
    private String descrizione;

    //getter and setter per tutti i campi
    ...
}
```

Il problema è quello di descrivere in modo sintetico sottoinsiemi di oggetti della classe Prodotto. Procederemo per gradi iniziando con il sottoinsieme diciamo Σ definito in modo sintetico, dalla descrizione seguente: il sottoinsieme di tutti gli oggetti la cui descrizione inizia con la parola 'manuale'. L'equivalente informatico di questa definizione è costiuito da un'instanza della classe seguente (che svolge il ruolo di Model Pattern):

```
public class ProdottoPattern  {

    private String inizioDescrizione;

    //getter and setter per tutti i campi
    ...
}
```

In particolare il sottoinsieme Σ può essere definito da un oggetto della classe ProdottoPattern che ha come proprietà inizioDescrizione la stringa 'manuale'. Si può notare che ProdottoPattern non contiene né il sottoinsieme Σ né il codice per estrarlo da database; si limita solo a fornirne una descrizione sintetica del sottoinsieme. Arricchiamo questo primo esempio includendo anche l'ordine degli oggetti contenuti nel sottoinsieme Σ, ad esempio richiedendo che siano ordinati in base alla proprietà id. In questo caso la classe Model Pattern potrebbe modificarsi così:

```
public class ProdottoPattern  {

    private String inizioDescrizione = ``manuale'';
    private String[] ordine = { ``id'' };

    //getter and setter per tutti i campi
    ...
}
```

In questo modo sarebbe in grado di descrivere il sottoinsieme ordinato Σ anche se in modo un po' rozzo infatti si trascura la possibilità di ordinare in modo ascendente o discendente. Comunque l'esempio dovrebbe rendere l'idea di come le classi Model Pattern si limitino a descrivere i sottoinsiemi senza contenere codice per l'estrazione dello stesso. Questo servizio è infatti fornito dalle classi a corredo come quelle fornite da Regola kit che consentono, dato un oggetto del tipo Model Pattern, di ottenere il sottoinsieme voluto. La sfida consiste nella predisposizione di un architettura che consenta di utilizzare dei Model Pattern semplici come quelli mostrati in questi esempi ma in grado di funzionare correttamente (ad esempio indicando che la proprietà inizioDescrizione si riferisce alla proprietà descrizione di Prodotto) su diversi orm. Si rimanda al paragrafo successivo per una descrizione esatta dell'architettura.
Fino a questo momento abbiamo trattato solo la capacità di Model Pattern di esprimere una selezione, adesso ci occuperemo della funzione di rappresentazione del sottoinsieme selezionato. Immaginiamo di dover mostrare in una pagina web il sottoinsieme Σ in modo sintetico (ovvero senza elencare in una colonna tutti i suoi elementi). Modifichiamo Model Pattern per aggiungere la proprietà sintesi.

```
public class ProdottoPattern  {

    public getSintesi()
    {
       if (inizioDescrizione == null)
       {
          return ``Prodotti con descrizione iniziante con: '' +
          inizioDescrizione;
       }
       else
       {
          return "Tutti di prodotti.";
       }
    }
```

```
14
}
```

Come si vede la proprietà sintesi contiene la logica per fornire una semplice rappresentazione del sottoinsieme Σ anche nel caso in cui inizioDescrizione sia nullo.

7.3.4 Architettura

Model Pattern propone di avvilire le forze del problema utilizzando una classe (Pattern) per ogni tipo nel modello che si occupi di due distinti aspetti:

1. individuare il sottoinsieme selezionato
2. fornire (zero o più) rappresentazione del sottoinsieme selezionato

Per quanto riguarda il primo punto bisogna precisare fin da subito che la classe Pattern non contiene in nessun modo il codice necessario all'estrazione della selezione o la selezione stessa. Invece si occupa di contenere le informazioni necessarie a descrivere in modo esatto il sottoinsieme, ovvero descriverlo in modo esatto rispetto ai seguenti punti:

1. quali istanze sono comprese nel sottoinsieme e quali esluse
2. con quali ordine le istanze entrano nel sottoinsieme
3. in che modo il sottoinsieme è paginato
4. quali proiezioni effettuare su ogni singola istanza del sottoinsieme

La classe Pattern si occupa anche di contenere zero o più rappresentazioni del sottoinsieme che descrive da utilizzare in altrettanti punti dell'applicazione (tipicamente un qualche livello di presentazione, o la produzione di report, ecc.). A tal proposito predispone il metodo

```
1     void init(Collection<Model> subset)
```

questo metodo prende in ingresso un sottoinsieme e provvede ad inizializzare le rappresentazioni di subset secondo la logica contenuta in pattern;

[TODO: nda Lorenzo, in futuro potrebbe essere utile utilizzare questo stesso metodo per inizializzare anche la parte di descrizione 1) in base ad un subset, ovvero produrre il filtro che estrarrebbe eventualmente subset?]

7.4 Generatori

TODO: Qui si elencano i generatori disponibili e cosa scrivano. Se il paragrafo diventasse troppo lungo allora lo mettiamo su un capitolo a parte.

Capitolo 8

Dao

8.1 Scopo

TODO: A cosa serve DAO?

8.2 GenericDao

TODO: Se ne descrivono interfacce e si presentano esempi d'uso (dei test d'unità) che si snodano tra i vari paragrafi.

8.3 Creare un custom dao

TODO: l'interfaccia, l'implementazione ed infine la configurazione di Spring. Si ricorda che esiste un generatore per questo.

8.4 Ricerche con Model Pattern

Nel paragrafo 7.3 nella pagina 31 è stato presentato in modo formale un nuovo design pattern chiamato Model Pattern; è stato spiegato come intenda risolvere il probema dell'estrazione di un sottoinsieme di oggetti e di come intenda fornire una rappresentazione sintetica di questo oggetto. La soluzione proposta, si è visto, prevede l'utilizzo di una classe che svolga il ruolo di ModelPattern, ovvero sia contemporaneamente in grado di individuare quali elementi estrarre e di rappresentarli senza però contenere al suo interno né il sottoinsieme né la logica per l'estrazione. La discussione è rimasta però a livello astratto rimandando la descrizione di come Model Pattern possa essere utilizzato concretamente all'interno di un'applicazione Regola kit; nei prossimi paragrafi vedremo quindi l'implementazione di riferimento di Model Pattern contenuta in Regola kit (precisamente nei moduli regola-core,

regola-dao e sottomoduli) per scoprire come progettare un classe ModelPattern con diversi criteri di filtraggio e ordinamento e la utilizzeremo lungo i diversi livelli applicativi, dal DAO alla presentazione. Per maggiore concretezza immagineremo di doverci occupare, diciamo, della classe Prodotto riportata di seguito.

```
class Prodotto {
    Integer id;
    String  descrizione;

    //seguono getter/setter per ogni campo
    ...
}
```

8.4.1 ModelPattern

Regola kit fornisce una classe di base org.regola.model.ModelPattern da cui derivare il nostro ModelPattern; questa classe presenta alcune facilitazioni per specificare gli ordinamenti e la paginazione ed è accettata quasi in ogni livello applicativo, dai GenericDao fino a controllori web che si occupano di disegnare la tabella con il sottoinsieme. Tenendo a mente la classe Prodotto una prima versione del nostro ModelPattern potrebbe essere la seguente:

```
class ProdottoPattern extends org.regola.model.ModelPattern
    implements Serializable {

    Integer  chiave;

    @Equals("id")
    public Integer getChiave() {
        return chiave;
    }

    ...
}
```

Soffermiamoci un attimo su alcuni elementi che trasformano la nostra classe in un ModelPattern utilizzabile con Regola kit:

nome della classe una convenzione di Regola kit prevede che ogni ModelPatter dovrebbe chiamarsi col nome della classe di modello a cui si riferisce (nel nostro caso Prodotto) seguito dal suffisso Pattern, per cui ProdottoPattern

derivazione la nostra classe deve derivare dalla classe org.regola.model.ModelPattern fornita con Regola kit

serilizzazione la nostra classe deve essere progettata per la serializzazione e quindi necessariamente implementare l'interfaccia di marker java.io.Serializable

La classe ProdottoPattern così realizzata è predisposta per individuare particolari sottoinsiemi di instanze Prodotto che presentano particolari valori

della proprietà id, in particolare tutte le istanze di Prodotto che hanno id uguale alla proprietà chiave di ProdottoPattern. Il legame tra chiave ed id è realizzato mediante l'annotazione @Equals (alla riga 6) che, posta sul getter della proprietà chiave di ProdottoPattern, unisce quest'ultima alla proprietà dell'oggetto di modello specificata dentro l'annotazione stessa (in questo caso id).
Bisogna chiarire da subito che non è la classe ModelPattern ad individuare un sottoinsieme ma ogni sua istanza per cui, tornando al nostro esempio, bisogna istanziare un oggetto del tipo ProdottoPattern.

```
ProdottoPattern pattern = new ProdottoPattern();

pattern.setChiave(234532); //ora pattern rappresenta un sottoinsieme
```

Dopo l'assegnazione pattern (l'oggetto) e non ProdottoPattern (la classe) è in grado di individuare un sottoinsieme di oggetti Prodotto aventi la proprietà id uguale a 234532 (essendo id una chiave primaria il sottoinsieme conterrà al più un elemento). Da notare come l'oggetto pattern non contenga in sé il sottoinsieme ma solo la descrizione sintetica di questo; per ottenere il sottoinsieme bisogna rivolgersi alla classe ProdottoDao invocando il metodo find.

```
ProdottoPattern pattern = new ProdottoPattern();

pattern.setChiave(234532);

List<Prodotto> prodotti = prodottoDao.find(pattern);
```

Ora che abbiamo visto concretamente come creare ed utilizzare una classe ModelPattern possiamo scendere nel dettaglio ricordando che Model Pattern deve individuare con esattezza il sottoinsieme, in particolare deve occuparsi di ordinamento, paginazione e proiezione. Nei prossimi paragrafi quindi scopriremo come specificare:

1. la selezione da effettuare
2. i criteri di ordinarmento
3. la paginazione da effettuare
4. le proprietà da comprendere nella proiezione
5. fornire rappresentazioni sintetiche del sottoinsieme indicato

8.4.2 La selezione

La selezione avviene impostando alcuni criteri di filtraggio sul ModelPattern. Ogni criterio è caratterizzato da:

1. la proprietà del modello a cui si applica
2. il tipo di criterio (ad esempio ugualianza, appartenenza, ecc.)
3. il valore da utilizzare per il confronto

Riprendendo l'esempio del paragrafo precedente troviamo un unico criterio di filtraggio, di tipo ugualianza, che confronta la proprietà di modello id con il valore della proprietà chiave di ProdottoPattern.

```
class ProdottoPattern extends org.regola.model.ModelPattern
    implements Serializable {

    Integer  chiave;

    @Equals("id")
    public Integer getChiave() {
        return chiave;
    }

    ...
}
```

La proprietà di modello può trovarsi direttamente sulla classe radice, ovvero quella a cui riferisce ModelPattern (nell'esempio Prodotto) oppure su classi a questa collegate tramite associazioni del tipo uno-a-uno, molti-a-uno od uno-a-molti. In generale si navigano le associazioni utilizzando come separatore il punto tranne che per relazioni uno-a-molti (collezioni) dove si utilizza il simbolo [] postfisso. Nel dettaglio:

radice	il nome della proprietà	id nome
uno-a-uno	il punto	indirizzo.via categoria.descrizione
molti-a-uno	il punto	cliente.nome fattura.progressivo
uno-a-molti	[] postfisso ed il punto	elmenti[].nome elementi[].dettagli[].progressivo

Il tipo del criterio è impostato scegliendo l'annotazione tra quelle presenti in Regola kit. Attualmente è possibile scegliere tra le annotazioni seguenti:

Equals	ugualianza
NotEquals	disugualianza
Like	il like
GreatherThan	maggiore
LessThan	minore
In	appartenenza

Il valore di ogni criterio una proprietà di ModelPattern individuata annotandone il getter. Il tipo di questa proprietà deve corrispondere a quello del modello tranne per il criterio In per cui è necessario specificare un'array. L'esempio seguente fornisce una rappresentazione piuttosto completa di quando esposto:

```
public class CustomerPattern extends ModelPattern
   implements Serializable {

  private Integer id;

  @Equals("id")
  public .Integer getId()
  {
    return id;
  }

  private String firstName;

  @Like(value = "firstName", caseSensitive = true)
  public String getFirstName()
  {
    return firstName;
  }

  private String[] lastNames;

  @In("lastName")
  public String[] getLastNames() {
    return lastNames;
  }

  private Integer invoiceId;

  @Equals("invoices[].id")
  public Integer getInvoiceId() {
    return invoiceId;
  }

}
```

8.4.3 Ordinamento

Sugli oggetti selezionati è possibile impostare criteri di ordinamento in base alle proprietà della radice del modello o sugli oggetti ad essa associati. L'implementazione di Model Pattern presente in Regola kit utilizza per specificare gli ordinamenti (così come le proiezioni) la classe ModelProperty, una specie di descrittore di una generica proprietà. Si instazia così:

```
   ModelProperty mp =  new
   ModelProperty("id","customer.column.",Order.asc);
```

ModelProperty contiene il nome della proprietà (in base a cui ordinare), un prefisso da utilizzare per individuare in modo univoco la proprietà all'interno di tutta l'applicazione ed, infine, la direzione dell'ordinamento (acendente o discendente). Nell'esempio si esprime un ordinamento ascentente sulla proprietà id, individuata nell'applicazione come customer.column.id. Per speficiare l'ordinamento basta popolare la lista sortedProperties di ModelPattern con tante istanze di ModelProperty in base all'ordinamento da realizzare. Ad esempio:

```
   ModelProperty id =  new
   ModelProperty("id","customer.column.",Order.asc);
```

```
ModelProperty street = new
ModelProperty("address.street","customer.column.",Order.desc);

modelPattern.getSortedProperties.clear();
modelPattern.getSortedProperties.add(id);
modelPattern.getSortedProperties.add(street);
```

In questo caso si impone un ordinamento crescente per id e descrescente per la proprietà strret dell'oggetto associato address.

8.4.4 Paginazione

Sugli oggetti così selezionati ed ordinati è possibile effettuare un'olteriore selezione dividendolo in blocchi contigui dette pagine contenenti al più n elementi. ModelPattern consente di impostare la dimensione delle pagine così come impostare la pagina da selezionare. Ad esempio:

```
modelPattern.setPageSize(20);
modelPattern.setCurrentPage(0);
```

Qui si suddivide l'insieme ordinato in pagina con al più di 20 oggetti, il primo blocco comprende gli oggetti da $[0, 19]$, il secondo da $[20, 39]$ e così via. Inoltre si seleziona la prima pagina (la numerazione delle pagine parte da 0), ovvero gli oggetti da $[0, 19]$.

8.4.5 Proiezione

Fino a qui abbiamo visto come effettuare la selezione, ordinarla e limitarla ad un certo numero di elementi. É inoltre possibile limitare non solo il numero ma anche le proprietà del singolo oggetto del sottoinisieme. Ad esempio può essere conveniente limitarsi a considerare solo la proprietà id e name piuttosto che il complesso di tutte le proprietà della radice del modello e/o degli oggetti associati. Questo genere di limitazione si chiama proiezione. Il meccanismo per specificare le proiezione dentro Regola kit si basa sempre sugli oggetti ModelProperty, basta popolare la lista visibleProperties di ModelPattern con le proprietà che si intende proiettare.

```
ModelProperty id = new
ModelProperty("id","customer.column.",Order.asc);
ModelProperty street = new
ModelProperty("address.street","customer.column.",Order.desc);

modelPattern.getVisibleProperties.clear();
modelPattern.getVisibleProperties.add(id);
modelPattern.getVisibleProperties.add(street);
```

Nell'esempio si effettua una proiezione comprendente esclusivamente le proprietà id e strett dell'oggetto associato street.

8.4.6 Rappresentazione

TODO:

Capitolo 9

Servizio

9.1 Scopo

TODO: A cosa serve il livello di Servizio? (spr)

9.2 GenericManager

TODO: Come crearlo. Si ricorda che esiste un generatore per questo.

9.3 Transazioni

TODO: come sono demarcate e come aggiungere politiche diverse per aprire e chiudere transazioni

9.4 Politiche di detach

Gli EJB, fin dal loro esordio nel 1998, hanno nascosto le classi di modello al livello di presentazione ed utilizzato in loro vece delle classi apposite, i Data Transfer Object. Questi DTO avevano il compito di raccogliere una visione appiattita del modello che però contenesse tutte e sole le informazioni necessarie al livello di presentazione. Erano oggetti di passaggio, tutti dati e nessuna logica che nascondevano la complessità del modello agli strati superiori, realizzando di fatto un disaccoppiamento forte tra questi livelli. Nonostante questo pregio i DTO hanno contribuito molto alla fama di pesantezza degli EJB in quanto, anche in progetti piccoli, risultava evidente che il lavoro per creare, popolare e sincronizzare i DTO con la classi di modello era di gran lunga superiore ai benefici. La domanda sorta spontanea a molti era: perché non passare ai livelli superiori direttamente le classi di modello? Molti

progetti hanno dimostrato che la strada è praticabile con successo e formalizzato i vari modi per farlo. In questo paragrafo presenteremo due pattern per utilizzare direttamente le classi di modello nel livello di presentazione conosciuti rispettivamente come pojo façade e modello esposto.

9.4.1 Pojo façade

Questo pattern per utilizzare le classi di modello mantiene l'architettura complessiva degli EJB interponendo tra la presentazione ed il modello un apposito livello (façade) il cui compito è ancora quello di fornire servizi per la presentazione. La differenza principale con gli EJB però risiede in due aspetti:

1. Il livello façade è costituito da semplici pojo
2. Invece che DTO si passano oggetti di modello, opportunamente trattati

Del primo punto bisogna sottolineare che il livello façade, come negli EJB, è ancora responsabile di aprire e chiudere le transazioni così come della sicurezza (autenticazione ed autorizzazione). La differenza è che non è necessario un container JEE per ottenere questi servizi ma sono realizzati tramite un container invertito (come Spring) e la programmazione orientata agli aspetti (AOP). Con il vantaggio di poter effettuare tutti i test fuori dal container. Il secondo punto invece richiede alcuni approfondimenti che discuterò di seguito.

incapsulamento del modello

Il modello, secondo la definizione più accettata di Model Driven Development (MDD), è una realtà attiva in grado di utilizzare risorse esterne, ad esempio può presentare dei metodi per salvarsi su database. Se il livello di presentazione chiamasse direttamente questi metodi verrebbe meno la funzione centralizzatrice dei pojo façade ed il sistema solleverebbe diverse eccezioni legate all'assenza di transazioni attive sul livello di presentazione. Per ovviare a questo problema esistono alcune tecniche:

convenzione si stabilisce che nessun sviluppatore chiami mai questi metodi e si limiti ad ignorarli. Per gruppi numerosi potrebbe essere necessario rafforzare la convenzione magari marcando i metodi da ignorare con delle annotazioni ed utilizzare un compilatore ad aspetti che riporti come errore ogni chiamata a questi metodi effettuata nel livello di presentazione.

visibilità dei metodi ovvero marcare i metodi come private o protected. Spesso però il modello è sparso in vari package Java per cui questa possibilità non è praticabile.

utilizzare interfacce il livello di presentazione è scritto non in termini delle classi del modello ma in termini di sotto interfacce che ne espongano

solo alcuni metodi nascondendone altri. Può essere realizzato in due modi:

realizzare le interfacce direttamente sul modello è una strada tecnicamente complessa perché richiede di scrivere metodi covarianti, di affrontare il problema delle collezioni immutabili ed inoltre non consente comunque di esporre quei metodi che richiedevano computazioni realizzabili solo nel livello di modello (perché ad esempio richiedono l'accesso al database)

creare degli adapter hanno lo svantaggio principale di essere terribilmente complicati e finiscono spesso per somigliare ai DTO (in termini dei soli svantaggi).

oggetti staccati (detached)

Gli elementi del modello possono effettuare implicitamente delle richieste al motore di persistenza in modo invisibile per i client. Il caso più esemplare è quello delle collezioni che gli ORM gestiscono a volte in modalità di caricamento differito (lazy o late loading). Ovvero gli elementi delle collezione sono recuperati dal database solo al momento del primo accesso ad un elemento, in modo trasparente. Il problema è che nel livello di presentazione sessioni ORM e transazioni non sono disponibili con conseguente errore a run time. Esistono in generale due tipi di soluzioni:

Eliminare i caricamenti differiti ovvero effettuare delle configurazioni per gli ORM (dette mappaggi) che carichino subito tutte le collezioni leggendole dal database assieme all'oggetto che le contiene. Bisogna però ricordare che questa strada non è percorribile quando la mole dei dati è notevole o lo schema del database non lo consenta; circostanze, queste, molte frequenti nella pratica.

Fare scattare i caricamenti differiti ovvero prima di passare al livello di presentazione gli oggetti del modello fare scattare tutti i caricamenti differiti. Spesso gli ORM hanno metodi appositi per fare questo (ad esempio Hibernate ha il metodo Hibernate.initialize()).

Vantaggi di Pojo Façade

Confrontando Pojo Façade, in particolare con gli EJB Façade, emergono diversi punti di forza che riassumo brevemente:

sviluppo più facile e (quindi) veloce per la possibilità di testare fuori dal container e l'assenza di DTO che, come visto, possono essere molto onerosi in termini di tempo.

possono eliminare la necessità di un container visto che non è più necessario per transazioni e sicurezza potrebbe non essere necessario utilizzare un container completo JEE nel progetto.

demarcazione flessibile delle transazioni la configurazione tramite AOP delle transazione può avere un livello di flessibilità inaudito nel mondo EJB.

Confrontando invece Pojo Façade con il pattern del modello esposto (presentato nel prossimo paragrafo) si possono isolare i seguenti vantaggi:

livello di presentazione facilitato perché non è richiesta una tecnica particolare di gestione delle transazioni sul livello di presentazione, come vedremo

vista coerente dei dati del database tutti i dati sono esposti dai pojo façade e sono quindi raccolti all'interno di un'unica transazione. Questo può non essere vero nel pattern del modello esposto.

Svantaggi di Pojo Façade

Gli svantaggi principali rispetto agli EJB façade sono:

mancanza di standard lo standard JEE non prevede Spring (per ora e purtroppo) quindi i problemi possono essere di vario genere. In verità Spring è largamente riconosciuto nella comunità di sviluppatori tanto che moltissime librerie o sono espressamente dedicata a Spring oppure sono facilmente integrabili. Inoltre molte configurazioni standard sono riconosciute da Spring (ad esempio le annotazioni per i web service o per la persistenza). Comunque a titolo di esempio i problemi legati alla mancanza di standard possono essere:

- **ottenere il pojo façade** il client potrebbe non sapere come ottenere un'istanza del façade, ad esempio un generatore che espone una classe come end point di un servizio web potrebbe non sapere come ottenere un'istanza del pojo façade.
- **sicurezza** la dichiarazione dei vincoli di sicurezza avviene in modo fuori standard e quindi non portabile.

transazioni iniziate su client remoti attualmente non è supportato da framework come Spring: i pojo façade non possono partecipare a transazioni avviate esternamente.

Con riferimento al pattern del modello esposto i principali svantaggi sono:

problemi con gli oggetti staccati come visto nei paragrafi precedenti può richiedere una codifica minuziosa e difficile accertare che tutti gli oggetti siano effettivamente staccati dal motore di persistenza. Inoltre gli errori si presentano solo a runtime, cosa che rende più difficile la loro individuazione.

incapsulamento del modello come abbiamo visto può risultare molto complesso nascondere alcuni metodi ai client.

9.4.2 Modello esposto

Un altra tecnica per evitare l'impiego dei DTO non si limita ad utilizzare le classi di modello per la presentazione ma imbocca la strada più radicale di rinunciare ad un livello di façade fornendo direttamente accesso ad un modello non più staccato (detached) ma attivo e transazionale. Il modello esposto è noto anche col nome di Open Session In View o anche Open Persistence Manager In View.

Quando è possibile utilizzare il modello esposto

Per poter utilizzare questo modello sono necessarie due condizioni inderogabili per il livello di presentazione che deve infatti:

- poter gestire la sessione dell'ORM
- accedere da locale al modello (sono escluse connessioni remote)

In assenza di queste due condizioni è necessario ripiegare su soluzioni alternative come pojo façade.

Come funziona

Per consentire al modello di funzionare nel livello di presentazione è necessario occuparsi di due aspetti fondamentali:

gestire la sessione La sessione utilizzata dall'ORM deve rimanere aperta per tutta la durata di una richiesta HTTP o addirittura per diverse richieste. Per fare questo è una soluzione consolidata ricorrere ad un filtro HTTP che si occupi prima di mantenere aperta la sessione ed infine di chiuderla anche in presenza di eccezioni. Spring presenta già dei filtri per svolgere questa funzione, tra cui OpenSessionInViewFilter. I vantaggi dell'impiego di un filtro sono:

sicurezza il filtro viene sempre invocato all'inizio ed alla fine della richiesta

riusabilità lo stesso filtro può essere riusato in diverse applicazioni

ortogonalità è possibile aggiungere e rimuovere il filtro senza modificare il codice del livello di presentazione.

gestire le transazioni La gestione delle transazioni potrebbe avvenire a livello di presentazione (ovvero essere delegata al filtro della sessione) oppure al livello del modello (ovvero gestita con AOP). Nel primo caso la transazione viene aperta all'inizio della richiesta dal filtro della sessione e chiusa al termine. Nel secondo caso (AOP) è possibile aprire e chiudere le transazioni in base ad uno schema molto più flessibile, come ad esempio attorno alle classi di modello che si occupano dei servizi. Entrambi gli approcci presentano vantaggi e svantaggi:

9.4.3 Conversazioni

Nelle applicazioni web una singola operazione di business può essere completata solo attraverso diverse richieste HTTP; ad esempio mediante la compilazione da parte dell'utente di alcune pagine in sequenza e solo al completamento dell'ultima l'operazione si considera o conclusa o annullata. In termini di interazione utente queste operazioni si chiamano conversazioni (o transazioni lunghe o anche transazioni utente) e sono gestite in modo completamente diverso nel pattern dei pojo façade piuttosto che nel modello esposto. Nei pojo façade la sessione dell'ORM inizia e si conclude all'interno della singola richiesta HTTP, per cui all'interno di una conversazione (che si compone di diverse richiesta HTTP) vengono usate sessioni diverse. Gli oggetti di modello prima di passare al livello di presentazione sono staccati (detached) dal motore di persistenza e poi riattaccati (reattached) nella richiesta HTTP successiva. Questa modalità operativa prevede quindi transazioni sul database limitate all'interno delle richieste HTTP e dedica molta attenzione alla fase di attach e detach degli oggetti dall'ORM. Nel pattern del modello esposto invece si utilizza la stessa sessione dell'ORM per tutta la durata della conversazione. Gli oggetti di modello passati al livello di presentazione non sono mai staccati (detached) ma rimangono sempre nel contesto di persistenza dell'ORM. Le transazioni sul database sono invece aperte e chiuse all'interno delle richieste HTTP per evitare dei lock sulle righe delle tabelle ed il conseguente degrado per tutta l'applicazione. Per assicurare la coerenza dei dati tra le varie transazioni è necessario abilitare un meccanismo per individuare eventuali variazione sui dati trattati, ad esempio una politica di lock ottimistico.

9.5 Web Services

TODO: esempi di configurazioni già pronte dentro Regola kit

Capitolo 10

Presentazione Web

10.1 Scopo

TODO: Di cosa si occupa questo livello?

10.2 Tecnologie

TODO: Panoramica brevissima su JSF, Spring WebFlow e Spring MVC.

10.3 Pagina di lista

TODO: Partendo da un esempio si provvede a spiegare quali sono i file coinvolti e come devono essere configurati. Si ricorda che esiste un generatore per questo.

10.4 Pagina di form

TODO: Partendo da un esempio si provvede a spiegare quali sono i file coinvolti e come devono essere configurati. Si ricorda che esiste un generatore per questo.

Bibliografia

[i18] Resource Bundles. http://java.sun.com/developer/technicalArticles/Intl/ResourceBundles/.

[mav] Maven2. http://maven.apache.org/.

[spr] Spring Framework. http://www.springframework.com.

Indice analitico