ORM: Object Relational Mapping

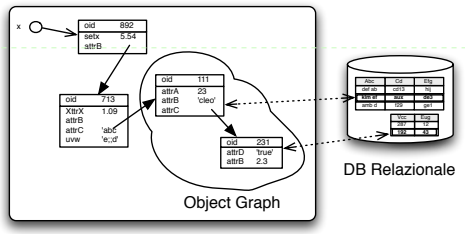
- Fenomeno dell'*impedence mismatch*:
  - modello dei dati relazionale ≠ modello dei dati del linguaggio
  - quindi occorre convertire i dati quando si trasferiscono fra DB e programma
- Si scrive un software di middleware che automatizzi il più possibile questo passaggio
- Approccio iniziata in Java, linguaggio orientato ad oggetti e trasferita ad altri linguaggi "ad oggetti" => Object Relational Mapping
- Si può usare per inserire (una parte del)le *business rules* nei programmi, sotto forma di metodi degli oggetti di mappatura
- Soluzione: Ricostruire in memoria un sottoinsieme del "grafo" della basi di dati, mano a mano che serve (si trasferisce solo ciò che è strettamente necessario), usando variabili del linguaggio
- Il sistema mantiene automaticamente la consistenza fra base di dati e "grafo" di oggetti in memoria, attraverso tecniche di sincronizzazione che tengano conto delle modifiche (transazioni)

Problematiche

- Tipi di mapping permessi (ricchezza del modello)
  - Gerarchie di oggetti (sottoclassi), attributi ripetuti o strutturati, relazioni molti a molti
- Specifiche del mapping
  - File di configurazione, annotazioni nel codice, strumenti grafici, mapping automatico
- Gestione delle sessioni di lavoro, transazioni
  - Differenza del modello di fallimento
- Livello di trasparenza della sincronizzazione
  - Le sessioni e le transazioni devono essere visibili
- Linguaggio di query
  - Potere espressivo rispetto a SQL

Object graph

- Si crea una struttura in memoria temporanea che riproduce sotto forma di oggetti una rappresentazione (di una parte) della base di dati (trasparenza della persistenza)
- Distinzione fra
  - definizione del mapping: automatica, semi-automatica, manuale
  - gestione del mapping: automatica o semi-automatica
-



Memoria temporanea

Sistemi

- Distinzione fra standard, implementazione di riferimento, implementazione
- Sistemi Open-Source e commerciali. Esempi:
  - Hibernate
    - Open source per l'application server JBoss (o GlassFish)
  - TopLink
    - TopLink, poi Oracle, poi "regalato" alla comunità open source (Glass Fish)
  - JDO
    - Standard Java, implementazione beta di Apache
- Standard
  - EJB 3.0 (modello di persistenza molto più semplice di EJB 2) => JPA
    - Utilizzo con JavaBeans oppure con normali oggetti Java (POJO)

Architettura

- Normalmente un ORM è una parte di un application server, anche se può essere usato da solo
  - JBoss
  - GlassFish
  - JRun

Hibernate (www.hibernate.org)

Classe persistente, può non essere un Java Bean

```
package events;

import java.util.Date;

public class Event {
    private Long id;
    private String title;
    private Date date;
    public Event() {}
    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id = id;
    }
    public Date getDate() {
        return date;
    }
    public void setDate(Date date) {
        this.date = date;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
}
```

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="events.Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="native"/>
        </id>
        <property name="date" type="timestamp"
                  column="EVENT_DATE"/>
        <property name="title"/>
    </class>
</hibernate-mapping>
```

chiave

mapping

EVENTS

EVENT_ID	EVENT_DATE	TITLE

tabella relazionale

Configurazione del sistema

```
<?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>
        <!-- Database connection settings -->
        <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
        <property name="connection.url">jdbc:hsqldb:hsql://localhost</property>
        <property name="connection.username">utente1</property>
        <property name="connection.password">passw1</property>
        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size">1</property>
        <!-- SQL dialect -->
        <property name="dialect">org.hibernate.dialect.HSQLDialect</property>
        <!-- Enable Hibernate's automatic session context management -->
        <property name="current_session_context_class">thread</property>
        <!-- Disable the second-level cache  -->
        <property name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>
        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property>
        <!-- Drop and re-create the database schema on startup -->
        <!-- <property name="hbm2ddl.auto">create</property> -->
       <mapping resource="events/Event.hbm.xml"/>
    </session-factory>

</hibernate-configuration>
```


Programma principale

```
package events;
import org.hibernate.Session;
import java.util.Date;
import java.util.List;
import util.HibernateUtil;

public class EventManager {
  public static void main(String[] args) {
    EventManager mgr = new EventManager();
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();
    Event theEvent = new Event();
    theEvent.setTitle("Nuovo Evento");
    theEvent.setDate(new Date());
    session.save(theEvent);
    session.getTransaction().commit();

    session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();
    List events = session.createQuery("from Event").list();
    session.getTransaction().commit();
    for (int i = 0; i < events.size(); i++) {
      theEvent = (Event) events.get(i);
      System.out.println("Event: " + theEvent.getTitle() +
       " Time: " + theEvent.getDate());
    }
    HibernateUtil.getSessionFactory().close();
  }
}
```

Sessioni

- Una sessione è lo “spazio” logico entro cui si svolge un’unità di lavoro. Nel caso più semplice c’è una corrispondenza uno a uno con una transazione.
- Creata da una “SessionFactory” (che è *threadsafe*): una sessione per *thread*
- Uso tipico:

```
Session session = factory.openSession();
Transaction transaction;
try {
  transaction = session.beginTransaction();
  ...
  transaction.commit();
} catch (Exception e) {
    if (transaction!=null) transaction.rollback();
    throw e;
} finally {
    session.close();
}
```


Mapping

- Può essere definito attraverso un file di configurazione XML
- Può essere definito attraverso annotazioni del programma JAVA (da Java 5.0 in poi, presenza di annotazioni con @…)


```
package it.fondamentidibasididati.eshibernate1;

import java.util.Set;

public class Studente {
	private Long Matricola;
	private String NomeCognome;
	private int AnnoCorso;
	private Set Esami;

	public Studente() {
	}

	public Long getMatricola() {
		return Matricola;
	}
	public void setMatricola(Long matricola1) {
		Matricola = matricola1;
	}
	public String getNomeCognome() {
		return NomeCognome;
	}
	public void setNomeCognome(String nomecognome1) {
		NomeCognome = nomecognome1;
	}
	public int getAnnoCorso() {
		return AnnoCorso;
	}
	public void setAnnoCorso(int annocorso1) {
		AnnoCorso = annocorso1;
	}
	public Set getEsami() {
		return Esami;
	}
	public void setEsami(Set nuoviEsami) {
		Esami = nuoviEsami;
	}
	public String toString() {
		return NomeCognome;
	}

}
```

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
	"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
	"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping
	package="it.fondamentidibasididati.eshibernate1">

	<class name="Studente" table="Studenti" lazy="true">

		<id name="Matricola"/>

		<property name="NomeCognome"
			not-null="true"/>

		<property name="AnnoCorso"/>

		<bag name="Esami"
			inverse="true"
			cascade="save-update,lock">
		    <key column="candidato"/>
		    <one-to-many class="Esame"/>
		</bag>

	</class>

</hibernate-mapping>
```



```
iimport java.util.Date;

public class Esame {
	private Long Codice;
	private String Materia;
	private Studente Candidato;
	private Date Data;
	private int Voto;
	private boolean Lode;

	public Esame() {
	}

	public Long getCodice() {
		return Codice;
	}
	public void setCodice(Long long1) {
		Codice = long1;
	}
	public String getMateria() {
		return Materia;
	}
	public void setMateria(String string) {
		Materia = string;
	}
	public Studente getCandidato() {
		return Candidato;
	}
	public void setCandidato(Studente studente) {
		Candidato = studente;
	}
	public Date getData() {
		return Data;
	}
	public void setData(Date data1) {
		Data = data1;
	}
	public int getVoto() {
		return Voto;
	}
	public void setVoto(int voto1) {
		Voto = voto1;
	}
	public boolean getLode() {
		return Lode;
	}
	public void setLode(boolean lode1) {
		Lode = lode1;
	}
	public String toString() {
		return Candidato.getNomeCognome() + " - " +
		Materia + ": " + Voto + (Lode ? " e lode" : "");
```

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
	"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
	"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping
	package="it.fondamentidibasididati.eshibernate1">

	<class name="Esame" table="Esami">

		<id name="Codice">
			<generator class="native"/>
		</id>
		<natural-id>
			<many-to-one name="Candidato" class="Studente".
			<property name="Materia"
				length="200"/>
		</natural-id>
		<property name="Data"/>
		<property name="Voto"/>
		<property name="Lode"/>

	</class>

</hibernate-mapping>
```

```
import java.util.List;

@Entity
@Table(name="Studenti")
public class Studente {
	private Long Matricola;
	private String NomeCognome;
	private int AnnoCorso;
	private List Esami;

	public Studente() {
	}

	@Id
	public Long getMatricola() {
		return Matricola;
	}
	public void setMatricola(Long matricola1) {
		Matricola = matricola1;
	}
	@NotNull
	public String getNomeCognome() {
		return NomeCognome;
	}
	public void setNomeCognome(String nomecognome1) {
		NomeCognome = nomecognome1;
	}
	@NotNull
	public int getAnnoCorso() {
		return AnnoCorso;
	}
	public void setAnnoCorso(int annocorso1) {
		AnnoCorso = annocorso1;
	}
	@OneToMany(cascade=ALL, mapped_by="candidato")
	public List<Esame> getEsami() {
		return Esami;
	}
	public void setEsami(List list) {
		Esami = list;
	}
	public String toString() {
		return NomeCognome;
	}

}
```


Aggiunta dei metodi alle classi persistenti

```
public double mediaEsami() {
    Iterator iter = this.getEsami().iterator();
    int sommavoti = 0; int numvoti = 0;
    while (iter.hasNext()) {
        Esame esame = (Esame) iter.next();
        sommavoti += esame.getVoto();
        numvoti += 1;
    }
    return sommavoti/(double)numvoti;
}
```


Nuovi costruttori

```
public Esame(Studente studente, String materia, int voto, boolean lode) {
    setCandidato(studente);
    setMateria(materia);
    setVoto(voto);
    if (voto==30) setLode(lode);
    setData(new Date());
    studente.getEsami().add(this);
}
```


Esempio

```
public class Main {

    public static void main(String[] args) throws Exception {

        Configuration cfg = new Configuration()
            .addClass(Esame.class)
            .addClass(Studente.class);
        SessionFactory factory = cfg.buildSessionFactory();
        Session s = factory.openSession();
        Transaction tx = null;

        try {
            // aggiungi un esame allo studente con matricola 6592
            tx = s.beginTransaction();
            Studente studente =
                (Studente) s.createQuery("from Studente as s where s.Matricola = :mat")
                            .setLong("mat", 6592)
                            .uniqueResult();
            new Esame(studente,"BD",30,true);
            tx.commit();
```

```
        // per ogni studente del 2 anno, calcola la media degli esami
        tx = s.beginTransaction();
        Iterator studIterator =
            s.createQuery("from Studente where AnnoCorso = :ac")
             .setDouble("ac", 2).iterate();
        while (studIterator.hasNext()) {
            studente = (Studente)(studIterator.next());
            System.out.println(studente.getNomeCognome() + " ha media "
                               + studente.mediaEsami());
        }
        tx.commit();
    }
    catch (Exception e) {
        if (tx!=null) tx.rollback();
        throw e;
    }
    finally {
        s.close();
    }
    factory.close();
```


Aspetti del “mapping”

- Hibernate distingue fra oggetti “entità” e oggetti “valore”
- La differenza è sull’identità
- Un oggetto valore è “posseduto” da un oggetto entità (o altro oggetto valore)
- Si possono avere liste, set, bag, maps di oggetti valore (attraverso associazioni)
- La persistenza può valere per entrambi: persistenza per raggiungibilità

Mapping - Oggetti non entità (valore)

- Persona può avere indirizzo di spedizione e indirizzo di pagamento
- Si può avere un oggetto persona con due oggetti indirizzo (non entità)

```
<class name="Persona" table="PERSONE">
  <id name="id" type="long"><generator class="native"/></id>
  <component name=indirizzoSpedizione class="Indirizzo">
    <property name="via" colonna="SPEDIZ_VIA" not-null="true"/>
    <property name="citta".../>
  </component>
  <component name=indirizzoPagamento class="Indirizzo">
    <property name="via" colonna="PAGAM_VIA" not-null="true"/>
    <property name="citta".../>
  </component>
  ...
</class>
```


Mapping: Associazioni uno-a-molti

- È necessario dichiarare la diretta e l’inversa, e la relazione fra di loro
- È necessario dichiarare la cardinalità dell’associazione
- È necessario definire la persistenza transitiva
- Si può definire la “dipendenza”
- Gestione della strategia di “fetch”

```
<class name="Esame" table="ESAMI">
  ...
  <many-to-one name="studente" column="CANDIDATO" class="Studente" not-null="true" lazy="true"/>
  ...
</class>

<class name="Studente" table="STUDENTI">
  ...
  <set name="esami" one-to-many class="Esame" inverse="true"
       key column="CANDIDATO" cascade="save-update"/> oppure cascade="all-delete-orphan"
  </set>
</class>
```

Associazioni uno-a-uno

- Caso particolare di uno a molti
  - Aggiunta di un vincolo di unicità
  - Specifica dell'associazione inversa come uno a uno.

```
<class name="Dipartimento" table="DIPARTIMENTI">
  ...
  <many-to-one name="direttore" column="DIRETTORE" class="Professori"
   cascade="save-update" unique="true"/>
  ...
</class>

<class name="Professore" table="PROFESSORI">
  ...
  <one-to-one name="dipartimentoDiretto" class="Dipartimento"
     property-ref="direttore"/>
  ...
</class>
```


Associazione molti a molti

- La tabella intermedia è invisibile dal programma
- Si dichiarano set da entrambe le parti

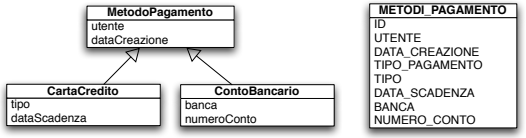
```
<class name="Studente" table="STUDENTI">
  ...
  <set name="CorsiFrequentati" table="STUDENTI_CORSI"
       lazy="true" cascade="save-update">
    <key column="MATRICOLA"/>
    <many-to-many class="Corso" column="NOME_CORSO"/>
  </set>
  ...
</class>

<class name="Corso" table="CORSI">
  ...
  <set name="studenti" table="STUDENTI_CORSI"
       lazy="true" inverse="true" cascade="save-update">
    <key column="NOME_CORSO"/>
    <many-to-many class="Studente" column="MATRICOLA"/>
  </set>
  ...
</class>
```


Gerarchie di classi

- Si può avere una gerarchia che viene mappata nelle tre modalità classiche:
  - Tabella unica
  - Partizionamento verticale
  - Partizionamento orizzontale

Tabella unica



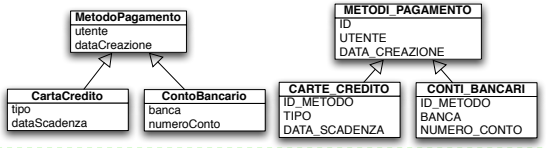
- Tabella unica con attributo discriminatore

```
<class name="MetodoPagamento" table="METODI_PAGAMENTO"
      discriminator-value="MP">
  <id name="id" type="long"><generator class="native"/></id>
  <discriminator column="TIPO_PAGAMENTO" type="string"/>
  <property name="utente" column="UTENTE" type="string"/>...
  <subclass name="CartaCredito" discriminator-value="CC">
    <property name="tipo" column="TIPO" type="string"/>
    ...
  </subclass>
  <subclass name="ContoBancario" ...>
```

Tabella unica

- Svantaggi:
  - si perdono i vincoli di valori nulli
  - si perde il vincolo della chiave esterna verso una sottoclasse
  - se la gerarchia è complessa si possono avere molti valori nulli

Partizionamento verticale



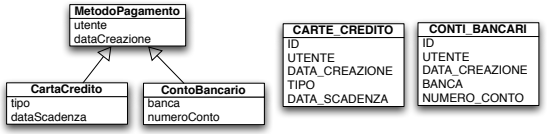
- Una tabella per classi e sottoclassi

```
<class name="MetodoPagamento" tabble="METODI_PAGAMENTO">
  <id name="id" type="long"><generator class="native"/></id>
  <property name="utente" column="UTENTE" type="string"/>
  <joined-subclass name="CartaCredito" table="CARTE_CREDITO">
    <key column="ID_METODO">
    <property name="tipo" column="TIPO" type="string"/>
    ...
  </joined-subclass>
  <joined-subclass name="ContoBancario" ...>
```


Problemi

- Complessa la generazione del codice da parte di Hibernate
  - Report di errori poco comprensibile
- Prestazioni possono creare problemi per gerarchie complesse

Partizionamento orizzontale



- Tabelle non collegate: le due tabelle non hanno alcuna interdipendenza
- Problemi
  - Associazioni con la superclasse non si possono fare (occorre avere due associazioni diverse)
  - Complessa la generazione del codice SQL
  - Rischio di perdere la sincronizzazione quando lo schema evolve
  - Non si può fare se ci sono elementi in comune nelle sottoclassi

Altri aspetti

- Gestione della granularità del mapping e dei tipi di dati coinvolti
- Gestione delle strategie di fetching (lazy - eager) sia nel mapping sia dinamicamente
- “Criteri” come classe per costruire query tipizzate
- Gestione delle transazioni più complessa (anche con approccio “ottimistico”)
- Caching dei dati

Gestione delle transazioni “ottimistica”

- Gestione *ottimistica* significa che alla terminazione di una transazione “interna” si salvano i dati nella base di dati ma controllando che i valori iniziali siano ancora presenti
- Se i valori sono cambiati, la transazione fallisce (viene fatta fallire esplicitamente dal gestore del mapping)
- Come si realizza?
  - Si salva una copia dei valori al momento della lettura (snapshot)
  - Quando si scrivono le modifiche si fa con:

    ```
    update tabella set att1 = nuovoVal1, att2 = nuovoVal2, ...
    where attkey = k1 AND att1 = vecchioVal1, att2 = vecchioVal2, ...
    ```

  - Si controlla che il risultato produca 1 (e non 0) (ennuple modificate)