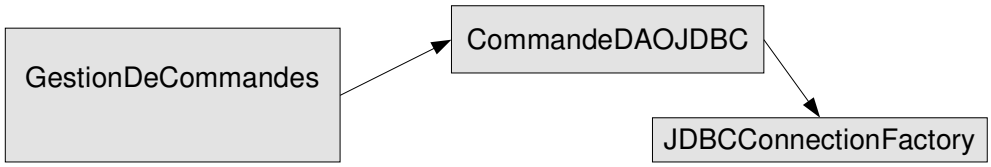
Le framework Spring
DUT info S4

Définitions

- Bibliothèque
  - peu d'incidence sur l'architecture des applications (facile à remplacer si code bien écrit)
- Framework
  - fournissent l'architecture d'une partie de l'application

Le problème des dépendances

- On souhaite avoir des composants les plus indépendants possibles. Mais :



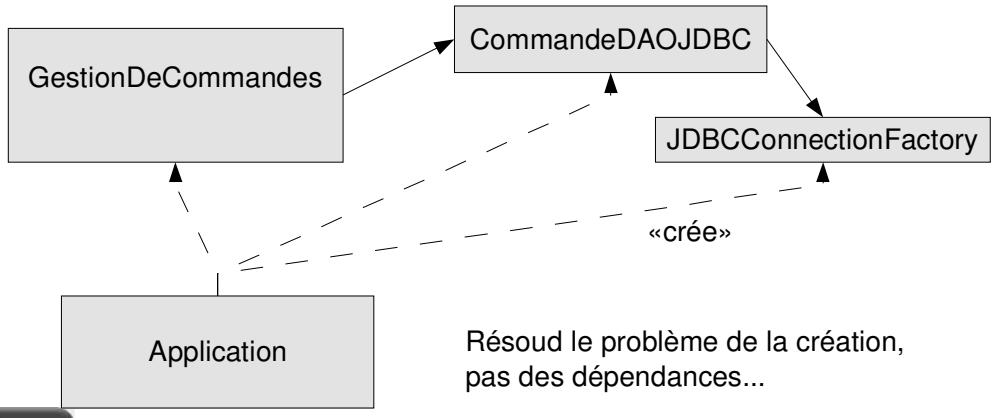
- La classe GestionDeCommande dépend de CommandeDAOJDBC et de JDBCConnectionFactory

Dépendances

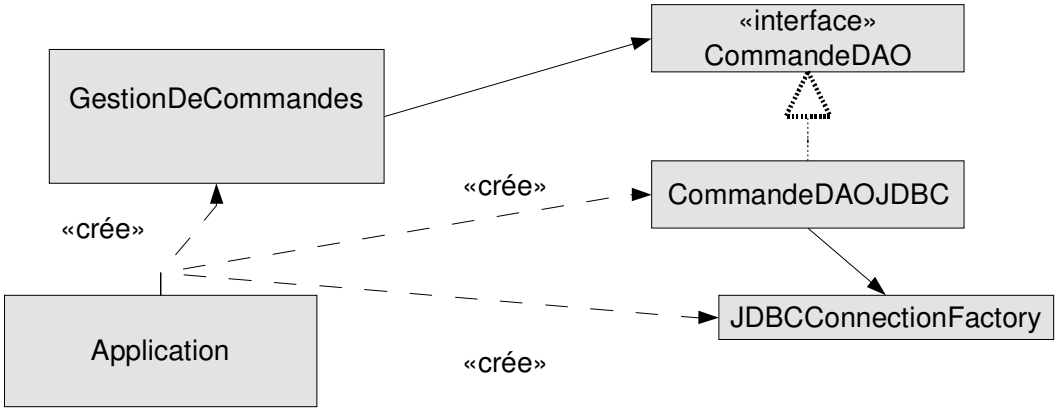
- C'est un problème, car
  - il est alors impossible de proposer plusieurs solutions de sauvegarde (XML, hibernate...) sans modifier GestionDeCommande
  - il est impossible de *tester* GestionDeCommande sans avoir écrit CommandeDAOJDBC
  - le test est compliqué : d'où proviennent les erreurs éventuelles ?
  - qui crée CommandeDAOJDBC ? qui crée GestionDeCommande ?

Solution 1

- Dans le programme de début d'année : l'application s'occupait de « cabler » les différents composants.



Solution 2



- Variante : GestionDeCommande admet ne dépend plus de CommandeDAOJDBC, mais seulement de son interface.

Inversion de contrôle

- Cette technique s'appelle Inversion de contrôle
- ou principe d'Hollywood : *ne nous appelez pas, nous vous appellerons*
- dans spring : injection de dépendance
- On remplace:

  dans GestionDeCommande

  dao= new CommandeDAOJDBC();

  par l'appel de la méthode setDAO(...) de GestionDeCommande.

Spring

- framework *léger :* impose peu de contraintes aux classes écrite par le programmeur
- basé sur l'inversion de contrôle
- framework modulaire pour traiter des aspects variés d'une application : couche web, client riche, persistence, transactions, sécurité...
- support de la programmation orientée aspect, en particulier pour les transactions.

Les beans

- Éléments de base des applications Spring. En gros, pour spring, des objets java dotés d'accesseurs.
- Spring permet de paramétrer les beans de manière déclarative (par ex. dans un fichier XML)

Création de l'infrastructure par XML

- Fichier de définition des beans

```
<beans>
    <bean id='commandeDAO' class='CommandeDAOJDBC'>
        <property name='connectionFactory' ref='cf'/>
    </bean>

    <bean id='cf' class='ConnectionFactory'>
        <property name='url' value='jdbc:hsqldb:hsql://localhost/db'/>
        .....
    </bean>

    <bean id='gestionDeCommande'>
        <property name='dao' ref='commandeDAO'/>
    </bean>
</beans>
```

Spring et les applications web

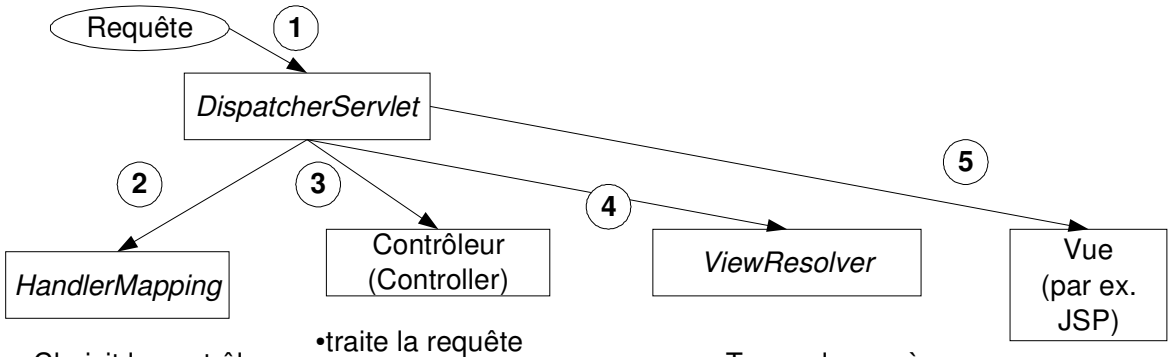
- Une application Spring web utilise typiquement la servlet

  org.springframework.web.servlet.DispatcherServlet

  (pattern dit « Front Controller »).
- Cette servlet, définie dans web.xml, a un nom, par exemple « monAppli ».
- le fichier de configuration des beans est alors

  monAppli-servlet.xml

Étapes du traitement d'une requête



Choisit le contrôleur

- traite la requête
- crée un objet modelAndView
- rempli de « modèle »
- choisit le *nom* de la vue

Trouve la vue à afficher à partir du *nom*.

classe fournie par Spring
classe écrite par le programmeur

Exemple de configuration (1)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
    <!-- Ensemble simple, mais complet et explicite, de beans pour une application web -->
    <!-- Un contrôleur -->
    <bean id="affiche4" class="simple.AfficheQuatreController"></bean>
    <!-- L'infrastructure -->
    <!-- Associe un contrôleur à une requête -->
    <bean id="handlerMapping"
            class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <value>
                /**/quatre=affiche4
            </value>
        </property>
    </bean>
    <!-- Associe à un nom de vue la jsp correspondante -->
    <bean id="viewResolver"
            class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
    </bean>
</beans>
```


Exemple de configuration (2)

```
<!-- un contrôleur -->
<bean name="/afficheQuatre" class="simple.AfficheQuatreController">
</bean>

<!-- L'infrastructure -->

<!-- Associe un contrôleur à une requête par le nom du contrôleur -->
<bean id="handlerMapping"
      class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">
</bean>

<!-- Associe à un nom de vue la jsp correspondante -->
<bean id="viewResolver"
          class="org.springframework.web.servlet.view.InternalResourceViewResolver">
 <property name="prefix" value="/WEB-INF/pages/"/>
 <property name="suffix" value=".jsp"/>
</bean>
```

Exemple de contrôleur

```
public class AfficheQuatreController extends AbstractController {


    @Override
    protected ModelAndView
        handleRequestInternal(HttpServletRequest request,
                              HttpServletResponse response)
    throws Exception {
        ModelAndView nombre=
            new ModelAndView("afficherNombre", "nombre", new Integer(4));
        return nombre;
    }
}
```

nom de la vue

nom d'un bean request

valeur du précédent

« Convention over Configuration » en Spring 2.0

- Inspiré de frameworks comme *Ruby on Rails*
- Pour les cas simples, on remplace la *configuration* XML par l'application de *conventions*, en gros :
- pour l'URI « /appli/affichequatre »
  - contrôle: AfficheQuatreController
  - vue « normale » : afficheQuatre

Conventions (de base)

- pour les contrôleurs simples :
  - le nom de la classe se termine par « Controller »
  - url =

  base + « / » + nom du contrôleur en *minuscule* sans « Controller ».

  AfficheQuatreController → /appli/affichequatre

Conventions (suite)

- Éléments du modèle
  - Dans le contrôleur (p.ex. **AfficheQuatre**Controller) :

  ModelAndView m= new ModelAndView();

  → la vue s'appellera « **afficheQuatre** »

  Personne p= ....; m.addObject(**p**);

  → l'élément du modèle s'appelle automatiquement « **personne** ».

  Set<Personne> tab= ....; m.addObject(tab);

  → l'élément s'appelle **personneList**

SimpleFormController

- Gère un formulaire
- les données sont sauvées dans un bean request, appelé la commande.

```
<bean class="facture.controle.CreerArticleController">
        <property name="commandClass" value="facture.modele.Article" />
        <property name="commandName" value="article"/>
        <property name="formView" value="creerArticle"/>
        <property name="successView" value="menu"/>
        <!-- «  valideur  » optionnel -->
        <property name="validator" ref="articleValidator"/>
        <!-- façade de la couche métier -->
        <property name="facturationFacade" ref="facturationFacade"/>
</bean>
```


SimpleFormController

```
public class CreerArticleController extends SimpleFormController {

    private FacturationFacade facturationFacade;

    @Override
    protected void doSubmitAction(Object command) throws Exception {
        Article a= (Article) command;
        facturationFacade.saveArticle(a);
    }

    public FacturationFacade getFacturationFacade() {
        return facturationFacade;
    }

    public void setFacturationFacade(FacturationFacade facturationFacade) {
        this.facturationFacade = facturationFacade;
    }
}
```

Les tags « form »

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<html>
<head>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Create user</title>
</head>
<body>
<form:form commandName="article">
    <p>Nom :<form:input path="designation" />
    <p>Adresse :<form:input path="prix" />
     <p><input type="submit" value="sauver" />
</form:form>
</body>
</html>
```

lit et écrit dans le bean «article» (c.f. configuration)

manipule la propriété article.prix

Validation (optionnelle)

```
public class ArticleValidator implements Validator {

    /**
     * Retourne vrai si le validateur gère la classe passée en argument.
     */
    public boolean supports(Class clazz) {
        return (clazz.equals(Article.class));
    }
    // accumule des erreurs si l'objet est mal formé.
    public void validate(Object target, Errors errors) {
        Article a= (Article) target;
        ValidationUtils.rejectIfEmpty(errors, "designation", "empty_field","empty field!");
        if (a.getPrix() <= 0)
            errors.rejectValue("prix", "incorrect_price", "incorrect price");
    }
}
```

Spring et les bases de données

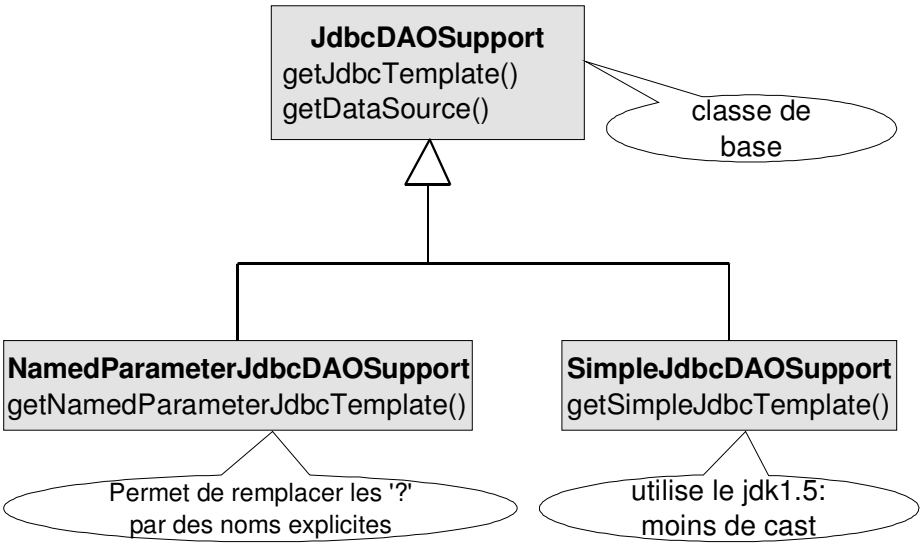
- DAOs, avec versions spécialisées : JdbcDaoSupport, HibernateDaoSupport...
- Exceptions unifiées et sémantiquement riches
- Mécanismes transactionnels

Spring et JDBC

- Les DAO étendent JDBCDAOSupport (ou NamedParameterJDBCDAOSupport)
- Les DAO sont des singletons
- La communication se fait à partir de JDBCTemplates.

```
// Le throws n'est pas obligatoire.
void createArticle(Article a) throws DataAccessException
{
      String sql= "insert into Article values (?, ?, ?);
      getJdbcTemplate().update(sql,
            new Object[] {a.getId(), a.getDescr(), a.getPrix()});
}
```

Classes de base pour les DAO



Configuration simple du dataSource

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
      <property name="driverClassName">
            <value>org.hsqldb.jdbcDriver</value>
      </property>
      <property name="url">
            <value>jdbc:hsqldb:hsql://localhost/db</value>
      </property>
      <property name="username">
            <value>sa</value>
      </property>
      <property name="password">
            <value></value>
      </property>
</bean>
```

Modification de la base

```
class JDBCArticleDAO implements ArticleDAO extends JdbcDAOSupport {

      public void deleteArticle(int id) {
            String sql= "delete from article where idArticle= ?";
            getJdbcTemplate().update(sql, new Object[] {Integer.valueOf(id)});
      }
}
```


Requête simple

```
class JDBCArticleDAO implements ArticleDAO extends JdbcDAOSupport {

      public void getMaxPrix() {
            String sql= "select max(prix) from article";
            return getJdbcTemplate().queryForInt(sql);
      }
}
```

Requêtes renvoyant un objet

- pb: extraction de l'objet depuis le resultset.
- Solution : utilisation d'un « Mapper »
- Deux versions, 1.5 et 1.4.

```
class ArticleMapper implements ParameterizedRowMapper<Article> {
        public Article mapRow(ResultSet rs, int rowNum) throws SQLException {
            Article a = new Article();
            a.setId(rs.getLong("id"));
            a.setDescription(rs.getString("descr"));
            a.setPrix(rs.getDouble("prix"));
            return a;
        }
    }
```


Requêtes renvoyant un objet

```
public Article findArticle(long id) {
    String sql = "select *  from Article where id = ?";

    SimpleJdbcTemplate simpleJdbcTemplate = new SimpleJdbcTemplate(this.getDataSource());
      return simpleJdbcTemplate.queryForObject(sql, new ArticleMapper(), id);
}
```

Méthode à nombre variable d'arguments (jdk 1.5)

Requêtes renvoyant *des* objets

```
public List<Article> findArticleMoinsCherQue(double prix) {
    String sql = "select *  from Article where prix < ?";
    SimpleJdbcTemplate simpleJdbcTemplate = new SimpleJdbcTemplate(this.getDataSource());
      return simpleJdbcTemplate.query(sql, new ArticleMapper(), prix);
}
```


Requête renvoyant des maps ou des listes

- Utiles en développement de prototype, en l'absence de modèle
- Les objets sont remplacés par des Maps<String, Object>. Pour chaque Article, on aurait
  - m["id"]= id de l'article
  - m["descr"]= description de l'article
  - m["prix"]= prix de l'article

NamedParameterJdbcTemplate

- permet de remplacer les « ? » par des noms du type :nomDeChamp
- Requêtes plus robustes et plus lisibles
- Possibilité de récupérer les valeurs dans une Map, ou automatiquement grâce à la classe BeanPropertySqlParameterSource

NamedParameterJdbcTemplate (exemple)

```
public int findStock(Article article) {
   String sql = "select sum(qte) from magasin where idArticle = :id";

   NamedParameterJdbcTemplate template =
                    new NamedParameterJdbcTemplate(this.getDataSource());
   SqlParameterSource param = new BeanPropertySqlParameterSource(article);

   return template.queryForInt(sql, param);
}
```

Ou

```
public int findStock(Article a) {
   String sql = "select sum(qte) from magasin where idArticle = :id";
   NamedParameterJdbcTemplate template =
                    new NamedParameterJdbcTemplate(this.getDataSource());
   Map param= new HashMap();
   param.put("id", a.getId());
   return template.queryForInt(sql, param);
}
```

Aspect Oriented Programming

- Ajout orthogonal de fonctionnalités.
- En gros, permet de demander l'exécution d'un bout de code avant, après, ou « autour » d'une exécution de méthode.
- Utilisations : sécurité, mise à jour d'interface, log, transactions...

Vocabulaire AOP

- point de coupe : endroit du programme (généralement une méthode)
- coupe : ensemble de points de coupes
- greffon *(advice)* : code à exécuter pour tous les appels d'une coupe
- tissage : association d'un greffon à une coupe.

Types de greffons :

- **avant** : exécuté avant un appel de méthode
- **après** : exécuté après un appel de méthode
- **autour** : exécuté avant *et* après un appel. Peut empêcher l'appel en question.
- **exception** : exécuté en cas de levée d'exception

Gestion des transactions

- Pb des transactions:
  - élément techniquement relié à la couche de persistance (« jdbcConnection.commit() » ???)
  - mais architecturalement relié à la couche de services : typiquement, une action dans un use case doit être dans une transaction unique.
  - pb. des transactions imbriquées (pas tjrs possibles)
- Comment concilier les deux ?
- Comment éviter la répétition de code ?

Spring, AOP et transaction

- Spring permet de spécifier que certaines méthodes de certaines classes seront exécutées dans une transaction
- En cas de levée d'exception, la transaction peut être annulée (rollback)
- Ce mécanisme utilise la programmation aspect.

Transactions et annotations

- Le mécanisme de java 1.5 permet un traitement très simple :

```
import org.springframework.transaction.annotation.*;

@Transactional
public class  FactureService {
      // Cest méthodes sont transactionnelles.
      public void sauverFacture() {...}
      public void detruireFacture() {...}
      // On précise que la méthode qui suit est read-only
      @Transactional(readOnly = true)
      public FactureDTO getFactureData()  {...}
}
```

Configuration correspondante

```
<!-- Démonstration de bean transactionnel -->
<bean id="factureService" class="demo.FactureService"></bean>

<!-- demande l'utilisation des annotations -->
<tx:annotation-driven transaction-manager="txManager" />

<!-- gestionnaire des transactions -->
<bean id="txManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
   <property name="dataSource" ref="dataSource"/>
</bean>
```


Commentaire

- très simple
- nécessite jdk1.5+
- informations de transaction dans la classe elle-même, mais peu invasif, et global (on peut déclarer d'un coup *toute* la classe comme transactionnelle).

Déclaration complète

```
<!-- La facade à rendre transactionnelle -->
<bean id="monService" class="monProg.MonService"/>

<aop:config> <!-- config aspect -->
  <!-- toute méthode de MonService... -->
  <aop:pointcut id="servOperation" expression="execution(* monProg.MonService.*(..))"/>
  <!-- est traitée par le gestionnaire de transactions tx. -->
  <aop:advisor  pointcut-ref="servOperation" advice-ref="tx"/>
</aop:config>

<tx:advice id="txAdvice" transaction-manager="tx">
  <tx:attributes>
    <!--les méthodes get sont en lecture seule. -->
    <tx:method name="get*" read-only="true"/>
    <!-- les autres méthodes sont aussi transactionnelles, mais en lecture/écriture.  -->
    <tx:method name="*"/>
  </tx:attributes>
</tx:advice>

<!-- déclaration d'un manager (adapté à la BD) -->
<bean id="tx" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

quelles méthodes ?

Qui les traite ?

Attributs de tx:method

- permettent de spécifier quelles méthodes sont transactionnelles, et comment.
- name : transactions concernées. C'est un motif.

  "get*" : toute méthode dont le nom commence par get

  "*" : valeurs par défaut.

  les méthodes qui ne correspondent à aucun motif ne sont pas transactionnelles.

Attributs de tx:method

- propagation : règle l'imbrication des transactions
  - REQUIRED : doit s'exécuter dans une transaction. Si une transaction existe, elle est utilisée
  - REQUIRED_NEW : s'exécute dans une *nouvelle* transaction. l'ancienne reprend après.
- isolation : isolement des transactions par rapport à d'autres transactions en parallèle

  DEFAULT; READ_UNCOMMITTED; READ_COMMITTED; REPEATABLE_READ; SERIALIZABLE;

Attributs de tx:method

- read-only: booléen, lecture seule (toujours pour l'utilisation en parallèle).
- rollback-for: exceptions qui déclenchent un RB, séparées par des virgules
- no-rollback-for: exceptions « acceptables ».

Hibernate

- Framework de persistence ORM (Object-Relationnal Mapping).
- http://www.hibernate.org/hib_docs/v3/reference/fr/html/

Mise en place

- Fichier de configuration du log à la racine du classpath (dans src)
- Fichier hibernate.cfg.xml idem. Les informations de ce dernier peuvent être fournies différemment.

Configuration générale

```
<?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="connection.driver_class">org.hsqldb.jdbcDriver</property>
      <property name="connection.url">jdbc:hsqldb:hsql://localhost/db</property>
      <property name="connection.username">sa</property>
      <property name="connection.password"></property>
      <property name="dialect">org.hibernate.dialect.HSQLDialect</property>
      <property name="current_session_context_class">thread</property>

      <!-- Fichiers de configuration des classes -->
      <mapping resource="demoHibernate/model/mapping.hbm.xml"/>

   </session-factory>

</hibernate-configuration>
```


Objets persistants

- disposent forcément d'un identifiant (normalement obligatoire en algèbre relationnelle) ;
- les modifications à un objet persistant sont répercutées dans la base de données

Déclaration des objets persistants

- en XML. Extensions pour le faire à l'aide d'annotations java 5 (ejb 3.0) actuellement en bêta.

```
<?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="demoHibernate.model">
      <class name="Article">
            <id name="id" column="idArticle">
                  <generator class="native"/> <!-- ou increment -->
            </id>
            <property name="designation" column="description"/>
            <property name="prix"></property>
      </class>

</hibernate-mapping>
```


Remarque

- identifiant: obligatoire. Très souple.
- Le setter de l'identifiant est souvent privé, pour que seul Hibernate le manipule.
- Pour que native fonctionne, il faut que la base « sache » autoincrémenter le champ. Sinon : increment.

Exemple de code

```
SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();

// B) On crée une session
Session session = sessionFactory.getCurrentSession();

// C) On commence à travailler
session.beginTransaction();

Article newArticle= new Article(); // On crée un objet
newArticle.setDesignation("toto");
newArticle.setPrix(2000);
session.save(newArticle); // On le sauve dans la base: il est lié à celle-ci !


newArticle.setPrix(2500); // Le prix est modifié dans la base !


// On ferme la session.
session.getTransaction().commit();
```


Remarque importante

- Un objet n'est *a priori* lié que le temps de la session.
- Une fois détaché de la session, les modifications qu'il subit ne sont plus prises en compte

Recherche : HQL

- Langage de recherche orienté objet

```
List<Article> l = session.createQuery(
        "from Article where prix > :minPrix").setDouble("minPrix", 100).list();

for (Object o : l)  {
            Article article= (Article)o;
            System.out.println(article);
}
```


Liens unidirectionnels

- Lien entre deux classes *dans une seule direction.* Par exemple, lien entre Facture et Client. La classe Facture a un lien vers un objet client, la classe Client n'a pas de lien vers la facture.

```
<class name="Facture">
    ....

    <many-to-one name="client" class="Client" column="clientId"/>
</class>
```

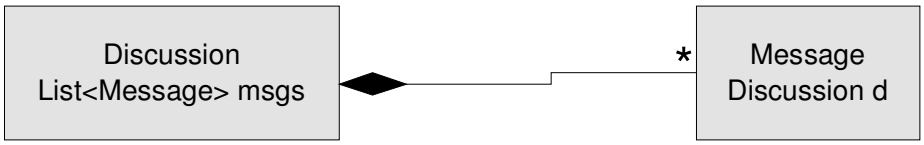
Liens unidirectionnels

- Relation n-n (ex. Une facture pour plusieurs clients)
  - Représentée par une collection de l'un des deux côtés

```
<class name="Facture">
    ...
  <set name="clients" table="FACTURE_VERS_CLIENT">
    <key column="idFacture"/> <!-- clef vers la facture dans la table de liaison -->
    <many-to-many column="idClient" class="Client"/>
  </set>
</class>
```


Liens bidirectionnels

- Les deux classes sont liées l'une à l'autre.



- Le code java doit garantir le lien dans les deux sens.

```
class Discussion {
      List msgs;
      ...
      void addMessage(Message m) {
            msgs.add(m);
            m.setDiscussion(this);
      }
}
```

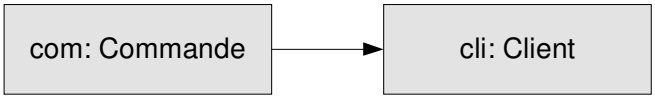
```
class Message {
      Discussion d;
      ...
      void setDiscussion(Discussion d) {
            this.d= d;
      }
}
```

```
<class name='Discussion'>
      ...
   <set name="msgs">
       <key column="idDiscussion" not-null="true"/>
       <one-to-many class="Message"/>
    </set>
</class>
<class name="Message">
      ...
      <many-to-one name='discussion'
            class="Message"
            column="messageId"
            not-null="true"
            inverse='true'
      />
</class>
```


Exercices

- Écriture et sauvegarde d'un client
- Écriture et sauvegarde d'une facture (en négligeant les articles)
- ajouter la gestion des articles, y compris dans la facture

Mise à jour récursives des objets



- pb: session.save(com) ne sauve pas le client (a priori).
- en réalité, rarement un vrai problème.
- Cependant...

```
<class name="Commande">
    ...
    <many-to-one name="client" column="loginClient"
                 class="Client"
                 cascade="persist,merge,save-update"/>
    ...
    </class>
```


Mise à jour récursives des objets

- L'attribut cascade permet de régler ce qui arrive aux objets liés à un objet persistant.
  - **none** (défaut) : rien n'est propagé
  - **persist, merge, delete... :** si le parent est passé à une méthode persist, delete..., alors l'objet lié l'est aussi.
  - **delete-orphan** : demande la destruction des données liées aux objets « orphelins », c'est à dire aux objets qui ne sont plus référencés par leur parent.

Session et objets

- Un objet java récupéré ou sauvé par hibernate est dit lié (à la session hibernate en cours)
- Les modifications sur cet objet sont répercutées dans la base (à la fin de la session)
- Une fois la session terminée, l'objet n'est plus lié
- On peut cependant rattacher un objet existant à une session

Rattacher un objet à une session

- session.update(obj) : rattache obj à la session. les données de la base concernant obj ne doivent pas avoir changé
- session.merge(obj) : rattache obj à la session, en écrasant d'éventuelles modifications dans la base

Le verrouillage pessimiste

- on bloque l'accès aux objets partagés
- sûr, mais trop coûteux:

  nbr de lecture >>>> nombre d'écriture !

Le verrouillage optimiste

- On attache à chaque entrée de la base un compteur.
- Si l'objet est modifié, le compteur augmente
- Pour savoir si un objet en mémoire est à jour, il suffit de comparer sa valeur du compteur à celle de la base
- Fonctionne bien si les objets sont surtout lus.

Objet versionné

- Pour utiliser le verrouillage optimiste, il suffit d'ajouter un champ de version dans les classes.
- Ce champ peut être de type divers: entier ou date
- les entiers sont plus fiables.

```
<class name='Client'>
    <id>....</id>
    <property ....>
    <version column='version'>
</class>
```


Le problème du chargement paresseux

- Par défaut, hibernate 3 charge partiellement les objets. La création réelle des objets liés ne se fait que lors de l'accès effectif aux champs.
- C'est transparent, sauf si la session se termine.
- On peut demander un chargement total dans la configuration XML
- Mais il est souvent plus pratique de le demander au niveau de la requête HQL

Contrôle du chargement en HQL

- Jointure interne (inner join):

R1=

idA	idB
A	1
B	2
C	3

R2=

idB	C
1	x
3	y
5	z

R1 inner join R2=

idA	idB	C
A	1	x
C	3	y

Contrôle du chargement en HQL

- Jointure ouverte à gauche (left outer join):

R1=

idA	idB
A	1
B	2
C	3

R2=

idB	C
1	x
3	y
5	z

R1 left outer join R2=

idA	idB	C
A	1	x
B	2	NULL
C	3	y

Contrôle du chargement en HQL

- sans chargement forcé:

  from Commande c join c.client where c.client.name ='turing'

- avec chargement forcé :

  from Commande c join **fetch** c.client where c.client.name ='turing'

- autre exemple:

  from Commande c **left join fetch** c.client

Spring et hibernate

- Configuration:

```
<bean id="sessionFactory"
        class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="mappingResources">
            <list>
                <value>/manuelDeCodage/hibernate/mdc.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.dialect">
                    org.hibernate.dialect.HSQLDialect
                </prop>
            </props>
        </property>
    </bean>
```

Spring et hibernate

```
<!-- Hibernate Transaction -->
    <bean id="transactionManager"
            class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>
```


Spring et hibernate

- Les implémentations des DAO étendent org.springframework.orm.hibernate3.support.HibernateDaoSupport

```
public class HibernateAuthorDAO extends HibernateDaoSupport implements
        AuthorDAO {

    public void create(Author author) {
        getHibernateTemplate().save(author);
    }
    public List<Author> findAuthorByBeginningOfName(String beginningOfName) {
        if (beginningOfName== null)
            beginningOfName="";
        String nameRegexp = beginningOfName.toUpperCase() + "%";
        return getHibernateTemplate().find("from Author where upper(surname) like ?",
                new Object[] { nameRegexp });}}
```

Session in Request

- pb: accès à des données non chargées dans la vue, alors que la session est terminée.
- Solution: la session englobe l'affichage.

```
<filter>
    <filter-name>hibernateFilter</filter-name>
    <filter-class>
        org.springframework.orm.hibernate3.support.OpenSessionInViewFilter
    </filter-class>
    <init-param>
        <param-name>singleSession</param-name>
        <param-value>true</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>hibernateFilter</filter-name>
    <url-pattern>/appli/*</url-pattern>
</filter-mapping>
```


Gestion du codage

```
<filter>
    <filter-class>
        org.springframework.web.filter.CharacterEncodingFilter
    </filter-class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>UTF-8</param-value>
    </init-param>
</filter>

<filter-mapping>
    <filter-name>Spring character encoding filter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```