



Universidade do Minho
Departamento de Informática

Engenharia de Aplicações

ORM: Hibernate

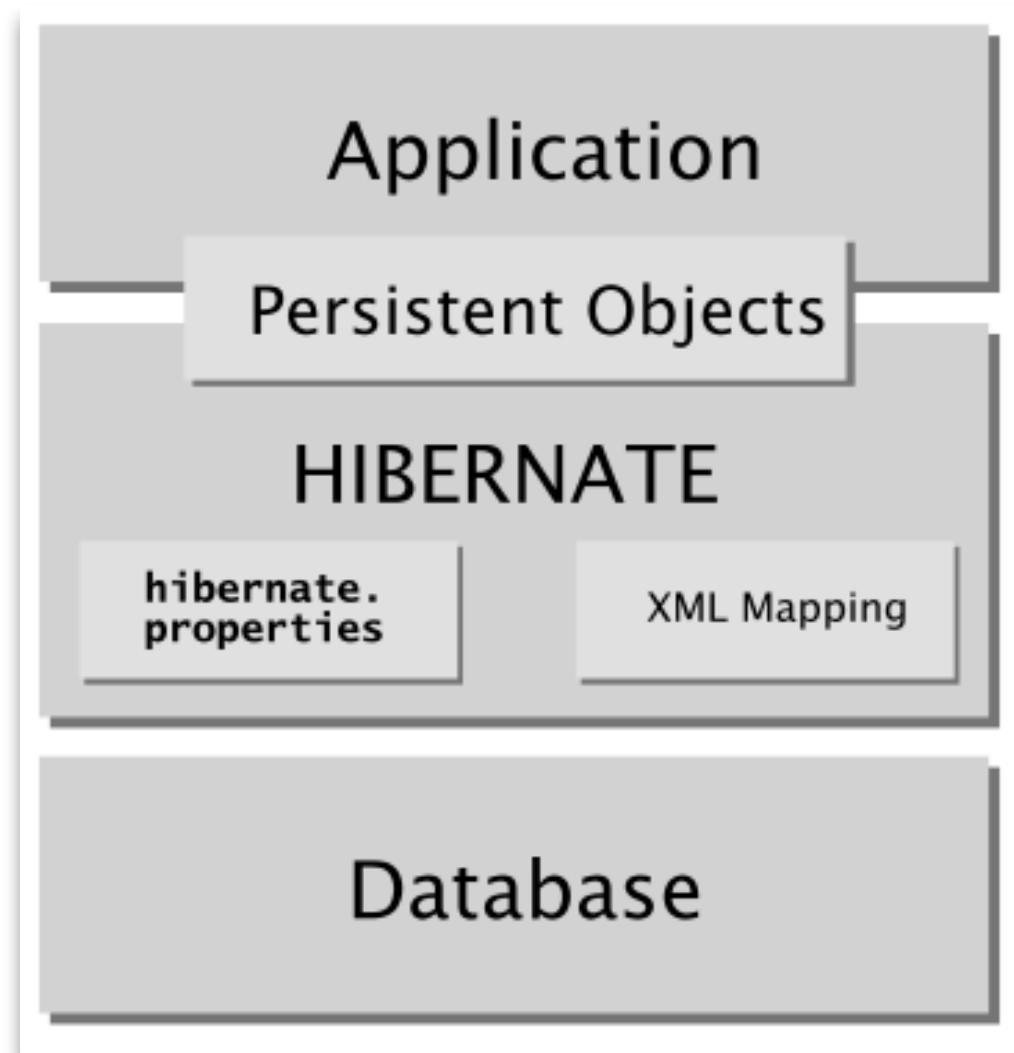
António Nestor Ribeiro
anr@di.uminho.pt

Introdução

- Hibernate é uma ferramenta de transformação Object-Relational, para a framework Java (e não só)
 - Possibilita a automatização do mapeamento de classes Java para tabelas de uma base de dados relacional
 - Disponibiliza uma linguagem de query, de alto nível, que permite interagir com a base de dados
- Além de especificar o mapeamento entre as classes e a BD, permite também estabelecer um nível de middleware entre as diferentes camadas
 - Esconde a utilização de tecnologias específicas, como o JDBC
 - Permite portabilidade entre diferentes produtos de base de dados
 - Tem um acréscimo de peso computacional

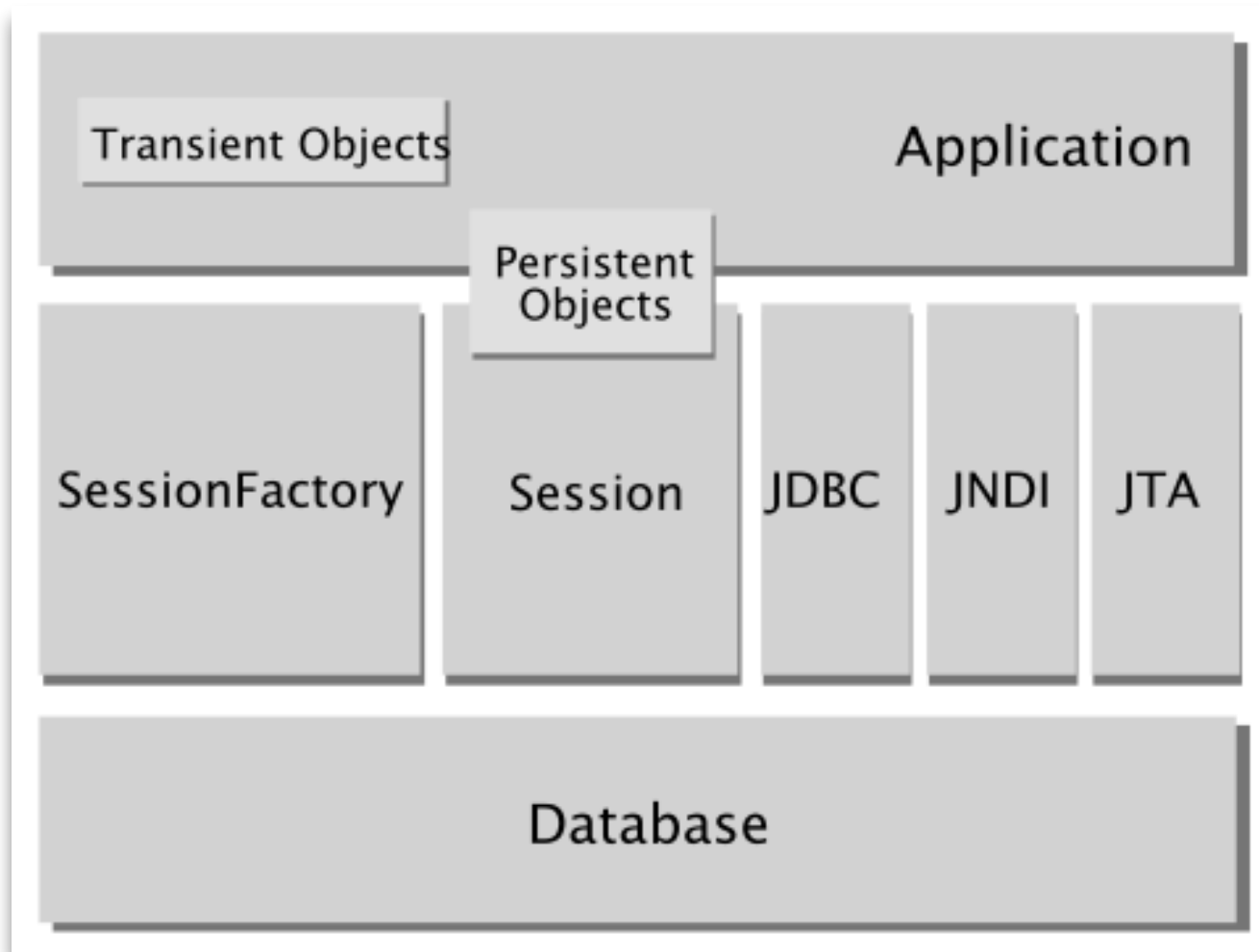
Arquitetura

- Arquitetura Geral



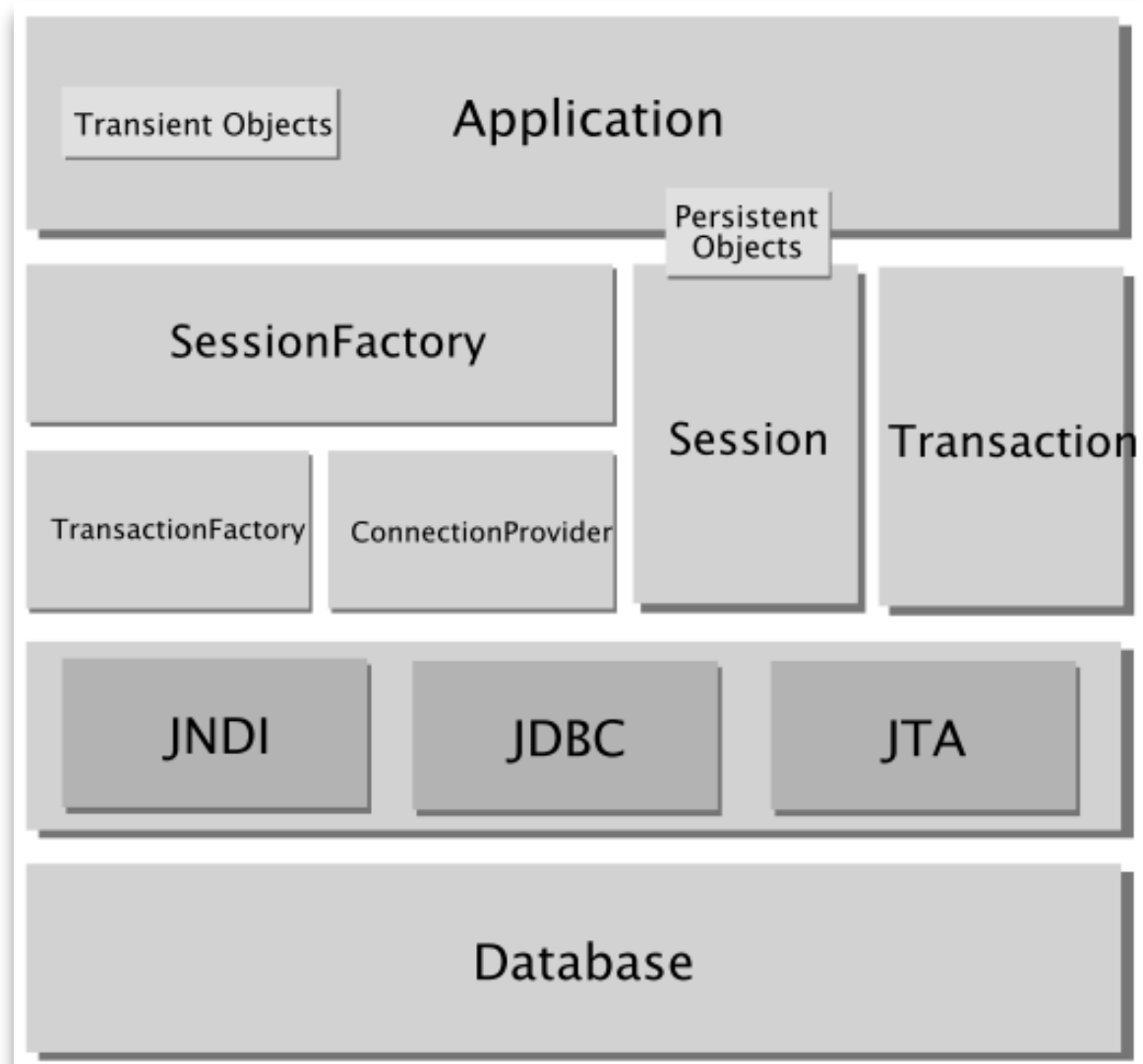
Arquitetura

- Detalhe da camada de transformação: versão 1



Arquitetura

- Detalhe da camada de transformação: versão 2



Arquitectura

- Session Factory
 - Funciona como uma cache de mapeamentos para uma determinada base de dados.
 - Permite criar sessões (instância de Session) e funciona como um cliente de um serviço de conexão à base de dados
- Session
 - objecto transiente de curta duração
 - permite a troca de informação entre a aplicação e a base de dados
 - encapsula uma ligação JDBC e permite criar transacções
 - contém uma cache, de primeiro nível, de referências a objectos
 - é possível
 - ter que uma sessão corresponde a uma transacção na BD
 - ter mais do que uma transacção por sessão

Arquitetura

- Persistent Objects and Collections
 - objectos, de curta duração, que possuem estado persistente e funcionalidades da camada de negócio.
 - são objectos Java *normais*, mas que estão associados a uma sessão e possuem uma identidade persistente (valor da chave primária)
 - para uma determinada sessão o Hibernate garante que a identidade de persistência é equivalente à identidade do objecto Java (em memória)
 - quando a sessão termina, deixam de estar associados à sessão e podem ser utilizados pela aplicação
- Transient and Detached Objects and Collections
 - instâncias de objectos persistentes que, no momento, não estão associados a nenhuma sessão.
 - Transient:
 - o objecto nunca foi associado a uma sessão e não possui qualquer identidade persistente
 - Detached:
 - o objecto já esteve associado a uma sessão, mas de momento a sessão terminou. Tem identidade persistente e uma entrada na base de dados.

Arquitectura

- Para além dos componentes base, existe na arquitectura um conjunto de componentes opcionais:
- Transaction
 - objecto utilizado para especificar transacções
 - permite abstrair a utilização de JDBC, JTA ou CORBA
 - uma sessão pode, se necessário, criar várias transacções
 - a aplicação tem de decidir se usa o suporte transaccional do Hibernate ou das frameworks JDBC, JTA ou CORBA
- Connection Provider
 - fornece uma factory para criar conexões JDBC
 - permite criar *pools* de conexões e efectuar a sua gestão
 - permite isolar a aplicação de conceitos como DriverManager ou DataSource
- TransactionFactory
 - fornece uma fábrica de transacções
- Extension Interfaces
 - interfaces, opcionais, que pode ser implementados para customizar o comportamento da camada persistente

Configuração e Mapeamento

- Alternativas de configuração do Hibernate:
 - ficheiro XML de configuração
 - anotações no código
- Em qualquer um dos casos podem ser configurados parâmetros como:
 - ligações via JDBC
 - transacções
 - propriedades da cache do hibernate
 - logging
 - mapeamento das classes para relações
 - dialecto de SQL que vai ser utilizado
 - etc.

Configuração e Mapeamento

- Ficheiro XML de configuração
 - cada classe a mapear deve ter um atributo único que irá ser mapeado numa chave primária

```
<hibernate-mapping>

    <class name="events.Event" table="EVENTS">

    </class>

</hibernate-mapping>
```

```
<hibernate-mapping>

    <class name="events.Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="native"/>
        </id>
    </class>

</hibernate-mapping>
```

Configuração e Mapeamento

- No ficheiro XML é possível definir:
 - chaves estrangeiras
 - nomes das colunas
 - tipos de dados SQL correspondentes a cada tipo de dados Java
 - tipos de associação:
 - bidireccional
 - unidireccional
 - de “um para um”
 - de “um para muitos”
 - de “muitos para muitos”
- Composição de classes
 - O Hibernate resolve o problema criando mais que uma relação e relacionando-as com as associações necessárias
 - Caso a associação seja de “muitos para muitos”, pode ser criada uma relação extra que associa as chaves primárias das várias tabelas.
 - O utilizador apenas tem de especificar no ficheiro de mapeamento que tipo de realidade é que está a modelar

Configuração e Mapeamento

- O utilizador descreve se é uma colecção e se é composição de objectos e escolhe o tipo de associação.

```
package events;

public class Person {

    private Long id;
    private int age;
    private String firstname;
    private String lastname;

    public Person() {}

    // Accessor methods for all properties, private setter for 'id'

}
```

- A classe deve ter os **get** e **set** codificados.

Configuração e Mapeamento

- O ficheiro de configuração, chamado **Person.hbm.xml**, tem o seguinte aspecto:

- ```
<hibernate-mapping>

 <class name="events.Person" table="PERSON">
 <id name="id" column="PERSON_ID">
 <generator class="native"/>
 </id>
 <property name="age"/>
 <property name="firstname"/>
 <property name="lastname"/>
 </class>

</hibernate-mapping>
```

# Configuração e Mapeamentos

- Uma pessoa assiste a muitos eventos (instância de Event) e um evento é frequentado por muitas pessoas (instância de Person)

```
public class Person {

 private Set events = new HashSet();

 public Set getEvents() {
 return events;
 }

 public void setEvents(Set events) {
 this.events = events;
 }
}
```

# Configuração e Mapeamento

```
<class name="events.Person" table="PERSON">
 <id name="id" column="PERSON_ID">
 <generator class="native"/>
 </id>
 <property name="age"/>
 <property name="firstname"/>
 <property name="lastname"/>
```

```
 <set name="events" table="PERSON_EVENT">
 <key column="PERSON_ID"/>
 <many-to-many column="EVENT_ID" class="events.Event"/>
 </set>
```

```
</class>
```

# Configuração e Mapeamento

- Quando existe a necessidade que a associação seja bidireccional

```
<set name="participants" table="PERSON_EVENT" inverse="true">
 <key column="EVENT_ID"/>
 <many-to-many column="PERSON_ID" class="events.Person"/>
</set>
```

- Além de <set>, existem como tipos de associações:
  - list
  - map
  - bag
  - array
  - primitive-array



# Mapeamento

- Unidireccional - muitos para um

```
<class name="Person">
 <id name="id" column="personId">
 <generator class="native"/>
 </id>
 <many-to-one name="address"
 column="addressId"
 not-null="true"/>
</class>

<class name="Address">
 <id name="id" column="addressId">
 <generator class="native"/>
 </id>
</class>
```

# Mapeamento

- um para um

```
<class name="Person">
 <id name="id" column="personId">
 <generator class="native"/>
 </id>
</class>

<class name="Address">
 <id name="id" column="personId">
 <generator class="foreign">
 <param name="property">person</param>
 </generator>
 </id>
 <one-to-one name="person" constrained="true"/>
</class>
```

# Mapeamento

- um para muitos

```
<class name="Person">
 <id name="id" column="personId">
 <generator class="native"/>
 </id>
 <set name="addresses">
 <key column="personId"
 not-null="true"/>
 <one-to-many class="Address"/>
 </set>
</class>

<class name="Address">
 <id name="id" column="addressId">
 <generator class="native"/>
 </id>
</class>
```

# Mapeamento

- muitos para muitos

```
<class name="Person">
 <id name="id" column="personId">
 <generator class="native"/>
 </id>
 <set name="addresses" table="PersonAddress">
 <key column="personId"/>
 <many-to-many column="addressId"
 class="Address"/>
 </set>
</class>

<class name="Address">
 <id name="id" column="addressId">
 <generator class="native"/>
 </id>
</class>
```

# Mapeamento da Herança

- As ferramentas de ORM costumam implementar as seguintes estratégias:
  - uma relação por hierarquia
  - uma relação por subclasse
  - uma relação por subclasse com discriminador
  - uma relação por classe concreta
- Uma relação por hierarquia
  - neste caso é apenas criada uma relação para descrever a classe e respectivas subclasses.
  - a limitação deste mapeamento é que as colunas das subclasses que não pertencem à superclasse devem poder ser nulas
  - caso exista muita informação a quantidade de nulls pode sobrecarregar a relação

# Herança e Polimorfismo

- Classe Payment com subclasses CreditCardPayment, CashPayment e ChequePayment

```
<class name="Payment" table="PAYMENT">
 <id name="id" type="long" column="PAYMENT_ID">
 <generator class="native"/>
 </id>
 <discriminator column="PAYMENT_TYPE" type="string"/>
 <property name="amount" column="AMOUNT"/>
 ...
 <subclass name="CreditCardPayment" discriminator-value="CREDIT">
 <property name="creditCardType" column="CCTYPE"/>
 ...
 </subclass>
 <subclass name="CashPayment" discriminator-value="CASH">
 ...
 </subclass>
 <subclass name="ChequePayment" discriminator-value="CHEQUE">
 ...
 </subclass>
</class>
```

# Herança e Polimorfismo

- Uma relação por subclasse
  - neste caso é criada uma relação para cada classe da hierarquia
  - as relações das subclasses têm chaves primárias que estão relacionadas com a relação de superclasse, numa lógica “um para um”

```
<class name="Payment" table="PAYMENT">
 <id name="id" type="long" column="PAYMENT_ID">
 <generator class="native"/>
 </id>
 <property name="amount" column="AMOUNT"/>
 ...
 <joined-subclass name="CreditCardPayment"
table="CREDIT_PAYMENT">
 <key column="PAYMENT_ID"/>
 <property name="creditCardType" column="CCTYPE"/>
 ...
 </joined-subclass>
 <joined-subclass name="CashPayment" table="CASH_PAYMENT">
 <key column="PAYMENT_ID"/>
 ...
 </joined-subclass>
 <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
 <key column="PAYMENT_ID"/>
```

# Herança e Polimorfismo

- Uma relação por subclasse com discriminador
  - esta implementação utiliza uma coluna adicional nas relações que classifica a classe
  - semelhante à alternativa anterior, mas mais simples de implementar

```
<class name="Payment" table="PAYMENT">
 <id name="id" type="long" column="PAYMENT_ID">
 <generator class="native"/>
 </id>
 <discriminator column="PAYMENT_TYPE" type="string"/>
 <property name="amount" column="AMOUNT"/>
 ...
 <subclass name="CreditCardPayment" discriminator-value="CREDIT">
 <join table="CREDIT_PAYMENT">
 <key column="PAYMENT_ID"/>
 <property name="creditCardType" column="CCTYPE"/>
 ...
 </join>
 </subclass>
 <subclass name="CashPayment" discriminator-value="CASH">
 <join table="CASH_PAYMENT">
 <key column="PAYMENT_ID"/>
 ...
 </join>
 </subclass>
 <subclass name="ChequePayment" discriminator-value="CHEQUE">
 <join table="CHEQUE_PAYMENT" fetch="select">
 <key column="PAYMENT_ID"/>
 ...
 </join>
 </subclass>
</class>
```



# Herança e Polimorfismo

- Uma relação por classe concreta
  - perde-se a noção de hierarquia e família que existe no mundo Java
  - se uma propriedade é mapeada na superclasse tem de existir nas relações das subclasses

```
<class name="Payment">
 <id name="id" type="long" column="PAYMENT_ID">
 <generator class="sequence"/>
 </id>
 <property name="amount" column="AMOUNT"/>
 ...
 <union-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
 <property name="creditCardType" column="CCTYPE"/>
 ...
 </union-subclass>
 <union-subclass name="CashPayment" table="CASH_PAYMENT">
 ...
 </union-subclass>
```